TIGHT WORST CASE FOR THE DOUBLE ORACLE ALGORITHM

Anonymous authors
Paper under double-blind review

ABSTRACT

The double oracle algorithm (DO) is an iterative algorithm that computes a mixed-strategy Nash equilibrium of a game without fully expanding the players' strategy spaces. It is the basis of Policy-Space Response Oracles (PSRO) and many related population-based methods for multiagent reinforcement learning, where each iteration trains a new policy. In this paper, we give a general, explicit construction that maximizes the number of iterations DO needs to converge. Specifically, for all integers $M, N \geq 2$, we give an explicit M-by-N matrix that forces DO, starting from the first row and column, to run for $M + N - 2$ iterations. This equals the trivial worst-case upper bound. DO does not specify how to choose among multiple Nash equilibria or best responses, and our construction works for every choice. It also works when the equilibrium and best-response oracles are inexact, as in PSRO, provided they are sufficiently accurate. To our knowledge, this is the first such general construction. We formalize these results in Lean.

1 INTRODUCTION

Game theory is widely used to model situations involving multiple agents. A central problem in game theory is to compute a Nash equilibrium of a given game. A Nash equilibrium is a choice of strategy for each agent such that no agent has an incentive to change its strategy. A common special case is the two-player zero-sum game, which models pure competition. A two-player zero-sum normal-form game can be solved using linear programming. However, if the players' strategy sets are large, this can be prohibitively expensive.

The *double oracle algorithm (DO)* (McMahan et al., 2003) computes an equilibrium of the game without fully expanding every player's action set, which reduces the cost of computation. It is widely used in multiagent learning. *Policy-Space Response Oracles (PSRO)* (Lanctot et al., 2017) adapts DO for complex environments by computing best responses via deep reinforcement learning. It underlies a line of population-based methods (Muller et al., 2020; McAleer et al., 2020; 2021; Marris et al., 2021; McAleer et al., 2024). Similar population-based training grows a population of agents by repeatedly training new ones against mixtures of existing ones. AlphaStar used such training to reach grandmaster level in StarCraft II (Vinyals et al., 2019). This iterative addition of best responses is a core strategy-exploration technique in *empirical game-theoretic analysis (EGTA)* (Wellman, 2006; Wellman et al., 2025). EGTA uses simulations to estimate payoffs and construct a simplified "empirical game". Because each iteration trains or simulates a new strategy, the total number of iterations is the main computational bottleneck. The advantage of the iterative expansion approach hinges on the premise that it converges long before the entire strategy space is expanded. If the algorithm has to expand the entire strategy space, it costs more than solving the full game once.

In this paper, we give, for any integers $M, N \geq 2$, an explicit $M \times N$ matrix on which the double oracle algorithm, starting from the first row and the first column, takes the maximum possible number of iterations, namely $M + N - 2$. Our construction works regardless of how the algorithm chooses Nash equilibria or best responses. The matrix is also defined when M or N is infinite, and our lemmas then give a run that never stops. Rows are added in order of their index, and so are columns. The matrix is not a knife edge: every matrix close enough to it entrywise forces the same run (Corollary 2). It plays the role for DO that the Klee–Minty cube (Klee & Minty, 1972) plays for the simplex method (Dantzig, 1951). It shows that the worst case can be forced, not that it is typical.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Section 2 reviews two-player zero-sum games and the double oracle algorithm, with the general game-theoretic definitions and the variants of DO in Appendix C. Section 3 reviews the prior work on its complexity. Section 4 sets up our proof. Section 5 presents our main result and shows that it survives approximate equilibrium and best-response oracles. Section 6 concludes with directions for future research. Appendix D describes our empirical validation. Appendix F presents our Lean proof.

2 BACKGROUND

2.1 MATHEMATICAL NOTATION USED IN THIS PAPER

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of natural numbers, including 0. Let $\mathbb{Q}$ be the set of rational numbers. Let $\mathbb{R}$ be the set of real numbers. If $n \in \mathbb{N}$, let $[n] = \{0, \ldots, n-1\}$ be the set of natural numbers that are strictly less than n. If ψ is a logical formula, let $[\psi]$ be its Iverson bracket, which is 1 if ψ holds and 0 otherwise. If S is a set, $\triangle S$ is the set of probability distributions on S. If A is a matrix, A^{T} is its transpose, such that $A_{ij} = (A^{\mathrm{T}})_{ji}$. If f is a function, the substitution $f[a \mapsto b]$ is the function such that $f[a \mapsto b](x) = b$ if $x = a$ and $f(x)$ otherwise.

2.2 TWO-PLAYER ZERO-SUM GAMES

In a *two-player zero-sum game*, the two players' utilities sum to zero. Such games model pure competition. von Neumann (1928) proved the *minimax theorem* for them. If X and Y are the players' strategy sets and $f : X \times Y \to \mathbb{R}$ is the payoff to Player 1, then

$$\sup_{x \in X} \inf_{y \in Y} f(x, y) = \inf_{y \in Y} \sup_{x \in X} f(x, y) \tag{1}$$

under suitable conditions on X, Y, and f (von Neumann & Morgenstern, 1947), and this common quantity is the *value of the game*.

Definition 1 (Saddle point)**.** A *saddle point* is a pair $(x^*, y^*) \in X \times Y$ for which

$$\sup_{x \in X} f(x, y^*) \leq f(x^*, y^*) \leq \inf_{y \in Y} f(x^*, y). \tag{2}$$

A saddle point is precisely a *Nash equilibrium (NE)* (Nash, 1951) of the game, that is, a strategy profile in which no player gains by deviating unilaterally.

Definition 2 (Duality gap)**.** Given a utility function $f : X \times Y \to \mathbb{R}$, a strategy $x \in X$ for the maximizing player, and a strategy $y \in Y$ for the minimizing player, the duality gap is

$$d(x, y) = \sup_{x' \in X} f(x', y) - \inf_{y' \in Y} f(x, y'). \tag{3}$$

The duality gap is nonnegative, and is zero exactly at Nash equilibria. For two players, it is the *exploitability*, the standard measure of how far a strategy profile is from being an NE (Lanctot et al., 2017; Lockhart et al., 2019; Walton & Lisý, 2021; Timbers et al., 2022).

The simplest two-player zero-sum games are *matrix games*, whose utilities are specified by a single *payoff matrix* $A \in \mathbb{R}^{M \times N}$, where $M \in \mathbb{N}$ is the number of actions for the row player and $N \in \mathbb{N}$ is the number of actions for the column player. If the row player chooses action $i \in [M]$ and the column player chooses action $j \in [N]$, the payoff to the former is A_{ij} and the payoff to the latter is $-A_{ij}$. Mixed strategies are points of the simplices $x \in \triangle[M]$ and $y \in \triangle[N]$, and the minimax theorem applies with $X = \triangle[M]$, $Y = \triangle[N]$, and the *expected payoff* $f(x, y) = x^{\mathrm{T}} A y$. A matrix game can be solved (that is, a Nash equilibrium (x, y) computed) by *linear programming (LP)* (Luce & Raiffa, 1957; Adler, 2013; Brooks & Reny, 2023). Solving large linear programs is expensive, which motivates algorithms that avoid it. Appendix C gives the general definitions of normal-form games, best responses, equilibria, regret, and exploitability.

2.3 THE DOUBLE ORACLE ALGORITHM

The *double oracle algorithm (DO)* (McMahan et al., 2003) computes a mixed-strategy NE of a game without fully expanding the players' strategy spaces. It is commonly used to approximate one in

games with large strategy spaces. It keeps a set of actions for each player, called an *active set*. The initial active set for each player is usually a singleton, containing just one strategy for that player. Each iteration computes an NE of the *subgame* spanned by the active sets. In the two-player zero-sum case, this can be done using linear programming. It then computes, for each player, a *best response (BR)* in the full game to that equilibrium, and adds it to that player's active set. It terminates when no active set changes, or when the exploitability falls below a given tolerance. We count the iterations until the active sets stop changing. Section 4 defines DO precisely for matrix games, our setting.

At any point, there may be multiple NE and/or BRs to choose from. DO itself does not specify how to choose them. The rules that choose them are the *equilibrium selection scheme* and the *best-response selection scheme*. Both are external to the algorithm.

Many variants of DO have since been proposed: for multiple players (Kroupa & Votroubek, 2022), for continuous strategy sets (Adam et al., 2021), for policies trained by reinforcement learning (Lanctot et al., 2017), with monotone exploitability (McAleer et al., 2022; 2024), and with no-regret learners in place of the linear program (Dinh et al., 2021). They keep DO's core loop of solving a subgame and adding BRs. Hence, their cost still grows with the number of iterations and with the size of the subgames they solve. Our construction shows how large both can get for DO itself: on an $M \times N$ matrix, the subgame can grow one strategy at a time until it is the full game. Appendix C.2 gives the multi-player version of the algorithm and what our construction implies for each of these variants.

3 Prior work on the complexity of the double oracle algorithm

Since DO aims to find a Nash equilibrium without expanding the full game, it is natural to ask how many iterations it can take in the worst case. Zhang & Sandholm (2024) proved exponential lower bounds on the number of iterations of DO in both *partially-observable stochastic games (POSGs)* and *extensive-form games (EFGs)*. Their results depend on what is assumed about the tie-breaking scheme, that is, which NE or BR is chosen when there are several. For EFGs, their lower bounds require adversarial tie-breaking. For POSGs, two of them apply regardless of how ties are broken. Our work differs from theirs in the following respects.

Explicit matrices, not compact games. Their games are POSGs and EFGs indexed by a parameter k, with only $O(k)$ or $\mathrm{poly}(k)$ states. Their bounds are exponential in k. This requires exponentially many pure strategies, since DO adds at least one per iteration until it stops. We work directly with matrix games, the setting for which DO was introduced (McMahan et al., 2003). For $M, N \geq 2$, our $M \times N$ matrix is given in closed form (Definition 8). Its entries are rational, and only $2\max(M, N)-1$ are nonzero.

Exact count, matching the trivial upper bound. None of their theorems shows that a game can force DO to reach the trivial upper bound. Their Theorem 3.2 gives $\Theta(2^k)$ iterations in expectation, and their Theorem 3.5 at least 2^{k-1}, in games where each player has 2^k pure strategies. On the n-bigger-number game underlying their Theorem 3.2, DO started from 0 for both players adds the next number for both players on every iteration, and stops after $n-1$ iterations. This yields half the $2n-2$ iterations that our construction forces when $M = N = n$. In their Theorem 3.1, the run they describe adds one strategy per iteration and so reaches the trivial upper bound. For $k \geq 2$, other BRs let DO add two strategies at once. On our matrix, every run adds exactly one strategy per iteration.

Rectangular, not just square. Each of their constructions is a one-parameter family in which both players have the same number of pure strategies. Ours covers every $M, N \geq 2$. When $N > M$, the corridor of Definition 8 makes the column player add the remaining $N - M$ columns one at a time, once every row is present.

Exact and deterministic under every selection rule. Their Theorem 3.1 holds for every choice of initialization, subgame equilibria, and BRs, with a bound of $2^{\Theta(k)}$ iterations to find an exact equilibrium. Their Theorem 3.2 makes no assumption on how subgame NE and BRs are chosen. It draws the initial strategies uniformly at random and bounds the expected number of iterations. Their other results assume that some of these choices are made adversarially. From our initial state, every state along the run has a unique successor, so every rule for selecting NE and BRs yields the same

run. The subgame equilibrium is in fact unique at every state, so the best-response rule is the only one with any freedom to begin with. Unlike their Theorem 3.1, ours fixes the starting row and column. By Theorem 4, no $M \times N$ matrix forces $M + N - 2$ iterations from every choice of starting row and column, unless $M = N = 2$.

Termination, not approximation. In their Theorems 3.2–3.5, the game has an NE in which each player mixes over at most two pure strategies. Yet, under the assumptions of each theorem, DO takes exponentially many iterations to find even an approximate equilibrium, for every tolerance below a threshold that is constant or $\Theta(1/k)$. We instead count the iterations until DO terminates with an exact NE. In this specific respect, our construction is weaker. After t iterations on our matrix, the duality gap is at most $4/(t+1)$ (Table 1), so DO finds a τ-NE within $4/\tau$ iterations. No matrix with entries in $[-1, 1]$ holds the gap away from zero at every shape. Proposition 1 bounds the gap along any full-length run by the shape of the matrix, and drives it to zero as one side outgrows the other.

The two papers are thus complementary. Theirs shows that DO can need exponentially many iterations in the size of a compact game, even to approximate an NE. Ours shows that, for all $M, N \geq 2$, some $M \times N$ matrix game makes DO reach the trivial upper bound from a fixed initial state under every selection rule.

4 Preliminaries of our proof

4.1 Double oracle on a matrix game

Definition 3 (State)**.** A *state* of the double oracle algorithm is a pair (R, C) of a set of rows $R \subseteq [M]$ and a set of columns $C \subseteq [N]$.

Definition 4 (Successor)**.** Let (x, y) be a mixed-strategy Nash equilibrium of the *subgame* $A|_{R\times C}$, the restriction of A to the rows R and the columns C. That is, $x \geq 0$ on R with $\sum_{i\in R} x_i = 1$, and $y \geq 0$ on C with $\sum_{j\in C} y_j = 1$. The row and column payoffs are

$$w_i = \sum_{j\in C} A_{ij} y_j \quad \forall i \in [M] \qquad\qquad u_j = \sum_{i\in R} A_{ij} x_i \quad \forall j \in [N], \tag{4}$$

and the equilibrium condition (i.e., strong duality) is $\min_{j\in C} u_j \geq \max_{i\in R} w_i$. Let $i_{\text{BR}} \in R_{\text{BR}} = \operatorname{argmax}_{i\in[M]} w_i$ be a BR row and $j_{\text{BR}} \in C_{\text{BR}} = \operatorname{argmin}_{j\in[N]} u_j$ be a BR column. Let $R' = R \cup \{i_{\text{BR}}\}$ and $C' = C \cup \{j_{\text{BR}}\}$. Then (R', C') is a *successor* of (R, C), written $(R, C) \to (R', C')$.

The algorithm stops when $R' = R$ and $C' = C$.

Definition 5 (Run)**.** A *run* is a sequence of states where each state after the first is a successor of the preceding state.

Definition 6 (Unique successor)**.** For states a and b, $a \xrightarrow{!} b$ means that b is the *unique* successor of a: $a \to b$, and $a \to c$ implies $c = b$.

Lemma 1 (Trivial upper bound)**.** Starting from $R = \{i\}, C = \{j\}$, the double oracle algorithm takes at most $M + N - 2$ iterations.

Proof. DO starts with 2 of the $M + N$ strategies and adds at least 1 per iteration until it stops. □

For an equilibrium (x, y) of $A|_{R\times C}$,

$$\min_{j\in C} u_j \leq \sum_{j\in C} y_j u_j = \sum_{i\in R}\sum_{j\in C} x_i A_{ij} y_j = \sum_{i\in R} x_i w_i \leq \max_{i\in R} w_i, \tag{5}$$

so equality holds in the equilibrium condition, and $v = \min_{j\in C} u_j = \max_{i\in R} w_i$ is the *value* of the subgame. A column $j \in C$ is *tight* if it pays exactly v against x and *slack* if it pays more. A row $i \in R$ is tight if it pays exactly v against y. Two standard consequences follow, proved in Appendix A.1: Lemma 2 verifies an equilibrium from unnormalized weights, and Lemma 3 says that a pure strategy in the support of one equilibrium is tight in every equilibrium.

Lemma 2 (Certificate)**.** Let $x : R \to \mathbb{R}$ and $y : C \to \mathbb{R}$ be nonnegative and nonzero, with payoffs $u_j = \sum_{i \in R} A_{ij} x_i$ and $w_i = \sum_{j \in C} A_{ij} y_j$, and let $v, \lambda \in \mathbb{R}$. If $u_j \geq v$ for all $j \in C$, with equality whenever $y_j > 0$, and $w_i = \lambda$ for all $i \in R$, then $x / \sum_{i \in R} x_i$ and $y / \sum_{j \in C} y_j$ form an equilibrium of $A|_{R \times C}$, with value $v / \sum_{i \in R} x_i$.

Scaling x by a positive constant scales every u_j by that constant, so best-response columns can be read off from an unnormalized x, which we do freely.

Lemma 3 (Complementary slackness)**.** Let (x, y) and (x', y') be equilibria of $A|_{R \times C}$, with payoffs u_j, w_i and u'_j, w'_i. Then they have the same value v. Moreover, $u'_j = v$ whenever $y_j > 0$, and $w'_i = v$ whenever $x_i > 0$.

4.2 Our construction

Definition 7 (Fibonacci)**.** Let $F : \mathbb{N} \to \mathbb{N}$ be the Fibonacci sequence. It satisfies $F(0) = 0$, $F(1) = 1$, and $F(n+2) = F(n) + F(n+1)$.

Definition 8 (Construction)**.** Let $M \in \mathbb{N}$ be a number of rows and $N \in \mathbb{N}$ be a number of columns. If $N \geq M$, let $A : [M] \times [N] \to \mathbb{Q}$ be the following matrix:

$$A_{ij} = \begin{cases} [i = j] - [i + 1 = j] & j \in \{0, \ldots, M-1\} \quad \text{(staircase)} \\ \frac{(-1)^n}{MF(n)}, \quad n = i + j - M + 4 & j \in \{M, \ldots, N-1\}, i \leq 1 \quad \text{(corridor)} \\ 0 & \text{otherwise} \end{cases}. \tag{6}$$

If $N < M$, take instead the negative transpose of the $N \times M$ matrix defined above with M and N exchanged, which switches the roles of the row and column players.

For example, if $M = 5$ and $N = 10$, then

$$A = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 1/15 & -1/25 & 1/40 & -1/65 & 1/105 \\ 0 & 1 & -1 & 0 & 0 & -1/25 & 1/40 & -1/65 & 1/105 & -1/170 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{7}$$

The first M columns form an upper-bidiagonal *staircase*. For $M, N \geq 2$, the remaining columns form a two-row *corridor*. The staircase makes its rows and columns enter alternately and in increasing order, each as the only new BR. The corridor, whose entries alternate in sign and shrink, then makes its columns enter one at a time.

Interpretation of the staircase. Row i beats column i and loses to column $i + 1$. Every other pair is a draw. So the column player gains by standing exactly one step above the row player, and gains nothing by standing further above. The row player gains by matching the column player's index. The global form of the staircase, in which any higher index wins, is the bigger-number game. There, DO needs only half as many iterations (Section 3), since both players move on every iteration. On the staircase, only one player can improve at each state, so the players alternate, and each iteration adds one strategy instead of two. The corridor has an interpretation of its own, as successive rational approximations to the row player's mixing ratio (Appendix A.3).

Sparsity. For $M, N \geq 2$, the matrix is sparse, with only $2 \max(M, N) - 1$ nonzero entries: $2 \min(M, N) - 1$ on the two diagonals of the staircase, and two in each corridor column.

Infinite cases. The matrix is well-defined when M or N is infinite. If both are infinite, it is an infinite staircase. If only one is, it is a finite staircase and infinite corridor.

From now on, $2 \leq M \leq N$, so A is given by the first case of Definition 8. The main theorem reduces $N < M$ to this case. Let $s(n) = \frac{(-1)^n}{MF(n)}$, so the corridor entries are $A_{ij} = s(i + j - M + 4)$. Corridor column j has $s(n)$ in row 0 and $s(n+1)$ in row 1, where $n = j - M + 4 \geq 4$ is its *index*. Row i has a 1 in column i, a -1 in column $i + 1$ if $i + 1 \leq M - 1$, and, if $i \leq 1$, the corridor entries.

Strategies are extended by zero outside R and C, with $x_{-1} = 0$. For $x_0, x_1 \in \mathbb{R}$, write $g(n) = (-1)^n(x_0F(n+1) - x_1F(n))$, the numerator of the corridor payoffs. Reading off the columns of A, column j pays against x

$$u_j = x_j - x_{j-1} \qquad \text{if } j \in [M], \tag{8}$$

$$u_j = x_0\, s(n) + x_1\, s(n+1) = \frac{g(n)}{MF(n)F(n+1)} \qquad \text{if } j \geq M, \text{ where } n = j - M + 4. \tag{9}$$

If $x \geq 0$, then (9) gives $|u_j| \leq \frac{x_0+x_1}{3M}$ for every corridor column, since $F(n) \geq F(4) = 3$. This crude bound is all we need while rows are still being added. It is the only way the corridor enters the staircase phase: any entries bounded by $\frac{1}{3M}$ in rows 0 and 1 would do. This changes once every row is present, when the first corridor column must itself become the BR. The bracket in g inherits the recurrence $F(n+2) = F(n+1) + F(n)$, and the factor $(-1)^n$ flips the sign of its middle term, giving $g(n+2) = g(n) - g(n+1)$. So once two consecutive values of g differ in sign, all later ones alternate, and so do the corridor payoffs. The next lemma, proved in Appendix A.1, adds that they shrink, so the first negative one is the column player's BR. This is what makes the corridor columns enter one at a time.

Lemma 4 (Alternation)**.** Let $n_0 \geq 4$. If $g(n_0 - 1) > 0$ and $g(n_0 - 1) \leq -2g(n_0)$, so that g has just turned negative and its magnitude has at most halved, then against x the corridor column of index n_0 pays a negative amount, and strictly less than every corridor column of higher index.

5 Main result

In this section, we present the main result.

Definition 9 (State families)**.** The states that DO encounters on our matrix fall into three families: the even staircase steps $S_k = ([k], [k])$ for $1 \leq k \leq M$, the odd ones $T_k = ([k], [k+1])$ for $1 \leq k \leq M-1$, and the corridor states $C_k = ([M], [k])$ for $M \leq k \leq N$. Note $S_M = C_M$.

Definition 10 (Garden path)**.** The *garden path* is the sequence of states

$$S_1, T_1, S_2, T_2, \ldots, S_{M-1}, T_{M-1}, S_M = C_M, C_{M+1}, \ldots, C_N. \tag{10}$$

The garden path starts at the initial state $(\{0\}, \{0\}) = S_1$. Theorem 1 shows that it is the only run DO can take from there. Examples of this run for different M and N are illustrated in Appendix D. In the *staircase phase*, the run alternately adds a column, $S_k \to T_k$ (a step right), and a row, $T_k \to S_{k+1}$ (a step down), until every row is present at S_M. In the *corridor phase*, it adds the corridor columns one at a time, $C_k \to C_{k+1}$, until every column is present at C_N. Each of the four lemmas below exhibits an equilibrium of the subgame. Lemma 3 transfers the tightness conditions of that equilibrium to *every* equilibrium. Each lemma then compares the payoffs of all M rows and all N columns, which leaves a single new BR at every state except the last. In each lemma, the tight columns are exactly $|R|$ in number and linearly independent over the rows of R, and the exhibited equilibrium puts positive weight on every tight column and on every row of R. By Lemma 3, each half of any other equilibrium then solves the same invertible system. Hence, the subgame at every state of the run has exactly *one* equilibrium, and the equilibrium selection scheme has nothing to select. It is the best-response scheme that the lemmas have to contend with. The proofs are in Appendix A, and the paragraphs that follow the statements say what drives each phase. Each lemma states everything the rest of the paper uses about its state: the unique successor, the value v, the *margin* m, and, at the corridor states, the weight of the tight column that Theorem 2 needs. The margin is the payoff of the new strategy, counted in its player's favor: the new column pays $-m$, and the new row pays m.

Lemma 5 (Staircase right)**.** Let $k \in \{1, \ldots, \min(M, N-1)\}$. Then $S_k \xrightarrow{!} ([k], [k+1])$, which is T_k if $k < M$ and C_{M+1} if $k = M$. The subgame has value $v = \frac{2}{k(k+1)}$, and the new column pays $-m$, where the margin m is $kv = \frac{2}{k+1}$ if $k < M$ and $\frac{2}{15M^2(M+1)}$ if $k = M$. For $k = M$, some equilibrium gives column 0 weight at least Mv.

Lemma 6 (Staircase down)**.** Let $k \in \{1, \ldots, M-1\}$. Then $T_k \xrightarrow{!} S_{k+1}$. The subgame has value $v = -\frac{2}{k(k+1)}$, and the new row pays the margin $m = \frac{2}{k+1}$.

The certificates are arithmetic progressions: $x_i = i + 1$ and $y_j = k - j$ at S_k, and $x_i = k - i$ and $y_j = j$ at T_k. Because the payoffs (8) telescope, a common difference of 1 makes every strategy of

the subgame tight, except column 0 at T_k. For $k < M$, the strategy just outside the subgame decides the step. At S_k, column k pays $-k$. The staircase columns after it pay 0, and the corridor columns pay at least $-\frac{x_0+x_1}{3M} > -k$. At T_k, row k pays $y_k = k$, while every row of the subgame pays -1.

In the corridor phase, the row player holds every row, so only columns move. The staircase columns $2, \ldots, M-1$, the *band*, are tight throughout and meet no corridor entry. From C_{M+1} on, each certificate is fixed by a 2×2 system. The two other tight columns meet rows 0 and 1 in a matrix H. The row weights (x_0, x_1) make those two columns pay the value, and their own weights make rows 0 and 1 pay it. With a single corridor column, C_{M+1} needs an equilibrium of its own, in which the two tight columns are staircase column 1 and that corridor column. The remaining states form a single family. There, the two tight columns are the two most recent corridor columns, the *frontier*. Then $\det H$ rests on one classical fact about Fibonacci numbers, Cassini's identity $F(n-1)F(n+1) - F(n)^2 = (-1)^n$ for $n \geq 1$ (Graham et al., 1994; Koshy, 2001), proved as Lemma 9 in Appendix A.3.

Lemma 7 (Handoff)**.** If $N \geq M+2$, then $C_{M+1} \xrightarrow{!} C_{M+2}$. The subgame has value $v = \frac{1}{M(8M+1)}$, the new column pays $-m$ with margin $m = \frac{45M-1}{80M^2(8M+1)}$, and some equilibrium gives column 1 weight at least $\frac{5M-2}{2M(8M+1)}$.

Lemma 8 (Corridor)**.** Let $k \in \{M+2, \ldots, N-1\}$, and write $K = k - M + 3 \geq 5$. Then $C_k \xrightarrow{!} C_{k+1}$. The subgame has value

$$v = V_k = \left(MF(K)\big(MF(K+1)^2 - F(K)^2\big) + \frac{(M-1)(M-2)}{2} \right)^{-1}, \tag{11}$$

the new column pays $-m$ with margin $m = \frac{V_k F(K)^2}{F(K+1)F(K+2)}$, and some equilibrium gives column $k-2$ weight at least $\frac{1}{6}$.

Cassini's sign $(-1)^n$ matches that of the corridor entries, so the frontier pays the same positive amount for every K, as does the band. Every earlier corridor column pays more, since g is positive below the frontier and decreasing in its index. At the column to be added, g has just turned negative and its magnitude has at most halved, so Lemma 4 makes it the unique BR.

Theorem 1 (Main theorem)**.** Let $M, N \geq 2$ be integers. Every run of the double oracle algorithm on the matrix A of Definition 8, starting from $R = \{0\}, C = \{0\}$, terminates at $R = [M], C = [N]$ after exactly $M + N - 2$ iterations. Furthermore, the rows and columns are added in increasing order.

Our theorem shows that exactly *one* strategy is added on each iteration, in order, and that no proper subgame along the way is sufficient.

Proof. Without loss of generality, $N \geq M$. If $N < M$, the negative transpose $-A^{\mathrm{T}}$ switches the roles of the two players. The initial state $(\{0\}, \{0\})$ is S_1. By the staircase, handoff, and corridor lemmas (Lemmas 5, 6, 7, and 8), each state in Definition 10 has the next one as its unique successor. Each transition adds exactly one strategy, the next row or column, so rows and columns are added in increasing order. The final state $C_N = ([M], [N])$ contains every row and column, so its best responses lie inside it and the algorithm stops there. The sequence has $2(M-1) + (N-M+1) = M + N - 1$ states, hence $M + N - 2$ iterations. □

Along this run, the intermediate subgames are also as large as possible: among runs of $M + N - 2$ iterations, ours maximizes $|R| \cdot |C|$ at every iteration, as Appendix A.4 shows.

The lemmas also fix the *duality gap* $d_t = \max_{i \in [M]} w_i - \min_{j \in [N]} u_j$ of the equilibrium after t iterations (Definition 2). It is the difference between the two best-response payoffs, and bounds the distance from v_t to the value of A. The player that does not move has best payoff v_t, attained in the subgame. The other player adds a column that pays $-m_t$ when $v_t > 0$, and a row that pays m_t when $v_t < 0$. So the gap is $d_t = |v_t| + m_t$. Table 1 in Appendix B collects all three at every state. The gap decays along the run, and that appendix asks how much of the decay is forced. No matrix with entries in $[-1, 1]$ holds the gap away from zero at every shape, since a run of $M + N - 2$ iterations whose gap never falls below γ forces $N \leq (1 + 2/\gamma)^M$. At $M = N$, where that leaves no constraint, the bigger-number game under a tilt comes arbitrarily close to the largest gap the entries allow.

Since every state along the run has a unique successor, the run also survives small errors in the equilibria and BRs. Such errors arise in PSRO, whose BRs are computed by reinforcement learning. For $\delta, \varepsilon_R, \varepsilon_C \geq 0$, the *approximate* double oracle algorithm relaxes the definition of a successor. The pair (x, y) need only be a δ-*equilibrium* of the subgame, with $\min_{j \in C} u_j \geq \max_{i \in R} w_i - \delta$. The row player need only answer to within ε_R, with $w_{i_{\text{BR}}} \geq \max_{i \in [M]} w_i - \varepsilon_R$, and the column player to within ε_C, with $u_{j_{\text{BR}}} \leq \min_{j \in [N]} u_j + \varepsilon_C$.

The bounds below use the value v_t and the margin m_t of Table 1 through the single quantity

$$\mu = \min_{t < M+N-2} \min(|v_t|, m_t). \tag{12}$$

The row player's threshold will involve only the staircase states, where that minimum is $|v_t|$, so write $\mu_S = \frac{2}{M(M-1)}$ for its least value there.

Theorem 2 (Oracle accuracy)**.** Let $2 \leq M \leq N$, and run the approximate double oracle algorithm on the matrix A of Definition 8, starting from $R = C = \{0\}$. If $\delta \leq \mu/64$, $\varepsilon_C \leq \mu/64$, and $\varepsilon_R \leq \mu_S/64$, then every run is the run of Theorem 1, and so takes $M + N - 2$ iterations.

Table 1 also bounds every term of (12) at once. This makes the threshold of Theorem 2 explicit: for $2 \leq M \leq N$,

$$\mu \geq \frac{1}{8M^3F(N - M + 3)^3}. \tag{13}$$

Appendix A.5 proves this and Theorem 2. The row player's threshold does not involve N. Rows are added only in the staircase phase. Once every row is present, no answer of the row oracle can change the state. So only δ and ε_C have to sharpen as the corridor lengthens.

Corollary 1 (Approximate oracles)**.** Let $M, N \geq 2$. There are some $\delta_0, \varepsilon_0 > 0$ such that Theorem 1 holds for the approximate double oracle algorithm whenever $\delta \in [0, \delta_0)$ and $\varepsilon_R, \varepsilon_C \in [0, \varepsilon_0)$.

Corollary 2 (Perturbed matrices)**.** Let $M, N \geq 2$. There is a $\rho > 0$ such that every run of the double oracle algorithm on any $M \times N$ matrix A' with $|A'_{ij} - A_{ij}| < \rho$ for all i, j, starting from $R = C = \{0\}$, is the run of Theorem 1.

The worst case is therefore not an artifact of the exact entries of Definition 8: every matrix in a ball around A forces it. That ball is small. Its radius ρ is proportional to μ, which by (13) and Table 1 is polynomially small in M when $N = M$ and exponentially small in $N - M$.

Theorem 3 (Oracle accuracy, necessity)**.** Let $2 \leq M \leq N$, and run the approximate double oracle algorithm on the matrix A of Definition 8, starting from $R = C = \{0\}$. Then some run takes fewer than $M + N - 2$ iterations in each of the following cases.

1. At some state t before the last, the tolerance of the player that moves there is at least d_t.
2. $\varepsilon_R \geq \mu_S$.
3. $M\varepsilon_R \leq 1$ and $\delta + \frac{2\varepsilon_R}{M+1} \geq \frac{4}{M^2-1}$.

For the row player's tolerance, the two theorems differ only by a constant. Case 2 of Theorem 3 breaks the run once ε_R reaches μ_S, the value at S_{M-1}, and Theorem 2 allows a fixed fraction of μ_S. This threshold is polynomially small in M and free of N. When $N = M$, the same holds for δ. Then $\mu = \mu_S$, and case 3 breaks the run once δ reaches a threshold below $2\mu_S$. In the corridor, the two theorems are further apart, since the margin there is a small fraction of the duality gap that case 1 needs. By cases 2 and 3, a sharper equilibrium oracle does not make up for a coarser best-response oracle, nor the other way round.

By Table 1, μ and the gaps d_t decay exponentially in $N - M$. So for $N > M$, the column player's oracle must be exponentially accurate, by case 1 of Theorem 3. Theorem 2 asks the same of the equilibrium oracle, but not of the row player's, whose threshold μ_S does not involve N. The argument for case 1 applies to any matrix. If every approximate run on it takes $M + N - 2$ iterations, then along each run the duality gap exceeds the moving player's tolerance at every state before the last. Proposition 1 forces that gap to vanish as N outgrows M, so no construction with entries in $[-1, 1]$ tolerates a fixed accuracy for all $M, N \geq 2$. For $N < M$, the negative transpose switches the players, as in Theorem 1.

6 CONCLUSION

We gave, for any integers $M, N \geq 2$, an explicit $M \times N$ matrix on which the double oracle algorithm, starting from the first row and the first column, takes exactly $M + N - 2$ iterations under every selection scheme. This exactly matches the trivial worst-case upper bound. The same holds when the oracles are approximate, as long as their tolerances stay below a fixed fraction of the smallest value or margin along the run (Theorem 2). It can fail once the row player's tolerance reaches the value at S_{M-1}, or once the moving player's tolerance reaches the duality gap at some state (Theorem 3).

The two phases are not equally robust to that inexactness. At the staircase states, the value and the margin of Table 1 are bounded below by an inverse polynomial in M. The argument for Theorem 2 is state by state. Hence, accuracies of order M^{-2} already force the first $2(M-1)$ iterations, regardless of N. The row player's oracle needs no more than that at any state, since rows stop being added once the corridor begins. At the corridor states, the value and the margin decay exponentially in $N - M$. By Theorem 3, the column player's oracle must then be exponentially accurate. The oracles of deep reinforcement learning do not grow exponentially accurate as a game grows. Against them, the part of this lower bound that survives is the staircase, of length $2\min(M, N) - 2$. The corridor can break, and the run can then stop early. For a practitioner, this is the benign direction: it means fewer iterations, not more. This suggests looking for bounds whose margins shrink polynomially in the size of the game, not exponentially. Theorems 3.2–3.5 of Zhang & Sandholm (2024) are of this kind, with tolerances that are constant or $\Theta(1/k)$ on games of size $\mathrm{poly}(k)$.

6.1 FUTURE RESEARCH

Open Problem 1 (Single matrix)**.** Is there a *single* infinite matrix whose every truncation guarantees the worst-case number of iterations? In other words, is there a construction A whose entries A_{ij} are independent of *both* M and N? (Ours is independent of the larger of M and N.)

Open Problem 2 (Large gap)**.** How large can the duality gap stay along a run of $M+N-2$ iterations, on a matrix with entries in $[-1, 1]$? Propositions 1 and 2 settle the square case. For $M < N$, the cap is $2/(N^{1/M} - 1)$, which still permits N exponential in M at a constant gap. Is that cap tight, and if not, how large can the gap stay as a function of M and N?

Open Problem 3 (Multiple players)**.** Let $P \in \mathbb{N}$ be a number of players. Given integers $N_i \geq 2$ for $i \in [P]$, give a payoff tensor A that maximizes the number of iterations that DO is forced to take from some initial state. (Our results are for two players, the most typical setting for DO.)

There, the trivial upper bound becomes $\sum_{i \in [P]} N_i - P$. In a zero-sum polymatrix game (Appendix C.1), every subgame still has a value and an equilibrium computable by linear programming. This is what Lemmas 2 and 3 rest on. A generalization of our construction would have to force exactly one player at a time to add an action.

Open Problem 4 (Compact representation)**.** Is there a game of size $\mathrm{poly}(k)$, say a partially-observable stochastic game or an extensive-form game, whose reduced normal form is the matrix of Definition 8 with $M = N = 2^k$? Double oracle on such a game, with oracles that return reduced pure strategies, is double oracle on that matrix. It would therefore take $2^{k+1} - 2$ iterations, exponentially many in the size of the description, under *every* selection rule. The extensive-form bounds of Zhang & Sandholm (2024) instead require adversarial tie-breaking. With $M = N$, the matrix is the staircase alone. Its payoff asks only whether the two chosen indices are equal or adjacent. A game tree can decide that in $O(k)$ steps, on indices written in binary. The smallest staircase value is of order M^{-2}, so by Theorem 3 the row oracle would still need accuracy exponential in k.

Open Problem 5 (Any start)**.** Given integers $M, N \geq 2$, give a matrix A that maximizes the *minimum* (or the average), over *all* initial states $R = \{i\}, C = \{j\}$, of the number of iterations that DO is forced to take from that state. (Our construction assumes that the initial state is $R = C = \{0\}$.)

Unless $M = N = 2$, this minimum is below $M + N - 2$ for every matrix (Theorem 4).

Open Problem 6 (DO variants)**.** Is there an analogous construction that forces the worst case for variants of DO, like *Anytime Double Oracle (ADO)* (McAleer et al., 2022; 2024) and *Online Double Oracle (ODO)* (Dinh et al., 2021)? (Ours does not once $N > M + 2$: on it, ADO stops after $\min(M + N - 2, 2M)$ iterations, as Appendix C.2 shows.)

AI USE STATEMENT

In this work, we used generative AI tools to polish writing, to retrieve some related work, to flesh out some of the proof sketches, to formalize the proofs in Lean, and to write the empirical verification code in Python. We have not used generative AI tools to generate synthetic data sets or to create the research ideas. We have reviewed all AI-assisted work. The LLM-generated Lean code was verified and tested for correctness. We take responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

Each numbered result states its assumptions, and Appendices A and B provide every proof that is not in the main text. Appendix F presents a formal proof in Lean of the results listed in Table 3, which names the declaration that proves each one. Appendix E presents a script that re-checks the quantitative claims over the rationals. Appendix D describes the script that runs DO on our matrix in exact arithmetic, and the one that produces Table 2 and Figure 1. The supplementary material contains the Lean proof and every script, with a README that gives the command for each.

REFERENCES

Lukáš Adam, Rostislav Horčík, Tomáš Kasl, and Tomáš Kroupa. Double oracle algorithm for computing equilibria in continuous games. *Association for the Advancement of Artificial Intelligence (AAAI)*, 35(6):5070–5077, 2021.

Ilan Adler. The equivalence of linear programs and zero-sum games. *International Journal of Game Theory*, 42(1):165–177, 2013.

Anthropic. Claude Fable 5.1, September 2026. Large Language Model.

David Bartl. A very short algebraic proof of the Farkas Lemma. *Mathematical Methods of Operations Research*, 75(1):101–104, 2012. doi: 10.1007/s00186-011-0377-y.

Benjamin Brooks and Philip J. Reny. A canonical game—75 years in the making—showing the equivalence of matrix games and linear programming. *Economic Theory Bulletin*, 11(2):171–180, 2023.

Yang Cai, Ozan Candogan, Constantinos Daskalakis, and Christos Papadimitriou. Zero-sum polymatrix games: A generalization of minmax. *Mathematics of Operations Research*, 41(2):648–655, 2016.

Xi Chen, Xiaotie Deng, and Shang-Hua Teng. Settling the complexity of computing two-player Nash equilibria. *Journal of the ACM (JACM)*, 56(3):1–57, 2009.

George B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In *Activity Analysis of Production and Allocation*, pp. 339–347. Wiley, 1951.

Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a Nash equilibrium. *Communications of the ACM*, 52(2):89–97, 2009.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE-28*, pp. 625–635. Springer International Publishing, 2021.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction – CADE-25*, pp. 378–388. Springer International Publishing, 2015.

Le Cong Dinh, Yaodong Yang, Stephen McAleer, Zheng Tian, Nicolas Perez Nieves, Oliver Slumbers, David Henry Mguni, Haitham Bou Ammar, and Jun Wang. Online double oracle. *arXiv:2103.07780*, 2021.

Yoav Freund and Robert E. Schapire. Adaptive game playing using multiplicative weights. *Games and Economic Behavior (GEB)*, 29(1-2):79–103, 1999.

Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics: a foundation for computer science*. Addison-Wesley, 1994.

John D. Hunter. Matplotlib: a 2D graphics environment. *Computing in Science & Engineering*, 9(3): 90–95, 2007.

Victor Klee and George J. Minty. How good is the simplex algorithm? In *Inequalities III*, pp. 159–175. Academic Press, 1972.

Thomas Koshy. *Fibonacci and Lucas numbers with applications*. Wiley, 2001.

Tomáš Kroupa and Tomáš Votroubek. Multiple oracle algorithm to solve continuous games. In *International Conference on Decision and Game Theory for Security (GameSec)*, pp. 149–167, 2022.

Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérolat, David Silver, and Thore Graepel. A unified game-theoretic approach to multiagent reinforcement learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

Edward Lockhart, Marc Lanctot, Julien Pérolat, Jean-Baptiste Lespiau, Dustin Morrill, Finbarr Timbers, and Karl Tuyls. Computing approximate equilibria in sequential adversarial games by exploitability descent. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.

R. Duncan Luce and Howard Raiffa. *Games and Decisions*. John Wiley and Sons, New York, 1957.

Luke Marris, Paul Muller, Marc Lanctot, Karl Tuyls, and Thore Graepel. Multi-agent training beyond zero-sum with correlated equilibrium meta-solvers. In *International Conference on Machine Learning (ICML)*, 2021.

Stephen McAleer, John B. Lanier, Roy Fox, and Pierre Baldi. Pipeline PSRO: A scalable approach for finding approximate Nash equilibria in large games. *Conference on Neural Information Processing Systems (NeurIPS)*, 33:20238–20248, 2020.

Stephen McAleer, John B. Lanier, Kevin A. Wang, Pierre Baldi, and Roy Fox. XDO: A double oracle algorithm for extensive-form games. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

Stephen McAleer, Kevin Wang, John Lanier, Marc Lanctot, Pierre Baldi, Tuomas Sandholm, and Roy Fox. Anytime PSRO for two-player zero-sum games. *arXiv:2201.07700*, 2022.

Stephen Marcus McAleer, JB Lanier, Kevin A. Wang, Pierre Baldi, Tuomas Sandholm, and Roy Fox. Toward optimal policy population growth in two-player zero-sum games. In *International Conference on Learning Representations (ICLR)*, 2024.

H. Brendan McMahan, Geoffrey J. Gordon, and Avrim Blum. Planning in the presence of cost functions controlled by an adversary. In *International Conference on Machine Learning (ICML)*, pp. 536–543, 2003.

Paul Muller, Shayegan Omidshafiei, Mark Rowland, Karl Tuyls, Julien Pérolat, Siqi Liu, Daniel Hennes, Luke Marris, Marc Lanctot, Edward Hughes, Zhe Wang, Guy Lever, Nicolas Heess, Thore Graepel, and Rémi Munos. A generalized training approach for multiagent learning. In *International Conference on Learning Representations (ICLR)*, 2020.

Roger B. Myerson. *Game theory: analysis of conflict*. Harvard University Press, 1991.

John F. Nash. Non-cooperative games. *Annals of Mathematics*, 54(2):286–295, 1951.

Eugene Nudelman, Jennifer Wortman, Yoav Shoham, and Kevin Leyton-Brown. Run the GAMUT: A comprehensive approach to evaluating game-theoretic algorithms. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2004.

Martin J. Osborne. *An introduction to game theory*. Oxford University Press, 2004.

Finbarr Timbers, Nolan Bard, Edward Lockhart, Marc Lanctot, Martin Schmid, Neil Burch, Julian Schrittwieser, Thomas Hubert, and Michael Bowling. Approximate exploitability: learning a best response. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2022.

Steven Vajda. *Fibonacci and Lucas numbers, and the golden section: theory and applications*. Ellis Horwood, 1989.

Guido Van Rossum and Fred Drake, Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019. doi: 10.1038/s41586-019-1724-z.

John von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100:295–320, 1928.

John von Neumann and Oskar Morgenstern. *Theory of games and economic behavior*. Princeton University Press, 1947.

Michael Walton and Viliam Lisý. Multi-agent reinforcement learning in OpenSpiel: a reproduction report. *arXiv:2103.00187*, 2021.

Michael P. Wellman. Methods for empirical game-theoretic analysis. In *Association for the Advancement of Artificial Intelligence (AAAI)*, volume 2, pp. 1552–1556, 2006.

Michael P. Wellman. Putting the agent in agent-based modeling. *Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 30(6):1175–1189, 2016.

Michael P. Wellman, Karl Tuyls, and Amy Greenwald. Empirical game theoretic analysis: A survey. *Journal of Artificial Intelligence Research (JAIR)*, 82:1017–1076, 2025.

Brian Hu Zhang and Tuomas Sandholm. Exponential lower bounds on the double oracle algorithm in zero-sum games. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 3032–3039, 2024. ISBN 978-1-956792-04-1.

A OMITTED PROOFS

A.1 PRELIMINARIES

Proof of Lemma 2. Let $X = \sum_{i \in R} x_i$ and $Y = \sum_{j \in C} y_j$. Then $vY = \sum_{j \in C} y_j u_j = \sum_{i \in R} x_i w_i = \lambda X$, using the equality on the support of y. So against x/X every column of C pays at least v/X, and against y/Y every row of R pays $\lambda/Y = v/X$, which is the equilibrium condition, with value v/X. □

Proof of Lemma 3. Let v and v' be the values. Then $v' = \min_{j \in C} u'_j \le \sum_{j \in C} y_j u'_j = \sum_{i \in R} x'_i w_i \le \max_{i \in R} w_i = v$, and symmetrically $v \le v'$. So $v = v'$, and $\sum_j y_j (u'_j - v) = 0$ with nonnegative terms gives $u'_j = v$ whenever $y_j > 0$. The row statement is symmetric. □

Proof of Lemma 4. Since $g(n+1) = g(n-1) - g(n)$, if $g(n-1)$ and $g(n)$ have opposite signs, then $g(n+1)$ has the sign of $g(n-1)$, and $|g(n+1)| = |g(n-1)| + |g(n)|$. By hypothesis, this starts at $n = n_0$. So from $n_0 - 1$ on the signs alternate, and from n_0 on the magnitudes are nondecreasing. With the hypothesis at n_0, this gives $|g(n-1)| \le 2|g(n)|$ for every $n \ge n_0$. Thus, by (9), the column of index $n \ge n_0$ pays a negative amount for even $n - n_0$ and a positive amount for odd $n - n_0$, of magnitude $\frac{|g(n)|}{MF(n)F(n+1)}$. From n to $n+2$, the numerator at most quadruples, since $|g(n+2)| = 2|g(n)| + |g(n-1)| \le 4|g(n)|$, while the denominator more than quadruples, since $F(n+2) \ge 2F(n)$ and $F(n+3) > 2F(n+1)$. So the magnitude strictly decreases every two steps, and the column of index n_0 pays less than every column of higher index. □

A.2 THE STAIRCASE PHASE

Proof of Lemma 5. Here $R = C = [k]$. That is, the subgame is the $k \times k$ staircase block.

Equilibrium. Let $x_i = i + 1$ and $y_j = k - j$ for $i, j \in [k]$. By (8), every column $j \in [k]$ pays $x_j - x_{j-1} = 1$ and every row $i \in [k]$ pays $y_i - y_{i+1} = 1$. Because payoffs telescope, an arithmetic progression with common difference 1 makes every opposing strategy tight. By Lemma 2, normalizing x and y gives an equilibrium.

Every equilibrium. Let (x', y') be any equilibrium. Its value is $v' > 0$, by Lemmas 2 and 3. Since y is positive on C, Lemma 3 makes every column of C tight for x': $x'_j - x'_{j-1} = v'$ for all $j \in [k]$, so $x' = v'x$.

Rows. A staircase column $j \in [k]$ is supported on rows $j-1$ and j, so every row $i \ge k$ pays $0 < v'$ against y', and $R_{\text{BR}} \subseteq R$. Thus no row is added.

Columns. Against x, every column of C pays $1 > 0$. If $k < M$, column k pays $x_k - x_{k-1} = -k$: it exploits the large weight on row $k-1$. Staircase columns $j > k$ pay $x_j - x_{j-1} = 0$, and corridor columns pay at least $-\frac{x_0 + x_1}{3M} \ge -\frac{1}{M} > -k$. If $k = M$, every staircase column is in C. Since $x_0 = 1$ and $x_1 = 2$, we have $g(3) = 2F(3) - F(4) = 1$ and $g(4) = F(5) - 2F(4) = -1$, so $g(3) > 0$ and $g(3) \le -2g(4)$. Lemma 4 with $n_0 = 4$ then shows that column M, of index 4, pays a negative amount and less than every later corridor column. Hence, $C_{\text{BR}} = \{k\}$, and the only successor is $([k], [k+1])$.

Value, margin, and weight. Both x and y have mass $\sum_{i \in [k]} (i+1) = \frac{k(k+1)}{2}$, so by Lemma 2 the value is $v = \frac{2}{k(k+1)}$ and column 0 has weight kv. Normalizing the payoffs above, column k pays $-kv$ if $k < M$, and column M, of index 4, pays $\frac{g(4)v}{MF(4)F(5)} = -\frac{v}{15M} = -\frac{2}{15M^2(M+1)}$ if $k = M$. □

Proof of Lemma 6. Here $R = [k]$ and $C = [k+1]$. The column player now has column k, whose only entry in the rows of R is the -1 in row $k-1$.

Equilibrium. Let $x_i = k - i$ for $i \in [k]$ and $y_j = j$ for $j \in [k+1]$. By (8), column 0 pays $x_0 = k > -1$ and columns $1 \le j \le k$ pay $x_j - x_{j-1} = -1$. Every row $i \in [k]$ pays $y_i - y_{i+1} = -1$, as column $i+1 \le k$ lies in C. By Lemma 2, normalizing x and y gives an equilibrium, in which column 0 is slack.

Every equilibrium. Let (x', y') be any equilibrium. Its value is $v' < 0$, by Lemmas 2 and 3. As $y_j > 0$ for $1 \le j \le k$, Lemma 3 gives $x'_j - x'_{j-1} = v'$ there, so $x' = |v'|x$. As $x_i > 0$ for $i \in [k]$, it gives $y'_i - y'_{i+1} = v'$ there. Column 0 is slack against x, so Lemma 3, with the two equilibria exchanged, gives $y'_0 = 0$. Hence, $y' = |v'|y$ and $y'_k = k|v'| > 0$.

Rows. Against y', every row of R pays $v' < 0$. Row k, whose only nonzero entry in the columns of C is $A_{kk} = 1$, pays $y'_k > 0$. Rows $i > k$ have no nonzero entry there and pay 0. So $R_{\text{BR}} = \{k\}$.

Columns. Against x, the columns of C pay at least -1. Staircase columns $j > k$ pay 0, and corridor columns pay at least $-\frac{x_0+x_1}{3M} = -\frac{2k-1}{3M} > -1$, because $2k - 1 < 3M$. Thus $C_{\text{BR}} \subseteq C$. No column is added, and the only successor is $([k+1], [k+1]) = S_{k+1}$.

Value and margin. Both x and y have mass $\frac{k(k+1)}{2}$, so the value is $v = -\frac{2}{k(k+1)}$, and row k pays $k|v| = \frac{2}{k+1}$ while every row of $[k]$ pays v. □

A.3 The corridor phase

Proof of Lemma 7. Here $R = [M]$, so no row can be added and only the best-response column matters. The only corridor column is column M, of index 4, with entries $s(4) = \frac{1}{3M}$ and $s(5) = -\frac{1}{5M}$.

Equilibrium. At the later corridor states, two tight corridor columns fix the ratio $x_0 : x_1$ (Lemma 8). Here, column M pairs with staircase column 1 instead, so

$$H = \begin{bmatrix} -1 & s(4) \\ 1 & s(5) \end{bmatrix}, \qquad \det H = -s(5) - s(4) = -\frac{2}{15M}, \qquad H^{-1} = \frac{1}{2}\begin{bmatrix} 3 & 5 \\ 15M & 15M \end{bmatrix}. \quad (14)$$

We solve at the value 2, which clears denominators. Let $(x_0, x_1)^{\mathrm{T}} = (H^{-1})^{\mathrm{T}}(2, 2)^{\mathrm{T}}$, which makes columns 1 and M pay 2. Let $x_i = x_1 + 2(i-1)$, an arithmetic progression as in the staircase. Then $x_i = 15M + 3 + 2i$, and the band pays $x_j - x_{j-1} = 2$. Column 0 pays $x_0 = 15M + 3 > 2$. Let y vanish except on the band and the two tight columns. On the band, $y_j = 2(M - j)$ for $2 \le j \le M - 1$, which puts $-y_2 = -2(M-2)$ on row 1, absent when $M = 2$. On the two tight columns, $(y_1, y_M)^{\mathrm{T}} = H^{-1}(2, 2M-2)^{\mathrm{T}}$, that is, $y_1 = 5M - 2$ and $y_M = 15M^2$. Rows 0 and 1 then pay 2 by construction, and rows $i \ge 2$ pay 2 as well ($y_{M-1} = 2$ for $i = M-1$, $y_i - y_{i+1} = 2$ otherwise). By Lemma 2, normalizing x and y gives an equilibrium.

Every equilibrium. Let (x', y') be any equilibrium, of value v'. Since y is positive on columns $1, \dots, M$, Lemma 3 gives $x'_j - x'_{j-1} = v'$ for $1 \le j \le M-1$ and $\frac{x'_0}{3M} - \frac{x'_1}{5M} = v'$. Hence, $x'_i = x'_0 + iv'$ and $x'_0 = \frac{15M+3}{2}v'$, so $x' = \frac{v'}{2}\,x$.

Columns. Against x, every column of C pays at least $2 > 0$. Also, $g(4) = 5x_0 - 3x_1 = 30M$ is positive, and $-2g(5) = 2(8x_0 - 5x_1) = 2(45M - 1) \ge g(4)$. Lemma 4 with $n_0 = 5$ then shows that column $M+1$, of index 5, pays a negative amount and less than every later corridor column. So $C_{\text{BR}} = \{M+1\}$, and the only successor is C_{M+2}.

Value, margin, and weight. Both x and y have mass $2M(8M+1)$, so the value is $v = \frac{2}{2M(8M+1)} = \frac{1}{M(8M+1)}$, column 1 has weight $\frac{5M-2}{2M(8M+1)}$, and column $M+1$ pays $\frac{g(5)}{MF(5)F(6)} \cdot \frac{1}{2M(8M+1)} = -\frac{45M-1}{80M^2(8M+1)}$. □

Lemma 9 (Cassini's identity)**.** For every $n \ge 1$,

$$F(n-1)F(n+1) - F(n)^2 = (-1)^n. \quad (15)$$

Proof. Induction on n. For $n = 1$, the left side is $0 - 1 = -1$. If the identity holds for n, then

$$F(n)F(n+2) - F(n+1)^2 = F(n)^2 - F(n+1)(F(n+1) - F(n)) \tag{16}$$
$$= -\big(F(n-1)F(n+1) - F(n)^2\big) = (-1)^{n+1}. \qquad \square$$

Proof of Lemma 8. Here $R = [M]$ and $C = [k]$, so only the best-response column matters. The corridor columns of C have indices $4, \dots, K$, where $K = k - M + 3 \geq 5$. The frontier has indices $K - 1$ and K, and column k, the one to be added, has index $K + 1$.

The frontier system. Rows 0 and 1 meet the frontier columns in the symmetric matrix

$$H = \begin{bmatrix} s(K-1) & s(K) \\ s(K) & s(K+1) \end{bmatrix}. \tag{17}$$

As in Lemma 7, both halves of the certificate solve a system with it, here at the value 1. The row weights make both frontier columns pay 1. The frontier weights contribute 1 to row 0 and $M-1$ to row 1, so that rows 0 and 1 pay what the band pays. By Lemma 9, $\det H = \frac{F(K)^2 - F(K-1)F(K+1)}{M^2F(K-1)F(K)^2F(K+1)} = \frac{(-1)^{K+1}}{M^2F(K-1)F(K)^2F(K+1)}$, so

$$H^{-1} = MF(K) \begin{bmatrix} F(K-1)F(K) & F(K-1)F(K+1) \\ F(K-1)F(K+1) & F(K)F(K+1) \end{bmatrix}. \tag{18}$$

Equilibrium. Let $(x_0, x_1)^{\mathrm{T}} = H^{-1}(1,1)^{\mathrm{T}}$, which makes both frontier columns pay 1, and let $x_i = x_1 + i - 1$ for $1 \leq i \leq M - 1$. By (18) and the identities $F(K) + F(K+1) = F(K+2)$ and $F(K-1) + F(K) = F(K+1)$, these are

$$x_0 = MF(K-1)F(K)F(K+2), \qquad x_1 = MF(K)F(K+1)^2. \tag{19}$$

The band pays $x_j - x_{j-1} = 1$. Column 0 pays $x_0 > 1$, and column 1 pays $x_1 - x_0 = MF(K)^3 > 1$, as $F(K+1)^2 - F(K-1)F(K+2) = F(K)^2$. Since the frontier pays 1, (9) gives $g(K-1) = MF(K-1)F(K)$ and $g(K) = MF(K)F(K+1)$. The recurrence in the form $g(n) = g(n+1) + g(n+2)$ makes every $g(n)$ with $n \leq K$ positive, and strictly decreasing in n for $n \leq K - 1$. Hence, the corridor column of index $n \leq K - 2$ pays

$$\frac{g(n)}{MF(n)F(n+1)} > \frac{g(K-1)}{MF(K-1)F(K)} = 1, \tag{20}$$

as $g(n) > g(K-1)$ and $F(n)F(n+1) < F(K-1)F(K)$. So every column of C pays at least 1, and the tight ones are exactly the band and the frontier.

Let y vanish except on the band, where $y_j = M - j$, and the frontier, where $(y_{k-2}, y_{k-1})^{\mathrm{T}} = H^{-1}(1, M-1)^{\mathrm{T}}$, that is

$$y_{k-2} = MF(K-1)F(K)(F(K) + (M-1)F(K+1)), \tag{21}$$
$$y_{k-1} = MF(K)F(K+1)(F(K-1) + (M-1)F(K)). \tag{22}$$

The frontier thus contributes 1 to row 0 and $M - 1$ to row 1, so every row pays 1: row 0 pays 1, row 1 pays $-(M-2) + (M-1) = 1$, where $M - 2$ is the weight of staircase column 2, absent when $M = 2$, and rows $i \geq 2$ pay 1 ($y_{M-1} = 1$ for $i = M - 1$, $y_i - y_{i+1} = 1$ otherwise). By Lemma 2, normalizing x and y gives an equilibrium.

Every equilibrium. Let (x', y') be any equilibrium. Its value is $v' > 0$, by Lemmas 2 and 3. Since y is positive on the frontier, Lemma 3 makes both frontier columns pay v' against x'. By (9), the g computed from x'_0 and x'_1 is therefore v' times the one above at $K - 1$ and at K. So $g(K+1) = g(K-1) - g(K) = v'MF(K)(F(K-1) - F(K+1)) = -v'MF(K)^2$.

Columns. Against x', every column of C pays at least $v' > 0$. Here $g(K) = v'MF(K)F(K+1)$ is positive and at most $-2g(K+1) = 2v'MF(K)^2$, as $F(K+1) \leq 2F(K)$. Lemma 4 with $n_0 = K + 1$ then shows that column k, of index $K + 1$, pays a negative amount and less than every later corridor column. So $C_{\mathrm{BR}} = \{k\}$, and the only successor is C_{k+1}.

Value, margin, and weight. The mass of x is $x_0 + \sum_{i=1}^{M-1} x_i = MF(K)\big(MF(K+1)^2 - F(K)^2\big) + \frac{(M-1)(M-2)}{2}$, using $F(K+1)^2 - F(K)^2 = F(K-1)F(K+2)$. So the value is V_k of (11), and column k, of index $K+1$, pays $\frac{g(K+1)V_k}{MF(K+1)F(K+2)} = -\frac{V_k F(K)^2}{F(K+1)F(K+2)}$. For the weight of column $k-2$, the band carries $\sum_{j=2}^{M-1}(M-j) = \frac{(M-1)(M-2)}{2}$, which is at most $\frac{y_{k-2}}{2}$. Also $y_{k-1} \le 3y_{k-2}$, since $F(K-1)+(M-1)F(K) \le F(K)+(M-1)F(K+1)$ and $F(K+1) \le 3F(K-1)$. The mass of y is therefore at most $6y_{k-2}$, so column $k-2$ has weight at least $\frac{1}{6}$. □

Interpretation of the corridor. By (9), the corridor column of index n pays $(-1)^n(x_0F(n+1) - x_1F(n))$ up to a positive factor. So it pays a negative amount exactly when the row player's ratio x_1/x_0 lies on one side of $\frac{F(n+1)}{F(n)}$, a convergent of the golden ratio φ: above it for even n, below it for odd n. The corridor thus offers the column player the successive rational approximations of φ. The equilibrium ratio at C_k is $\frac{F(K+1)^2}{F(K-1)F(K+2)}$, which tends to φ, alternating around it. Every partial quotient of φ is 1, so the denominators $F(n)$ of its convergents grow as slowly as a continued fraction allows.

Beyond its recurrence, the Fibonacci sequence enters only through Lemma 9. Lemma 9 is the case $b = a+1$ of d'Ocagne's identity (Koshy, 2001; Vajda, 1989), $F(a+1)F(b) - F(a)F(b+1) = (-1)^a F(b-a)$. In rows 0 and 1, the corridor columns of any two indices $a < b$ therefore meet in a matrix of determinant $\frac{(-1)^{b+1}F(b-a)}{M^2F(a)F(a+1)F(b)F(b+1)} \neq 0$. This covers Lemma 7 too, where staircase column 1 is M times the corridor column of index 1. No two corridor columns are parallel, which is what lets each of them become the BR in turn. Reciprocals of a geometric sequence would instead make them all proportional, and the first two would suffice for the column player.

A.4 Intermediate game sizes

Let (R_t, C_t) be the state after t iterations, for $0 \le t \le M+N-2$. The state is $S_{t/2+1}$ for even $t \le 2M-2$, $T_{(t+1)/2}$ for odd $t \le 2M-3$, and C_{t-M+2} for $t \ge 2M-2$, so

$$|R_t| = \min\{\lfloor t/2 \rfloor + 1,\, M\}, \qquad |C_t| = t + 2 - |R_t|. \tag{23}$$

Any run of $M+N-2$ iterations adds exactly one strategy per iteration, so after t iterations it has $|R|+|C| = t+2$ with $|R| \le M$ and $|C| \le N$. Under these constraints, $|R| \cdot |C|$ is maximized by the split above, which keeps the subgame as square as possible.

A.5 Approximate oracles

Lemma 10 (Perturbed equilibria)**.** Let (x, y) be an equilibrium of $A|_{R\times C}$ of value v, and let (x', y') be a δ-equilibrium of it, with payoffs u'_j and w'_i. Then $\max_{i\in R} w'_i \ge v \ge \min_{j\in C} u'_j$, and

$$u'_j \ge v - \delta \quad (j \in C), \qquad w'_i \le v + \delta \quad (i \in R), \qquad u'_j \le v + \delta/y_j \quad (y_j > 0). \tag{24}$$

Proof. Every column of C pays at least v against x, so $v \le \sum_{j\in C} y'_j u_j = \sum_{i\in R} x_i w'_i \le \max_{i\in R} w'_i$, which is at most $u'_j + \delta$ for every $j \in C$. Symmetrically, every row of R pays at most v against y, so $\sum_{j\in C} y_j u'_j = \sum_{i\in R} x'_i w_i \le v$, whence $\min_{j\in C} u'_j \le v$, and $w'_i \le \sum_{j\in C} y_j u'_j + \delta \le v + \delta$ for every $i \in R$. For the last bound, $y_j(u'_j - v) \le -\sum_{l\neq j} y_l(u'_l - v) \le \delta \sum_{l\neq j} y_l \le \delta$, using $u'_l \ge v - \delta$. □

Lemma 11 (Staircase steps)**.** Let $2 \le M \le N$ and $1 \le k < M$, let $v = \frac{2}{k(k+1)}$, the magnitude of the value at S_k and at T_k, and let $b \ge 0$ bound the columns after k from below: every column j with $k < j < N$ pays at least $-b$ against the row strategy of any δ-equilibrium of the subgame. If $\varepsilon_R < v$ and $\varepsilon_C + b < k(v - \delta)$, then the only successor of S_k is T_k. If $\varepsilon_C + b < v$ and $\varepsilon_R < k(v - \delta)$, then the only successor of T_k is S_{k+1}.

Proof. Let (x', y') be a δ-equilibrium of the subgame, and apply Lemma 10 to the equilibrium of Lemma 5 or 6. Either hypothesis gives $\delta < v$.

At S_k, of value v, the rows outside $[k]$ pay 0 while some row of $[k]$ pays at least $v > \varepsilon_R$, so the row player adds nothing. Every column $j \in [k]$ pays $x'_j - x'_{j-1} \geq v - \delta$, so $x'_{k-1} = \sum_{j\in[k]} u'_j \geq k(v-\delta) > \varepsilon_C + b$. Column k pays $-x'_{k-1} < -b - \varepsilon_C$, while the columns of $[k]$ pay at least $v - \delta > 0$ and those after k pay at least $-b$, so it is the column player's only answer.

At T_k, of value $-v$, some column of $[k+1]$ pays at most $-v$ while every column outside pays at least $-b > -v + \varepsilon_C$, so the column player adds nothing. Every row $i \in [k]$ pays $y'_i - y'_{i+1} \leq -v + \delta$, so $y'_k \geq y'_0 + k(v-\delta) > \varepsilon_R$, while the rows of $[k]$ pay less than 0 and those after k pay 0. Thus row k is the row player's only answer. □

Proof of Theorem 2. Fix a state of the run before the last, of value v and margin m, so δ and ε_C are at most $\min(|v|, m)/64$, and let (x', y') be a δ-equilibrium of its subgame. At a staircase state, $\mu_S \leq |v|$, so ε_R is at most $\min(|v|, m)/64$ there as well.

Staircase states. At S_k, where $m = kv = \frac{2}{k+1} \geq \frac{2}{M}$, the staircase columns after k pay 0, and the corridor columns pay at least $-\frac{x'_0+x'_1}{3M} \geq -\frac{1}{3M} \geq -\frac{m}{6}$. Hence, Lemma 11 applies with $b = \frac{m}{6}$, as $\varepsilon_R < v$ and $k(v-\delta) \geq \frac{63}{64}m > \frac{m}{6} + \frac{m}{64} \geq \frac{m}{6} + \varepsilon_C$. At T_k, column k pays $-x'_{k-1} \geq -v-\delta$, and every earlier column gives $x'_i \leq x'_{i+1} + v + \delta$. Hence, $x'_0 + x'_1 \leq (2k-1)(v+\delta)$. As $\frac{2k-1}{3M} \leq \frac{2}{3}$, the corridor columns pay at least $-\frac{2}{3}(v+\delta) \geq -\frac{3}{4}v$. Lemma 11 applies with $b = \frac{3}{4}v$, as $\varepsilon_C + b \leq \frac{v}{64} + \frac{3}{4}v < v$ and $k(v-\delta) \geq \frac{63}{64}m > \varepsilon_R$.

Corridor states. At $S_M = C_M$, at C_{M+1}, and at C_k with $k \geq M+2$, the row player holds every row, so only the column player moves, and by (9) the corridor payoffs are fixed by x'_0 and x'_1. In each case, the new column pays u'_{new}, the same combination of the payoffs of two tight columns of the subgame:

$$u'_M = \frac{2u'_0 - 3u'_1}{15M}, \qquad u'_{M+1} = -\frac{9}{16}u'_M + \frac{u'_1}{80M}, \tag{25}$$

$$u'_k = \frac{F(K)\big(F(K-1)u'_{k-2} - F(K+1)u'_{k-1}\big)}{F(K+1)F(K+2)}, \tag{26}$$

by (8) and (9). Substituting v throughout the right sides gives $-m$, by Lemmas 5, 7, and 8, using $F(K+1) - F(K-1) = F(K)$ for the last one. Every column of the subgame pays at least $v - \delta$. The one that enters with a positive coefficient is the column whose weight those lemmas bound below, by Mv, $\frac{5M-2}{2M(8M+1)}$, and $\frac{1}{6}$ in the three cases. Hence, Lemma 10 also puts its payoff at most δ/y_j above v. So

$$u'_{\text{new}} \leq -m + \tfrac{9}{2}\delta, \tag{27}$$

and the new column beats every column of the subgame by at least $v + m - \frac{11}{2}\delta > \varepsilon_C$. The hypotheses of Lemma 4 at the new column's index still hold. By the identities above, and since every tight column pays at least $v - \delta > 0$, they reduce to $3u'_0 \leq 4u'_1$ at C_M, to $u'_1 \leq 30Mu'_M$ at C_{M+1}, and to $2F(K-1)u'_{k-2} \leq F(K+1)u'_{k-1}$ at C_k. Each holds at $\delta = 0$, and the bounds on u' above keep it for $\delta \leq \mu/64$. As $\frac{F(n)}{F(n+1)} \leq \frac{5}{8}$ for $n \geq 4$, two corridor steps multiply a payoff by at most $\frac{4F(n)F(n+1)}{F(n+2)F(n+3)} \leq \frac{5}{8}$ in magnitude. Hence, the new column beats every later corridor column by at least $\frac{3}{8}\big(m - \frac{9}{2}\delta\big) > \varepsilon_C$.

Hence, every state of the run has the same unique successor as in Theorem 1, and the final state contains every strategy, so the algorithm stops there. □

We turn to (13). At S_k and T_k, the smaller of the two quantities is $|v_t| = \frac{2}{k(k+1)} \geq \frac{2}{M^2}$, at C_M it is $\frac{2}{15M^2(M+1)} \geq \frac{1}{15M^3}$, and at C_{M+1} it is $\frac{45M-1}{80M^2(8M+1)} \geq \frac{11}{180M^2}$. Each of these is at least $\frac{1}{15M^3}$, which is more than the right side of (13), as $F(N-M+3) \geq F(3) = 2$. At C_k with $k \geq M+2$, the smaller one is m_t, and $\frac{1}{V_k} \leq M^2F(K)F(K+1)^2 + \frac{M^2}{2} \leq \frac{3}{2}M^2F(K)F(K+1)^2$, so

$$m_t = \frac{V_kF(K)^2}{F(K+1)F(K+2)} \geq \frac{2F(K)}{3M^2F(K+1)^3F(K+2)} \geq \frac{2}{9M^2F(K+1)^3}, \tag{28}$$

using $F(K+2) \leq 3F(K)$. Every state before the last has $K+1 \leq N-M+3$, and $\frac{2}{9M^2} \geq \frac{1}{8M^3}$.

Proof of Corollary 1. As in Theorem 1, assume $N \geq M$. Take δ_0 and ε_0 to be the smallest of the three thresholds of Theorem 2, which is positive by (13). □

Proof of Corollary 2. Assume $N \geq M$ as before, and let ρ be half the smallest of the three thresholds of Theorem 2. Fix a state (R, C) and an equilibrium (x, y) of $A'|_{R\times C}$. Since x and y are probability distributions, every row and column payoff under A differs from the one under A' by less than ρ. Hence, $\min_{j\in C} u_j \geq \max_{i\in R} w_i - 2\rho$ under A, so (x, y) is a 2ρ-equilibrium of $A|_{R\times C}$. Likewise, a best response under A' is a 2ρ-best response under A. Every run on A' is therefore an approximate run on A with all three parameters at most 2ρ, and Theorem 2 applies. □

Proof of Theorem 3. At each state before the last, all best responses of one player lie in the subgame, while the other player's best response improves on its tight strategies, which pay the value, by exactly $d_t = |v_t| + m_t$. Thus if that player's tolerance is at least d_t, it may pick a tight strategy instead, and the algorithm stops there.

At S_{M-1}, every row of the subgame pays the value $\frac{2}{M(M-1)}$ against the equilibrium, and row $M-1$ pays 0. So if $\varepsilon_R \geq \mu_S$, the row player may add row $M-1$ alongside column $M-1$, and the run reaches S_M an iteration early.

At T_{M-1}, let $c = \frac{2(1-M\varepsilon_R)}{M(M+1)}$ and $y'_j = c(M-j) + \varepsilon_R$ for $j \in [M]$, a mixed strategy of the column player, as $c\frac{M(M+1)}{2} + M\varepsilon_R = 1$. Every row of the subgame pays c against y', and row $M-1$ pays $c + \varepsilon_R$, so row 0 is within ε_R of best. The column player faces the equilibrium x, so by Lemma 6 it may still add nothing. As the value at T_{M-1} is $-\frac{2}{M(M-1)}$, the pair (x, y') is a δ-equilibrium whenever $\delta \geq \frac{4}{M^2-1} - \frac{2\varepsilon_R}{M+1}$, and then the algorithm stops at T_{M-1}. □

A.6 Other initial states

Theorem 4 (Any start, negative result)**.** Let $M, N \geq 2$ with $(M, N) \neq (2, 2)$, and let A be an $M \times N$ matrix. Then some run of the double oracle algorithm on A, starting from some $R = \{i\}, C = \{j\}$, takes fewer than $M + N - 2$ iterations.

Proof. At the initial state $(\{i\}, \{j\})$, the subgame is the single entry A_{ij}, whose only equilibrium is $x_i = y_j = 1$, so $R_{\text{BR}} = \operatorname{argmax}_{k\in[M]} A_{kj}$ and $C_{\text{BR}} = \operatorname{argmin}_{l\in[N]} A_{il}$. At most one initial state per column j has $R_{\text{BR}} = \{i\}$, and at most one per row i has $C_{\text{BR}} = \{j\}$. As $(M-1)(N-1) > 1$, that is, $MN > M + N$, some initial state has neither. A run from it that chooses $i_{\text{BR}} \neq i$ and $j_{\text{BR}} \neq j$ adds two strategies in its first iteration and at least one in each later iteration, so it takes at most $M + N - 3$ iterations. □

The hypothesis $(M, N) \neq (2, 2)$ cannot be dropped. For $M = N = 2$, every run on the matrix $\left[\begin{smallmatrix} 1 & -1 \\ 0 & 1 \end{smallmatrix}\right]$ of Definition 8 takes 2 iterations, whatever its initial state $(\{i\}, \{j\})$. The reason is that the unique best responses to a pure profile (k, l) are row l and column $1 - k$.

Table 1: Value, duality gap, and margin of the subgame along the run. At the final state, C_N, the gap is 0.

State	Value v_t	Duality gap d_t	Margin $m_t = d_t - \lvert v_t \rvert$
S_k, $k < M$	$\dfrac{2}{k(k+1)}$	$\dfrac{2}{k}$	$\dfrac{2}{k+1}$
T_k	$-\dfrac{2}{k(k+1)}$	$\dfrac{2}{k}$	$\dfrac{2}{k+1}$
C_M	$\dfrac{2}{M(M+1)}$	$\dfrac{2(15M+1)}{15M^2(M+1)}$	$\dfrac{2}{15M^2(M+1)}$
C_{M+1}	$\dfrac{1}{M(8M+1)}$	$\dfrac{125M-1}{80M^2(8M+1)}$	$\dfrac{45M-1}{80M^2(8M+1)}$
C_k, $k \geq M+2$	V_k	$V_k\left(1 + \dfrac{F(K)^2}{F(K+1)F(K+2)}\right)$	$V_k \dfrac{F(K)^2}{F(K+1)F(K+2)}$

B THE DUALITY GAP ALONG A RUN

Table 1 gives the value, the duality gap, and the margin at each state of the run, each row two independent numbers and their sum. The gap decays along the run. How much of that decay is forced is the question of this appendix. The first result below bounds the gap along any full-length run by the shape of the matrix. The second shows that at $M = N$, where the first leaves no constraint, the gap can come arbitrarily close to the largest value the entries allow.

Proposition 1 (Large gap, negative result)**.** Let A be an $M \times N$ matrix with entries in $[-1, 1]$. Suppose that a run of DO takes $M + N - 2$ iterations, and that the duality gap is at least $\gamma > 0$ at every state before the last. Then any two columns of A differ by at least γ in some row, and any two rows differ by at least γ in some column. Hence, $N \leq (1 + 2/\gamma)^M$ and $M \leq (1 + 2/\gamma)^N$.

Equivalently, some state of any full-length run has gap at most $2/(N^{1/M} - 1)$. So no matrix holds the gap away from zero at every shape, and with $M = 2$ it must fall to at most $2/(\sqrt{N} - 1)$. The gap is at most 2 for entries in $[-1, 1]$, so the bound is vacuous once $M \geq \log_2 N$, and in particular when $M = N$. There, it is not the shape but the entries that bind, and Proposition 2 comes arbitrarily close to their bound.

Proof of Proposition 1. An initial state with more than one strategy for some player leaves fewer than $M + N - 2$ strategies to add, so the run starts from a singleton for each player. It then adds $M + N - 2$ strategies in $M + N - 2$ iterations, hence exactly one on each. Consider an iteration that adds the column j', at a state (R, C) with equilibrium (x, y) and value v. No row is added there, so the row best response lies in R, and $\max_{i \in [M]} w_i = v$. The gap is therefore $v - u_{j'} \geq \gamma$, while every column $j \in C$ has $u_j \geq v$. Subtracting, $\sum_{i \in R} x_i (A_{ij} - A_{ij'}) \geq \gamma$, and as x is a distribution, $A_{ij} - A_{ij'} \geq \gamma$ for some row i. Any two columns are of this form, since one of them is added at an iteration at which the other is already present. The rows are symmetric. The N columns are thus points of $[-1, 1]^M$ at sup-norm distance at least γ from one another. Split $[-1, 1]$ into $\lfloor 2/\gamma \rfloor + 1$ half-open intervals of length γ, which cover it as $(\lfloor 2/\gamma \rfloor + 1)\gamma > 2$. Two columns lying in the same interval in every row would differ by less than γ in every row, which the separation forbids. Distinct columns thus occupy distinct cells, and $N \leq (\lfloor 2/\gamma \rfloor + 1)^M \leq (1 + 2/\gamma)^M$. □

Proposition 2 (Large gap, square case)**.** Let $M \geq 2$ and $0 < \eta < \frac{1}{M-1}$, let $D_i = 1 - i\eta$, and let B be the $M \times M$ matrix

$$B_{ij} = \begin{cases} D_i & j \leq i \\ -D_j & j > i \end{cases} . \tag{29}$$

Its entries lie in $[-1, 1]$. Every run of the double oracle algorithm on B, starting from $R = \{0\}, C = \{0\}$, takes exactly $2M - 2$ iterations, and the duality gap at every state before the last is at least $2D_{M-1}$.

In words, the larger index wins, ties go to the row player, and the prize D shrinks with the winner's index. This is the bigger-number game of Section 3 with a tilt. Without the tilt, both players move on

the same iteration, and the run is half as long. With it, only one player can improve at each state, and its only best response is the next index. Since η is free, $2D_{M-1} = 2 - 2(M-1)\eta$ comes as close to 2 as one likes, and 2 is the most that entries in $[-1, 1]$ allow.

Proof of Proposition 2. The sequence D is strictly decreasing with $D_i \in (0, 1]$, so every entry lies in $[-1, 1]$. Let $S_k = ([k], [k])$ and $T_k = ([k], [k+1])$, so that the initial state is S_1.

At S_k, row $k-1$ pays D_{k-1} against every column of C, while column $k-1$ pays $-D_{k-1}$ against every other row of R, so the value is D_{k-1}. Any x with $x_{k-1} < 1$ pays $(2x_{k-1} - 1)D_{k-1} < D_{k-1}$ against column $k-1$, so the row player has the single equilibrium strategy $x = e_{k-1}$. Against it, column j pays D_{k-1} for $j < k$ and $-D_j$ for $j \geq k$, least at $j = k$, so column k is the only best response, and it beats the value by $D_{k-1} + D_k$. Every row $i \geq k$ pays $D_i < D_{k-1}$ against every y on C, so the row best response lies in R, whichever equilibrium is selected. Hence, $S_k \xrightarrow{!} T_k$.

At T_k, column k pays $-D_k$ against every row of R, while row $k-1$ pays D_{k-1} against every other column of C, so the value is $-D_k$. Any y with $y_k < 1$ lets row $k-1$ pay more than that, so the column player has the single equilibrium strategy $y = e_k$. Against it, row i pays $-D_k$ for $i < k$ and D_i for $i \geq k$, greatest at $i = k$, so row k is the only best response, and it beats the value by $2D_k$. Every column $j > k$ pays $-D_j > -D_k$ against every x on R, while some column of C pays $-D_k$ against an equilibrium x, so the column best response lies in C. Hence, $T_k \xrightarrow{!} S_{k+1}$.

The run is therefore $S_1, T_1, \ldots, S_{M-1}, T_{M-1}, S_M$, which is $2M - 2$ iterations, and it stops at S_M, where every row and column is present. At S_k, the row best response lies in the subgame, so the gap is $D_{k-1} + D_k$, and at T_k, the column best response does, so the gap is $2D_k$. Both are at least $2D_{M-1}$. □

C Additional background

C.1 Game theory

Game theory is the mathematical study of strategic interactions between agents (Myerson, 1991). In the context of game theory, a *game* is a formal model of a setting or situation in which such agents interact. In these models, agents may have independent interests, creating incentives for pure cooperation, pure competition, or a complicated mixture of the two.

Empirical game-theoretic analysis (EGTA) (Wellman, 2006; 2016; Wellman et al., 2025) is a computational framework for multiagent games that are too large or complex to specify and solve analytically. It analyzes them with simulation data and statistical modeling. It allows researchers to identify approximate equilibria and evaluate strategic behavior in highly complex environments.

The simplest kind of game is a *normal-form game (NFG)*, in which the players choose their strategies simultaneously and are paid immediately.

Definition 11 (Normal-form game)**.** A normal-form game is a tuple (I, S, u) whose elements are as follows.

- I is a set of *players*.
- S_i is a set of *strategies* for each player $i \in I$. A *strategy profile* is a map from each player to an element of its strategy set. That is, it is an element of $\prod_{i \in I} S_i$.
- $u : \prod_{i \in I} S_i \to \mathbb{R}^I$ is a *utility function*, which maps strategy profiles to utility profiles.

Definition 12 (Best response)**.** A *best response (BR)* is a strategy that maximizes a player's utility given the other players' strategies. The set of best responses by player i to a strategy profile s is denoted by

$$\mathrm{BR}_i(u, s) = \operatorname*{argmax}_{x \in S_i} u(s[i \mapsto x])_i. \tag{30}$$

The central solution concept in game theory is the *Nash equilibrium (NE)* (Nash, 1951; Osborne, 2004), at which no player has an incentive to change its strategy unilaterally.

Definition 13 (Nash equilibrium)**.** A *Nash equilibrium (NE)* is a strategy profile for which each player's strategy is a best response to the other players' strategies. Thus

$$s \in \mathrm{NE} \quad \iff \quad \forall i \in I : s_i \in \mathrm{BR}_i(u, s). \tag{31}$$

Definition 14 (Player regret)**.** A player's *regret* is the supremum utility it could gain by unilaterally changing its strategy. It is denoted by

$$\mathrm{Reg}_i(s) = \sup_{x \in S_i} u(s[i \mapsto x])_i - u(s)_i. \tag{32}$$

Definition 15 (Exploitability)**.** The *exploitability* of a strategy profile is the aggregate regret across players. It is denoted by

$$\mathrm{Expl}(s) = \sum_{i \in I} \mathrm{Reg}_i(s). \tag{33}$$

Exploitability is commonly used in the literature as a measure of the closeness of a strategy profile to being an NE (Lanctot et al., 2017; Lockhart et al., 2019; Walton & Lisý, 2021; Timbers et al., 2022).

Definition 16 (Mixed strategy)**.** A *mixed strategy* is a probability distribution over (pure) strategies. We extend utility functions to mixed-strategy profiles by using the *expected* utility. If $\sigma \in \prod_{i \in I} \triangle S_i$ is a mixed-strategy profile, then

$$u(\sigma) = \sum_{s \in \prod_{i \in I} S_i} u(s) \prod_{i \in I} \sigma_i(s_i). \tag{34}$$

This is a standard overloading of notation in game theory.

Beyond two players, some classes keep the tractability of the two-player zero-sum case. In a *zero-sum polymatrix game* (Cai et al., 2016), each edge of a graph carries a bimatrix game, and each player plays one strategy across its edges. Equilibria of such games are still computable by linear programming, and they still satisfy a minimax theorem. By contrast, general-sum games are PPAD-hard already for two players (Daskalakis et al., 2009; Chen et al., 2009).

Algorithm 1 Multi-player double oracle algorithm.

input (I, S, u) is a normal-form game
input $X_i \subseteq S_i$ is an initial active set for each player $i \in I$
input $\tau \geq 0$ an exploitability tolerance
loop
  $\sigma \in \text{NE}(u|_X)$ ▷ Compute an NE of the subgame
  $X' \leftarrow X$ ▷ Initialize the next active set before the loop
  for $i \in I$ **do**
    $s_i \in \text{BR}_i(u, \sigma)$ ▷ Compute a BR in the full game against the other players
    $X'_i \leftarrow X_i \cup \{s_i\}$ ▷ Add it to the active set
  if $X' = X$ or $\text{Expl}(\sigma) \leq \tau$ **then**
    break ▷ If active sets did not change or exploitability is below tolerance, stop
  $X \leftarrow X'$
output σ

C.2 The double oracle algorithm and its variants

McMahan et al. (2003) introduced the original version of DO, which was for two-player zero-sum normal-form games, i.e., matrix games. Kroupa & Votroubek (2022) generalized it to multiple players, calling it *Multiple Oracle (MO)*. This algorithm is shown in Algorithm 1. The algorithm maintains a set of actions for each player, called an *active set*. Each active set starts as a small, nonempty set, usually a singleton. Each iteration restricts the full game to the current active sets, forming the *subgame* $u|_X$. It computes a mixed-strategy Nash equilibrium of that subgame, by linear programming in the two-player zero-sum case. It then computes, for each player, a best response *in the full game* to that equilibrium, and adds it to that player's active set. It terminates when no active set changes, or when the exploitability falls below a given threshold.

Two best-response selection schemes are common. Typically, in the case of finite normal-form games, the smallest best response is chosen, under some canonical ordering of the actions. Alternatively, some schemes favor choosing a best response that is already in the current active set, if one exists, to avoid expanding the active set.

We now describe the variants most relevant to the number of iterations.

Kroupa & Votroubek (2022) prove that MO recovers an approximate equilibrium in a finite number of iterations, and converges in the Wasserstein metric to an equilibrium of the original continuous game. The BR computation is based on global solvers for special classes of utility functions. According to the authors, "the choice of best response oracle and the method for solving finite subgames should be fine-tuned for every particular class of games [...] our method converges fast when the dimensions of strategy spaces are small and the generated subgames are not large". Our construction is an instance where the second condition fails: every strategy ends up in the subgame.

Adam et al. (2021) introduced an extension of DO that computes NE of two-player zero-sum games with *continuous*, compact strategy sets. In the authors' words, "The equilibrium of each finite subgame is found by solving a linear program. The best responses were computed by selecting the best point of a uniform discretization for the one-dimensional problems and by using a mixed-integer linear programming reformulation for the Colonel Blotto games." This method converges to an NE, and recovers an *approximate* equilibrium in a finite number of iterations.

Lanctot et al. (2017) introduced *Policy-Space Response Oracles (PSRO)*. It generalizes DO to the case where the "actions" are policies in a multi-step, multiagent environment. Each PSRO iteration trains a new policy with reinforcement learning, so iterations are expensive and their number matters even more than for DO. With exact best responses and a Nash meta-solver, PSRO on a two-player zero-sum game with finitely many deterministic policies is DO on the corresponding matrix game. Any matrix game is such a game with a single step. Hence, our construction also forces PSRO, starting from the first policy of each player, to add every policy. By Theorem 2, this remains true when the meta-solver returns δ-equilibria and the oracles return deterministic approximate best responses, as long as the three tolerances stay below the thresholds of that theorem. These thresholds shrink as M and N grow.

McAleer et al. (2022; 2024) introduced *Anytime Double Oracle (ADO)*, a tabular DO algorithm for two-player zero-sum games that is guaranteed to converge to an NE without increasing exploitability from one iteration to the next. They also introduced *Anytime PSRO (APSRO)*, a version of ADO that calculates best responses via reinforcement learning. Monotone exploitability matters most when the algorithm is stopped before it terminates, which is the practical situation when, as on our construction, running DO to termination would expand every strategy.

Our construction does not force the worst case for ADO. Once every row has entered the population, ADO's row distribution is already optimal in the full game. Hence, the only columns it can still add are the ones tight against it. Among the corridor columns, only the last two are tight against it. ADO therefore stops at most two iterations after the rows run out, taking $\min(M + N - 2, 2M)$ iterations. The formalization proves this for all $2 \leq M \leq N$ (Appendix F), and `code/variants.py` checks it exactly for $2 \leq M \leq 8$ and $M \leq N \leq M + 6$. It reaches the worst case only while $N \leq M + 2$. The smallest matrix of Definition 8 on which it falls short is the 2×5 one, where ADO takes four iterations instead of five, adding the last two columns and never column 2. Forcing the worst case for ADO would need an equilibrium with many tight columns, since those are the only strategies left to add once one population is complete.

Dinh et al. (2021) introduced *Online Double Oracle (ODO)*, which replaces the exact LP solver at each step with no-regret learning algorithms, like *Multiplicative Weights Update (MWU)* (Freund & Schapire, 1999). Their regret bound depends on the size of the *effective strategy set*, the set of strategies that are eventually added, rather than on the size of the full game. For DO itself, our construction shows that this set can be the entire game.

Table 2: Iterations of DO on matrices from six families, as mean and maximum over 1000 draws, against the trivial upper bound. The bigger-number game is deterministic.

Family	10×10	20×20	40×40	10×40
Uniform on $[-1, 1]$	7.5 / 12	14.1 / 19	27.3 / 34	12.6 / 19
Standard normal	7.5 / 12	14.1 / 19	27.3 / 34	12.4 / 18
Sparse, one fifth nonzero	4.0 / 13	10.3 / 23	27.3 / 42	12.4 / 20
Rank two	3.6 / 5	3.5 / 5	3.4 / 5	3.5 / 6
Colonel Blotto	6.7 / 12	12.0 / 18	20.3 / 28	10.8 / 18
Bigger number	9	19	39	10
Definition 8	18	38	78	48

D EMPIRICAL VALIDATION

Correctness is settled by the formal proof of Appendix F. The figures here show what a run looks like. We also verified our construction computationally for all $M, N \in \{2, \ldots, 50\}$, with the script `code/runs.py`. It runs DO from $(\{0\}, \{0\})$ in Python (Van Rossum & Drake, 1995), with exact rational arithmetic throughout, including in the linear program solver for the subgames. It checks that the run is the one of Theorem 1, state by state. The figures were drawn with Matplotlib (Hunter, 2007). Appendix E gives a second script, which re-checks the quantitative claims of the paper rather than the run.

How far is the worst case from the typical case? Table 2 counts the iterations that DO takes from $(\{0\}, \{0\})$, with the smallest-index best response, on matrices drawn from several standard families by the script `code/distributions.py`. It solves the subgames in floating point. The families are entries uniform on $[-1, 1]$, standard normal entries, a sparse and a rank-two variant, Colonel Blotto with 15 soldiers on three battlefields, and the bigger-number game of Section 3. The uniform family is the random zero-sum family of the GAMUT suite (Nudelman et al., 2004). Colonel Blotto is the family on which Adam et al. (2021) tested DO. None of them comes close to $M + N - 2$, as Figure 1 shows for the five random families at two of the shapes. The sparse family also has a long lower tail. The uniform and normal families use about a third of it, Colonel Blotto somewhat less, and the rank-two family far less. In the square case, the bigger-number game has the highest mean of the six, and it uses exactly half. The shortfall is largest on the rectangular shape, where no family reaches half. There, the bound relies on the corridor, whose columns are the convergents of the golden ratio (Appendix A.3). So sampling from natural distributions does not exhibit the worst case, and an explicit construction is needed to show that it can be forced. By Corollary 2, the matrices that force it are not isolated points. Determining the average count on such families mathematically, rather than empirically, remains open.

Figures 2–3 show examples of states for a run of the double oracle algorithm. Dark gray cells denote action profiles not in the subgame. Light gray cells denote action profiles not in the equilibrium. White cells denote action profiles in the equilibrium. Each row and column is labeled with its equilibrium probability (blank if zero). Arrows indicate which actions are currently best responses. *Filled* arrows are best responses with *strictly* better payoff than the current game value. Figure 4 shows the corresponding game values and duality gaps. The duality gap drops to zero on the final state, and thus does not appear on the logarithmic-scale duality gap plot.

Figure 5 shows some states for a larger matrix. Figure 6 shows the corresponding game values and duality gaps. Figure 7 shows the game values and duality gaps for an even larger matrix.

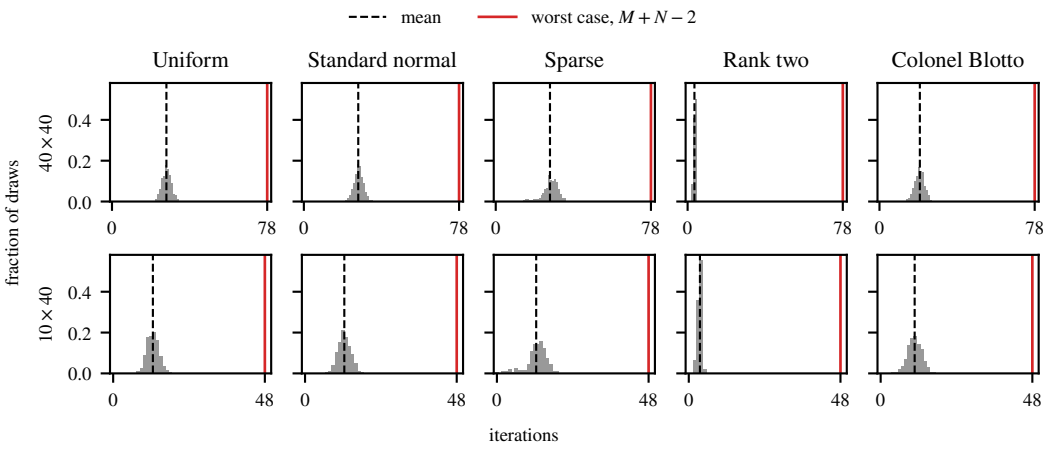


Figure 1: Iterations of DO over the 1000 draws of Table 2, for the five random families at two of its shapes. The dashed line is the mean. The solid line is the worst case, $M + N - 2$, which the matrix of Definition 8 forces.

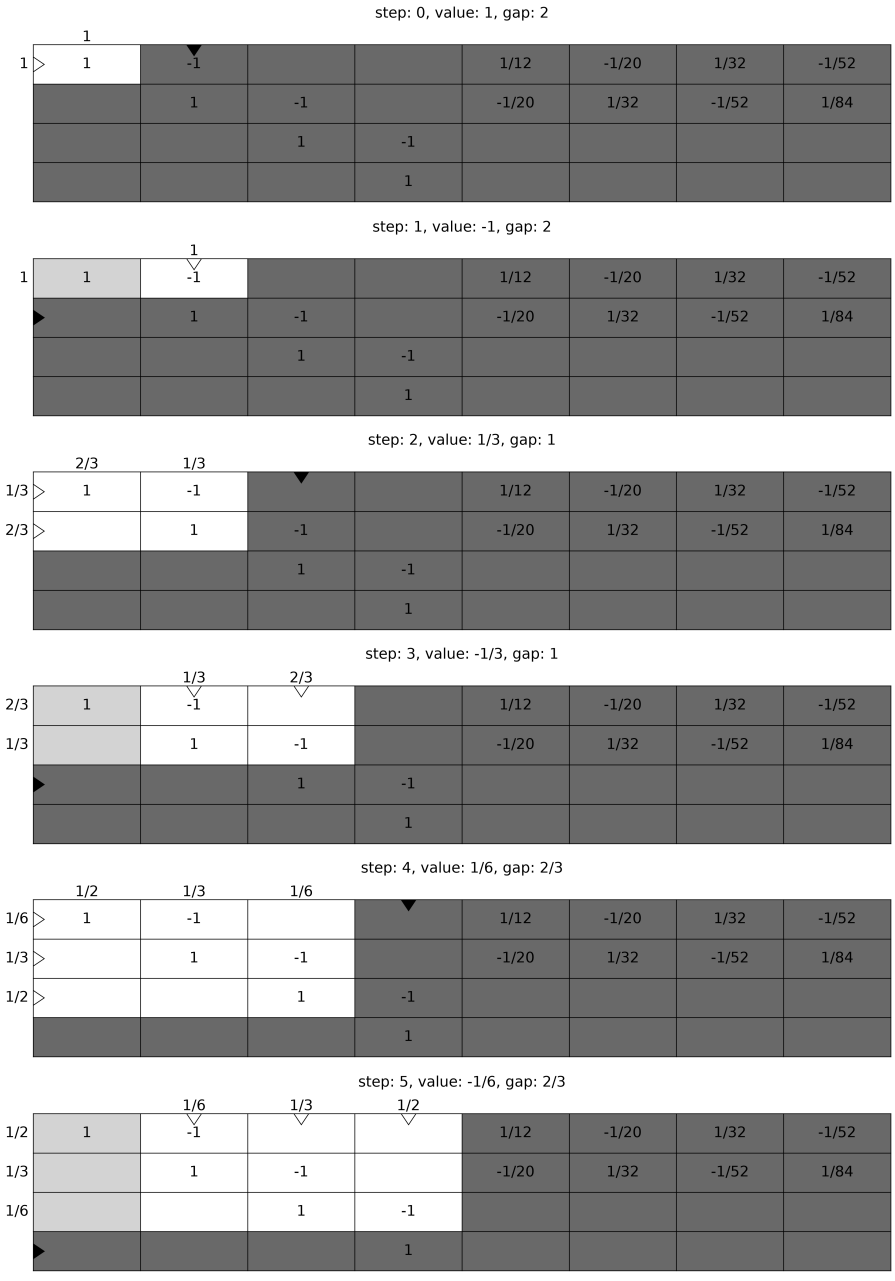


Figure 2: States of the double oracle algorithm on the 4×8 matrix for steps 0–5.

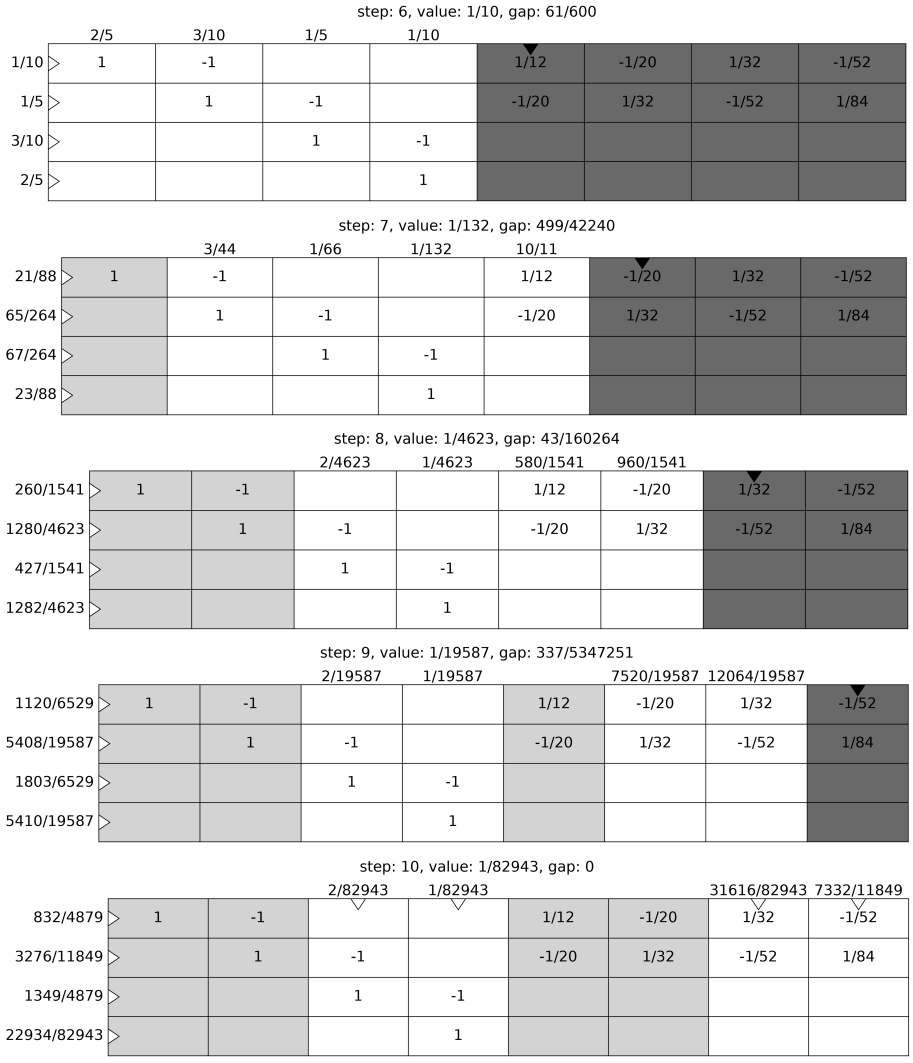


Figure 3: States of the double oracle algorithm on the 4×8 matrix for steps 6–10.

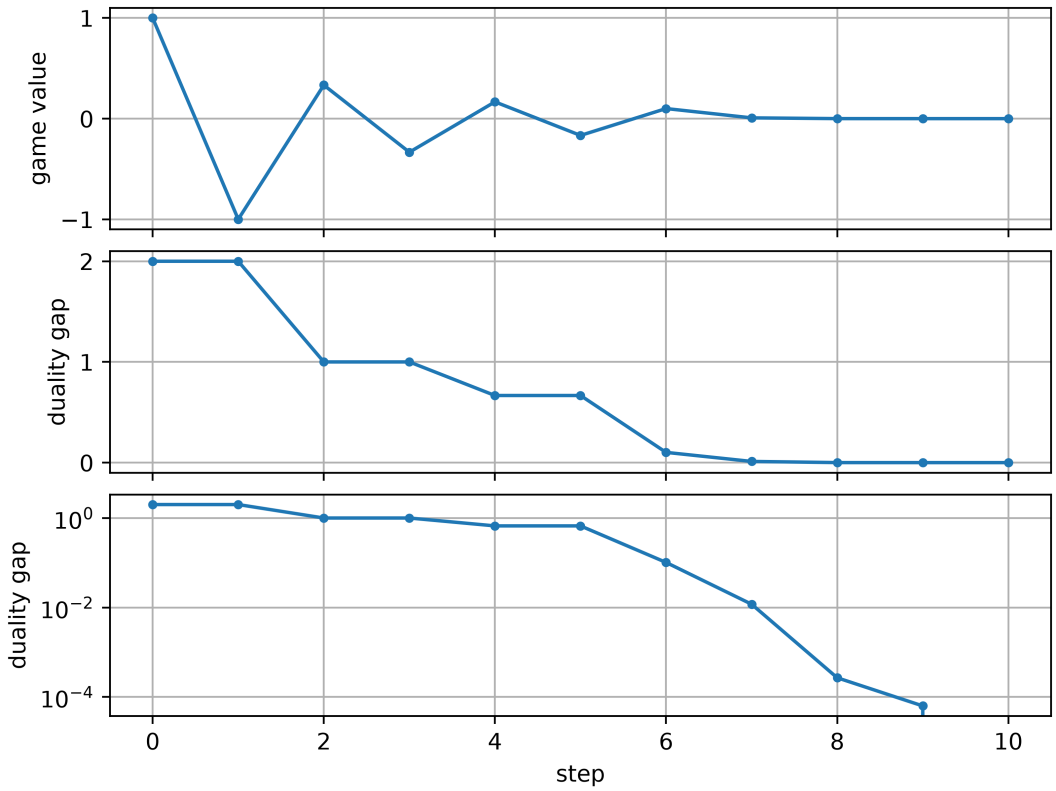


Figure 4: Game value and duality gap for each state on the 4×8 matrix. Notice the transition at $(\min(M, N) - 1) \times 2 = 6$.

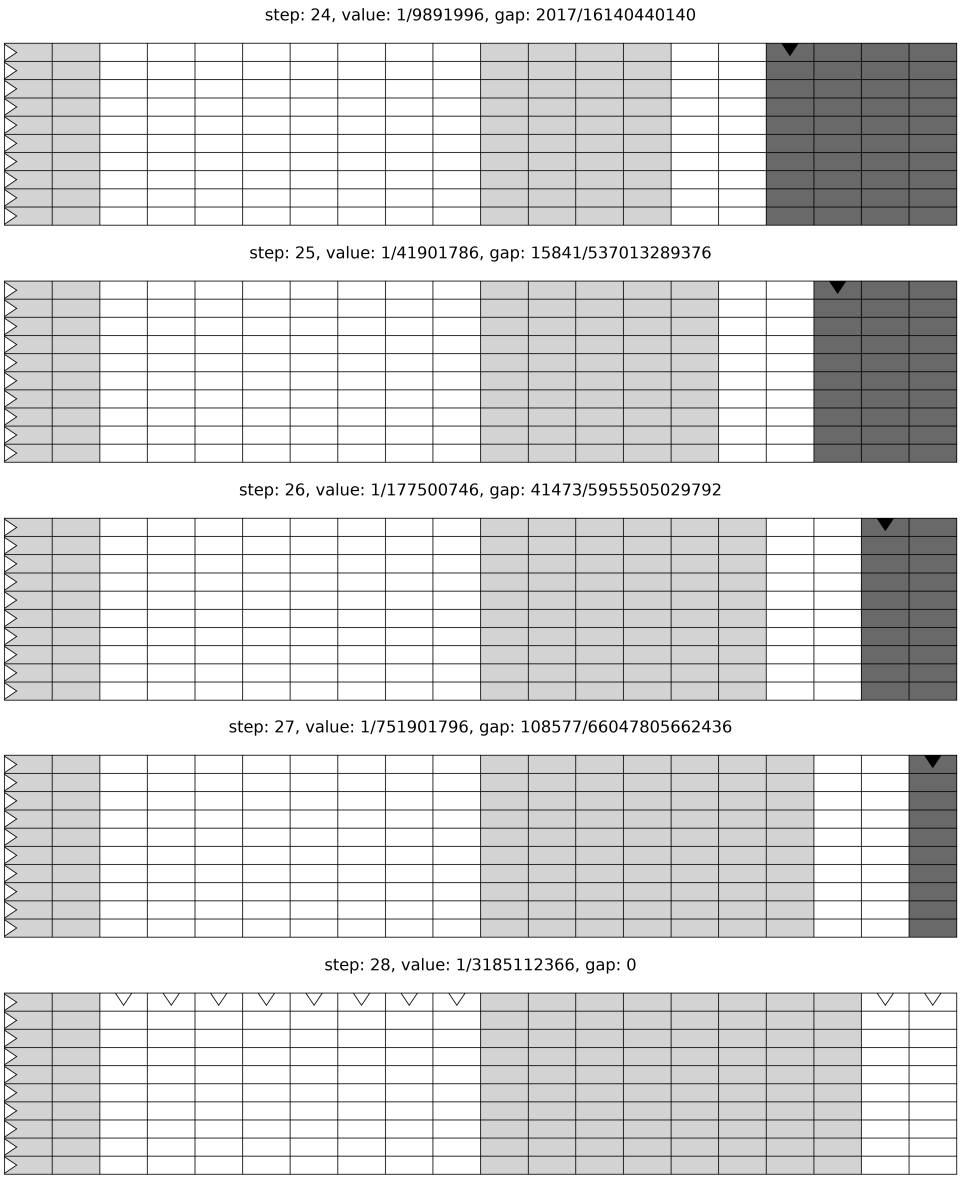


Figure 5: States of the double oracle algorithm on the 10×20 matrix for steps 24–28. Matrix entries and action probabilities are omitted here, due to crowding.

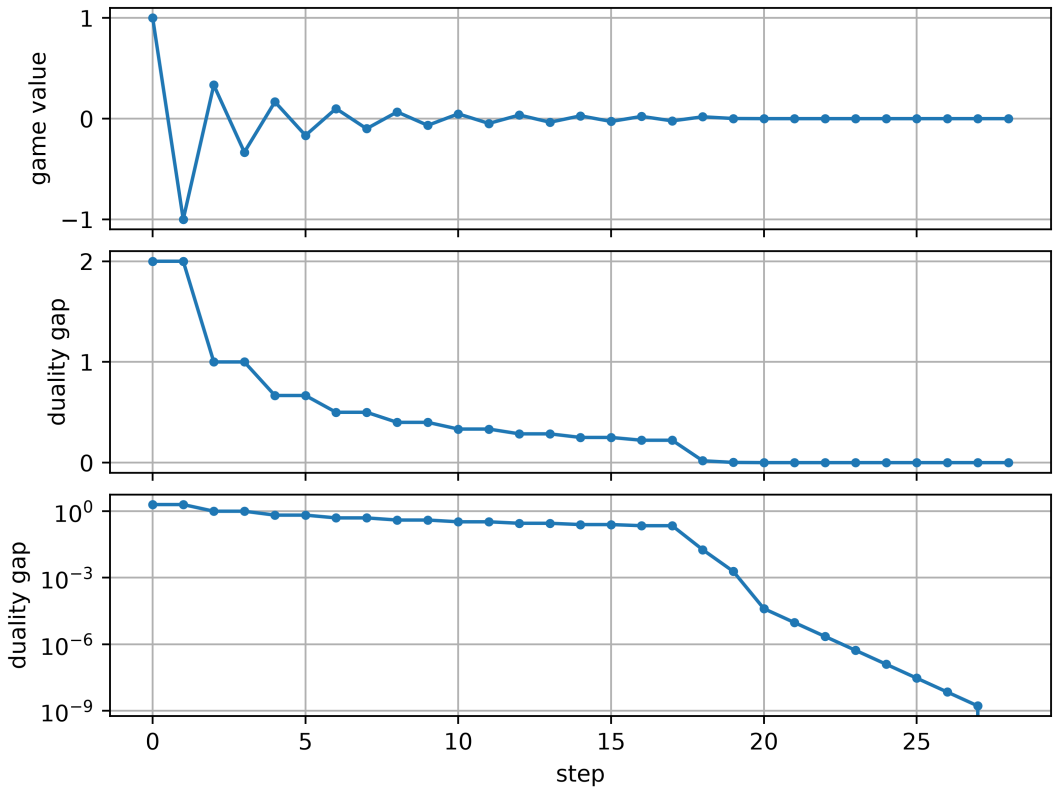


Figure 6: Game value and duality gap for each state on the 10×20 matrix. Notice the transition at $(\min(M, N) - 1) \times 2 = 18$.

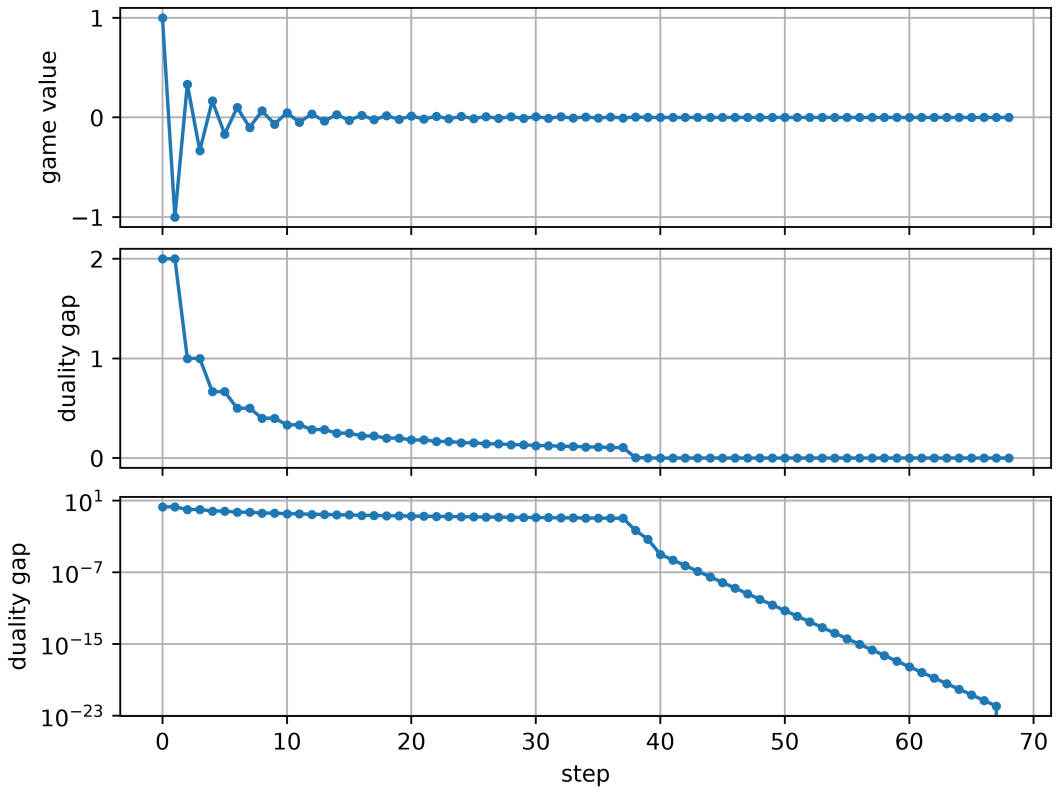


Figure 7: Game value and duality gap for each state on the 20×50 matrix. Notice the transition at $(\min(M, N) - 1) \times 2 = 38$.

E CHECKS IN PYTHON

The Python (Van Rossum & Drake, 1995) script below, `code/verify.py`, was created with the assistance of Claude Fable 5.1 (Anthropic, 2026). It re-checks the quantitative claims over the rationals: the example matrix, the payoff identities (8) and (9), each certificate of Lemmas 5 to 8 with the value, margin, and weight that its lemma states and with the stated strategy as the unique new best response, every row of Table 1, the bound (13), every constant asserted in the proof of Theorem 2, and the two witnesses of Theorem 3. It also checks that at each state the columns tight against the certificate number exactly $|R|$ and are linearly independent over the rows of R, and that the certificate weights them all positively. By Lemma 3, the subgame then has exactly one equilibrium, so at that state the equilibrium selection scheme has no freedom. Every quantity is a `Fraction`, so a passing run is a proof for the shapes it sweeps. These are all the shapes with $2 \leq M \leq N \leq 50$. The rank checks settle the quantification over *every* equilibrium at each state swept, and Appendix F settles it for all M and N. It can be run via the following shell command.

```
python3 code/verify.py
```

```
"""Exact-arithmetic checks of the quantitative claims in the paper.

Everything here is computed over the rationals, so a passing run is a proof for
the ranges swept, not a numerical indication. What is checked:

  * def:construction    the printed 5 x 10 example equals A
  * eq:pay_stair,       the payoff formulas, against the matrix itself
    eq:pay_corr
  * lem:staircase_right each certificate is an equilibrium of its subgame,
    ... lem:corridor    with the stated value, margin, and weight, and with
                        the stated strategy as the unique new best response
  * tab:value_gap       value, gap, and margin at each state, d = |v| + m
  * eq:mu_bound         the lower bound on mu
  * thm:accuracy        its constants and the reduced hypotheses of
                        lem:alternation, at delta = eps = min(|v|, m) / 64
                        at each state, which covers mu / 64, and the
                        identities and decay bounds of its proof
  * thm:accuracy_sharp  the witnesses at S_{M-1} and at T_{M-1}
  * uniqueness          at each state the tight columns number |R|, have
                        full rank, and carry positive weight, as do the rows,
                        so by lem:cs the subgame has exactly one equilibrium:
                        equilibrium selection has no freedom

The lemmas quantify over every equilibrium of the subgame. The rank checks
settle that quantification at each state swept, since a full-rank tight system
has one solution, but only the Lean proof settles it for all M and N.

It sweeps every shape with 2 <= M <= N <= 50, the case N < M being the negative
transpose, on all cores.

Usage: python3 verify.py [max_size]
"""

import sys
from collections.abc import Collection
from concurrent.futures import ProcessPoolExecutor
from fractions import Fraction as Q
from functools import lru_cache
from itertools import repeat

Weights = dict[int, Q]  # a strategy, unnormalized, indexed by pure strategy
Row = tuple[Q, Q, Q]  # value, gap, and margin of one state
Table = dict[tuple[str, int], Row]


@lru_cache(None)
def F(n: int) -> int:
    """Fibonacci, F(0) = 0, F(1) = 1."""
    return n if n < 2 else F(n - 1) + F(n - 2)


def s(M: int, n: int) -> Q:
    """The corridor entry s(n) = (-1)^n / (M F(n))."""
    return Q((-1) ** n, M * F(n))


def A(M: int, N: int, i: int, j: int) -> Q:
    """The matrix of def:construction, for 2 <= M <= N."""
```

```
    if j < M:
        return Q(int(i == j) - int(i + 1 == j))
    return s(M, i + j - M + 4) if i <= 1 else Q(0)


def rank(rows: list[list[Q]]) -> int:
    """Rank over the rationals, by Gaussian elimination."""
    rows = [r[:] for r in rows]
    width = len(rows[0]) if rows else 0
    r = 0
    for c in range(width):
        piv = next((i for i in range(r, len(rows)) if rows[i][c] != 0), None)
        if piv is None:
            continue
        rows[r], rows[piv] = rows[piv], rows[r]
        for i in range(len(rows)):
            if i != r and rows[i][c] != 0:
                f = rows[i][c] / rows[r][c]
                rows[i] = [a - f * b for a, b in zip(rows[i], rows[r])]
        r += 1
    return r


def col_pay(M: int, N: int, R: Collection[int], x: Weights, j: int) -> Q:
    return sum((x[i] * A(M, N, i, j) for i in R), Q(0))


def row_pay(M: int, N: int, C: Collection[int], y: Weights, i: int) -> Q:
    return sum((A(M, N, i, j) * y[j] for j in C), Q(0))


def check_state(M: int, N: int, R: Collection[int], C: Collection[int],
                x: Weights, y: Weights, value: int, new_kind: str,
                new_index: int, margin: Q,
                weight_of: tuple[int, Q] | None = None) -> Row:
    """Check a certificate against lem:certificate, and read off its row.

    `x` and `y` are unnormalized, `value` is the value before normalizing, and
    `new_kind` is 'col' or 'row'. Returns (normalized value, gap, margin), and
    checks that the subgame has exactly one equilibrium.
    """
    assert all(x[i] >= 0 for i in R) and any(x[i] > 0 for i in R)
    assert all(y[j] >= 0 for j in C) and any(y[j] > 0 for j in C)

    # lem:certificate: every column of C pays at least the value, with
    # equality on supp y, and every row of R pays the same amount.
    lam = {row_pay(M, N, C, y, i) for i in R}
    assert len(lam) == 1, (M, N, "rows of R pay different amounts")
    for j in C:
        u = col_pay(M, N, R, x, j)
        assert u >= value and (u == value or y[j] == 0), (M, N, j, "tight")

    X, Y = sum((x[i] for i in R), Q(0)), sum((y[j] for j in C), Q(0))
    v = Q(value, 1) / X

    # the best responses in the full game, from the normalized certificate
    cols = {j: col_pay(M, N, R, x, j) / X for j in range(N)}
    rows = {i: row_pay(M, N, C, y, i) / Y for i in range(M)}
    br_col = [j for j in cols if cols[j] == min(cols.values())]
    br_row = [i for i in rows if rows[i] == max(rows.values())]
    if new_kind == "col":
        assert br_col == [new_index], (M, N, br_col, "column BR")
        assert set(br_row) <= set(R), (M, N, br_row, "a row outside R is a BR")
        assert cols[new_index] == -margin, (M, N, cols[new_index], -margin)
    else:
        assert br_row == [new_index], (M, N, br_row, "row BR not unique/new")
        assert set(br_col) <= set(C), (M, N, br_col, "BR outside C")
        assert rows[new_index] == margin, (M, N, rows[new_index], margin)

    gap = max(rows.values()) - min(cols.values())
    assert gap == abs(v) + margin, (M, N, "d = |v| + m fails")
    if weight_of is not None:
        j, bound = weight_of
        assert Q(y[j], Y) >= bound, (M, N, j, "weight below the stated bound")

    # Uniqueness. The tight columns number |R|, are linearly independent over
    # the rows of R, and carry positive weight, as does every row of R. By
    # lem:cs, every equilibrium (x', y') then has u'_j = v on the tight
    # columns, w'_i = v on the rows, and y'_j = 0 on the slack columns, so
    # each half solves the same invertible system as the certificate.
    tight = [j for j in C if cols[j] == v]
```

```
    assert len(tight) == len(list(R)), (M, N, "tight columns")
    assert all(y[j] > 0 for j in tight), (M, N, "unweighted tight column")
    assert all(x[i] > 0 for i in R), (M, N, "unweighted row")
    square = [[A(M, N, i, j) for i in R] for j in tight]
    assert rank(square) == len(tight), (M, N, "tight columns dependent")
    return v, gap, margin


def check_example() -> None:
    """The 5 x 10 matrix printed after def:construction."""
    rows: list[list[Q | int]] = [
        [1, -1, 0, 0, 0, Q(1, 15), Q(-1, 25), Q(1, 40), Q(-1, 65), Q(1, 105)],
        [0, 1, -1, 0, 0, Q(-1, 25), Q(1, 40), Q(-1, 65), Q(1, 105),
         Q(-1, 170)],
        [0, 0, 1, -1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 1, -1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    ]
    for i in range(5):
        for j in range(10):
            assert A(5, 10, i, j) == rows[i][j], (i, j)


def check_payoff_formulas(M: int, N: int) -> None:
    """eq:pay_stair and eq:pay_corr: u_j from x, against the matrix."""
    x: Weights = {i: Q(i + 3, 7) for i in range(M)}
    x[-1] = Q(0)
    for j in range(N):
        u = col_pay(M, N, range(M), x, j)
        if j < M:
            assert u == x[j] - x[j - 1], (M, N, j)
        else:
            n = j - M + 4
            g = (-1) ** n * (x[0] * F(n + 1) - x[1] * F(n))
            assert u == x[0] * s(M, n) + x[1] * s(M, n + 1)
            assert u == Q(g, M * F(n) * F(n + 1))


def table_2(M: int, N: int) -> Table:
    """Every row of tab:value_gap, checked against its certificate."""
    out: Table = {}
    for k in range(1, min(M, N - 1) + 1):                          # Lemma 5: S_k
        x: Weights = {i: Q(i + 1) for i in range(k)}
        x[-1] = Q(0)
        y = {j: Q(k - j) for j in range(k)}
        m = Q(2, k + 1) if k < M else Q(2, 15 * M ** 2 * (M + 1))
        w = (0, Q(2, M + 1)) if k == M else None  # weight M v at C_M
        out[("S", k)] = check_state(M, N, range(k), range(k), x, y, 1,
                                    "col", k, m, w)
        assert out[("S", k)][0] == Q(2, k * (k + 1))
    for k in range(1, M):                                          # Lemma 6: T_k
        x = {i: Q(k - i) for i in range(k)}
        x[-1] = Q(0)
        y = {j: Q(j) for j in range(k + 1)}
        out[("T", k)] = check_state(M, N, range(k), range(k + 1), x, y, -1,
                                    "row", k, Q(2, k + 1))
        assert out[("T", k)][0] == -Q(2, k * (k + 1))
    if N >= M + 2:  # Lemma 7: C_{M+1}
        x = {i: Q(15 * M + 3 + 2 * i) for i in range(M)}
        x[-1] = Q(0)
        y = {0: Q(0), 1: Q(5 * M - 2), M: Q(15 * M ** 2)}
        y.update({j: Q(2 * (M - j)) for j in range(2, M)})
        m = Q(45 * M - 1, 80 * M ** 2 * (8 * M + 1))
        w = (1, Q(5 * M - 2, 2 * M * (8 * M + 1)))
        out[("C", M + 1)] = check_state(M, N, range(M), range(M + 1), x, y,
                                        2, "col", M + 1, m, w)
        assert out[("C", M + 1)][0] == Q(1, M * (8 * M + 1))
    for k in range(M + 2, N):                                      # Lemma 8: C_k
        K = k - M + 3
        x1 = M * F(K) * F(K + 1) ** 2
        x = {0: Q(M * F(K - 1) * F(K) * F(K + 2))}
        x.update({i: Q(x1 + i - 1) for i in range(1, M)})
        x[-1] = Q(0)
        y = {j: Q(0) for j in range(k)}
        y.update({j: Q(M - j) for j in range(2, M)})
        y[k - 2] = Q(M * F(K - 1) * F(K) * (F(K) + (M - 1) * F(K + 1)))
        y[k - 1] = Q(M * F(K) * F(K + 1) * (F(K - 1) + (M - 1) * F(K)))
        mass = M * F(K) * (M * F(K + 1) ** 2 - F(K) ** 2)
        V = Q(1, mass + (M - 1) * (M - 2) // 2)
        m = V * Q(F(K) ** 2, F(K + 1) * F(K + 2))
        out[("C", k)] = check_state(M, N, range(M), range(k), x, y, 1,
```

```
                                    "col", k, m, (k - 2, Q(1, 6)))
        assert out[("C", k)][0] == V
    return out


def check_mu_bound(M: int, N: int, table: Table) -> Q:
    """eq:mu_bound: mu >= 1 / (8 M^3 F(N - M + 3)^3)."""
    mu: Q = min(min(abs(v), m) for v, _, m in table.values())
    assert mu >= Q(1, 8 * M ** 3 * F(N - M + 3) ** 3), (M, N, "mu bound fails")
    return mu


def check_theorem_2(M: int, N: int, table: Table) -> Q:
    """The constants in thm:accuracy's proof, at delta = eps = min(|v|, m) / 64
    at each state, which covers the thresholds of the theorem."""
    for k in range(1, M):
        v, m = Q(2, k * (k + 1)), Q(2, k + 1)
        d = e = min(v, m) / 64
        assert m >= Q(2, M) and Q(1, 3 * M) <= m / 6  # b = m / 6 at S_k
        assert e < v and e + m / 6 < k * (v - d)
        assert Q(2 * k - 1, 3 * M) <= Q(2, 3)  # b = 3v/4 at T_k
        assert Q(2, 3) * (v + d) <= Q(3, 4) * v
        assert e + Q(3, 4) * v < v and e < k * (v - d)
    worst = Q(0)
    for (fam, k), (v, _, m) in table.items():
        if fam == "T" or (fam == "S" and k < M):
            continue
        d = e = min(abs(v), m) / 64
        # c is the coefficient of delta, and hyp the hypothesis of
        # lem:alternation at the new column, reduced to the tight columns
        if fam == "S":                                          # C_M
            c = Q(2, 15 * M) / (M * v) + Q(3, 15 * M)
            hyp = 3 * (v + d / (M * v)) <= 4 * (v - d)
        elif k == M + 1:
            w = Q(5 * M - 2, 2 * M * (8 * M + 1))
            c = Q(1, 80 * M) / w + Q(9, 16)
            hyp = v + d / w <= 30 * M * (v - d)
        else:
            K = k - M + 3
            c = 6 * Q(F(K) * F(K - 1), F(K + 1) * F(K + 2)) + Q(F(K), F(K + 2))
            hyp = 2 * F(K - 1) * (v + 6 * d) <= F(K + 1) * (v - d)
        worst = max(worst, c)
        assert c <= Q(9, 2), (M, N, k, "needs more than 9/2 delta")
        assert hyp, (M, N, k, "alternation hypothesis fails")
        assert abs(v) + m - Q(11, 2) * d > e  # beats the subgame
        assert Q(3, 8) * (m - Q(9, 2) * d) > e  # beats later columns
    return worst


def check_identities(M: int, N: int) -> None:
    """The identities in thm:accuracy's proof, against a generic strategy:
    the new column's payoff from two tight columns, the alternation
    hypothesis reduced to those columns, and the decay by 5 / 8."""
    x: Weights = {i: Q(3 * i * i + 1, 2 * i + 5) for i in range(M)}
    x[-1] = Q(0)

    def u(j: int) -> Q:
        return col_pay(M, N, range(M), x, j)

    def g(n: int) -> Q:
        sign = 1 if n % 2 == 0 else -1
        return sign * (x[0] * F(n + 1) - x[1] * F(n))

    if N > M:  # C_M, where column M, of index 4, enters
        assert 15 * M * u(M) == 2 * u(0) - 3 * u(1), (M, N)
        assert -2 * g(4) - g(3) == 4 * u(1) - 3 * u(0), (M, N)
    if N > M + 1:  # C_{M+1}, where column M + 1, of index 5, enters
        assert 80 * M * u(M + 1) == -45 * M * u(M) + u(1), (M, N)
        assert -2 * g(5) - g(4) == 30 * M * u(M) - u(1), (M, N)
    for k in range(M + 2, N):  # C_k, where column k, of index K + 1, enters
        K = k - M + 3
        a, b = u(k - 2), u(k - 1)
        assert (F(K + 1) * F(K + 2) * u(k)
                == F(K) * (F(K - 1) * a - F(K + 1) * b)), (M, N, k)
        assert (-2 * g(K + 1) - g(K)
                == M * F(K) * (F(K + 1) * b - 2 * F(K - 1) * a)), (M, N, k)
    for n in range(4, N - M + 4):
        assert 8 * F(n) <= 5 * F(n + 1)  # F(n) / F(n + 1) <= 5 / 8
        assert 32 * F(n) * F(n + 1) <= 5 * F(n + 2) * F(n + 3)
```

```python
def check_theorem_3(M: int, N: int) -> None:
    """The two witnesses of thm:accuracy_sharp, at S_{M-1} and at T_{M-1}."""
    k = M - 1
    # at S_{M-1}: row M-1 pays 0 while the subgame pays the value, so eps >= v
    # lets the row player add it a step early
    y: Weights = {j: Q(k - j) for j in range(k)}
    Y = sum(y.values(), Q(0))
    pay = [row_pay(M, N, range(k), y, i) / Y for i in range(M)]
    v = Q(2, k * (k + 1))
    assert all(p == v for p in pay[:k]) and pay[k] == 0, (M, "S_{M-1} payoffs")

    # at T_{M-1}: y' makes row 0 an eps-best response, and (x, y') is a
    # delta-equilibrium once delta >= 4 / (M^2 - 1) - 2 eps / (M + 1)
    for num in range(M):  # eps = num / M^2 < 1 / M
        eps = Q(num, M ** 2)
        c = Q(2, M * (M + 1)) * (1 - M * eps)
        yp = {j: c * (M - j) + eps for j in range(M)}
        assert sum(yp.values(), Q(0)) == 1 and all(w >= 0 for w in yp.values())
        pay = [row_pay(M, N, range(M), yp, i) for i in range(M)]
        assert all(p == c for p in pay[:k]) and pay[k] == c + eps
        assert pay[0] >= max(pay) - eps  # row 0 is an eps-BR
        delta = Q(4, M ** 2 - 1) - Q(2, M + 1) * eps
        value = -Q(2, M * (M - 1))
        assert max(pay[:k]) - value <= delta  # a delta-equilibrium


def sweep(M: int, top: int) -> tuple[Q, int, int]:
    """Every check at the shapes M x N with M <= N <= top. Returns the worst
    coefficient of delta, the number of shapes, and the number of states."""
    worst, states = Q(0), 0
    for N in range(M, top + 1):
        check_payoff_formulas(M, N)
        table = table_2(M, N)
        check_mu_bound(M, N, table)
        worst = max(worst, check_theorem_2(M, N, table))
        check_identities(M, N)
        states += len(table)
    check_theorem_3(M, M + 3)
    return worst, top + 1 - M, states


def main() -> None:
    top = int(sys.argv[1]) if len(sys.argv) > 1 else 50
    check_example()
    print("def:construction: the printed 5 x 10 example matches A")
    with ProcessPoolExecutor() as pool:
        rows = list(pool.map(sweep, range(2, top + 1), repeat(top)))
    worst = max(w for w, _, _ in rows)
    checked, states = sum(c for _, c, _ in rows), sum(n for *_, n in rows)
    print(f"payoffs, certificates, tab:value_gap, eq:mu_bound, thm:accuracy,"
          f" thm:accuracy_sharp: all {checked} shapes with"
          f" 2 <= M <= N <= {top}")
    print(f"worst coefficient of delta in u'_new <= -m + c delta:"
          f" {float(worst):.3f} (the proof states 9/2)")
    print(f"equilibrium uniqueness: at all {states} states, the tight columns"
          f" number |R|, have full rank, and carry positive weight")
    print("all checks passed")


if __name__ == "__main__":
    main()
```

The script `code/gap.py` checks Proposition 1 the same way: on our matrix at every shape it sweeps, and on random matrices whose run happens to reach full length, which is where the bound could fail. It also checks Proposition 2 on the tilted matrices of that proposition. The script `code/variants.py` runs Anytime Double Oracle on our matrix, as Appendix C.2 reports.

F PROOF IN LEAN

With the assistance of Claude Fable 5.1 (Anthropic, 2026), we obtained a formal proof in Lean 4.32.2 (de Moura et al., 2015; de Moura & Ullrich, 2021) of the results in Table 3. It can be run via the following shell command.

```
cd code && lake env lean proof.lean
```

Table 3 names the theorem that proves each result, so any one of them can be checked against the formalization on its own.

Table 3: Each result and the declaration in `code/proof.lean` that establishes it.

Paper	`code/proof.lean`
Lemma 2 (Certificate)	`certificate`
Lemma 3 (Complementary slackness)	`cs`
Sparsity (Definition 8)	`nonzero_card`
Infinite N (Definition 8)	`double_oracle_cols_inf`, `runInf_card`
Infinite M (Definition 8)	`double_oracle_rows_inf`, `runInf_card`
Infinite M and N (Definition 8)	`double_oracle_inf`, `runStair_card`
Lemma 4 (Alternation)	`alternation`
One equilibrium per subgame (Section 5)	`eq_unique`
Lemma 5 (Staircase right)	`staircase_right`, `pin_S`
Lemma 6 (Staircase down)	`staircase_down`, `pin_T`
Lemma 7 (Handoff)	`handoff`, `pin_H`
Lemma 9 (Cassini's identity)	`cassini`
Lemma 8 (Corridor)	`corridor`, `cert_corridor`, `pin_C`
d'Ocagne's identity (Appendix A.3)	`docagne`, `det_corr`, `stair_one`
Geometric corridor (Appendix A.3)	`geom_corr`
Golden ratio (Appendix A.3)	`pay_neg_iff`, `ratio_C`, `ratio_golden`
Convergents of φ (Appendix A.3)	`goldenRatio_convs`
Their denominators (Appendix A.3)	`goldenRatio_dens_le`
Theorem 1 (Main theorem)	`double_oracle_worst_case`, `run_card`
Theorem 1, case $N < M$	`double_oracle_swap`
Intermediate sizes (Appendix A.4)	`run_sizes`
Table 1, value	`tableValue`, `value_run`
Table 1, gap and margin	`tableMargin`, `gap_run`
Gap at most $4/(t+1)$ (Section 3)	`gap_run_le`, `gap_run_le_swap`
Theorem 2 (Oracle accuracy)	`double_oracle_accuracy`
Equation (13)	`margin_ge`
Corollary 1 (Approximate oracles)	`double_oracle_approx`
Corollary 1, case $N < M$	`double_oracle_approx_swap`
Corollary 2 (Perturbed matrices)	`double_oracle_perturbed`
Corollary 2, case $N < M$	`double_oracle_perturbed_swap`
Theorem 3 (Oracle accuracy, necessity)	`double_oracle_accuracy_sharp`
Proposition 1 (Large gap, negative result)	`gap_shape_run`, `gap_shape`
Proposition 2 (Large gap, square case)	`tilt_worst_case`, `tilt_gap`
Theorem 4 (Any start, negative result)	`any_start`, `any_start_two`
ADO iterations (Appendix C.2)	`ado_iterations`

Several rows leave the matrix of Definition 8 behind. The formalization runs double oracle on an arbitrary matrix, which on ours is the same algorithm. Every subgame there has an equilibrium by the minimax theorem, which `exists_isEqB` proves over any ordered field, from Farkas' lemma by Bartl's induction (Bartl, 2012). So runs exist from every state, as Theorem 4 needs. Theorem 4, the intermediate sizes, Corollary 2, and Proposition 1 are then statements about runs on any matrix. The rows for $N < M$ run double oracle on the negative transpose, whose steps, exact or approximate, are those of the $N \times M$ matrix with the players exchanged (`stepOn_swap_iff`, `stepApproxB_swap_iff`). For Proposition 1, `gap_shape_run` derives from a run the separation of the columns, and of the rows, that `card_le_of_sep` counts, as $\mathtt{k}^{|R|}$ for any integer `k` with

$k \cdot d > 2$. Proposition 2 is about the tilted matrix instead, and `tilt_worst_case` identifies its runs with the same state sequence `run M M` that the main theorem uses.

Four rows need steps of their own. On an infinite game, a step also needs best responses among infinitely many strategies, and `StepOn` asks for them. Every large enough truncation takes them among the strategies of the next state, so the finite runs supply them (`stepOn_iff`). ADO is `StepADO`, which adds a novel best response whenever there is one. The golden-ratio rows are the only ones over $\mathbb{R}$.

The proof is shown below.

```
/-
  Explicit worst case for the double oracle algorithm: the main theorem,
    formally.

  For `2 ≤ M ≤ N`, the runs of double oracle on the `M × N` matrix `A`
    below, starting from
  `({0}, {0})`, are exactly the explicit sequence `run M N`
    (`double_oracle_worst_case`).
  It adds one strategy per iteration, in increasing order (`run_card`),
    and stops at the full
  game after exactly `M + N - 2` iterations (`run_last`).

  As in the paper, each state has a unique successor (`staircase_right`,
    `staircase_down`,
  `handoff`, `corridor`): an explicit equilibrium of the subgame fixes,
    by complementary
  slackness, what every equilibrium pays (`certificate`), and a
    comparison of payoffs then
  leaves a single best response. In fact, every subgame along the run
    has exactly one
  equilibrium (`eq_unique`), which `pin_S`, `pin_T`, `pin_H`, and
    `pin_C` write out.

  The same holds with `δ`-equilibria and approximate best responses
    whenever `δ` and the column
  player's `εC` stay below a fixed fraction of `min (|v_t|, m_t)` at
    every state before the last
  (`margin`, which `margin_eq_min` identifies with that minimum of the
    two columns of Table 1),
  and the row player's `εR` below a fixed fraction of `2 / (M (M - 1))`,
    which is free of `N`
  (`double_oracle_accuracy`), hence for all small enough tolerances
    (`double_oracle_approx`),
  `margin` itself being at least `1 / (8 M^3 F(N - M + 3)^3)`
    (`margin_ge`). Table 1 is tied
  to the run by `value_run` and `gap_run`: at every state before the
    last, every equilibrium of
  that subgame has the value `tableValue M t` in magnitude and the
    duality gap
  `tableValue M t + tableMargin M t`, which is at most `4 / (t + 1)`
    (`gap_run_le`).
  Some run of the approximate algorithm stops early once the tolerance
    of the player that moves
  at some state reaches that gap, once `εR` reaches the value at
    `S_{M-1}`, or once `δ` and `εR`
  reach a joint threshold of the same order at `T_{M-1}`
    (`double_oracle_accuracy_sharp`).
  So the bound on `εR` is of the right order.

  Three sections then leave our matrix behind. Along any run of `M + N -
    2` iterations on any
  matrix with entries in `[-1, 1]`, a duality gap that never falls below
    `d` keeps the columns
```

```
  `d` apart in the sup norm, which bounds how many of them there can be
    (`gap_shape`,
  `gap_shape_run`). That bound is vacuous when `M = N`, and there the
    bigger-number game with a
  tilt `η` runs the full `2 M - 2` iterations (`tilt_worst_case`) with a
    gap of at least
  `2 (1 - (M - 1) η)` at every state before the last (`tilt_gap`). The
    third runs double oracle
  on every matrix (`StepB`). Farkas' lemma gives the minimax theorem
    over any ordered field
  (`exists_isEqB`). No matrix forces `M + N - 2` iterations from every
    pure start unless
  `M = N = 2` (`any_start`, `any_start_two`). No run of `M + N - 2`
    iterations has larger
  subgames than `run M N` (`run_sizes`). And every matrix close enough
    to `A` has the same runs
  (`double_oracle_perturbed`).

  Four closing sections return to our matrix. Its runs never stop when
    `N`, or `M`, or both are
  infinite (`double_oracle_cols_inf`, `double_oracle_rows_inf`,
    `double_oracle_inf`). For `N < M`,
  it is the negative transpose, and the main theorem, both corollaries,
    and the gap bound hold with
  the players exchanged (`double_oracle_swap`,
    `double_oracle_approx_swap`,
  `double_oracle_perturbed_swap`, `gap_run_le_swap`). Its corridor
    offers the column player the
  convergents of the golden ratio (`pay_neg_iff`, `goldenRatio_convs`),
    and by d'Ocagne's identity
  no two of its columns are parallel (`det_corr`). Anytime double oracle
    stops on it after
  `min (M + N - 2) (2 M)` iterations (`ado_iterations`). The matrix
    itself has
  `2 max (M, N) - 1` nonzero entries (`nonzero_card`).

  Payoffs and mixed strategies live in an arbitrary ordered field `K`,
    such as `ℚ`, over which
  the whole development is computable. Only the golden ratio needs `ℝ`.
-/

import Mathlib

open Finset Nat

-- Some lemmas below do not use the order on `K`.
set_option linter.unusedSectionVars false

variable {M N c i j k n : ℕ}

-- The payoffs live in an arbitrary ordered field, such as `ℚ` or `ℝ`.
variable {K : Type*} [Field K] [LinearOrder K] [IsStrictOrderedRing K]

/-! ## The construction -/

/-- The corridor entry `s(n) = (-1)^n / (M F(n))`. -/
def s (M n : ℕ) : K := (-1) ^ n / (M * fib n)

/-- The matrix: an upper-bidiagonal staircase in the columns `j < M`,
    then a two-row corridor. -/
def A (M i j : ℕ) : K :=
  if j < M then (if i = j then 1 else 0) - (if i + 1 = j then 1 else 0)
  else if i ≤ 1 then s M (i + j - M + 4) else 0

/-! ## Sparsity -/
```

```
/-- The corridor entries are nonzero. -/
theorem s_ne_zero (hM : 0 < M) (hn : 0 < n) : (s M n : K) ≠ 0 := by
  have : (fib n : K) ≠ 0 := by exact_mod_cast (fib_pos.2 hn).ne'
  have : (M : K) ≠ 0 := by exact_mod_cast hM.ne'
  simp [s, *]

/-- Column `0` has one nonzero entry, and every other column two: a `1`
    and a `-1` in the
    staircase, or its two corridor entries. -/
theorem col_nonzero (hM : 2 ≤ M) (j : ℕ) :
    ((range M).filter fun i => (A M i j : K) ≠ 0).card = if j = 0 then 1
    else 2 := by
  rcases lt_or_ge j M with hj | hj
  · have e : ∀ i, (A M i j : K) ≠ 0 ↔ i = j ∨ i + 1 = j := fun i => by
      simp only [A, if_pos hj]
      split_ifs <;> simp_all
    rcases Nat.eq_zero_or_pos j with rfl | hj0
    · rw [if_pos rfl, card_eq_one]
      exact ⟨0, by ext i; simp only [mem_filter, mem_range, e,
    mem_singleton]; omega⟩
    · rw [if_neg (by omega), card_eq_two]
      exact ⟨j - 1, j, by omega, by
        ext i; simp only [mem_filter, mem_range, e, mem_insert,
    mem_singleton]; omega⟩
  · rw [if_neg (by omega), card_eq_two]
    refine ⟨0, 1, by omega, ?_⟩
    ext i
    simp only [mem_filter, mem_range, A, if_neg (show ¬ j < M by omega),
    mem_insert,
      mem_singleton]
    constructor
    · rintro ⟨-, h⟩
      split_ifs at h with h1
      · omega
      · exact absurd rfl h
    · rintro (rfl | rfl) <;>
        exact ⟨by omega, by rw [if_pos (by omega)]; exact s_ne_zero (by
    omega) (by omega)⟩

/-- **Sparsity.** For `M, N ≥ 2`, the matrix has `2 max(M, N) - 1`
    nonzero entries. For `N < M`,
    it is the negative transpose of the `N × M` one. -/
theorem nonzero_card (hM : 2 ≤ M) (hN : 2 ≤ N) :
    ((range M ×ˢ range N).filter fun p =>
      (if M ≤ N then A M p.1 p.2 else -A N p.2 p.1 : K) ≠ 0).card = 2 *
    max M N - 1 := by
  -- for `M ≤ N`, count column by column
  have key : ∀ {M N : ℕ}, 2 ≤ M → M ≤ N →
      ((range M ×ˢ range N).filter fun p => (A M p.1 p.2 : K) ≠ 0).card =
     2 * N - 1 := by
    intro M N hM hMN
    rw [card_filter, sum_product_right]
    simp_rw [← card_filter]
    rw [sum_congr rfl fun j _ => col_nonzero hM j]
    obtain ⟨n, rfl⟩ : ∃ n, N = n + 1 := ⟨N - 1, by omega⟩
    rw [sum_range_succ', if_pos rfl, sum_congr rfl fun j _ => if_neg
    (Nat.succ_ne_zero j)]
    simp
    omega
  by_cases h : M ≤ N
  · simp only [if_pos h, max_eq_right h]
    exact key hM h
  -- for `N < M`, transpose
  simp only [if_neg h, neg_ne_zero, max_eq_left (by omega : N ≤ M)]
```

```
  rw [← key hN (by omega : N ≤ M)]
  refine card_nbij' Prod.swap Prod.swap (fun p hp => ?_) (fun p hp =>
    ?_) (fun p _ => rfl)
    (fun p _ => rfl)
  · simp only [coe_filter, mem_product, mem_range, Set.mem_setOf_eq] at
    hp ⊢
    exact ⟨⟨hp.1.2, hp.1.1⟩, hp.2⟩
  · simp only [coe_filter, mem_product, mem_range, Set.mem_setOf_eq] at
    hp ⊢
    exact ⟨⟨hp.1.2, hp.1.1⟩, hp.2⟩

/-! ## Double oracle -/

/-- The payoff of column `j` against `x` on the rows `R`. -/
def colPay (M : ℕ) (R : Finset ℕ) (x : ℕ → K) (j : ℕ) : K := ∑ i ∈ R,
    x i * A M i j

/-- The payoff of row `i` against `y` on the columns `C`. -/
def rowPay (M : ℕ) (C : Finset ℕ) (y : ℕ → K) (i : ℕ) : K := ∑ j ∈ C,
    A M i j * y j

/-- `w` is a mixed strategy on `S`. -/
def IsMix (S : Finset ℕ) (w : ℕ → K) : Prop := (∀ i ∈ S, 0 ≤ w i) ∧ ∑
    i ∈ S, w i = 1

/-- `(x, y)` is an equilibrium of the subgame on `R × C`. -/
def IsEq (M : ℕ) (R C : Finset ℕ) (x y : ℕ → K) : Prop :=
  IsMix R x ∧ IsMix C y ∧ ∀ i ∈ R, ∀ j ∈ C, rowPay M C y i ≤ colPay M R
    x j

/-- `q` is a successor of the state `p`: add a best-response row and a
    best-response column
    to an equilibrium of the subgame. -/
def Step (K : Type*) [Field K] [LinearOrder K] [IsStrictOrderedRing K]
    (M N : ℕ)
    (p q : Finset ℕ × Finset ℕ) : Prop :=
  ∃ (x y : ℕ → K) (i j : ℕ), IsEq M p.1 p.2 x y ∧
    i < M ∧ (∀ i' < M, rowPay M p.2 y i' ≤ rowPay M p.2 y i) ∧
    j < N ∧ (∀ j' < N, colPay M p.1 x j ≤ colPay M p.1 x j') ∧ q =
    (insert i p.1, insert j p.2)

/-- A unique successor, from an equilibrium and the best responses
    against every equilibrium. -/
theorem step_iff (hM : 0 < M) (hN : 0 < N) {R C R' C' : Finset ℕ}
    (hE : ∃ x y : ℕ → K, IsEq M R C x y)
    (hR : ∀ x y : ℕ → K, IsEq M R C x y → ∀ i, (∀ i' < M, rowPay M C y
    i' ≤ rowPay M C y i) →
      i < M → insert i R = R')
    (hC : ∀ x y : ℕ → K, IsEq M R C x y → ∀ j, (∀ j' < N, colPay M R x
    j ≤ colPay M R x j') →
      j < N → insert j C = C')
    (q : Finset ℕ × Finset ℕ) : Step K M N (R, C) q ↔ q = (R', C') := by
  refine ⟨fun ⟨x, y, i, j, h, hi, hi', hj, hj', hq⟩ => ?_, fun hq => ?_⟩
  · rw [hq, hR x y h i hi' hi, hC x y h j hj' hj]
  obtain ⟨x, y, h⟩ := hE
  obtain ⟨i, hi, hi'⟩ := exists_max_image (range M) (rowPay M C y)
    (nonempty_range_iff.2 hM.ne')
  obtain ⟨j, hj, hj'⟩ := exists_min_image (range N) (colPay M R x)
    (nonempty_range_iff.2 hN.ne')
  simp only [mem_range] at hi hj hi' hj'
  exact ⟨x, y, i, j, h, hi, hi', hj, hj', by rw [hq, hR x y h i hi' hi,
    hC x y h j hj' hj]⟩

/-! ## Certificates -/
```

```
section

variable {R C : Finset ℕ}

theorem mix_le {S : Finset ℕ} {w f : ℕ → K} {c : K} (hw : IsMix S w)
    (hf : ∀ i ∈ S, f i ≤ c) :
    Σ i ∈ S, w i * f i ≤ c :=
  (sum_le_sum fun i hi => mul_le_mul_of_nonneg_left (hf i hi) (hw.1 i
    hi)).trans
    (by rw [← sum_mul, hw.2, one_mul])

theorem mix_ge {S : Finset ℕ} {w f : ℕ → K} {c : K} (hw : IsMix S w)
    (hf : ∀ i ∈ S, c ≤ f i) :
    c ≤ Σ i ∈ S, w i * f i := by
  have := mix_le (f := fun i => -f i) hw fun i hi => neg_le_neg (hf i hi)
  simp only [mul_neg, sum_neg_distrib] at this; linarith

/-- A mixture that averages `f ≥ c` to at most `c` sees `f = c` on its
    support. -/
theorem mix_tight {S : Finset ℕ} {w f : ℕ → K} {c : K} (hw : IsMix S w)
    (hf : ∀ i ∈ S, c ≤ f i)
    (hc : Σ i ∈ S, w i * f i ≤ c) {i : ℕ} (hi : i ∈ S) (hpos : 0 < w i)
    : f i = c := by
  have hnn : ∀ i ∈ S, 0 ≤ w i * (f i - c) := fun i hi =>
    mul_nonneg (hw.1 i hi) (sub_nonneg.2 (hf i hi))
  have h0 : Σ i ∈ S, w i * (f i - c) = 0 := le_antisymm (by
    simp only [mul_sub, sum_sub_distrib, ← sum_mul, hw.2]; linarith)
    (sum_nonneg hnn)
  have := (mul_eq_zero.1 ((sum_eq_zero_iff_of_nonneg hnn).1 h0 i
    hi)).resolve_left hpos.ne'
  linarith

theorem sum_range_cast_succ (k : ℕ) : Σ i ∈ range k, ((i : K) + 1) = k *
    (k + 1) / 2 := by
  induction k with
  | zero => simp
  | succ k ih => rw [sum_range_succ, ih]; push_cast; ring

theorem sum_range_cast_sub (k : ℕ) : Σ i ∈ range k, ((i : K) - k) = -(k *
     (k + 1) / 2) := by
  have h := sum_range_cast_succ (K := K) k
  rw [sum_add_distrib, sum_const, card_range, nsmul_eq_mul, mul_one] at h
  rw [sum_sub_distrib, sum_const, card_range, nsmul_eq_mul]
  linarith

theorem sum_range_cast_rev (k : ℕ) : Σ j ∈ range k, ((k : K) - j) = k *
    (k + 1) / 2 := by
  have h : Σ j ∈ range k, ((k : K) - j) = -Σ j ∈ range k, ((j : K) - k)
    := by
    rw [← sum_neg_distrib]; exact sum_congr rfl fun j _ => by ring
  rw [h, sum_range_cast_sub]; ring

theorem sum_range_cast (k : ℕ) : Σ i ∈ range k, (i : K) = k * (k - 1) /
    2 := by
  have h := sum_range_cast_succ (K := K) k
  rw [sum_add_distrib, sum_const, card_range, nsmul_eq_mul, mul_one] at h
  linarith

theorem sum_swap (x y : ℕ → K) :
    Σ i ∈ R, x i * rowPay M C y i = Σ j ∈ C, y j * colPay M R x j := by
  simp only [rowPay, colPay, mul_sum]
  rw [sum_comm]; exact sum_congr rfl fun _ _ => sum_congr rfl fun _ _ =>
    by ring
```

```
/-- Complementary slackness: against an equilibrium `(x', y')`, the
    columns in the support of
    an equilibrium `(x, y)` of value `v` pay `v`, and so do the rows in
    its support. -/
theorem cs {x y x' y' : ℕ → K} {v : K} (hx : IsMix R x) (hy : IsMix C y)
    (hu : ∀ j ∈ C, v ≤ colPay M R x j) (hw : ∀ i ∈ R, rowPay M C y i ≤
    v)
    (h : IsEq M R C x' y') :
    (∀ j ∈ C, v ≤ colPay M R x' j) ∧ (∀ j ∈ C, 0 < y j → colPay M R x'
    j = v) ∧
      (∀ i ∈ R, 0 < x i → rowPay M C y' i = v) := by
  have ha := (sum_swap x' y).symm.trans_le (mix_le h.1 hw)
  have hb := (mix_ge h.2.1 hu).trans_eq (sum_swap x y').symm
  have hu' : ∀ j ∈ C, v ≤ colPay M R x' j := fun j hj =>
    hb.trans (mix_le hx fun i hi => h.2.2 i hi j hj)
  refine ⟨hu', fun j hj => mix_tight hy hu' ha hj, fun i hi hpos => ?_⟩
  have := mix_tight (f := fun i => -rowPay M C y' i) (c := -v) hx
    (fun i hi => neg_le_neg ((mix_ge hy fun j hj => h.2.2 i hi j
    hj).trans ha))
    (by simp only [mul_neg, sum_neg_distrib]; linarith) hi hpos
  linarith

/-- Certificate: nonnegative weights `x, y` against which every column
    pays at least `v`, with
    equality on the support of `y`, and every row pays `l`, normalize to
    an equilibrium of value
    `v / Σ x`. By complementary slackness, this pins down what every
    equilibrium pays. -/
theorem certificate {x y : ℕ → K} {v l : K} (hx : ∀ i ∈ R, 0 ≤ x i) (hy
    : ∀ j ∈ C, 0 ≤ y j)
    (hX : 0 < Σ i ∈ R, x i) (hY : 0 < Σ j ∈ C, y j)
    (hu : ∀ j ∈ C, v ≤ colPay M R x j) (hyu : ∀ j ∈ C, 0 < y j → colPay
    M R x j = v)
    (hw : ∀ i ∈ R, rowPay M C y i = l) :
    IsEq M R C (fun i => x i / Σ i ∈ R, x i) (fun j => y j / Σ j ∈ C, y
    j) ∧
      ∀ x' y' : ℕ → K, IsEq M R C x' y' →
      (∀ j ∈ C, v / Σ i ∈ R, x i ≤ colPay M R x' j) ∧
      (∀ j ∈ C, 0 < y j → colPay M R x' j = v / Σ i ∈ R, x i) ∧
      (∀ i ∈ R, 0 < x i → rowPay M C y' i = v / Σ i ∈ R, x i) := by
  set X := Σ i ∈ R, x i
  set Y := Σ j ∈ C, y j
  have hl : l * X = v * Y := by
    have := sum_swap (M := M) (R := R) (C := C) x y
    rw [sum_congr rfl fun i hi => by rw [hw i hi], ← sum_mul,
      sum_congr rfl fun j hj => (?_ : y j * colPay M R x j = y j * v), ←
     sum_mul] at this
    · linarith
    · rcases (hy j hj).lt_or_eq with h | h
      · rw [hyu j hj h]
      · rw [← h, zero_mul, zero_mul]
  have mx : IsMix R fun i => x i / X :=
    ⟨fun i hi => div_nonneg (hx i hi) hX.le, by rw [← sum_div, div_self
    hX.ne']⟩
  have my : IsMix C fun j => y j / Y :=
    ⟨fun j hj => div_nonneg (hy j hj) hY.le, by rw [← sum_div, div_self
    hY.ne']⟩
  have hu' : ∀ j ∈ C, v / X ≤ colPay M R (fun i => x i / X) j := fun j
    hj => by
    simp only [colPay, div_mul_eq_mul_div, ← sum_div]
    exact div_le_div_of_nonneg_right (hu j hj) hX.le
  have hw' : ∀ i ∈ R, rowPay M C (fun j => y j / Y) i ≤ v / X := fun i
    hi => by
    simp only [rowPay, mul_div_assoc', ← sum_div]
    rw [← rowPay, hw i hi, div_le_div_iff₀ hY hX]; linarith
```

```
  refine ⟨⟨mx, my, fun i hi j hj => (hw' i hi).trans (hu' j hj)⟩, fun x'
    y' h => ?_⟩
  obtain ⟨h1, h2, h3⟩ := cs mx my hu' hw' h
  exact ⟨h1, fun j hj hp => h2 j hj (div_pos hp hY), fun i hi hp => h3 i
    hi (div_pos hp hX)⟩

end

/-! ## Payoffs

  Staircase payoffs telescope: column `j` pays `x j - x (j - 1)` and row
    `i` pays
  `y i - y (i + 1)`, with strategies extended by zero outside the
    subgame. -/

theorem colPay_zero (hM : 0 < M) (hk : 0 < k) (x : ℕ → K) : colPay M
    (range k) x 0 = x 0 := by
  simp [colPay, A, hM, hk]

theorem colPay_succ (hj : j + 1 < M) (x : ℕ → K) : colPay M (range k) x
    (j + 1) =
    (if j + 1 < k then x (j + 1) else 0) - if j < k then x j else 0 := by
  simp [colPay, A, hj, mul_sub, sum_sub_distrib]

theorem rowPay_stair (hc : c ≤ M) (y : ℕ → K) :
    rowPay M (range c) y i = (if i < c then y i else 0) - if i + 1 < c
    then y (i + 1) else 0 := by
  rw [rowPay, sum_congr rfl fun j hj => by rw [A, if_pos (by simp at hj;
    omega)]]
  simp [sub_mul, sum_sub_distrib]

theorem rowPay_corr (y : ℕ → K) (i d : ℕ) : rowPay M (range (M + d)) y
    i =
    rowPay M (range M) y i + Σ t ∈ range d, A M i (M + t) * y (M + t) :=
    by
  rw [rowPay, sum_range_add]; rfl

theorem A_corr (hi : i ≤ 1) (t : ℕ) : (A M i (M + t) : K) = s M (i + t +
    4) := by
  simp [A, hi, show i + (M + t) - M + 4 = i + t + 4 by omega]

theorem A_high (hi : 2 ≤ i) (t : ℕ) : (A M i (M + t) : K) = 0 := by
  simp [A, show ¬ i ≤ 1 by omega]

/-- The payoff of the corridor column of index `n` against rows `0` and
    `1` weighted `a, b`. -/
def pay (M : ℕ) (a b : K) (n : ℕ) : K := a * s M n + b * s M (n + 1)

/-- A corridor column reads only rows `0` and `1`. -/
theorem colPay_corr (hk : 2 ≤ k) (hj : M ≤ j) (x : ℕ → K) :
    colPay M (range k) x j = pay M (x 0) (x 1) (j - M + 4) := by
  obtain ⟨k, rfl⟩ : ∃ k', k = k' + 2 := ⟨k - 2, by omega⟩
  simp [colPay, pay, sum_range_succ', A, show ¬ j < M by omega, add_comm,
    show j + 1 - M + 4 = j - M + 4 + 1 by omega]

/-- The corridor column `M + t` pays `pay M (x 0) (x 1) (t + 4)`. -/
theorem colPay_add (hk : 2 ≤ k) (x : ℕ → K) (t : ℕ) :
    colPay M (range k) x (M + t) = pay M (x 0) (x 1) (t + 4) := by
  rw [colPay_corr hk (by omega), Nat.add_sub_cancel_left]

theorem s_ge (hM : 0 < M) (hn : 4 ≤ n) : (-1 : K) ≤ 3 * M * s M n := by
  have h3 : (3 : K) ≤ fib n := by exact_mod_cast (fib_mono hn : fib 4 ≤
    fib n)
  have hM' : (0 : K) < M := by exact_mod_cast hM
  rw [s, mul_div_assoc', le_div_iff₀ (by positivity)]
```

```
  rcases neg_one_pow_eq_or K n with h | h <;> rw [h] <;> nlinarith

/-- The Fibonacci numbers the handoff uses. -/
theorem fib_vals : fib 3 = 2 ∧ fib 4 = 3 ∧ fib 5 = 5 ∧ fib 6 = 8 := by
    decide

/-- `M F(n) s(n) = (-1)^n`. -/
theorem fib_mul_s (hM : 0 < M) (hn : 0 < n) : (M : K) * fib n * s M n =
    (-1) ^ n := by
  have : (fib n : K) ≠ 0 := by simp; omega
  have : (M : K) ≠ 0 := by simp; omega
  simp only [s]; field_simp

/-- Against weights with `x 0, x 1 ≥ 0`, a corridor column pays at least
    `-(x 0 + x 1) / (3M)`. -/
theorem corr_ge (hM : 0 < M) (hk : 0 < k) (hj : M ≤ j) {x : ℕ → K} (h0
    : 0 ≤ x 0)
    (h1 : 0 ≤ x 1) : -(x 0 + x 1) ≤ 3 * M * colPay M (range k) x j := by
  have e0 := mul_le_mul_of_nonneg_left (s_ge hM (by omega : 4 ≤ j - M +
    4)) h0
  have e1 := mul_le_mul_of_nonneg_left (s_ge hM (by omega : 4 ≤ j - M +
    4 + 1)) h1
  rcases lt_or_ge k 2 with hk2 | hk2
  · obtain rfl : k = 1 := by omega
    simp [colPay, A, show ¬ j < M by omega]; nlinarith
  · rw [colPay_corr hk2 hj, pay]; nlinarith

/-! ## Alternation -/

/-- `g(n) = (-1)^n (a F(n+1) - b F(n))`, the numerator of the corridor
    payoff. -/
def g (a b : K) (n : ℕ) : K := (-1) ^ n * (a * fib (n + 1) - b * fib n)

theorem pay_g (hM : 0 < M) (hn : 0 < n) (a b : K) :
    pay M a b n = g a b n / (M * fib n * fib (n + 1)) := by
  have : (fib n : K) ≠ 0 := by simp; omega
  have : (fib (n + 1) : K) ≠ 0 := by simp
  have : (M : K) ≠ 0 := by simp; omega
  simp only [pay, s, g]; field_simp; ring

/-- Carrying the sign makes `g` run backwards along the Fibonacci
    recurrence, so the corridor
    payoffs alternate once `g` does. -/
theorem g_rec (a b : K) (n : ℕ) : g a b (n + 2) = g a b n - g a b (n +
    1) := by
  simp only [g]; push_cast [fib_add_two]; ring

/-- For `n ≥ 4`, `3 F(n+1) ≤ 5 F(n)` and `8 F(n) ≤ 5 F(n+1)`. -/
theorem fib_ratio (hn : 4 ≤ n) : 3 * fib (n + 1) ≤ 5 * fib n ∧ 8 * fib
    n ≤ 5 * fib (n + 1) := by
  induction n, hn using Nat.le_induction with
  | base => decide
  | succ n hn ih =>
    obtain ⟨h1, h2⟩ := ih
    rw [fib_add_two]
    omega

/-- Two steps of the corridor multiply a payoff by at most `5 / 8` in
    magnitude, when its
    numerator `g` at most quadruples. -/
theorem fib_prod (hn : 4 ≤ n) :
    32 * (fib n * fib (n + 1)) ≤ 5 * (fib (n + 2) * fib (n + 3)) := by
  obtain ⟨h1, h2⟩ := fib_ratio hn
  obtain ⟨h3, h4⟩ := fib_ratio (show 4 ≤ n + 1 by omega)
  have e2 : fib (n + 2) = fib n + fib (n + 1) := fib_add_two
```

```
  have e3 : fib (n + 3) = fib (n + 1) + fib (n + 2) := fib_add_two
  nlinarith [fib_pos.2 (show 0 < n by omega), fib_pos.2 (show 0 < n + 1
    by omega)]

-- The quantitative decay makes this proof heavier than the default
    budget allows.
set_option maxHeartbeats 400000 in
/-- If `g(m) > 0` and `g(m) ≤ -2 g(m+1)`, so that `g` has just turned
    negative and its magnitude
    has at most halved, then the corridor column of index `m + 1` pays a
    negative amount, less
    than every column of higher index. -/
theorem alternation (hM : 0 < M) {a b : K} {m : ℕ} (hm : 3 ≤ m) (h0 : 0
    < g a b m)
    (h1 : g a b m ≤ -2 * g a b (m + 1)) :
    pay M a b (m + 1) < 0 ∧ (∀ n, m + 1 < n → pay M a b (m + 1) < pay M
    a b n) ∧
      ∀ n, m + 1 < n → pay M a b (m + 1) * (5 / 8) ≤ pay M a b n := by
  -- `G t = |g(m + t)|`
  set G : ℕ → K := fun t => (-1) ^ t * g a b (m + t)
  have hrec : ∀ t, G (t + 2) = G (t + 1) + G t := fun t => by
    simp only [G, ← add_assoc, g_rec]; ring
  -- for all `t`, `g(m + t)` has the sign `(-1)^t` and `|g(m + t)| ≤ 2
    |g(m + t + 1)|`
  have inv : ∀ t, 0 < G t ∧ G t ≤ 2 * G (t + 1) := fun t => by
    induction t with
    | zero => constructor <;> norm_num [G] <;> linarith
    | succ t ih =>
      show 0 < G (t + 1) ∧ G (t + 1) ≤ 2 * G (t + 2)
      have := hrec t
      constructor <;> linarith [ih.1, ih.2]
  have hM' : (0 : K) < M := by exact_mod_cast hM
  have hF : ∀ n, 0 < n → (0 : K) < fib n := fun n hn => by
    exact_mod_cast fib_pos.2 hn
  have hD : ∀ t, (0 : K) < M * fib (m + 1 + t) * fib (m + 1 + t + 1) :=
    fun t =>
    mul_pos (mul_pos hM' (hF _ (by omega))) (hF _ (by omega))
  -- the magnitude of the payoff of index `m + 1 + t`
  set T : ℕ → K := fun t => G (t + 1) / (M * fib (m + 1 + t) * fib (m +
    1 + t + 1))
  have hT : ∀ t, 0 < T t := fun t => div_pos (inv (t + 1)).1 (hD t)
  have hP : ∀ t, pay M a b (m + 1 + t) = -((-1) ^ t * T t) := fun t => by
    rw [pay_g hM (by omega)]
    simp only [T, G, show m + (t + 1) = m + 1 + t by ring, pow_succ]
    field_simp
    rw [show (((-1 : K)) ^ t) ^ 2 = 1 from by rw [← pow_mul, pow_mul'];
    norm_num, mul_one]
  -- over two steps the numerator at most quadruples, and the
    denominator more than quadruples
  have hdecay : ∀ t, T (t + 2) < T t := fun t => by
    have hG4 : G (t + 2 + 1) ≤ 4 * G (t + 1) := by
      have e1 : G (t + 2 + 1) = G (t + 2) + G (t + 1) := hrec (t + 1)
      have e2 := hrec t
      linarith [(inv t).2]
    have hD4 : 4 * ((M : K) * fib (m + 1 + t) * fib (m + 1 + t + 1)) <
        M * fib (m + 1 + t + 2) * fib (m + 1 + t + 2 + 1) := by
      have f1 : (fib (m + 1 + t + 2) : K) = fib (m + 1 + t) + fib (m + 1
    + t + 1) := by
        exact_mod_cast fib_add_two
      have f2 : (fib (m + 1 + t + 2 + 1) : K) = fib (m + 1 + t + 1) +
    fib (m + 1 + t + 2) := by
        exact_mod_cast fib_add_two
      have f3 : (fib (m + 1 + t) : K) ≤ fib (m + 1 + t + 1) := by
    exact_mod_cast fib_le_fib_succ
      have := hF (m + 1 + t + 1) (by omega)
```

```
      rw [f2, f1]
      nlinarith [mul_pos hM' (mul_pos this this),
        mul_nonneg hM'.le (mul_nonneg this.le (sub_nonneg.2 f3))]
    simp only [T]
    rw [div_lt_div_iff₀ (hD (t + 2)) (hD t), show m + 1 + (t + 2) = m +
    1 + t + 2 by omega]
    nlinarith [mul_le_mul_of_nonneg_right hG4 (hD t).le,
    mul_lt_mul_of_pos_left hD4 (inv (t + 1)).1]
  -- two steps multiply the magnitude by at most `5 / 8`
  have hdecay8 : ∀ t, 8 * T (t + 2) ≤ 5 * T t := fun t => by
    have hG4 : G (t + 2 + 1) ≤ 4 * G (t + 1) := by
      have e1 : G (t + 2 + 1) = G (t + 2) + G (t + 1) := hrec (t + 1)
      have e2 := hrec t
      linarith [(inv t).2]
    have hD32 : 32 * ((M : K) * fib (m + 1 + t) * fib (m + 1 + t + 1)) ≤
        5 * (M * fib (m + 1 + t + 2) * fib (m + 1 + t + 2 + 1)) := by
      have hnat := fib_prod (show 4 ≤ m + 1 + t by omega)
      have hcast : (32 : K) * (fib (m + 1 + t) * fib (m + 1 + t + 1)) ≤
          5 * (fib (m + 1 + t + 2) * fib (m + 1 + t + 2 + 1)) := by
    exact_mod_cast hnat
      nlinarith [hcast, hM', mul_pos hM' hM']
    simp only [T]
    rw [mul_div_assoc', mul_div_assoc',
      div_le_div_iff₀ (hD (t + 2)) (hD t), show m + 1 + (t + 2) = m + 1 +
     t + 2 by omega]
    nlinarith [mul_le_mul_of_nonneg_right hG4 (hD t).le,
      mul_le_mul_of_nonneg_left hD32 (inv (t + 1)).1.le]
  have h00 : pay M a b (m + 1) = -T 0 := by simpa using hP 0
  refine ⟨by linarith [hT 0], fun n hn => ?_, fun n hn => ?_⟩
  obtain ⟨n, rfl⟩ : ∃ k, n = m + 1 + k := ⟨n - (m + 1), by omega⟩
  rw [h00, hP]
  rcases Nat.even_or_odd n with ⟨t, rfl⟩ | ⟨t, rfl⟩
  · obtain ⟨t, rfl⟩ : ∃ u, t = u + 1 := ⟨t - 1, by omega⟩
    have : ∀ t, T (2 * t + 2) < T 0 := fun t => by
      induction t with
      | zero => exact hdecay 0
      | succ t ih => exact (hdecay (2 * t + 2)).trans ih
    rw [Even.neg_one_pow ⟨t + 1, rfl⟩, show t + 1 + (t + 1) = 2 * t + 2
    by ring]
    linarith [this t]
  · rw [Odd.neg_one_pow ⟨t, rfl⟩]; linarith [hT 0, hT (2 * t + 1)]
  obtain ⟨n, rfl⟩ : ∃ k, n = m + 1 + k := ⟨n - (m + 1), by omega⟩
  rw [h00, hP]
  have hev : ∀ t, 8 * T (2 * t + 2) ≤ 5 * T 0 := fun t => by
    induction t with
    | zero => exact hdecay8 0
    | succ t ih =>
      have h1 := hdecay8 (2 * t + 2)
      have h2 := hT (2 * t + 2)
      have h3 := hT 0
      rw [show 2 * (t + 1) + 2 = 2 * t + 2 + 2 by ring]
      linarith
  rcases Nat.even_or_odd n with ⟨t, rfl⟩ | ⟨t, rfl⟩
  · obtain ⟨t, rfl⟩ : ∃ u, t = u + 1 := ⟨t - 1, by omega⟩
    rw [Even.neg_one_pow ⟨t + 1, rfl⟩, show t + 1 + (t + 1) = 2 * t + 2
    by ring]
    linarith [hev t]
  · rw [Odd.neg_one_pow ⟨t, rfl⟩]; linarith [hT 0, hT (2 * t + 1)]

/-! ## Staircase -/

/-- At `S_k = ([k], [k])`, every equilibrium plays `x i = (i + 1) V`
    with `V > 0`, against
    which every row and every column of the subgame pays `V`. -/
theorem cert_S (hk : 0 < k) (hkM : k ≤ M) :
```

```
    (∃ x y : ℕ → K, IsEq M (range k) (range k) x y) ∧
      ∀ x' y' : ℕ → K, IsEq M (range k) (range k) x' y' →
        ∃ V > 0, (∀ i < k, x' i = (i + 1) * V) ∧ (∀ j < k, colPay M
    (range k) x' j = V) ∧
          ∀ i < k, rowPay M (range k) y' i = V := by
  have hcol : ∀ j ∈ range k, colPay M (range k) (fun i => (i : K) + 1) j
    = 1 := fun j hj => by
    simp only [mem_range] at hj
    rcases j with _ | j
    · simp [colPay_zero (hk.trans_le hkM) hk]
    · simp [colPay_succ (j := j) (by omega : j + 1 < M), hj, show j < k
    by omega]
  have hrow : ∀ i ∈ range k, rowPay M (range k) (fun j => (k : K) - j) i
    = 1 := fun i hi => by
    simp only [mem_range] at hi
    rw [rowPay_stair hkM]
    by_cases h : i + 1 < k
    · simp [hi, h]
    · have : (i : K) + 1 = k := by exact_mod_cast (by omega : i + 1 = k)
      simp [hi, h]; linarith
  have hX : 0 < Σ i ∈ range k, ((i : K) + 1) :=
    sum_pos (fun i _ => by positivity) (nonempty_range_iff.2 hk.ne')
  obtain ⟨hE, h⟩ := certificate (x := fun i => (i : K) + 1) (y := fun j =
    > (k : K) - j)
    (fun i _ => by positivity) (fun j hj => sub_nonneg.2 (by simp at hj;
    exact_mod_cast hj.le)) hX
    (sum_pos (fun j hj => sub_pos.2 (by simp at hj; exact_mod_cast hj))
      (nonempty_range_iff.2 hk.ne'))
    (fun j hj => (hcol j hj).ge) (fun j hj _ => hcol j hj) hrow
  refine ⟨⟨_, _, hE⟩, fun x' y' h' => ?_⟩
  obtain ⟨-, h2, h3⟩ := h x' y' h'
  set V := 1 / Σ i ∈ range k, ((i : K) + 1)
  have hc : ∀ j < k, colPay M (range k) x' j = V := fun j hj =>
    h2 j (mem_range.2 hj) (sub_pos.2 (by exact_mod_cast hj))
  refine ⟨V, one_div_pos.2 hX, fun i hi => ?_, hc,
    fun i hi => h3 i (mem_range.2 hi) (by positivity)⟩
  induction i with
  | zero => simpa [colPay_zero (hk.trans_le hkM) hk] using hc 0 hk
  | succ i ih =>
    have := hc (i + 1) hi
    simp [colPay_succ (j := i) (by omega : i + 1 < M), hi, show i < k by
    omega] at this
    push_cast; linarith [ih (by omega)]

/-- At `S_k` with `k < M`, column `k` pays strictly less than every
    other column: the staircase
    columns of `[k]` pay `V > 0`, those after `k` pay `0`, and the
    corridor pays at least
    `-3 V / (3 M)`, while column `k` pays `-k V`. -/
theorem cert_S_out (hM : 2 ≤ M) (hk : 0 < k) (hkM : k < M) {x y : ℕ → K}
    (h : IsEq M (range k) (range k) x y) {j : ℕ} (hj : j ≠ k) :
    colPay M (range k) x k < colPay M (range k) x j := by
  obtain ⟨V, hV, hx, hc, -⟩ := (cert_S hk hkM.le).2 x y h
  obtain ⟨m, rfl⟩ : ∃ m, k = m + 1 := ⟨k - 1, by omega⟩
  have hpay : colPay M (range (m + 1)) x (m + 1) = -(((m : K) + 1) * V)
    := by
    rw [colPay_succ hkM, hx m (by omega)]; simp
  rw [hpay]
  rcases lt_or_gt_of_ne hj with hlt | hgt
  · rw [hc j hlt]; nlinarith [Nat.cast_nonneg (α := K) m]
  by_cases hjM : j < M
  · obtain ⟨t, rfl⟩ : ∃ t, j = t + 1 := ⟨j - 1, by omega⟩
    rw [colPay_succ hjM]
    simp only [if_neg (show ¬ t + 1 < m + 1 by omega), if_neg (show ¬ t
    < m + 1 by omega), sub_zero]
```

```
    nlinarith [Nat.cast_nonneg (α := K) m]
  · have e : colPay M (range (m + 1)) x j
        = colPay M (range (m + 1)) (fun i => ((i : K) + 1) * V) j :=
      sum_congr rfl fun i hi => by simp [hx i (mem_range.1 hi)]
    have h3 := corr_ge (x := fun i => ((i : K) + 1) * V) (by omega) (by
    omega) (not_lt.1 hjM)
      (mul_nonneg (by positivity) hV.le) (mul_nonneg (by positivity)
    hV.le)
    have hM2 : (2 : K) ≤ M := by exact_mod_cast hM
    rw [e]
    by_contra hcon
    rw [not_lt] at hcon
    push_cast at h3
    nlinarith [mul_le_mul_of_nonneg_left hcon (by positivity : (0 : K) ≤
    3 * M),
      mul_le_mul_of_nonneg_right hM2 hV.le, mul_nonneg (Nat.cast_nonneg
    (α := K) (M * m)) hV.le]

/-- Staircase right: `S_k → ([k], [k + 1])`, which is `T_k` if `k < M`
    and `C_{M+1}` if `k = M`. -/
theorem staircase_right (hM : 2 ≤ M) (hk : 0 < k) (hkM : k ≤ M) (hkN :
    k < N)
    (q : Finset ℕ × Finset ℕ) : Step K M N (range k, range k) q ↔ q =
    (range k, range (k + 1)) := by
  refine step_iff (by omega) (by omega) (cert_S hk hkM).1
    (fun x y h i hi _ => insert_eq_of_mem (mem_range.2 ?_)) (fun x y h j
    hj _ => ?_) q
  · obtain ⟨V, hV, -, -, hr⟩ := (cert_S hk hkM).2 x y h
    by_contra hik
    have := hi 0 (by omega)
    rw [hr 0 hk, rowPay_stair hkM] at this
    simp [show ¬ i < k by omega, show ¬ i + 1 < k by omega] at this
    linarith
  suffices j = k by rw [this, range_add_one]
  obtain ⟨V, hV, hx, hc, -⟩ := (cert_S hk hkM).2 x y h
  by_contra hne
  rcases lt_or_eq_of_le hkM with hkM | rfl
  -- `k < M`: column `k` pays less than every other column
  · exact absurd (hj k hkN) (not_le.2 (cert_S_out hM hk hkM h hne))
  -- `k = M`: corridor column `M` pays a negative amount, below every
    later one
  · have hcol : ∀ t, colPay k (range k) x (k + t) = pay k (x 0) (x 1) (t +
     4) :=
      fun t => colPay_add (by omega) x t
    have hg : ∀ n, g (x 0) (x 1) n = (-1) ^ n * (V * (fib (n + 1) - 2 *
    fib n)) := fun n => by
      simp only [g, hx 0 (by omega), hx 1 (by omega)]; push_cast; ring
    obtain ⟨f3, f4, f5, -⟩ := fib_vals
    obtain ⟨hneg, hmin, -⟩ := alternation (M := k) (m := 3) (a := x 0) (b
    := x 1) (by omega)
      (by omega)
      (by rw [hg, f4, f3]; norm_num; linarith) (by rw [hg, hg, f5, f4,
    f3]; norm_num; linarith)
    have h4 := hcol 0
    simp only [add_zero, zero_add, show 4 = 3 + 1 from rfl] at h4
    have hle := hj k hkN
    rw [h4] at hle
    rcases lt_or_gt_of_ne hne with hlt | hgt
    · linarith [hc j hlt]
    · obtain ⟨t, rfl⟩ : ∃ t, j = k + t := ⟨j - k, by omega⟩
      rw [hcol] at hle
      linarith [hmin (t + 4) (by omega)]

/-- At `T_k = ([k], [k + 1])`, every equilibrium plays `x` i = (i - k)
    V` with `V < 0`, against
```

```
    which every row of `[k]` pays `V`. -/
theorem cert_T (hk : 0 < k) (hkM : k < M) :
    (∃ x y : ℕ → K, IsEq M (range k) (range (k + 1)) x y) ∧
      ∀ x' y' : ℕ → K, IsEq M (range k) (range (k + 1)) x' y' →
        ∃ V < 0, (∀ i < k, x' i = (i - k) • V) ∧ ∀ i < k, rowPay M
    (range (k + 1)) y' i = V := by
  have hcol : ∀ j ∈ range (k + 1), 0 < j → colPay M (range k) (fun i =>
    (k : K) - i) j = -1 :=
    fun j hj hj0 => by
      obtain ⟨t, rfl⟩ : ∃ t, j = t + 1 := ⟨j - 1, by omega⟩
      simp only [mem_range] at hj
      rw [colPay_succ (j := t) (by omega)]
      by_cases h : t + 1 < k
      · simp [h, show t < k by omega]
      · have : (t : K) + 1 = k := by exact_mod_cast (by omega : t + 1 = k)
        simp [h, show t < k by omega]; linarith
  have hX : 0 < Σ i ∈ range k, ((k : K) - i) :=
    sum_pos (fun i hi => sub_pos.2 (by simp at hi; exact_mod_cast hi))
      (nonempty_range_iff.2 hk.ne')
  obtain ⟨hE, h⟩ := certificate (x := fun i => (k : K) - i) (y := fun j =
    > (j : K))
    (v := -1) (l := -1)
    (fun i hi => sub_nonneg.2 (by simp at hi; exact_mod_cast hi.le))
    (fun j _ => by positivity) hX
    (sum_pos' (fun j _ => by positivity) ⟨k, by simp, by simpa⟩)
    (fun j hj => by
      rcases Nat.eq_zero_or_pos j with rfl | hj0
      · rw [colPay_zero (by omega) hk]; simp; linarith
      · rw [hcol j hj hj0])
    (fun j hj hy => hcol j hj (by simpa using hy))
    (fun i hi => by
      simp at hi; simp [rowPay_stair (M := M) (c := k + 1) (by omega),
    show i < k + 1 by omega, hi])
  refine ⟨⟨_, _, hE⟩, fun x' y' h' => ?_⟩
  obtain ⟨-, h2, h3⟩ := h x' y' h'
  set V := -1 / Σ i ∈ range k, ((k : K) - i)
  have hc : ∀ t < k, colPay M (range k) x' (t + 1) = V :=
    fun t ht => h2 (t + 1) (by simp; omega) (by positivity)
  have ap : ∀ i < k, x' i = x' 0 + i • V := fun i hi => by
    induction i with
    | zero => simp
    | succ i ih =>
      have := hc i (by omega)
      simp [colPay_succ (j := i) (by omega : i + 1 < M), hi, show i < k
    by omega] at this
      push_cast; linarith [ih (by omega)]
  obtain ⟨m, rfl⟩ : ∃ m, k = m + 1 := ⟨k - 1, by omega⟩
  have htop := hc m (by omega)
  simp [colPay_succ (j := m) hkM, ap m (by omega)] at htop
  refine ⟨V, div_neg_of_neg_of_pos (by norm_num) hX, fun i hi => ?_,
    fun i hi => h3 i (by simp; omega) (sub_pos.2 (by exact_mod_cast hi))⟩
  rw [ap i hi]; push_cast; linarith

/-- At `T_k`, column `k` pays `V < 0`, strictly less than every column
    after it: those in the
    staircase pay `0`, and the corridor pays at least `(2 k - 1) V / (3
    M) > V`. -/
theorem cert_T_out (hk : 0 < k) (hkM : k < M) {x y : ℕ → K}
    (h : IsEq M (range k) (range (k + 1)) x y) {j : ℕ} (hj : k < j) :
    colPay M (range k) x k < colPay M (range k) x j := by
  obtain ⟨V, hV, hx, -⟩ := (cert_T hk hkM).2 x y h
  obtain ⟨m, rfl⟩ : ∃ m, k = m + 1 := ⟨k - 1, by omega⟩
  have hpay : colPay M (range (m + 1)) x (m + 1) = V := by
    rw [colPay_succ hkM, hx m (by omega)]; push_cast; ring_nf; simp
  rw [hpay]
```

```
  by_cases hjM : j < M
  · obtain ⟨t, rfl⟩ : ∃ t, j = t + 1 := ⟨j - 1, by omega⟩
    rw [colPay_succ hjM]
    simp only [if_neg (show ¬ t + 1 < m + 1 by omega), if_neg (show ¬ t
    < m + 1 by omega), sub_zero]
    exact hV
  · have hmM : (m : K) + 1 < M := by exact_mod_cast hkM
    have hm0 : (0 : K) ≤ m := Nat.cast_nonneg m
    have e : colPay M (range (m + 1)) x j
        = colPay M (range (m + 1)) (fun i => ((i : K) - (m + 1)) * V) j
    :=
      sum_congr rfl fun i hi => by simp [hx i (mem_range.1 hi)]
    have h3 := corr_ge (x := fun i => ((i : K) - (m + 1)) * V) (by
    omega) (by omega) (not_lt.1 hjM)
      (mul_nonneg_of_nonpos_of_nonpos (by push_cast; linarith) hV.le)
      (mul_nonneg_of_nonpos_of_nonpos (by push_cast; linarith) hV.le)
    rw [e]
    by_contra hcon
    rw [not_lt] at hcon
    push_cast at h3
    nlinarith [mul_le_mul_of_nonneg_left hcon (by positivity : (0 : K) ≤
    3 * M),
      mul_lt_mul_of_neg_right (by linarith : 2 * (m : K) + 1 < 3 * M) hV]

/-- Staircase down: `T_k → S_{k+1}`. -/
theorem staircase_down (hk : 0 < k) (hkM : k < M) (hkN : k < N) (q :
    Finset ℕ × Finset ℕ) :
    Step K M N (range k, range (k + 1)) q ↔ q = (range (k + 1), range
    (k + 1)) := by
  refine step_iff (by omega) (by omega) (cert_T hk hkM).1 (fun x y h i
    hi _ => ?_)
    (fun x y h j hj _ => insert_eq_of_mem (mem_range.2 ?_)) q
  · suffices i = k by rw [this, range_add_one]
    obtain ⟨V, hV, -, hr⟩ := (cert_T hk hkM).2 x y h
    obtain ⟨m, rfl⟩ : ∃ m, k = m + 1 := ⟨k - 1, by omega⟩
    -- row `k` pays `y` k > 0`, since row `k - 1` pays `y` (k - 1) - y`
    k = V < 0`
    have hyk : 0 < rowPay M (range (m + 1 + 1)) y (m + 1) := by
      have h1 := hr m (by omega)
      rw [rowPay_stair (by omega)] at h1 ⊢
      simp at h1 ⊢
      linarith [h.2.1.1 m (by simp)]
    have := hi (m + 1) (by omega)
    by_contra hne
    rcases lt_or_gt_of_ne hne with hlt | hgt
    · rw [hr i hlt] at this; linarith
    · rw [rowPay_stair (i := i) (by omega)] at this
      simp [show ¬ i < m + 1 + 1 by omega, show ¬ i + 1 < m + 1 + 1 by
    omega] at this; linarith
  · -- column `k` pays less than every column after it
    by_contra hge
    exact absurd (hj k hkN) (not_le.2 (cert_T_out hk hkM h (by omega)))

/-! ## Handoff -/

theorem s4 : (s M 4 : K) = 1 / (3 * M) := by simp [s, fib_vals.2.1]; ring
theorem s5 : (s M 5 : K) = -1 / (5 * M) := by simp [s, fib_vals.2.2.1];
    ring
theorem s6 : (s M 6 : K) = 1 / (8 * M) := by simp [s, fib_vals.2.2.2];
    ring

set_option maxHeartbeats 400000 in
/-- At `C_{M+1} = ([M], [M + 1])`, every equilibrium has the value `1 /
    (M (8 M + 1))`, and makes
```

```
    column `1`, corridor column `M`, and the band tight. The certificate
    gives column `1`
    weight at least `(5 M - 2) / (20 M^2)`. -/
theorem cert_handoff (hM : 2 ≤ M) :
    (∃ x y : ℕ → K, IsEq M (range M) (range (M + 1)) x y ∧
        (5 * (M : K) - 2) / (20 * M ^ 2) ≤ y 1) ∧
      ∀ x' y' : ℕ → K, IsEq M (range M) (range (M + 1)) x' y' →
        (∀ j < M + 1, 1 / ((M : K) * (8 * M + 1)) ≤ colPay M (range M)
    x' j) ∧
          x' 1 - x' 0 = 1 / ((M : K) * (8 * M + 1)) ∧
          pay M (x' 0) (x' 1) 4 = 1 / ((M : K) * (8 * M + 1)) ∧
          ∀ j, 2 ≤ j → j < M → colPay M (range M) x' j = 1 / ((M : K) *
     (8 * M + 1)) := by
  have hM' : (2 : K) ≤ M := by exact_mod_cast hM
  set x : ℕ → K := fun i => 15 * M + 3 + 2 * i
  set y : ℕ → K := fun j => if j = 0 then 0 else if j = 1 then 5 * M -
    2 else
    if j < M then 2 * (M - j) else if j = M then 15 * M ^ 2 else 0
  have hx : ∀ i, 0 < x i := fun i => by simp only [x]; positivity
  have hcol : ∀ j < M + 1, 0 < j → colPay M (range M) x j = 2 := fun j
    hj hj0 => by
    obtain ⟨t, rfl⟩ : ∃ t, j = t + 1 := ⟨j - 1, by omega⟩
    by_cases ht : t + 1 < M
    · simp [x, colPay_succ (j := t) ht, ht, show t < M by omega]; ring
    · rw [show t + 1 = M by omega, colPay_corr hM le_rfl, Nat.sub_self,
    zero_add, pay, s4, s5]
      simp only [x]; field_simp; ring
  have hrow : ∀ i ∈ range M, rowPay M (range (M + 1)) y i = 2 := fun i
    hi => by
    simp only [mem_range] at hi
    rw [rowPay_corr, sum_range_one, rowPay_stair le_rfl]
    rcases lt_or_ge i 2 with hi2 | hi2
    · interval_cases i
      · rw [A_corr (by norm_num), show 0 + 0 + 4 = 4 from rfl, s4]
        simp [y, show M ≠ 0 by omega, show M ≠ 1 by omega, show 1 < M
    by omega]
        field_simp; ring
      · rw [A_corr le_rfl, show 1 + 0 + 4 = 5 from rfl, s5]
        by_cases h2 : 2 < M
        · simp [y, h2, show M ≠ 0 by omega, show M ≠ 1 by omega,
            show 1 < M by omega]
          field_simp; ring
        · obtain rfl : M = 2 := by omega
          simp [y]; norm_num
    · rw [A_high hi2]
      by_cases h : i + 1 < M
      · simp [y, h, hi, show i ≠ 0 by omega, show i ≠ 1 by omega]; ring
      · have : (i : K) + 1 = M := by exact_mod_cast (by omega : i + 1 = M)
        simp [y, h, hi, show i ≠ 0 by omega, show i ≠ 1 by omega];
    linarith
  have hy : ∀ j, 0 ≤ y j := fun j => by
    simp only [y]; split_ifs with h0 h1 h2
    · exact le_rfl
    · linarith
    · have : (j : K) < M := by exact_mod_cast h2
      linarith
    · positivity
    · exact le_rfl
  have hyM : y M = 15 * M ^ 2 := by simp [y, show M ≠ 0 by omega, show M
    ≠ 1 by omega]
  have hX : 0 < Σ i ∈ range M, x i := sum_pos (fun i _ => hx i)
    (nonempty_range_iff.2 (by omega))
  have hY : 0 < Σ j ∈ range (M + 1), y j :=
    sum_pos' (fun j _ => hy j) ⟨M, by simp, by rw [hyM]; positivity⟩
  -- the total masses: `Σ x = 2 M (8 M + 1)`, and `Σ y ≤ 20 M^2`
```

```
  have hXval : Σ i ∈ range M, x i = 2 * (M : K) * (8 * M + 1) := by
    simp only [x]
    rw [sum_add_distrib, sum_const, card_range, nsmul_eq_mul, ←
    mul_sum, sum_range_cast]
    field_simp
    ring
  have hYle : Σ j ∈ range (M + 1), y j ≤ 20 * (M : K) ^ 2 := by
    rw [sum_range_succ, hyM]
    have hb : Σ j ∈ range M, y j ≤ Σ _j ∈ range M, 5 * (M : K) :=
    sum_le_sum fun j hj => by
      have hjM : j < M := mem_range.1 hj
      have hj0 : (0 : K) ≤ j := Nat.cast_nonneg j
      have hjM' : (j : K) ≤ M := by exact_mod_cast hjM.le
      simp only [y]
      split_ifs <;> first | omega | linarith
    rw [sum_const, card_range, nsmul_eq_mul] at hb
    nlinarith
  obtain ⟨hE, h⟩ := certificate (x := x) (y := y) (v := 2) (l := 2) (fun
    i _ => (hx i).le)
    (fun j _ => hy j) hX hY
    (fun j hj => by
      rcases Nat.eq_zero_or_pos j with rfl | hj0
      · rw [colPay_zero (by omega) (by omega)]; simp only [x]; push_cast;
    linarith
      · exact (hcol j (by simpa using hj) hj0).ge)
    (fun j hj hyj => hcol j (by simpa using hj) (by
      rcases Nat.eq_zero_or_pos j with rfl | h
      · simp [y] at hyj
      · exact h)) hrow
  have hval : (2 : K) / Σ i ∈ range M, x i = 1 / ((M : K) * (8 * M + 1))
    := by
    rw [hXval]; field_simp
  refine ⟨⟨_, _, hE, ?_⟩, fun x' y' h' => ?_⟩
  · refine le_trans (div_le_div_of_nonneg_left (by linarith) hY hYle) ?_
    rw [show y 1 = 5 * (M : K) - 2 by simp [y]]
  obtain ⟨h1, h2, -⟩ := h x' y' h'
  rw [hval] at h1 h2
  refine ⟨fun j hj => h1 j (mem_range.2 hj), ?_, ?_, fun j hj2 hjM => ?_⟩
  · have := h2 1 (by simp; omega) (by simp [y]; linarith)
    simpa [colPay_succ (j := 0) (by omega : 0 + 1 < M), show 1 < M by
    omega,
      show 0 < M by omega] using this
  · have := h2 M (by simp) (by rw [hyM]; positivity)
    rwa [colPay_corr hM le_rfl, Nat.sub_self, zero_add] at this
  · refine h2 j (by simp; omega) ?_
    have : (j : K) < M := by exact_mod_cast hjM
    simp only [y, if_neg (show j ≠ 0 by omega), if_neg (show j ≠ 1 by
    omega), if_pos hjM]
    linarith

theorem cert_handoff_ex (hM : 2 ≤ M) : ∃ x y : ℕ → K, IsEq M (range M)
    (range (M + 1)) x y := by
  obtain ⟨x, y, h, -⟩ := (cert_handoff (K := K) hM).1
  exact ⟨x, y, h⟩

/-- The handoff: `C_{M+1} → C_{M+2}`. -/
theorem handoff (hM : 2 ≤ M) (hN : M + 1 < N) (q : Finset ℕ × Finset ℕ)
    :
    Step K M N (range M, range (M + 1)) q ↔ q = (range M, range (M +
    2)) := by
  refine step_iff (by omega) (by omega) (cert_handoff_ex hM)
    (fun x y h i _ hi => insert_eq_of_mem (mem_range.2 hi)) (fun x' y' h
    j hj _ => ?_) q
  suffices j = M + 1 by rw [this, ← range_add_one]
  obtain ⟨hge, h1, h2, -⟩ := (cert_handoff hM).2 x' y' h
```

```
  have hM' : (0 : K) < M := by exact_mod_cast (show 0 < M by omega)
  set V : K := 1 / ((M : K) * (8 * M + 1)) with hV_def
  have hV : 0 < V := by rw [hV_def]; positivity
  have hcol : ∀ t, colPay M (range M) x' (M + t) = pay M (x' 0) (x' 1)
    (t + 4) :=
    fun t => colPay_add (by omega) x' t
  rw [pay, s4, s5] at h2
  obtain ⟨-, f4, f5, f6⟩ := fib_vals
  have hg4 : x' 0 * 5 - x' 1 * 3 = 15 * M * V := by field_simp at h2;
    linarith
  have hMV : V ≤ M * V := le_mul_of_one_le_left hV.le (by exact_mod_cast
    (by omega : 1 ≤ M))
  obtain ⟨hneg, hmin, -⟩ := alternation (M := M) (m := 4) (a := x' 0) (b
    := x' 1) (by omega)
    (by omega)
    (by norm_num [g, f4, f5]; nlinarith)
    (by norm_num [g, f4, f5, f6]; nlinarith)
  have h5 := hcol 1
  simp only [show 1 + 4 = 4 + 1 from rfl] at h5
  have hle := hj (M + 1) hN
  rw [h5] at hle
  by_contra hne
  rcases lt_or_gt_of_ne hne with hlt | hgt
  · linarith [hge j hlt]
  · obtain ⟨t, rfl⟩ : ∃ t, j = M + t := ⟨j - M, by omega⟩
    rw [hcol] at hle
    linarith [hmin (t + 4) (by omega)]

/-! ## Corridor -/

/-- **d'Ocagne's identity**, `F(a + 1) F(b) - F(a) F(b + 1) = (-1)^a F(b
    - a)`, by induction
    on `a`. -/
theorem docagne {a b : ℕ} (hab : a ≤ b) :
    (fib (a + 1) : K) * fib b - fib a * fib (b + 1) = (-1) ^ a * fib (b
    - a) := by
  obtain ⟨d, rfl⟩ : ∃ d, b = a + d := ⟨b - a, by omega⟩
  rw [Nat.add_sub_cancel_left]
  clear hab
  induction a with
  | zero => simp
  | succ a ih =>
    have h1 : (fib (a + 2) : K) = fib a + fib (a + 1) := by
    exact_mod_cast fib_add_two
    have h2 : (fib (a + d + 2) : K) = fib (a + d) + fib (a + d + 1) := by
      exact_mod_cast fib_add_two
    rw [show a + 1 + 1 = a + 2 from rfl, show a + 1 + d = a + d + 1 by
    omega,
      show a + d + 1 + 1 = a + d + 2 from rfl, h1, h2, pow_succ]
    linear_combination -ih

/-- Cassini's identity, the case `b = a + 1` of d'Ocagne's. -/
theorem cassini (n : ℕ) : (fib n : K) * fib (n + 2) - fib (n + 1) ^ 2 =
    (-1) ^ (n + 1) := by
  have h := docagne (K := K) (a := n) (b := n + 1) (by omega)
  rw [Nat.add_sub_cancel_left, fib_one, Nat.cast_one, mul_one,
    show n + 1 + 1 = n + 2 from rfl] at h
  rw [pow_succ]
  linear_combination -h

/-- If the corridor columns of indices `m` and `m + 1` pay the same `v >
    0`, every column of
    lower index `n ≥ 1` pays more than `v`. -/
theorem pay_below (hM : 0 < M) {a b v : K} (hv : 0 < v) {m : ℕ} (hm : 0
    < m)
```

```
    (h1 : pay M a b m = v) (h2 : pay M a b (m + 1) = v) {n : ℕ} (hn : 0
    < n) (hnm : n < m) :
    v < pay M a b n := by
  have hM' : (0 : K) < M := by exact_mod_cast hM
  have hD : ∀ n, 0 < n → (0 : K) < M * fib n * fib (n + 1) := fun n hn =
    >
    mul_pos (mul_pos hM' (by exact_mod_cast fib_pos.2 hn)) (by
    exact_mod_cast fib_pos.2 (by omega))
  rw [pay_g hM hm, div_eq_iff (hD m hm).ne'] at h1
  rw [pay_g hM (by omega), div_eq_iff (hD _ (by omega)).ne'] at h2
  -- the payoffs run backwards: `g n = g (n + 1) + g (n + 2)`
  have hrec : ∀ n, g a b n = g a b (n + 1) + g a b (n + 2) := fun n => by
    rw [g_rec]; ring
  have hm0 : 0 < g a b m := by rw [h1]; exact mul_pos hv (hD m hm)
  have hm1 : 0 < g a b (m + 1) := by rw [h2]; exact mul_pos hv (hD _ (by
    omega))
  have key : ∀ d n, n + 1 + d = m → 0 < g a b (n + 1) ∧ g a b m < g a b
    n := fun d => by
    induction d with
    | zero => intro n hn; subst hn; exact ⟨hm0, by rw [hrec n]; linarith⟩
    | succ d ih =>
      intro n hn
      obtain ⟨h3, h4⟩ := ih (n + 1) (by omega)
      exact ⟨by linarith, by rw [hrec n]; linarith⟩
  obtain ⟨-, hk⟩ := key (m - n - 1) n (by omega)
  rw [pay_g hM hn, lt_div_iff₀ (hD n hn)]
  have hmono : (fib n : K) * fib (n + 1) ≤ fib m * fib (m + 1) := by
    exact_mod_cast Nat.mul_le_mul (fib_mono (by omega)) (fib_mono (by
    omega))
  nlinarith [mul_le_mul_of_nonneg_left hmono (mul_pos hv hM').le]

/-- If the corridor columns of indices `m` and `m + 1` pay the same `v >
    0`, the column of index
    `m + 2` pays a negative amount, less than every column of higher
    index. -/
theorem pay_above (hM : 0 < M) {a b v : K} (hv : 0 < v) {m : ℕ} (hm : 2 ≤
    m)
    (h1 : pay M a b m = v) (h2 : pay M a b (m + 1) = v) :
    pay M a b (m + 1 + 1) < 0 ∧ (∀ n, m + 1 + 1 < n → pay M a b (m + 1 +
    1) < pay M a b n) ∧
      ∀ n, m + 1 + 1 < n → pay M a b (m + 1 + 1) * (5 / 8) ≤ pay M a b
    n := by
  have hM' : (0 : K) < M := by exact_mod_cast hM
  have hF : ∀ n, 0 < n → (0 : K) < fib n := fun n hn => by
    exact_mod_cast fib_pos.2 hn
  have hD : ∀ n, 0 < n → (0 : K) < M * fib n * fib (n + 1) := fun n hn =
    >
    mul_pos (mul_pos hM' (hF n hn)) (hF _ (by omega))
  rw [pay_g hM (by omega), div_eq_iff (hD m (by omega)).ne'] at h1
  rw [pay_g hM (by omega), div_eq_iff (hD _ (by omega)).ne'] at h2
  have f : (fib (m + 1 + 1) : K) = fib m + fib (m + 1) := by
    exact_mod_cast fib_add_two
  have fm : (fib m : K) ≤ fib (m + 1) := by exact_mod_cast
    fib_le_fib_succ
  have hrec : g a b (m + 1 + 1) = g a b m - g a b (m + 1) := g_rec a b m
  apply alternation hM (by omega)
  · rw [h2]; exact mul_pos hv (hD _ (by omega))
  · rw [hrec, h1, h2, f]
    nlinarith [mul_le_mul_of_nonneg_left fm (mul_pos (mul_pos hv hM')
    (hF (m + 1) (by omega))).le]

/-- The total mass of the corridor certificate at `C_{M+e+2}`, with `K =
    e + 5`:
    `M F(K) (M F(K+1)^2 - F(K)^2) + (M - 1) (M - 2) / 2`. The value
    there is its reciprocal. -/
```

```
def corrMass (M e : ℕ) : K :=
  (M : K) * fib (e + 5) * (M * ((fib (e + 4) : K) + fib (e + 5)) ^ 2 -
    (fib (e + 5) : K) ^ 2)
    + ((M : K) - 1) * ((M : K) - 2) / 2

/-- The corridor mass is positive, so the value `1 / corrMass M e` is. -/
theorem corrMass_pos (hM : 2 ≤ M) (e : ℕ) : (0 : K) < corrMass M e := by
  have hp : (0 : K) < (fib (e + 4) : K) := by exact_mod_cast fib_pos.2
    (by omega)
  have hq : (0 : K) < (fib (e + 5) : K) := by exact_mod_cast fib_pos.2
    (by omega)
  have hM2 : (2 : K) ≤ M := by exact_mod_cast hM
  have h1 : (0 : K) < (M : K) * ((fib (e + 4) : K) + fib (e + 5)) ^ 2 -
    (fib (e + 5) : K) ^ 2 := by
    nlinarith
  simp only [corrMass]
  nlinarith [mul_pos (mul_pos (show (0 : K) < M by linarith) hq) h1,
    mul_nonneg (by linarith : (0 : K) ≤ (M : K) - 1) (by linarith : (0 :
    K) ≤ (M : K) - 2)]

-- Instance search for a general ordered field makes this long proof
    slow.
set_option maxHeartbeats 800000 in
/-- At `C_{M+e+2}`, every equilibrium has the value `1 / corrMass M e`
    and makes the two frontier
    columns `M + e` and `M + e + 1` tight, and the band too. The
    certificate gives the first
    frontier column weight at least `1 / 6`. -/
theorem cert_corridor (hM : 2 ≤ M) (e : ℕ) :
    (∃ x y : ℕ → K, IsEq M (range M) (range (M + e + 2)) x y ∧ 1 / 6 ≤
    y (M + e)) ∧
      ∀ x' y' : ℕ → K, IsEq M (range M) (range (M + e + 2)) x' y' →
        (∀ j < M + e + 2, 1 / corrMass M e ≤ colPay M (range M) x' j) ∧
          pay M (x' 0) (x' 1) (e + 4) = 1 / corrMass M e ∧
          pay M (x' 0) (x' 1) (e + 4 + 1) = 1 / corrMass M e ∧
          ∀ j, 2 ≤ j → j < M → colPay M (range M) x' j = 1 / corrMass
    M e := by
  have hM0 : 0 < M := by omega
  have hM1 : (1 : K) ≤ M := by exact_mod_cast (show 1 ≤ M by omega)
  obtain ⟨p, hp_def⟩ : ∃ p : K, p = fib (e + 4) := ⟨_, rfl⟩
  obtain ⟨q, hq_def⟩ : ∃ q : K, q = fib (e + 5) := ⟨_, rfl⟩
  obtain ⟨σ, hσ_def⟩ : ∃ σ : K, σ = (-1) ^ (e + 4) := ⟨_, rfl⟩
  have hp : 1 ≤ p := by rw [hp_def]; exact_mod_cast fib_pos.2 (by omega)
  have hq : 1 ≤ q := by rw [hq_def]; exact_mod_cast fib_pos.2 (by omega)
  have hp3 : 3 ≤ p := by rw [hp_def]; exact_mod_cast (fib_mono (by omega
    : 4 ≤ e + 4))
  have hq5 : 5 ≤ q := by rw [hq_def]; exact_mod_cast (fib_mono (by omega
    : 5 ≤ e + 5))
  have hpq : p ≤ q := by rw [hp_def, hq_def]; exact_mod_cast
    fib_le_fib_succ
  have hqp : q ≤ 2 * p := by
    rw [hp_def, hq_def, show e + 5 = e + 3 + 2 from rfl, fib_add_two]
    have : fib (e + 3) ≤ fib (e + 4) := fib_le_fib_succ
    push_cast
    have h4 : ((fib (e + 3) : K)) ≤ fib (e + 4) := by exact_mod_cast this
    rw [show e + 4 = e + 3 + 1 from rfl]
    linarith
  have f6 : (fib (e + 6) : K) = p + q := by
    rw [hp_def, hq_def]; exact_mod_cast fib_add_two
  have hcas : p * (p + q) - q ^ 2 = -σ := by
    have := cassini (K := K) (e + 4)
    rw [show e + 4 + 2 = e + 6 from rfl, show e + 4 + 1 = e + 5 from
    rfl, f6, ← hp_def, ← hq_def,
      pow_succ (-1 : K), ← hσ_def] at this
    linarith
```

```
  have hσ : σ ^ 2 = 1 := by rw [hσ_def, ← pow_mul]; exact
    Even.neg_one_pow ⟨e + 4, by ring⟩
  have hs4 : M * p * s M (e + 4) = σ := by rw [hp_def, hσ_def]; exact
    fib_mul_s hM0 (by omega)
  have hs5 : M * q * s M (e + 5) = -σ := by
    rw [hq_def, hσ_def, fib_mul_s hM0 (by omega), show e + 5 = e + 4 + 1
    from rfl, pow_succ]; ring
  have hs6 : M * (p + q) * s M (e + 6) = σ := by
    rw [← f6, hσ_def, fib_mul_s hM0 (by omega), show e + 6 = e + 4 + 1 +
     1 from rfl, pow_succ,
      pow_succ]
    ring
  set x : ℕ → K := fun i =>
    if i = 0 then M * p * q * (p + 2 * q) else M * q * (p + q) ^ 2 + (i
    - 1)
  set y : ℕ → K := fun j => if 2 ≤ j ∧ j < M then M - j else
    if j = M + e then M * p * q * (q + (M - 1) * (p + q)) else
    if j = M + e + 1 then M * q * (p + q) * ((M - 1) * q + p) else 0
  have mul_ge : ∀ {a b : K}, 1 ≤ a → 1 ≤ b → 1 ≤ a * b := fun ha hb =>
    ha.trans (le_mul_of_one_le_right (by linarith) hb)
  have hMp : 1 ≤ M * p := mul_ge hM1 hp
  have hMq : 1 ≤ M * q := mul_ge hM1 hq
  have hMpq : 1 ≤ M * p * q := mul_ge hMp hq
  have hα : 0 < M * p * q * (q + (M - 1) * (p + q)) := by
    have : 0 ≤ (M - 1) * (p + q) := mul_nonneg (by linarith) (by
    linarith)
    exact mul_pos (by linarith) (by linarith)
  have hβ : 0 < M * q * (p + q) * ((M - 1) * q + p) := by
    have : 0 ≤ (M - 1) * q := mul_nonneg (by linarith) (by linarith)
    exact mul_pos (mul_pos (by linarith) (by linarith)) (by linarith)
  have hpay4 : pay M (x 0) (x 1) (e + 4) = 1 := by
    show x 0 * s M (e + 4) + x 1 * s M (e + 5) = 1
    simp only [x, if_pos, if_neg one_ne_zero]; push_cast
    linear_combination q * (p + 2 * q) * hs4 + (p + q) ^ 2 * hs5 - σ *
    hcas + hσ
  have hpay5 : pay M (x 0) (x 1) (e + 4 + 1) = 1 := by
    show x 0 * s M (e + 5) + x 1 * s M (e + 6) = 1
    simp only [x, if_pos, if_neg one_ne_zero]; push_cast
    linear_combination p * (p + 2 * q) * hs5 + q * (p + q) * hs6 - σ *
    hcas + hσ
  have hband : ∀ t, 1 ≤ t → t + 1 < M → colPay M (range M) x (t + 1) =
    1 := fun t ht htM => by
    simp [colPay_succ (j := t) htM, htM, show t < M by omega, x, show t ≠
     0 by omega]
  have hfront : ∀ t, t < 2 → colPay M (range M) x (M + e + t) = 1 :=
    fun t ht => by
    rw [colPay_corr hM (by omega), show M + e + t - M + 4 = e + 4 + t by
    omega]
    interval_cases t
    · exact hpay4
    · exact hpay5
  have hge : ∀ j < M + e + 2, 1 ≤ colPay M (range M) x j := fun j hj =>
    by
    rcases lt_or_ge j M with hjM | hjM
    · rcases j with _ | _ | t
      · rw [colPay_zero hM0 hM0]; simp [x]
        exact mul_ge hMpq (by linarith)
      · rw [colPay_succ (j := 0) (by omega)]
        simp [x, show 1 < M by omega, hM0]
        rw [show (M : K) * q * (p + q) ^ 2 - M * p * q * (p + 2 * q) = M
    * q * q ^ 2 by ring]
        exact mul_ge hMq (one_le_pow₀ (by linarith))
      · exact (hband (t + 1) (by omega) hjM).ge
    · obtain ⟨t, rfl⟩ : ∃ t, j = M + t := ⟨j - M, by omega⟩
      rcases lt_or_ge t e with ht | ht
```

```
    · rw [colPay_add (by omega)]
      exact (pay_below hM0 one_pos (by omega) hpay4 hpay5 (by omega)
  (by omega)).le
    · obtain ⟨u, rfl⟩ : ∃ u, t = e + u := ⟨t - e, by omega⟩
      rw [← add_assoc, hfront u (by omega)]
have hyz : ∀ j < 2, y j = 0 := fun j hj => by
  simp only [y]; rw [if_neg (by omega), if_neg (by omega), if_neg (by
  omega)]
have hyb : ∀ j, 2 ≤ j → j < M → y j = M - j := fun j h1 h2 => by
  simp only [y]; rw [if_pos ⟨h1, h2⟩]
have hyc : ∀ t < e, y (M + t) = 0 := fun t ht => by
  simp only [y]; rw [if_neg (by omega), if_neg (by omega), if_neg (by
  omega)]
have hya : y (M + e) = M * p * q * (q + (M - 1) * (p + q)) := by simp
  [y]
have hyb' : y (M + (e + 1)) = M * q * (p + q) * ((M - 1) * q + p) := by
  simp [y, ← add_assoc, show ¬ M + e + 1 < M by omega]
have hy : ∀ j, 0 ≤ y j := fun j => by
  simp only [y]; split_ifs with h1
  · have : (j : K) < M := by exact_mod_cast h1.2
    linarith
  all_goals linarith
have hx : ∀ i, 0 < x i := fun i => by
  simp only [x]; split_ifs with h
  · linarith [mul_ge hMpq (show (1 : K) ≤ p + 2 * q by linarith)]
  · have : (1 : K) ≤ i := by exact_mod_cast (by omega : 1 ≤ i)
    linarith [mul_ge hMq (one_le_pow₀ (n := 2) (show (1 : K) ≤ p + q
  by linarith))]
have hr0 : s M (e + 4) * (M * p * q * (q + (M - 1) * (p + q))) +
    s M (e + 5) * (M * q * (p + q) * ((M - 1) * q + p)) = 1 := by
  linear_combination q * (q + (M - 1) * (p + q)) * hs4 + (p + q) * ((M
  - 1) * q + p) * hs5 -
    σ * hcas + hσ
have hr1 : s M (e + 5) * (M * p * q * (q + (M - 1) * (p + q))) +
    s M (e + 6) * (M * q * (p + q) * ((M - 1) * q + p)) = M - 1 := by
  linear_combination p * (q + (M - 1) * (p + q)) * hs5 + q * ((M - 1) *
   q + p) * hs6 -
    (M - 1) * σ * hcas + (M - 1) * hσ
have hrow : ∀ i ∈ range M, rowPay M (range (M + e + 2)) y i = 1 := fun
  i hi => by
  simp only [mem_range] at hi
  rw [show M + e + 2 = M + (e + 1 + 1) from rfl, rowPay_corr,
  sum_range_succ, sum_range_succ,
    sum_eq_zero fun t ht => by rw [hyc t (mem_range.1 ht), mul_zero],
  zero_add, hya, hyb',
    rowPay_stair le_rfl]
  rcases lt_or_ge i 2 with hi2 | hi2
  · interval_cases i
    · rw [A_corr (by norm_num), A_corr (by norm_num), show 0 + e + 4 =
  e + 4 by omega,
      show 0 + (e + 1) + 4 = e + 5 by omega, if_pos hM0, if_pos (by
  omega), hyz 0 (by omega),
      hyz (0 + 1) (by omega), hr0]
      ring
    · rw [A_corr le_rfl, A_corr le_rfl, show 1 + e + 4 = e + 5 by omega,
      show 1 + (e + 1) + 4 = e + 6 by omega, if_pos (by omega), hyz
  1 (by omega), hr1]
      by_cases h2 : 1 + 1 < M
      · rw [if_pos h2, hyb (1 + 1) le_rfl h2]; push_cast; ring
      · have : (M : K) = 2 := by exact_mod_cast (by omega : M = 2)
        rw [if_neg h2]; linarith
  · rw [A_high hi2, A_high hi2, if_pos hi, hyb i hi2 hi]
    by_cases h : i + 1 < M
    · rw [if_pos h, hyb (i + 1) (by omega) h]; push_cast; ring
    · have : (i : K) + 1 = M := by exact_mod_cast (by omega : i + 1 = M)
```

```
      rw [if_neg h]; linarith
have hX : 0 < Σ i ∈ range M, x i := sum_pos (fun i _ => hx i)
  (nonempty_range_iff.2 (by omega))
-- the total masses: `Σ x = corrMass M e`, and `Σ y ≤ 6 y (M + e)` at
  the first frontier column
have hXval : Σ i ∈ range M, x i = corrMass M e := by
  obtain ⟨t, rfl⟩ : ∃ t, M = t + 1 := ⟨M - 1, by omega⟩
  simp only [corrMass, ← hp_def, ← hq_def]
  rw [sum_range_succ']
  simp only [x, if_neg (Nat.succ_ne_zero _), if_pos rfl]
  rw [sum_add_distrib, sum_const, card_range, nsmul_eq_mul,
    sum_congr rfl (fun i _ => by push_cast; ring :
      ∀ i ∈ range t, ((i + 1 : ℕ) : K) - 1 = (i : K)), sum_range_cast]
  push_cast
  ring
have hYsplit : Σ j ∈ range (M + e + 2), y j
    = (Σ j ∈ range M, y j) + (y (M + e) + y (M + e + 1)) := by
  rw [show M + e + 2 = M + (e + 1 + 1) from rfl, sum_range_add,
  sum_range_succ, sum_range_succ,
    sum_eq_zero fun t ht => hyc t (mem_range.1 ht), zero_add,
    show M + (e + 1) = M + e + 1 by omega]
have hYst : Σ j ∈ range M, y j ≤ (M : K) ^ 2 := by
  have hb : Σ j ∈ range M, y j ≤ Σ _j ∈ range M, (M : K) :=
  sum_le_sum fun j hj => by
    have hjM : j < M := mem_range.1 hj
    have hj0 : (0 : K) ≤ j := Nat.cast_nonneg j
    simp only [y]
    split_ifs with h1 h2 h3 <;> linarith
  rw [sum_const, card_range, nsmul_eq_mul] at hb
  rw [sq]
  exact hb
obtain ⟨hE, h⟩ := certificate (x := x) (y := y) (v := 1) (l := 1) (fun
  i _ => (hx i).le)
  (fun j _ => hy j) hX (sum_pos' (fun j _ => hy j) ⟨M + e, by simp, by
  rw [hya]; exact hα⟩)
  (fun j hj => hge j (mem_range.1 hj))
  (fun j hj hyj => by
    simp only [y] at hyj
    split_ifs at hyj with h1 h2 h3
    · obtain ⟨t, rfl⟩ : ∃ t, j = t + 1 := ⟨j - 1, by omega⟩
      exact hband t (by omega) h1.2
    · rw [h2]; exact hfront 0 (by omega)
    · rw [h3]; exact hfront 1 (by omega)
    · exact absurd hyj (lt_irrefl 0)) hrow
have hY : 0 < Σ j ∈ range (M + e + 2), y j :=
  sum_pos' (fun j _ => hy j) ⟨M + e, by simp, by rw [hya]; exact hα⟩
have hMK : (2 : K) ≤ M := by exact_mod_cast hM
have hp0 : (0 : K) < p := by linarith
have hq0 : (0 : K) < q := by linarith
-- the rest of the mass is at most five times the first frontier column
-- `q (p + q) (5 p - q) ≥ 5 · 8 · 9`, and the factor `M (M - 1)` covers
  `M^2`
have hfac : ∀ c : K, 9 ≤ c → (M : K) ^ 2 ≤ M * q * (((M : K) - 1) *
  (p + q) * c) := by
  intro c hc
  have g1 : (8 : K) * 9 ≤ (p + q) * c :=
    mul_le_mul (by linarith) hc (by linarith) (by linarith)
  have g2 : (5 : K) * (8 * 9) ≤ q * ((p + q) * c) :=
    mul_le_mul hq5 g1 (by linarith) (by linarith)
  linarith only [mul_le_mul_of_nonneg_left g2
      (mul_nonneg (by linarith : (0 : K) ≤ (M : K)) (by linarith : (0
  : K) ≤ (M : K) - 1)),
    mul_le_mul_of_nonneg_left hMK (by linarith : (0 : K) ≤ (M : K))]
have hab : (M : K) ^ 2 + y (M + e + 1) ≤ 5 * y (M + e) := by
  rw [hya, show M + e + 1 = M + (e + 1) from rfl, hyb']
```

```
    have key : 5 * ((M : K) * p * q * (q + ((M : K) - 1) * (p + q)))
        - (M : K) * q * (p + q) * (((M : K) - 1) * q + p)
        = M * q * (((M : K) - 1) * (p + q) * (5 * p - q)) + M * q * (p *
    (4 * q - p)) := by ring
    linarith only [key, hfac (5 * p - q) (by linarith),
      mul_nonneg (mul_nonneg (by linarith : (0 : K) ≤ (M : K)) hq0.le)
        (mul_nonneg hp0.le (by linarith : (0 : K) ≤ 4 * q - p))]
  refine ⟨⟨_, _, hE, ?_⟩, fun x' y' h' => ?_⟩
  · rw [le_div_iff₀ hY, hYsplit]
    linarith [hYst, hab]
  obtain ⟨h1, h2, -⟩ := h x' y' h'
  rw [hXval] at h1
  have hf : ∀ t, t < 2 → pay M (x' 0) (x' 1) (e + 4 + t) = 1 / corrMass
    M e :=
    fun t ht => by
    have hyt : 0 < y (M + e + t) := by
      interval_cases t
      · rw [add_zero, hya]; exact hα
      · rw [show M + e + 1 = M + (e + 1) from rfl, hyb']; exact hβ
    have := h2 (M + e + t) (by simp; omega) hyt
    rwa [colPay_corr hM (by omega), show M + e + t - M + 4 = e + 4 + t
    by omega, hXval] at this
  refine ⟨fun j hj => h1 j (mem_range.2 hj), hf 0 (by omega), hf 1 (by
    omega),
    fun j hj2 hjM => ?_⟩
  have : (j : K) < M := by exact_mod_cast hjM
  have := h2 j (by simp; omega) (by rw [hyb j hj2 hjM]; linarith)
  rwa [hXval] at this

theorem cert_corridor_ex (hM : 2 ≤ M) (e : ℕ) :
    ∃ x y : ℕ → K, IsEq M (range M) (range (M + e + 2)) x y := by
  obtain ⟨x, y, h, -⟩ := (cert_corridor (K := K) hM e).1
  exact ⟨x, y, h⟩

/-- The corridor: `C_{M+e+2} → C_{M+e+3}`. -/
theorem corridor (hM : 2 ≤ M) (e : ℕ) (hN : M + e + 2 < N) (q : Finset ℕ
     × Finset ℕ) :
    Step K M N (range M, range (M + e + 2)) q ↔ q = (range M, range (M +
     e + 3)) := by
  refine step_iff (by omega) (by omega) (cert_corridor_ex hM e)
    (fun x y h i _ hi => insert_eq_of_mem (mem_range.2 hi)) (fun x' y' h
    j hj _ => ?_) q
  suffices j = M + e + 2 by rw [this, ← range_add_one]
  obtain ⟨hge, h4, h5, -⟩ := (cert_corridor hM e).2 x' y' h
  have hV : (0 : K) < 1 / corrMass M e := one_div_pos.2 (corrMass_pos hM
    e)
  obtain ⟨hneg, hmin, -⟩ := pay_above (by omega) hV (by omega) h4 h5
  have hcol : ∀ t, colPay M (range M) x' (M + t) = pay M (x' 0) (x' 1)
    (t + 4) :=
    fun t => colPay_add (by omega) x' t
  have hle := hj (M + e + 2) hN
  rw [show M + e + 2 = M + (e + 2) from rfl, hcol, show e + 2 + 4 = e +
    4 + 1 + 1 from rfl]
    at hle
  by_contra hne
  rcases lt_or_gt_of_ne hne with hlt | hgt
  · linarith [hge j hlt]
  · obtain ⟨t, rfl⟩ : ∃ t, j = M + t := ⟨j - M, by omega⟩
    rw [hcol] at hle
    linarith [hmin (t + 4) (by omega)]

/-! ## The run -/

/-- The state after `t` iterations: `S_1, T_1, ..., T_{M-1}, S_M = C_M,
    C_{M+1}, ..., C_N`,
```

```
    after which it stays at `C_N`. -/
def run (M N t : ℕ) : Finset ℕ × Finset ℕ :=
  if t + 2 < 2 * M then (range (t / 2 + 1), range ((t + 1) / 2 + 1))
  else (range M, range (min (t + 2 - M) N))

/-- One strategy is added per iteration, until iteration `M + N - 2` ...
    -/
theorem run_card (hM : 2 ≤ M) (hMN : M ≤ N) (t : ℕ) :
    (run M N t).1.card + (run M N t).2.card = min t (M + N - 2) + 2 := by
  simp only [run]; split_ifs <;> simp <;> omega

/-- ... from which on the state is the full game. -/
theorem run_last (hMN : M ≤ N) (ht : M + N - 2 ≤ t) : run M N t =
    (range M, range N) := by
  simp [run, show ¬ t + 2 < 2 * M by omega, show N ≤ t + 2 - M by omega]

/-- Each state of the run has the next one as its unique successor. -/
theorem double_oracle (hM : 2 ≤ M) (hMN : M ≤ N) (t : ℕ) (q : Finset ℕ ×
     Finset ℕ) :
    Step K M N (run M N t) q ↔ q = run M N (t + 1) := by
  by_cases ht : t + 2 < 2 * M
  · rcases Nat.even_or_odd' t with ⟨k, rfl | rfl⟩
    · have e1 : run M N (2 * k) = (range (k + 1), range (k + 1)) := by
        simp only [run, if_pos ht, show 2 * k / 2 = k by omega, show (2 *
     k + 1) / 2 = k by omega]
      have e2 : run M N (2 * k + 1) = (range (k + 1), range (k + 1 + 1))
    := by
        simp only [run, if_pos (show 2 * k + 1 + 2 < 2 * M by omega),
          show (2 * k + 1) / 2 = k by omega, show (2 * k + 1 + 1) / 2 =
    k + 1 by omega]
      rw [e1, e2]; exact staircase_right hM (by omega) (by omega) (by
    omega) q
    · have e1 : run M N (2 * k + 1) = (range (k + 1), range (k + 1 + 1))
    := by
        simp only [run, if_pos ht, show (2 * k + 1) / 2 = k by omega,
          show (2 * k + 1 + 1) / 2 = k + 1 by omega]
      have e2 : run M N (2 * k + 1 + 1) = (range (k + 1 + 1), range (k +
    1 + 1)) := by
        simp only [run]
        split_ifs with h
        · rw [show (2 * k + 1 + 1) / 2 = k + 1 by omega,
            show (2 * k + 1 + 1 + 1) / 2 = k + 1 by omega]
        · rw [show 2 * k + 1 + 1 + 2 - M = M by omega, min_eq_left hMN,
    show M = k + 1 + 1 by omega]
      rw [e1, e2]; exact staircase_down (by omega) (by omega) (by omega)
    q
  have hrun : ∀ t, ¬ t + 2 < 2 * M → run M N t = (range M, range (min
    (t + 2 - M) N)) :=
    fun t ht => if_neg ht
  rw [hrun t ht, hrun (t + 1) (by omega)]
  rcases le_or_gt N (t + 2 - M) with hN | hN
  · rw [min_eq_right hN, min_eq_right (by omega)]
    obtain ⟨d, rfl⟩ : ∃ d, N = M + d := ⟨N - M, by omega⟩
    refine step_iff (by omega) (by omega) ?_
      (fun x y h i _ hi => insert_eq_of_mem (mem_range.2 hi))
      (fun x y h j _ hj => insert_eq_of_mem (mem_range.2 hj)) q
    rcases d with _ | _ | e
    · exact (cert_S (by omega) le_rfl).1
    · exact cert_handoff_ex hM
    · exact cert_corridor_ex hM e
  · obtain ⟨d, hd⟩ : ∃ d, t + 2 - M = M + d := ⟨t + 2 - 2 * M, by omega⟩
    rw [min_eq_left hN.le, min_eq_left (show t + 1 + 2 - M ≤ N by omega),
      show t + 1 + 2 - M = M + d + 1 by omega, hd]
    rcases d with _ | _ | e
    · exact staircase_right hM (by omega) le_rfl (by omega) q
```

```
    · exact handoff hM (by omega) q
    · exact corridor hM e (by omega) q

/-- A sequence is the only run from its first state when each of its
    states has the next as its
    unique successor. -/
theorem runs_iff {step : Finset ℕ × Finset ℕ → Finset ℕ × Finset ℕ →
    Prop}
    {f : ℕ → Finset ℕ × Finset ℕ} (h : ∀ t q, step (f t) q ↔ q = f (t +
     1))
    (s : ℕ → Finset ℕ × Finset ℕ) : (s 0 = f 0 ∧ ∀ t, step (s t) (s (t +
     1))) ↔ s = f := by
  refine ⟨fun ⟨hs0, hs⟩ => funext fun t => ?_, ?_⟩
  · induction t with
    | zero => exact hs0
    | succ t ih => exact (h t _).1 (ih ▷ hs t)
  · rintro rfl
    exact ⟨rfl, fun t => (h t _).2 rfl⟩

/-- **Main theorem.** For `2 ≤ M ≤ N`, the runs of double oracle from
    `({0}, {0})` are exactly
    `run M N`. The case `M > N`, where the matrix is the negative
    transpose, which switches the
    roles of the two players, is `double_oracle_swap`. -/
theorem double_oracle_worst_case (hM : 2 ≤ M) (hMN : M ≤ N) (s : ℕ →
    Finset ℕ × Finset ℕ) :
    (s 0 = ({0}, {0}) ∧ ∀ t, Step K M N (s t) (s (t + 1))) ↔ s = run M
    N := by
  have h0 : run M N 0 = ({0}, {0}) := by simp [run, show 2 < 2 * M by
    omega]
  rw [← h0]
  exact runs_iff (double_oracle hM hMN) s

/-! ## The equilibrium along the run

  Every subgame along the run has exactly one equilibrium, the
    normalized certificate: `pin_S`,
  `pin_T`, `pin_H`, and `pin_C` write every equilibrium out, and
    `eq_unique` concludes. So the
  equilibrium selection scheme has nothing to select, and only the
    best-response scheme can
  differ between runs. The weights these lemmas give are the ones the
    step lemmas of the paper
  state, such as column `1` at `C_{M+1}`, which carries `(5 M - 2) / (2
    M (8 M + 1))`. -/

/-- At `S_k`, every equilibrium is `x i = (i + 1) W` and `y j = (k - j)
    W`, where
    `W = 2 / (k (k + 1))` is the value. -/
theorem pin_S (hk : 0 < k) (hkM : k ≤ M) {x y : ℕ → K} (h : IsEq M
    (range k) (range k) x y) :
    (∀ i < k, x i = ((i : K) + 1) * (2 / (k * (k + 1)))) ∧
      ∀ j < k, y j = ((k : K) - j) * (2 / (k * (k + 1))) := by
  obtain ⟨V, hV, hx, -, hr⟩ := (cert_S (K := K) hk hkM).2 x y h
  have hk' : (0 : K) < k := by exact_mod_cast hk
  have hsum : Σ i ∈ range k, ((i : K) + 1) * V = 1 :=
    (sum_congr rfl fun i hi => (hx i (mem_range.1 hi)).symm).trans h.1.2
  rw [← sum_mul, sum_range_cast_succ] at hsum
  have hVval : V = 2 / (k * (k + 1)) := by
    field_simp at hsum ⊢; linarith
  -- every row pays `V`: `y (k - 1) = V` and `y j - y (j + 1) = V`
  have hy : ∀ d, d < k → y (k - 1 - d) = ((d : K) + 1) * V := by
    intro d
    induction d with
    | zero =>
```

```
      intro _
      have h1 := hr (k - 1) (by omega)
      rw [rowPay_stair hkM, if_pos (by omega), if_neg (by omega)] at h1
      simpa using h1
    | succ d ih =>
      intro hd
      have h1 := hr (k - 1 - (d + 1)) (by omega)
      rw [rowPay_stair hkM, if_pos (by omega), if_pos (by omega),
        show k - 1 - (d + 1) + 1 = k - 1 - d by omega, ih (by omega)] at
    h1
      push_cast
      linarith
  refine ⟨fun i hi => by rw [hx i hi, hVval], fun j hj => ?_⟩
  have h1 := hy (k - 1 - j) (by omega)
  rw [show k - 1 - (k - 1 - j) = j by omega, Nat.sub_sub, Nat.cast_sub
    (by omega)] at h1
  rw [h1, hVval]
  push_cast
  ring

/-- At `T_k`, every equilibrium is `x i = (k - i) W` and `y j = j W`,
    where `-W = -2 / (k (k + 1))`
    is the value. -/
theorem pin_T (hk : 0 < k) (hkM : k < M) {x y : ℕ → K}
    (h : IsEq M (range k) (range (k + 1)) x y) :
    (∀ i < k, x i = ((k : K) - i) * (2 / (k * (k + 1)))) ∧
      ∀ j < k + 1, y j = (j : K) * (2 / (k * (k + 1))) := by
  obtain ⟨V, hV, hx, hr⟩ := (cert_T (K := K) hk hkM).2 x y h
  have hk' : (0 : K) < k := by exact_mod_cast hk
  have hsum : Σ i ∈ range k, ((i : K) - k) * V = 1 :=
    (sum_congr rfl fun i hi => (hx i (mem_range.1 hi)).symm).trans h.1.2
  rw [← sum_mul, sum_range_cast_sub] at hsum
  have hVval : V = -(2 / (k * (k + 1))) := by
    field_simp at hsum ⊢; linarith
  -- every row of `[k]` pays `V`, so `y j = y 0 - j V`
  have hy : ∀ j, j < k + 1 → y j = y 0 - j * V := by
    intro j
    induction j with
    | zero => intro _; simp
    | succ j ih =>
      intro hj
      have h1 := hr j (by omega)
      rw [rowPay_stair (by omega : k + 1 ≤ M), if_pos (by omega), if_pos
    (by omega)] at h1
      push_cast
      linarith [ih (by omega)]
  -- and the weights sum to `1`, which forces `y 0 = 0`
  have hy0 : y 0 = 0 := by
    have hs : Σ j ∈ range (k + 1), (y 0 - j * V) = 1 :=
      (sum_congr rfl fun j hj => (hy j (mem_range.1 hj)).symm).trans
    h.2.1.2
    rw [sum_sub_distrib, sum_const, card_range, nsmul_eq_mul, ←
    sum_mul, sum_range_cast,
      hVval] at hs
    push_cast at hs
    field_simp at hs
    nlinarith
  refine ⟨fun i hi => by rw [hx i hi, hVval]; ring, fun j hj => by rw
    [hy j hj, hy0, hVval]; ring⟩

/-- At a corridor state, rows `2` to `M - 1` meet only the staircase, so
    if every one of them
    pays `V`, the band carries `y j = (M - j) V`. -/
theorem band_pin (hM : 2 ≤ M) {d : ℕ} {y : ℕ → K} {V : K}
    (hr : ∀ i, 2 ≤ i → i < M → rowPay M (range (M + d)) y i = V) :
```

```
    ∀ j, 2 ≤ j → j < M → y j = ((M : K) - j) * V := by
  have hrow : ∀ i, 2 ≤ i → i < M →
      (if i + 1 < M then y i - y (i + 1) else y i) = V := fun i h2 hi =>
    by
    have h := hr i h2 hi
    rw [rowPay_corr, rowPay_stair le_rfl, if_pos hi,
      sum_eq_zero fun t _ => by rw [A_high h2, zero_mul], add_zero] at h
    split_ifs with h1
    · rw [if_pos h1] at h; exact h
    · rw [if_neg h1, sub_zero] at h; exact h
  -- downward from `y (M - 1) = V`
  have key : ∀ e, 2 ≤ M - 1 - e → y (M - 1 - e) = ((e : K) + 1) * V :=
    by
    intro e
    induction e with
    | zero =>
      intro h2
      have h := hrow (M - 1) (by omega) (by omega)
      rw [if_neg (by omega)] at h
      simpa using h
    | succ e ih =>
      intro h2
      have h := hrow (M - 1 - (e + 1)) h2 (by omega)
      rw [if_pos (by omega), show M - 1 - (e + 1) + 1 = M - 1 - e by
    omega,
        ih (by omega)] at h
      push_cast
      linarith
  intro j h2 hjM
  have h := key (M - 1 - j) (by omega)
  rw [show M - 1 - (M - 1 - j) = j by omega, Nat.cast_sub (by omega),
    Nat.cast_sub (by omega)] at h
  rw [h]
  push_cast
  ring

/-- At `C_{M+1}`, with `V = 1 / (M (8 M + 1))`, every equilibrium is `x
    i = (15 M + 3 + 2 i) V / 2`,
    and `y` is `0` on column `0`, `(5 M - 2) V / 2` on column `1`, `(M -
    j) V` on the band, and
    `15 M^2 V / 2` on column `M`: the certificate of `cert_handoff`,
    normalized. -/
theorem pin_H (hM : 2 ≤ M) {x y : ℕ → K} (h : IsEq M (range M) (range
    (M + 1)) x y) :
    (∀ i < M, x i = (15 * (M : K) + 3 + 2 * i) * (1 / ((M : K) * (8 * M +
     1))) / 2) ∧
      y 0 = 0 ∧ y 1 = (5 * (M : K) - 2) * (1 / ((M : K) * (8 * M + 1)))
    / 2 ∧
      (∀ j, 2 ≤ j → j < M → y j = ((M : K) - j) * (1 / ((M : K) * (8 *
    M + 1)))) ∧
      y M = 15 * (M : K) ^ 2 * (1 / ((M : K) * (8 * M + 1))) / 2 := by
  obtain ⟨hge, h1, h2, hband⟩ := (cert_handoff (K := K) hM).2 x y h
  obtain ⟨V, hV_def⟩ : ∃ V : K, V = 1 / ((M : K) * (8 * M + 1)) := ⟨_,
    rfl⟩
  rw [← hV_def] at hge h1 h2 hband ⊢
  have hM' : (2 : K) ≤ M := by exact_mod_cast hM
  have hV0 : 0 < V := by rw [hV_def]; positivity
  -- rows `0` and `1` from the tight columns `1` and `M`
  have h15 : 15 * (M : K) * pay M (x 0) (x 1) 4 = 5 * x 0 - 3 * x 1 := by
    have : (M : K) ≠ 0 := by exact_mod_cast (show M ≠ 0 by omega)
    rw [pay, s4, s5]; field_simp; ring
  rw [h2] at h15
  have hx0 : x 0 = (15 * (M : K) + 3) * V / 2 := by linarith
  -- the band telescopes
  have hxi : ∀ i, 1 ≤ i → i < M → x i = x 1 + ((i : K) - 1) * V := by
```

```
    intro i
    induction i with
    | zero => intro h; omega
    | succ i ih =>
      intro _ hi
      rcases Nat.eq_zero_or_pos i with rfl | hi0
      · simp
      · have hb := hband (i + 1) (by omega) hi
        rw [colPay_succ hi, if_pos hi, if_pos (by omega)] at hb
        rw [show x (i + 1) = x i + V by linarith, ih hi0 (by omega)]
        push_cast
        ring
  have hx : ∀ i < M, x i = (15 * (M : K) + 3 + 2 * i) * V / 2 := fun i
    hi => by
    rcases Nat.eq_zero_or_pos i with rfl | hi0
    · rw [hx0]; simp
    · rw [hxi i hi0 hi]; linarith
  -- complementary slackness at this very equilibrium
  have hw : ∀ i ∈ range M, rowPay M (range (M + 1)) y i ≤ V := fun i hi =
    > by
    have := h.2.2 i hi 1 (mem_range.2 (by omega))
    rwa [colPay_succ (show 0 + 1 < M by omega), if_pos (by omega),
    if_pos (by omega), h1] at this
  obtain ⟨-, hsupp, hrows⟩ := cs h.1 h.2.1 (fun j hj => hge j
    (mem_range.1 hj)) hw h
  have hrow : ∀ i < M, rowPay M (range (M + 1)) y i = V := fun i hi =>
    hrows i (mem_range.2 hi) (by rw [hx i hi]; positivity)
  -- column '0' pays more than 'V', so it carries no weight
  have hy0 : y 0 = 0 := by
    refine le_antisymm (not_lt.1 fun hpos => ?_) (h.2.1.1 0 (mem_range.2
    (by omega)))
    have := hsupp 0 (mem_range.2 (by omega)) hpos
    rw [colPay_zero (by omega) (by omega), hx 0 (by omega)] at this
    push_cast at this
    nlinarith
  have hyb := band_pin (d := 1) hM fun i h2 hi => hrow i hi
  -- rows '0' and '1' fix the weights of the two tight columns
  have hr0 := hrow 0 (by omega)
  rw [rowPay_corr, rowPay_stair le_rfl, sum_range_one, A_corr (by
    norm_num), if_pos (by omega),
    if_pos (by omega), hy0, show 0 + 0 + 4 = 4 from rfl, s4] at hr0
  have hr1 := hrow 1 (by omega)
  rw [rowPay_corr, rowPay_stair le_rfl, sum_range_one, A_corr le_rfl,
    if_pos (by omega),
    show 1 + 0 + 4 = 5 from rfl, s5] at hr1
  have hy2 : (if 1 + 1 < M then y (1 + 1) else 0) = ((M : K) - 2) * V :=
    by
    split_ifs with h3
    · rw [hyb 2 le_rfl h3]; push_cast; ring
    · have : (M : K) = 2 := by exact_mod_cast (show M = 2 by omega)
      rw [this]; ring
  rw [hy2] at hr1
  simp only [Nat.zero_add, Nat.add_zero, zero_sub] at hr0 hr1
  have hM0 : (M : K) ≠ 0 := by positivity
  field_simp at hr0 hr1
  have hyM : y M = 15 * (M : K) ^ 2 * V / 2 := by linarith
  refine ⟨hx, hy0, mul_left_cancel₀ hM0 ?_, hyb, hyM⟩
  rw [hyM] at hr0
  linarith

set_option maxHeartbeats 800000 in
/-- At `C_{M+e+2}`, with `K = e + 5`, `p = F(K - 1)`, `q = F(K)` and `V =
    1 / corrMass M e`, every
    equilibrium is the certificate of `cert_corridor`, normalized: `x 0 =
    M p q (p + 2 q) V` and
```

```
    `x i = (M q (p + q)^2 + i - 1) V` for `i ≥ 1`, while `y` vanishes on
    columns `0` and `1` and
    below the frontier, is `(M - j) V` on the band, and on the frontier
    solves a `2 × 2` system
    with the same matrix as `x`, whose determinant is Cassini's. -/
theorem pin_C (hM : 2 ≤ M) (e : ℕ) {x y : ℕ → K} (h : IsEq M (range M)
    (range (M + e + 2)) x y)
    {p q V : K} (hp : p = fib (e + 4)) (hq : q = fib (e + 5)) (hV : V =
    1 / corrMass M e) :
    x 0 = M * p * q * (p + 2 * q) * V ∧
      (∀ i, 1 ≤ i → i < M → x i = (M * q * (p + q) ^ 2 + ((i : K) -
    1)) * V) ∧
      y 0 = 0 ∧ y 1 = 0 ∧ (∀ j, 2 ≤ j → j < M → y j = ((M : K) - j) *
    V) ∧
      (∀ t < e, y (M + t) = 0) ∧
      y (M + e) = M * p * q * (q + (M - 1) * (p + q)) * V ∧
      y (M + e + 1) = M * q * (p + q) * ((M - 1) * q + p) * V := by
  obtain ⟨hge, h4, h5, hband⟩ := (cert_corridor (K := K) hM e).2 x y h
  rw [← hV] at hge h4 h5 hband
  have hM0 : 0 < M := by omega
  have hM2 : (2 : K) ≤ M := by exact_mod_cast hM
  have hV0 : 0 < V := by rw [hV]; exact one_div_pos.2 (corrMass_pos hM e)
  -- the corridor entries at the frontier, and Cassini
  obtain ⟨σ, hσ_def⟩ : ∃ σ : K, σ = (-1) ^ (e + 4) := ⟨_, rfl⟩
  have hp3 : 3 ≤ p := by rw [hp]; exact_mod_cast (fib_mono (by omega : 4
    ≤ e + 4))
  have hq5 : 5 ≤ q := by rw [hq]; exact_mod_cast (fib_mono (by omega : 5
    ≤ e + 5))
  have f6 : (fib (e + 6) : K) = p + q := by rw [hp, hq]; exact_mod_cast
    fib_add_two
  have hcas : p * (p + q) - q ^ 2 = -σ := by
    have := cassini (K := K) (e + 4)
    rw [show e + 4 + 2 = e + 6 from rfl, show e + 4 + 1 = e + 5 from
    rfl, f6, ← hp, ← hq,
      pow_succ (-1 : K), ← hσ_def] at this
    linarith
  have hσ : σ ^ 2 = 1 := by rw [hσ_def, ← pow_mul]; exact
    Even.neg_one_pow ⟨e + 4, by ring⟩
  have hs4 : M * p * s M (e + 4) = σ := by rw [hp, hσ_def]; exact
    fib_mul_s hM0 (by omega)
  have hs5 : M * q * s M (e + 5) = -σ := by
    rw [hq, hσ_def, fib_mul_s hM0 (by omega), show e + 5 = e + 4 + 1
    from rfl, pow_succ]; ring
  have hs6 : M * (p + q) * s M (e + 6) = σ := by
    rw [← f6, hσ_def, fib_mul_s hM0 (by omega), show e + 6 = e + 4 + 1 +
     1 from rfl, pow_succ,
      pow_succ]
    ring
  -- the two frontier columns pay `V`, which fixes `x 0` and `x 1`
  have e4 : x 0 * s M (e + 4) + x 1 * s M (e + 5) = V := h4
  have e5 : x 0 * s M (e + 5) + x 1 * s M (e + 6) = V := h5
  have E1 : σ * (x 0 * q - x 1 * p) = M * p * q * V := by
    linear_combination (M * p * q) * e4 - (x 0 * q) * hs4 - (x 1 * p) *
    hs5
  have E2 : σ * (x 1 * q - x 0 * (p + q)) = M * q * (p + q) * V := by
    linear_combination (M * q * (p + q)) * e5 - (x 0 * (p + q)) * hs5 -
    (x 1 * q) * hs6
  have hx0 : x 0 = M * p * q * (p + 2 * q) * V := by
    linear_combination q * E1 + p * E2 - x 0 * hσ + σ * x 0 * hcas
  have hx1 : x 1 = M * q * (p + q) ^ 2 * V := by
    linear_combination (p + q) * E1 + q * E2 - x 1 * hσ + σ * x 1 * hcas
  -- the band telescopes
  have hxi : ∀ i, 1 ≤ i → i < M → x i = x 1 + ((i : K) - 1) * V := by
    intro i
    induction i with
```

```
    | zero => intro h; omega
    | succ i ih =>
      intro _ hi
      rcases Nat.eq_zero_or_pos i with rfl | hi0
      · simp
      · have hb := hband (i + 1) (by omega) hi
        rw [colPay_succ hi, if_pos hi, if_pos (by omega)] at hb
        rw [show x (i + 1) = x i + V by linarith, ih hi0 (by omega)]
        push_cast
        ring
  have hx : ∀ i, 1 ≤ i → i < M → x i = (M * q * (p + q) ^ 2 + ((i : K)
    - 1)) * V :=
    fun i h1 hi => by rw [hxi i h1 hi, hx1]; ring
  -- complementary slackness at this very equilibrium
  have hcol : ∀ t, colPay M (range M) x (M + t) = pay M (x 0) (x 1) (t +
    4) :=
    fun t => colPay_add hM x t
  have hw : ∀ i ∈ range M, rowPay M (range (M + e + 2)) y i ≤ V := fun i
    hi => by
    have := h.2.2 i hi (M + e) (mem_range.2 (by omega))
    rwa [hcol e, h4] at this
  obtain ⟨-, hsupp, hrows⟩ := cs h.1 h.2.1 (fun j hj => hge j
    (mem_range.1 hj)) hw h
  have hzero : ∀ j < M + e + 2, V < colPay M (range M) x j → y j = 0 :=
    fun j hj hlt =>
    le_antisymm (not_lt.1 fun hpos => by
      rw [hsupp j (mem_range.2 hj) hpos] at hlt; exact lt_irrefl _ hlt)
      (h.2.1.1 j (mem_range.2 hj))
  have hp0 : (0 : K) < p := by linarith
  have hq0 : (0 : K) < q := by linarith
  have hMp : (6 : K) ≤ M * p := by
    have := mul_le_mul hM2 hp3 (by norm_num) (by linarith : (0 : K) ≤
    M); linarith
  have hMpq : (30 : K) ≤ M * p * q := by
    have := mul_le_mul hMp hq5 (by norm_num) (by linarith : (0 : K) ≤ M *
     p); linarith
  have hc0 : (390 : K) ≤ M * p * q * (p + 2 * q) := by
    have := mul_le_mul hMpq (show (13 : K) ≤ p + 2 * q by linarith) (by
    norm_num)
      (by linarith : (0 : K) ≤ M * p * q)
    linarith
  have hc1 : (250 : K) ≤ M * q ^ 3 := by
    have hq3 : (125 : K) ≤ q ^ 3 := by
      have := pow_le_pow_left₀ (by norm_num : (0 : K) ≤ 5) hq5 3;
    norm_num at this; linarith
    have := mul_le_mul hM2 hq3 (by norm_num) (by linarith : (0 : K) ≤
    M); linarith
  have hy0 : y 0 = 0 := hzero 0 (by omega) (by
    rw [colPay_zero hM0 hM0, hx0]
    have := mul_lt_mul_of_pos_right (show (1 : K) < M * p * q * (p + 2 *
    q) by linarith) hV0
    linarith)
  have hy1 : y 1 = 0 := hzero 1 (by omega) (by
    rw [colPay_succ (show 0 + 1 < M by omega), if_pos (by omega), if_pos
    (by omega),
      show 0 + 1 = 1 from rfl, hx 1 le_rfl (by omega), hx0]
    have := mul_lt_mul_of_pos_right (show (1 : K) < M * q ^ 3 by
    linarith) hV0
    push_cast
    linarith)
  have hylow : ∀ t < e, y (M + t) = 0 := fun t ht => hzero (M + t) (by
    omega) (by
    rw [hcol t]; exact pay_below hM0 hV0 (by omega) h4 h5 (by omega) (by
    omega))
```

```
  have hrow : ∀ i < M, rowPay M (range (M + e + 2)) y i = V := fun i hi =
    > by
    refine hrows i (mem_range.2 hi) ?_
    rcases Nat.eq_zero_or_pos i with rfl | hi0
    · rw [hx0]; exact mul_pos (by linarith) hV0
    · rw [hx i hi0 hi]
      have : (1 : K) ≤ i := by exact_mod_cast hi0
      have : 0 < (M : K) * q * (p + q) ^ 2 :=
        mul_pos (mul_pos (by linarith) hq0) (pow_pos (by linarith) 2)
      exact mul_pos (by linarith) hV0
  have hyb := band_pin (d := e + 2) hM fun i h2 hi => hrow i hi
  -- rows `0` and `1` fix the frontier weights
  have hsplit : ∀ i, i ≤ 1 → rowPay M (range (M + e + 2)) y i
      = (if i < M then y i else 0) - (if i + 1 < M then y (i + 1) else 0)
        + (s M (i + e + 4) * y (M + e) + s M (i + (e + 1) + 4) * y (M +
    e + 1)) := fun i hi => by
    rw [show M + e + 2 = M + (e + 1 + 1) from rfl, rowPay_corr,
    sum_range_succ, sum_range_succ,
      sum_eq_zero fun t ht => by rw [hylow t (mem_range.1 ht),
    mul_zero], zero_add,
      rowPay_stair le_rfl, A_corr hi, A_corr hi, show M + (e + 1) = M +
    e + 1 from rfl]
  have r0 := hrow 0 (by omega)
  rw [hsplit 0 (by omega), if_pos (by omega), if_pos (by omega), hy0,
    show 0 + 1 = 1 from rfl, hy1,
    show 0 + e + 4 = e + 4 by omega, show 0 + (e + 1) + 4 = e + 5 by
    omega] at r0
  have r1 := hrow 1 (by omega)
  rw [hsplit 1 le_rfl, if_pos (by omega), hy1, show 1 + e + 4 = e + 5 by
    omega,
    show 1 + (e + 1) + 4 = e + 6 by omega] at r1
  have hy2 : (if 1 + 1 < M then y (1 + 1) else 0) = ((M : K) - 2) * V :=
    by
    split_ifs with h3
    · rw [hyb 2 le_rfl h3]; push_cast; ring
    · have : (M : K) = 2 := by exact_mod_cast (show M = 2 by omega)
      rw [this]; ring
  rw [hy2] at r1
  have g0 : s M (e + 4) * y (M + e) + s M (e + 5) * y (M + e + 1) = V :=
    by linarith
  have g1 : s M (e + 5) * y (M + e) + s M (e + 6) * y (M + e + 1) = ((M
    : K) - 1) * V := by
    linarith
  have F1 : σ * (y (M + e) * q - y (M + e + 1) * p) = M * p * q * V := by
    linear_combination (M * p * q) * g0 - (y (M + e) * q) * hs4 - (y (M +
     e + 1) * p) * hs5
  have F2 : σ * (y (M + e + 1) * q - y (M + e) * (p + q)) = M * q * (p +
    q) * ((M - 1) * V) := by
    linear_combination (M * q * (p + q)) * g1 - (y (M + e) * (p + q)) *
    hs5 -
      (y (M + e + 1) * q) * hs6
  refine ⟨hx0, hx, hy0, hy1, hyb, hylow, ?_, ?_⟩
  · linear_combination q * F1 + p * F2 - y (M + e) * hσ + σ * y (M + e) *
    hcas
  · linear_combination (p + q) * F1 + q * F2 - y (M + e + 1) * hσ + σ * y
    (M + e + 1) * hcas

/-- The states of the run, family by family: `S_{k+1}` at `2 k`,
    `T_{k+1}` at `2 k + 1`, and
    `([M], [M + d])` once every row is present. -/
theorem run_even (h : 2 * k + 2 < 2 * M) : run M N (2 * k) = (range (k +
    1), range (k + 1)) := by
  simp only [run, if_pos h, show 2 * k / 2 = k by omega, show (2 * k +
    1) / 2 = k by omega]
```

```
theorem run_odd (h : 2 * k + 1 + 2 < 2 * M) :
    run M N (2 * k + 1) = (range (k + 1), range (k + 1 + 1)) := by
  simp only [run, if_pos h, show (2 * k + 1) / 2 = k by omega,
    show (2 * k + 1 + 1) / 2 = k + 1 by omega]

theorem run_corr (hMN : M ≤ N) (h : ¬ t + 2 < 2 * M) :
    ∃ d, run M N t = (range M, range (M + d)) ∧ M + d = min (t + 2 - M)
    N := by
  refine ⟨min (t + 2 - M) N - M, ?_, by omega⟩
  simp only [run, if_neg h]
  congr 2
  omega

/-- **Every subgame along the run has exactly one equilibrium**, the
    certificate: `pin_S`,
    `pin_T`, `pin_H`, and `pin_C` describe every equilibrium explicitly.
    So the equilibrium
    selection scheme has nothing to select at any state of the run. -/
theorem eq_unique (hM : 2 ≤ M) (hMN : M ≤ N) (t : ℕ) {x y x' y' : ℕ →
    K}
    (h : IsEq M (run M N t).1 (run M N t).2 x y) (h' : IsEq M (run M N
    t).1 (run M N t).2 x' y') :
    (∀ i ∈ (run M N t).1, x i = x' i) ∧ ∀ j ∈ (run M N t).2, y j = y' j
    := by
  by_cases ht : t + 2 < 2 * M
  · rcases Nat.even_or_odd' t with ⟨k, rfl | rfl⟩
    · rw [run_even ht] at h h' ⊢
      obtain ⟨hx, hy⟩ := pin_S (by omega) (by omega) h
      obtain ⟨hx', hy'⟩ := pin_S (by omega) (by omega) h'
      exact ⟨fun i hi => by rw [hx i (mem_range.1 hi), hx' i
    (mem_range.1 hi)],
        fun j hj => by rw [hy j (mem_range.1 hj), hy' j (mem_range.1
    hj)]⟩
    · rw [run_odd ht] at h h' ⊢
      obtain ⟨hx, hy⟩ := pin_T (by omega) (by omega) h
      obtain ⟨hx', hy'⟩ := pin_T (by omega) (by omega) h'
      exact ⟨fun i hi => by rw [hx i (mem_range.1 hi), hx' i
    (mem_range.1 hi)],
        fun j hj => by rw [hy j (mem_range.1 hj), hy' j (mem_range.1
    hj)]⟩
  obtain ⟨d, hrun, -⟩ := run_corr (N := N) hMN ht
  rw [hrun] at h h' ⊢
  rcases d with _ | _ | e
  · obtain ⟨hx, hy⟩ := pin_S (by omega) le_rfl h
    obtain ⟨hx', hy'⟩ := pin_S (by omega) le_rfl h'
    exact ⟨fun i hi => by rw [hx i (mem_range.1 hi), hx' i (mem_range.1
    hi)],
      fun j hj => by rw [hy j (mem_range.1 hj), hy' j (mem_range.1 hj)]⟩
  · obtain ⟨hx, h0, h1, hb, hM'⟩ := pin_H hM h
    obtain ⟨hx', h0', h1', hb', hM''⟩ := pin_H hM h'
    refine ⟨fun i hi => by rw [hx i (mem_range.1 hi), hx' i (mem_range.1
    hi)], fun j hj => ?_⟩
    rw [mem_range] at hj
    rcases lt_or_ge j 2 with hj2 | hj2
    · interval_cases j
      · rw [h0, h0']
      · rw [h1, h1']
    · rcases lt_or_ge j M with hjM | hjM
      · rw [hb j hj2 hjM, hb' j hj2 hjM]
      · rw [show j = M by omega, hM', hM'']
  · obtain ⟨hx0, hx, h0, h1, hb, hlow, hf0, hf1⟩ := pin_C hM e h rfl rfl
    rfl
    obtain ⟨hx0', hx', h0', h1', hb', hlow', hf0', hf1'⟩ := pin_C hM e h'
    rfl rfl rfl
    refine ⟨fun i hi => ?_, fun j hj => ?_⟩
```

```
    · rcases Nat.eq_zero_or_pos i with rfl | hi0
      · rw [hx0, hx0']
      · rw [hx i hi0 (mem_range.1 hi), hx' i hi0 (mem_range.1 hi)]
    · rw [mem_range, show M + (e + 1 + 1) = M + e + 2 from rfl] at hj
      rcases lt_or_ge j 2 with hj2 | hj2
      · interval_cases j
        · rw [h0, h0']
        · rw [h1, h1']
      rcases lt_or_ge j M with hjM | hjM
      · rw [hb j hj2 hjM, hb' j hj2 hjM]
      obtain ⟨u, rfl⟩ : ∃ u, j = M + u := ⟨j - M, by omega⟩
      rcases lt_or_ge u e with hu | hu
      · rw [hlow u hu, hlow' u hu]
      rcases eq_or_lt_of_le hu with rfl | hu'
      · rw [hf0, hf0']
      · rw [show M + u = M + e + 1 by omega, hf1, hf1']

/-! ## Approximate oracles

  With `δ`-equilibria and approximate best responses, the run is the
    same as long as `δ` and the
  column player's `εC` stay below a fixed fraction of `min (|v_t|, m_t)`
    (`margin`) at every state
  before the last, and the row player's `εR` below a fixed fraction of
    `2 / (M (M - 1))`
  (`double_oracle_accuracy`). In particular, it survives all small
    enough tolerances
  (`double_oracle_approx`). Some run stops early once the tolerance of
    the player that moves at
  some state reaches the duality gap there, once `εR` reaches the value
    at `S_{M-1}`, or once `δ`
  and `εR` reach a joint threshold at `T_{M-1}`
    (`double_oracle_accuracy_sharp`). So the bound on
  `εR` is of the right order. -/

/-- `(x, y)` is a `δ`-equilibrium of the subgame on `R × C`. -/
def IsEqApprox (M : ℕ) (R C : Finset ℕ) (δ : K) (x y : ℕ → K) : Prop :=
  IsMix R x ∧ IsMix C y ∧ ∀ i ∈ R, ∀ j ∈ C, rowPay M C y i ≤ colPay M R
    x j + δ

/-- `q` is an approximate successor of the state `p`: to a
    `δ`-equilibrium of the subgame, add a
    row within `εR` of best and a column within `εC` of best. -/
def StepApprox (M N : ℕ) (δ εR εC : K) (p q : Finset ℕ × Finset ℕ) :
    Prop :=
  ∃ x y i j, IsEqApprox M p.1 p.2 δ x y ∧
    i < M ∧ (∀ i' < M, rowPay M p.2 y i' ≤ rowPay M p.2 y i + εR) ∧
    j < N ∧ (∀ j' < N, colPay M p.1 x j ≤ colPay M p.1 x j' + εC) ∧
    q = (insert i p.1, insert j p.2)

/-- For `δ, ε ≥ 0`, every successor is an approximate successor. -/
theorem Step.approx {p q : Finset ℕ × Finset ℕ} (h : Step K M N p q) {δ ε
    R εC : K} (hδ : 0 ≤ δ)
    (hεR : 0 ≤ εR) (hεC : 0 ≤ εC) : StepApprox M N δ εR εC p q := by
  obtain ⟨x, y, i, j, ⟨hx, hy, he⟩, hi, hi', hj, hj', hq⟩ := h
  exact ⟨x, y, i, j, ⟨hx, hy, fun a ha b hb => by linarith [he a ha b
    hb]⟩, hi,
    fun a ha => by linarith [hi' a ha], hj, fun b hb => by linarith [hj'
    b hb], hq⟩

/-! ## An explicit guarantee

  The run survives `δ`-equilibria and approximate best responses as soon
    as `δ` and `εC` stay
```

```
  below a fixed fraction of `min (|v_t|, m_t)` (Table 1) at every state
    before the last, and `εR`
  below a fixed fraction of the value at `S_{M-1}`.
    `double_oracle_accuracy` fixes the fraction. -/

/-- For `δ ≥ 0`, an equilibrium is a `δ`-equilibrium. -/
theorem IsEq.approx {R C : Finset ℕ} {x y : ℕ → K} (h : IsEq M R C x y)
    {δ : K} (hδ : 0 ≤ δ) :
    IsEqApprox M R C δ x y :=
  ⟨h.1, h.2.1, fun i hi j hj => by linarith [h.2.2 i hi j hj]⟩

/-- Against any mixed strategy of the column player, some row of the
    subgame pays at least the
    value certified by an equilibrium strategy `x`. -/
theorem exists_row_ge {R C : Finset ℕ} {x y' : ℕ → K} {v : K} (hx :
    IsMix R x) (hy' : IsMix C y')
    (hu : ∀ j ∈ C, v ≤ colPay M R x j) (hR : R.Nonempty) : ∃ i ∈ R, v ≤
    rowPay M C y' i := by
  obtain ⟨i, hi, hi'⟩ := exists_max_image R (rowPay M C y') hR
  exact ⟨i, hi, ((mix_ge hy' hu).trans_eq (sum_swap x y').symm).trans
    (mix_le hx fun a ha => hi' a ha)⟩

/-- Symmetrically, some column of the subgame pays at most the value. -/
theorem exists_col_le {R C : Finset ℕ} {y x' : ℕ → K} {v : K} (hy :
    IsMix C y) (hx' : IsMix R x')
    (hw : ∀ i ∈ R, rowPay M C y i ≤ v) (hC : C.Nonempty) : ∃ j ∈ C,
    colPay M R x' j ≤ v := by
  obtain ⟨j, hj, hj'⟩ := exists_min_image C (colPay M R x') hC
  exact ⟨j, hj, ((mix_ge hy fun b hb => hj' b hb).trans_eq (sum_swap x'
    y).symm).trans
    (mix_le hx' hw)⟩

/-- Against a `δ`-equilibrium, every column of the subgame pays at least
    `v - δ` and every row
    pays at most `v + δ`. -/
theorem approx_bounds {R C : Finset ℕ} {x y x' y' : ℕ → K} {v δ : K}
    (hx : IsMix R x)
    (hy : IsMix C y) (hu : ∀ j ∈ C, v ≤ colPay M R x j) (hw : ∀ i ∈ R,
    rowPay M C y i ≤ v)
    (h : IsEqApprox M R C δ x' y') :
    (∀ j ∈ C, v - δ ≤ colPay M R x' j) ∧ (∀ i ∈ R, rowPay M C y' i ≤ v +
    δ) := by
  constructor
  · intro j hj
    have h1 : v ≤ Σ i ∈ R, x i * rowPay M C y' i :=
      (mix_ge h.2.1 hu).trans_eq (sum_swap x y').symm
    have h2 : Σ i ∈ R, x i * rowPay M C y' i ≤ colPay M R x' j + δ :=
      mix_le hx fun i hi => h.2.2 i hi j hj
    linarith
  · intro i hi
    have h1 : rowPay M C y' i ≤ Σ j ∈ C, y j * (colPay M R x' j + δ) :=
      mix_ge hy fun j hj => h.2.2 i hi j hj
    have h2 : Σ j ∈ C, y j * (colPay M R x' j + δ) = (Σ j ∈ C, y j *
    colPay M R x' j) + δ := by
      simp [mul_add, sum_add_distrib, ← sum_mul, hy.2]
    have h3 : Σ j ∈ C, y j * colPay M R x' j ≤ v :=
      (sum_swap x' y).symm.trans_le (mix_le h.1 hw)
    linarith

/-- Approximate complementary slackness: against a `δ`-equilibrium, a
    column of weight `y j` in
    an equilibrium pays at most `v + δ / y j`. -/
theorem approx_cs {R C : Finset ℕ} {x y x' y' : ℕ → K} {v δ : K} (hx :
    IsMix R x) (hy : IsMix C y)
```

```
    (hu : ∀ j ∈ C, v ≤ colPay M R x j) (hw : ∀ i ∈ R, rowPay M C y i ≤
    v)
    (h : IsEqApprox M R C δ x' y') {j : ℕ} (hj : j ∈ C) (hδ : 0 ≤ δ) :
    y j * (colPay M R x' j - v) ≤ δ := by
  have hlow := (approx_bounds hx hy hu hw h).1
  have hsum : Σ l ∈ C, y l * colPay M R x' l ≤ v := (sum_swap x'
    y).symm.trans_le (mix_le h.1 hw)
  have hall : Σ l ∈ C, y l * (colPay M R x' l - v) ≤ 0 := by
    simp only [mul_sub, sum_sub_distrib, ← sum_mul, hy.2, one_mul]
    linarith
  have herase : -δ ≤ Σ l ∈ C.erase j, y l * (colPay M R x' l - v) := by
    have h1 : ∀ l ∈ C.erase j, -(δ * y l) ≤ y l * (colPay M R x' l - v)
    := fun l hl => by
      have := hlow l (mem_of_mem_erase hl)
      nlinarith [hy.1 l (mem_of_mem_erase hl)]
    have h2 : Σ l ∈ C.erase j, y l ≤ 1 := by
      rw [← hy.2]
      exact sum_le_sum_of_subset_of_nonneg (erase_subset _ _) fun l hl _
    => hy.1 l hl
    have := sum_le_sum h1
    rw [sum_neg_distrib, ← mul_sum] at this
    nlinarith [hy.1 j hj]
  rw [← add_sum_erase C _ hj] at hall
  linarith

/-- A unique approximate successor, from the approximate best responses
    against every
`δ`-equilibrium. -/
theorem stepApprox_iff (hM : 0 < M) (hN : 0 < N) {R C R' C' : Finset ℕ}
    {δ εR εC : K}
    (hεR : 0 ≤ εR) (hεC : 0 ≤ εC)
    (hE : ∃ x y : ℕ → K, IsEqApprox M R C δ x y)
    (hR : ∀ x y : ℕ → K, IsEqApprox M R C δ x y → ∀ i,
      (∀ i' < M, rowPay M C y i' ≤ rowPay M C y i + εR) → i < M →
    insert i R = R')
    (hC : ∀ x y : ℕ → K, IsEqApprox M R C δ x y → ∀ j,
      (∀ j' < N, colPay M R x j ≤ colPay M R x j' + εC) → j < N →
    insert j C = C')
    (q : Finset ℕ × Finset ℕ) : StepApprox M N δ εR εC (R, C) q ↔ q =
    (R', C') := by
  refine ⟨fun ⟨x, y, i, j, h, hi, hi', hj, hj', hq⟩ => ?_, fun hq => ?_⟩
  · rw [hq, hR x y h i hi' hi, hC x y h j hj' hj]
  obtain ⟨x, y, h⟩ := hE
  obtain ⟨i, hi, hi'⟩ := exists_max_image (range M) (rowPay M C y)
    (nonempty_range_iff.2 hM.ne')
  obtain ⟨j, hj, hj'⟩ := exists_min_image (range N) (colPay M R x)
    (nonempty_range_iff.2 hN.ne')
  simp only [mem_range] at hi hj hi' hj'
  have hi'' : ∀ i' < M, rowPay M C y i' ≤ rowPay M C y i + εR :=
    fun i' h' => by linarith [hi' i' h']
  have hj'' : ∀ j' < N, colPay M R x j ≤ colPay M R x j' + εC :=
    fun j' h' => by linarith [hj' j' h']
  exact ⟨x, y, i, j, h, hi, hi'', hj, hj'', by rw [hq, hR x y h i hi''
    hi, hC x y h j hj'' hj]⟩

/-- At `S_k`, an equilibrium certifies the value `V = 2 / (k (k + 1))`,
    and its column weights
    telescope to `y j = (k - j) V`. -/
theorem value_S (hk : 0 < k) (hkM : k ≤ M) :
    ∃ x y : ℕ → K, IsEq M (range k) (range k) x y ∧
      (∀ j ∈ range k, 2 / (k * (k + 1)) ≤ colPay M (range k) x j) ∧
      (∀ i ∈ range k, rowPay M (range k) y i ≤ 2 / (k * (k + 1))) ∧
      (∀ j < k, y j = ((k : K) - j) * (2 / (k * (k + 1)))) := by
  obtain ⟨x, y, hxy⟩ := (cert_S (K := K) hk hkM).1
  obtain ⟨V, hV, hx, hc, hr⟩ := (cert_S hk hkM).2 x y hxy
```

```
  have hk' : (0 : K) < k := by exact_mod_cast hk
  have hsum : Σ i ∈ range k, ((i : K) + 1) * V = 1 :=
    (sum_congr rfl fun i hi => (hx i (mem_range.1 hi)).symm).trans
    hxy.1.2
  rw [← sum_mul, sum_range_cast_succ] at hsum
  have hVval : V = 2 / (k * (k + 1)) := by
    field_simp at hsum ⊢; linarith
  -- `y (k - 1) = V` and `y j - y (j + 1) = V`, so `y (k - 1 - d) = (d +
    1) V`
  have hy : ∀ d, d < k → y (k - 1 - d) = ((d : K) + 1) * V := by
    intro d
    induction d with
    | zero =>
      intro _
      have h1 := hr (k - 1) (by omega)
      rw [rowPay_stair hkM, if_pos (by omega), if_neg (by omega)] at h1
      simpa using h1
    | succ d ih =>
      intro hd
      have h1 := hr (k - 1 - (d + 1)) (by omega)
      rw [rowPay_stair hkM, if_pos (by omega), if_pos (by omega),
        show k - 1 - (d + 1) + 1 = k - 1 - d by omega, ih (by omega)] at
    h1
      push_cast
      linarith
  refine ⟨x, y, hxy, fun j hj => ?_, fun i hi => ?_, fun j hj => ?_⟩
  · rw [hc j (mem_range.1 hj), hVval]
  · rw [hr i (mem_range.1 hi), hVval]
  · have h1 := hy (k - 1 - j) (by omega)
    rw [show k - 1 - (k - 1 - j) = j by omega, Nat.sub_sub, Nat.cast_sub
    (by omega)] at h1
    rw [h1, hVval]
    push_cast
    ring

/-- At `T_k`, an equilibrium certifies the value `-2 / (k (k + 1))`. -/
theorem value_T (hk : 0 < k) (hkM : k < M) :
    ∃ x y : ℕ → K, IsEq M (range k) (range (k + 1)) x y ∧
      (∀ j ∈ range (k + 1), -(2 / (k * (k + 1))) ≤ colPay M (range k) x
    j) ∧
      (∀ i ∈ range k, rowPay M (range (k + 1)) y i ≤ -(2 / (k * (k +
    1)))) := by
  obtain ⟨x, y, hxy⟩ := (cert_T (K := K) hk hkM).1
  obtain ⟨V, hV, hx, hr⟩ := (cert_T hk hkM).2 x y hxy
  have hk' : (0 : K) < k := by exact_mod_cast hk
  have hsum : Σ i ∈ range k, ((i : K) - k) * V = 1 :=
    (sum_congr rfl fun i hi => (hx i (mem_range.1 hi)).symm).trans
    hxy.1.2
  rw [← sum_mul, sum_range_cast_sub] at hsum
  have hVval : V = -(2 / (k * (k + 1))) := by
    field_simp at hsum ⊢; linarith
  refine ⟨x, y, hxy, fun j hj => ?_, fun i hi => ?_⟩
  · have := hxy.2.2 0 (mem_range.2 hk) j hj
    rwa [hr 0 hk, hVval] at this
  · rw [hr i (mem_range.1 hi), hVval]

/-- Against nonnegative weights on `[k]` whose first two entries sum to
    at most `c`, a corridor
    column pays at least `-c / (3 M)`. -/
theorem corr_ge_mix (hM : 2 ≤ M) (hk : 0 < k) (hj : M ≤ j) {x : ℕ → K}
    {c : K}
    (hx : ∀ i ∈ range k, 0 ≤ x i) (hc : Σ i ∈ range (min k 2), x i ≤ c)
    :
    -c ≤ 3 * (M : K) * colPay M (range k) x j := by
```

```
  obtain ⟨z, hz⟩ : ∃ z : ℕ → K, z = fun i => if i < k then x i else 0 :=
    ⟨_, rfl⟩
  have hcol : colPay M (range k) z j = colPay M (range k) x j :=
    sum_congr rfl fun i hi => by rw [hz]; simp [mem_range.1 hi]
  have hz0 : 0 ≤ z 0 := by
    rw [hz]; simp only [if_pos hk]; exact hx 0 (mem_range.2 hk)
  have hz1 : 0 ≤ z 1 := by
    rw [hz]
    by_cases h1 : 1 < k
    · simp only [if_pos h1]; exact hx 1 (mem_range.2 h1)
    · simp only [if_neg h1]; exact le_rfl
  have hsum : z 0 + z 1 ≤ c := by
    refine le_trans (le_of_eq ?_) hc
    rcases lt_or_ge k 2 with hk2 | hk2
    · obtain rfl : k = 1 := by omega
      rw [hz]; simp
    · rw [show min k 2 = 2 by omega, sum_range_succ, sum_range_one, hz]
      simp [show 0 < k by omega, show 1 < k by omega]
  have := corr_ge (x := z) (by omega) hk hj hz0 hz1
  rw [hcol] at this
  linarith

/-- Staircase right, with approximate oracles: at `S_k`, if `εR` is
    below the value `v`, and
    `εC + b` below `k (v - δ)` where every column after `k` pays at
    least `-b`, the only
    approximate successor is `T_k`. -/
theorem staircase_right_approx (hk : 0 < k) (hkM : k < M) (hkN : k < N)
    {δ εR εC b : K}
    (hδ : 0 ≤ δ) (hεR : 0 ≤ εR) (hεC : 0 ≤ εC) (hc : 0 ≤ b)
    (h1 : εR < 2 / (k * (k + 1)))
    (h2 : εC + b < k * (2 / (k * (k + 1)) - δ))
    (hout : ∀ x' : ℕ → K, IsMix (range k) x' → ∀ j, k < j → j < N →
      -b ≤ colPay M (range k) x' j)
    (q : Finset ℕ × Finset ℕ) :
    StepApprox M N δ εR εC (range k, range k) q ↔ q = (range k, range (k
    + 1)) := by
  obtain ⟨m, rfl⟩ : ∃ m, k = m + 1 := ⟨k - 1, by omega⟩
  obtain ⟨x, y, ⟨hx, hy, -⟩, hu, hw, -⟩ := value_S (K := K) (M := M) hk
    hkM.le
  have hk' : (0 : K) < (m : K) + 1 := by positivity
  have hvd : 0 < 2 / ((m + 1 : ℕ) * ((m + 1 : ℕ) + 1)) - δ := by
    push_cast at h2 ⊢
    nlinarith
  refine stepApprox_iff (by omega) (by omega) hεR hεC
    ⟨x, y, hx, hy, fun i hi j hj => by linarith [hu j hj, hw i hi]⟩ (fun
    x' y' h i hi' hi => ?_)
    (fun x' y' h j hj' hj => ?_) q
  · -- no row outside the subgame is an `εR`-best response
    obtain ⟨i₀, hi₀, hvi₀⟩ := exists_row_ge hx h.2.1 hu
    (nonempty_range_iff.2 (by omega))
    have hi₀' : i₀ < m + 1 := mem_range.1 hi₀
    refine insert_eq_of_mem (mem_range.2 ?_)
    by_contra hik
    have hzero : rowPay M (range (m + 1)) y' i = 0 := by
      rw [rowPay_stair hkM.le]
      simp [show ¬ i < m + 1 by omega, show ¬ i + 1 < m + 1 by omega]
    have := hi' i₀ (by omega)
    rw [hzero] at this
    push_cast at h1 hvi₀
    linarith
  · -- the best-response column is the new one
    obtain ⟨hcol, hrow⟩ := approx_bounds hx hy hu hw h
    push_cast at hcol h1 h2 hvd
```

```
    have hxlow : ∀ i, i < m + 1 → ((i : K) + 1) * (2 / ((m + 1) * (m +
    1 + 1)) - δ) ≤ x' i := by
      intro i
      induction i with
      | zero =>
        intro _
        have h3 := hcol 0 (mem_range.2 (by omega))
        rw [colPay_zero (by omega) (by omega)] at h3
        push_cast; linarith
      | succ i ih =>
        intro hik
        have h3 := hcol (i + 1) (mem_range.2 hik)
        rw [colPay_succ (by omega : i + 1 < M)] at h3
        rw [if_pos hik, if_pos (show i < m + 1 by omega)] at h3
        have h4 := ih (by omega)
        push_cast at h4 ⊢
        linarith
    have hxm : ((m : K) + 1) * (2 / ((m + 1) * (m + 1 + 1)) - δ) ≤ x' m
    := hxlow m (by omega)
    have hpayk : colPay M (range (m + 1)) x' (m + 1) = -x' m := by
      rw [colPay_succ (by omega : m + 1 < M)]
      simp [show m < m + 1 by omega]
    suffices j = m + 1 by rw [this, ← range_add_one]
    by_contra hne
    have hle := hj' (m + 1) (by omega)
    rw [hpayk] at hle
    rcases lt_or_gt_of_ne hne with hlt | hgt
    · have := hcol j (mem_range.2 hlt)
      nlinarith
    · have := hout x' h.1 j (by omega) hj
      nlinarith

/-- Staircase down, with approximate oracles: at `T_k`, with the roles
    of `εR` and `εC`
    exchanged, so that `εC + b` is below `v` and `εR` below `k (v - δ)`,
    the only approximate
    successor is `S_{k+1}`. -/
theorem staircase_down_approx (hk : 0 < k) (hkM : k < M) (hkN : k < N)
    {δ εR εC b : K}
    (hδ : 0 ≤ δ) (hεR : 0 ≤ εR) (hεC : 0 ≤ εC)
    (h1 : εC + b < 2 / (k * (k + 1)))
    (h2 : εR < k * (2 / (k * (k + 1)) - δ))
    (hout : ∀ x' : ℕ → K, IsMix (range k) x' →
      (∀ j ∈ range (k + 1), -(2 / (k * (k + 1))) - δ ≤ colPay M (range
    k) x' j) →
      ∀ j, k < j → j < N → -b ≤ colPay M (range k) x' j)
    (q : Finset ℕ × Finset ℕ) :
    StepApprox M N δ εR εC (range k, range (k + 1)) q ↔ q = (range (k +
    1), range (k + 1)) := by
  obtain ⟨m, rfl⟩ : ∃ m, k = m + 1 := ⟨k - 1, by omega⟩
  obtain ⟨x, y, ⟨hx, hy, -⟩, hu, hw⟩ := value_T (K := K) (M := M) hk hkM
  have hvd : 0 < 2 / ((m + 1 : ℕ) * ((m + 1 : ℕ) + 1)) - δ := by
    push_cast at h2 ⊢
    nlinarith
  refine stepApprox_iff (by omega) (by omega) hεR hεC
    ⟨x, y, hx, hy, fun i hi j hj => by linarith [hu j hj, hw i hi]⟩ (fun
    x' y' h i hi' hi => ?_)
    (fun x' y' h j hj' hj => ?_) q
  · -- the best-response row is the new one
    obtain ⟨hcol, hrow⟩ := approx_bounds hx hy hu hw h
    push_cast at hrow h1 h2 hvd
    have hylow : ∀ i, i < m + 2 → (i : K) * (2 / ((m + 1) * (m + 1 +
    1)) - δ) ≤ y' i := by
      intro i
      induction i with
```

```
      | zero => intro _; simpa using h.2.1.1 0 (mem_range.2 (by omega))
      | succ i ih =>
        intro hik
        have h3 := hrow i (mem_range.2 (by omega))
        rw [rowPay_stair (by omega : m + 1 + 1 ≤ M)] at h3
        rw [if_pos (show i < m + 1 + 1 by omega), if_pos (show i + 1 < m
    + 1 + 1 by omega)] at h3
        have h4 := ih (by omega)
        push_cast at h4 ⊢
        linarith
    have hym : ((m : K) + 1) * (2 / ((m + 1) * (m + 1 + 1)) - δ) ≤ y' (m
    + 1) := by
      have := hylow (m + 1) (by omega); push_cast at this ⊢; linarith
    have hpayk : rowPay M (range (m + 1 + 1)) y' (m + 1) = y' (m + 1) :=
    by
      rw [rowPay_stair (by omega : m + 1 + 1 ≤ M)]
      simp [show m + 1 < m + 1 + 1 by omega]
    suffices i = m + 1 by rw [this, ← range_add_one]
    by_contra hne
    have hge := hi' (m + 1) (by omega)
    rw [hpayk] at hge
    rcases lt_or_gt_of_ne hne with hlt | hgt
    · have := hrow i (mem_range.2 hlt)
      nlinarith
    · have hi0 : rowPay M (range (m + 1 + 1)) y' i = 0 := by
        rw [rowPay_stair (by omega : m + 1 + 1 ≤ M)]
        simp [show ¬ i < m + 1 + 1 by omega, show ¬ i + 1 < m + 1 + 1 by
    omega]
      rw [hi0] at hge
      nlinarith
  · -- no column outside the subgame is an `εC`-best response
    obtain ⟨j₀, hj₀, hvj₀⟩ := exists_col_le hy h.1 hw
    (nonempty_range_iff.2 (by omega))
    have hj₀' : j₀ < m + 1 + 1 := mem_range.1 hj₀
    obtain ⟨hcol, -⟩ := approx_bounds hx hy hu hw h
    refine insert_eq_of_mem (mem_range.2 ?_)
    by_contra hjk
    have hge := hout x' h.1 (fun j' hj' => by linarith [hcol j' hj']) j
    (by omega) hj
    have := hj' j₀ (by omega)
    push_cast at h1 hvj₀ hge
    linarith

/-- Approximate complementary slackness, in the form used at the
    corridor states: against a
    `δ`-equilibrium, a column of weight at least `w > 0` in an
    equilibrium pays at most
    `v + δ / w`. -/
theorem approx_cs_le {R C : Finset ℕ} {x y x' y' : ℕ → K} {v δ w : K}
    (hx : IsMix R x)
    (hy : IsMix C y) (hu : ∀ j ∈ C, v ≤ colPay M R x j) (hw : ∀ i ∈ R,
    rowPay M C y i ≤ v)
    (h : IsEqApprox M R C δ x' y') {j : ℕ} (hj : j ∈ C) (hδ : 0 ≤ δ)
    (hw0 : 0 < w)
    (hwy : w ≤ y j) : colPay M R x' j ≤ v + δ / w := by
  rcases le_or_gt (colPay M R x' j) v with h' | h'
  · linarith [div_nonneg hδ hw0.le]
  · rw [← sub_le_iff_le_add', le_div_iff₀ hw0, mul_comm]
    exact (mul_le_mul_of_nonneg_right hwy (by linarith)).trans
      (approx_cs hx hy hu hw h hj hδ)

/-- The corridor payoff of index `4`, the new column at `C_M`. -/
theorem pay_four (hM : 0 < M) (a b : K) : 15 * (M : K) * pay M a b 4 = 5
    * a - 3 * b := by
  have : (M : K) ≠ 0 := by positivity
```

```
  rw [pay, s4, s5]; field_simp; ring

/-- `g(3)` and `g(4)`, the quantities `alternation` compares at `C_M`. -/
theorem g_three (a b : K) : g a b 3 = 2 * b - 3 * a ∧ g a b 4 = 5 * a -
    3 * b := by
  obtain ⟨f3, f4, f5, -⟩ := fib_vals
  constructor <;> simp [g, f3, f4, f5] <;> ring

/-- If the new corridor column pays at most `-m + 2 δ`, and a later one
    at least `5 / 8` of what it
    pays, as `alternation` gives, then the new column beats the later
    one by more than `εC`. -/
theorem later_beats {P Q m δ ε : K} (hm : 0 < m) (hδm : 64 * δ ≤ m) (hεm
    : 64 * ε ≤ m)
    (hP : P ≤ -m + 2 * δ) (hQ : P * (5 / 8) ≤ Q) : P + ε < Q := by
  linarith

/-- At a corridor state, the row player holds every row, so if the new
    column `M + d` beats every
    other column by more than `εC` against every `δ`-equilibrium, the
    only approximate successor
    adds it. -/
theorem corridor_step {d : ℕ} (hM : 2 ≤ M) (hdN : M + d < N) {δ εR εC :
    K} (hδ : 0 ≤ δ)
    (hεR : 0 ≤ εR) (hεC : 0 ≤ εC)
    (hE : ∃ x y : ℕ → K, IsEq M (range M) (range (M + d)) x y)
    (hbeat : ∀ x' y' : ℕ → K, IsEqApprox M (range M) (range (M + d)) δ
    x' y' → ∀ j < N,
      j ≠ M + d → colPay M (range M) x' (M + d) + εC < colPay M (range
    M) x' j)
    (q : Finset ℕ × Finset ℕ) :
    StepApprox M N δ εR εC (range M, range (M + d)) q ↔ q = (range M,
    range (M + d + 1)) := by
  obtain ⟨x, y, hxy⟩ := hE
  refine stepApprox_iff (by omega) (by omega) hεR hεC ⟨x, y, hxy.approx
    hδ⟩
    (fun x' y' h i _ hi => insert_eq_of_mem (mem_range.2 hi)) (fun x' y'
    h j hj hjN => ?_) q
  suffices j = M + d by rw [this, ← range_add_one]
  by_contra hne
  linarith [hj (M + d) hdN, hbeat x' y' h j hjN hne]

/-- The arithmetic at `C_M`: from `15 M u_M = 5 x_0 - 3 x_1` and the
    bounds on columns `0` and
    `1`, the new column pays at most `-m + δ`. -/
theorem corridor_M_new (hM : 2 ≤ M) {a b P v m δ : K} (hδ : 0 ≤ δ)
    (hmv : 15 * (M : K) * m = v) (ha : 2 * (a - v) ≤ δ * ((M : K) + 1))
    (hb : v - δ ≤ b - a)
    (hid : 15 * (M : K) * P = 5 * a - 3 * b) : P ≤ -m + δ := by
  have hM2 : (2 : K) ≤ M := by exact_mod_cast hM
  refine le_of_mul_le_mul_left ?_ (by linarith : (0 : K) < 15 * (M : K))
  have h1 : (0 : K) ≤ δ * (14 * (M : K) - 4) := mul_nonneg hδ (by
    linarith)
  linarith only [hid, hmv, ha, hb, h1]

set_option maxHeartbeats 400000 in
/-- At `C_M = S_M`, with approximate oracles: the new column `M` pays
    about the negated margin
    of Table 1, so it is the only answer within tolerance. -/
theorem corridor_M_approx (hM : 2 ≤ M) (hMN : M < N) {δ εR εC : K} (hδ :
    0 ≤ δ)
    (hεR : 0 ≤ εR) (hεC : 0 ≤ εC)
    (hδm : 64 * δ ≤ 2 / (15 * (M : K) ^ 2 * ((M : K) + 1)))
    (hεm : 64 * εC ≤ 2 / (15 * (M : K) ^ 2 * ((M : K) + 1))) (q : Finset
    ℕ × Finset ℕ) :
```

```
    StepApprox M N δ εR εC (range M, range M) q ↔ q = (range M, range (M
    + 1)) := by
  obtain ⟨x, y, hxy, hu, hw, hyw⟩ := value_S (K := K) (M := M) (k := M)
    (by omega) le_rfl
  have hM' : (2 : K) ≤ M := by exact_mod_cast hM
  have hM0 : (0 : K) < M := by linarith
  obtain ⟨v, hv_def⟩ : ∃ v : K, v = 2 / ((M : K) * ((M : K) + 1)) := ⟨_,
    rfl⟩
  obtain ⟨m, hm_def⟩ : ∃ m : K, m = 2 / (15 * (M : K) ^ 2 * ((M : K) +
    1)) := ⟨_, rfl⟩
  rw [← hv_def] at hu hw hyw
  rw [← hm_def] at hδm hεm
  have hv0 : 0 < v := by rw [hv_def]; positivity
  have hm0 : 0 < m := by rw [hm_def]; positivity
  have hmv : 15 * (M : K) * m = v := by rw [hm_def, hv_def]; field_simp
  have hδM : 0 ≤ δ * (M : K) := mul_nonneg hδ hM0.le
  -- `δ` and `εC` are far below the value
  have hδv : δ * (2 * (M : K) + 10) ≤ v := by
    rw [hv_def, le_div_iff₀ (by positivity)]
    rw [hm_def, le_div_iff₀ (by positivity)] at hδm
    nlinarith [mul_nonneg (mul_nonneg hδ (by positivity : (0 : K) ≤ (M :
    K) * ((M : K) + 1)))
      (by linarith : (0 : K) ≤ 958 * (M : K) - 10)]
  refine corridor_step (d := 0) hM (by omega) hδ hεR hεC ⟨x, y, hxy⟩ (fun
    x' y' h j hjN hne => ?_) q
  obtain ⟨hlow, -⟩ := approx_bounds hxy.1 hxy.2.1 hu hw h
  have hcol0 : colPay M (range M) x' 0 = x' 0 := colPay_zero (by omega)
    (by omega) x'
  have hcol1 : colPay M (range M) x' 1 = x' 1 - x' 0 := by
    rw [colPay_succ (show 0 + 1 < M by omega)]
    simp [show (1 : ℕ) < M by omega, show 0 < M by omega]
  have hcolM : colPay M (range M) x' (M + 0) = pay M (x' 0) (x' 1) 4 :=
    colPay_add (by omega) x' 0
  -- column `0` carries weight `M v`, so it pays at most `v + δ (M + 1)
    / 2`
  have hx0 : 2 * (x' 0 - v) ≤ δ * ((M : K) + 1) := by
    have hcs := approx_cs hxy.1 hxy.2.1 hu hw h (mem_range.2 (show 0 < M
    by omega)) hδ
    rw [hcol0, hyw 0 (by omega), Nat.cast_zero, sub_zero,
      show (M : K) * v = 2 / ((M : K) + 1) by rw [hv_def]; field_simp,
      div_mul_eq_mul_div, div_le_iff₀ (by linarith : (0 : K) < (M : K) +
    1)] at hcs
    exact hcs
  have hx1 : v - δ ≤ x' 1 - x' 0 := by
    rw [← hcol1]; exact hlow 1 (mem_range.2 (by omega))
  -- so the new column pays at most `-m + δ`
  have hnew : colPay M (range M) x' (M + 0) ≤ -m + δ := by
    have hpay := pay_four (M := M) (by omega) (x' 0) (x' 1)
    rw [← hcolM] at hpay
    exact corridor_M_new hM hδ hmv hx0 hx1 hpay
  obtain ⟨e3, e4⟩ := g_three (x' 0) (x' 1)
  obtain ⟨hneg, -, hquant⟩ := alternation (M := M) (m := 3) (a := x' 0)
    (b := x' 1) (by omega)
    (by omega) (by rw [e3]; linarith) (by rw [e3, e4]; linarith)
  rcases lt_or_gt_of_ne hne with hlt | hgt
  · -- a column of the subgame, which pays at least `v - δ`
    linarith only [hlow j (mem_range.2 (by omega : j < M)), hnew, hδm,
    hεm, hmv, hm0,
      mul_le_mul_of_nonneg_right hM' hm0.le]
  · -- a later corridor column, which `alternation` keeps above `5 / 8`
    of the new one
    obtain ⟨t, rfl⟩ : ∃ t, j = M + t := ⟨j - M, by omega⟩
    rw [hcolM, colPay_add (by omega) x' t]
    exact later_beats hm0 hδm hεm (by rw [← hcolM]; linarith)
      (hquant (t + 4) (by omega) : pay M (x' 0) (x' 1) 4 * (5 / 8) ≤ _)
```

```
/-- The arithmetic at `C_{M+1}`: from the identity `240 M u_{M+1} = -9
    (15 M u_M) + 3 u_1` and
    the bounds on the two tight columns, the new column pays at most `-m
    + δ`, and column `1`
    pays at most `30 M` times column `M`, which is the alternation
    hypothesis there. -/
theorem handoff_new (hM : 2 ≤ M) {A B P v m δ : K} (hδ : 0 ≤ δ) (hv : 0
    < v)
    (hmv : 80 * (M : K) * m = (45 * M - 1) * v) (hδv : 64 * δ ≤ v) (hA :
    v - δ ≤ A)
    (hB : (B - v) * (5 * (M : K) - 2) ≤ δ * (20 * M ^ 2))
    (hid : 240 * (M : K) * P = -9 * (15 * M * A) + 3 * B) :
    P ≤ -m + δ ∧ B ≤ 30 * (M : K) * A := by
  have hM2 : (2 : K) ≤ M := by exact_mod_cast hM
  have hM0 : (0 : K) < M := by linarith
  have h1 : ∀ c : K, 0 ≤ c → c * (5 * (M : K) - 2) * (v - δ) ≤ c * (5 *
    M - 2) * A := fun c hc =>
    mul_le_mul_of_nonneg_left hA (mul_nonneg hc (by linarith))
  constructor
  · refine le_of_mul_le_mul_left ?_
      (mul_pos (by linarith : (0 : K) < 5 * (M : K) - 2) (by linarith :
    (0 : K) < 240 * M))
    have h2 : (5 * (M : K) - 2) * (240 * M * P) = (5 * M - 2) * (-9 *
    (15 * M * A) + 3 * B) := by
      rw [hid]
    have h3 : (5 * (M : K) - 2) * (240 * M * m) = (5 * M - 2) * (3 *
    ((45 * M - 1) * v)) := by
      rw [show (240 : K) * M * m = 3 * (80 * M * m) by ring, hmv]
    have h4 : (0 : K) ≤ δ * ((M : K) * (465 * M - 210)) :=
      mul_nonneg hδ (mul_nonneg hM0.le (by linarith))
    linarith only [h1 (135 * (M : K)) (by positivity), h2, h3, h4, hB]
  · refine le_of_mul_le_mul_right ?_ (by linarith : (0 : K) < 5 * (M : K)
    - 2)
    have h2 : (M : K) ^ 2 * (64 * δ) ≤ (M : K) ^ 2 * v :=
      mul_le_mul_of_nonneg_left hδv (sq_nonneg _)
    have h3 : 2 * (M : K) * v ≤ (M : K) ^ 2 * v :=
      mul_le_mul_of_nonneg_right (by nlinarith) hv.le
    linarith only [h1 (30 * (M : K)) (by positivity), h2, h3, hB, hv,
    mul_nonneg hM0.le hδ,
      mul_nonneg (sq_nonneg (M : K)) hv.le]

set_option maxHeartbeats 400000 in
/-- At `C_{M+1}`, with approximate oracles: the new column `M + 1` pays
    about the negated
    margin of Table 1, so it is the only answer within tolerance. -/
theorem corridor_handoff_approx (hM : 2 ≤ M) (hMN : M + 1 < N) {δ εR εC
    : K} (hδ : 0 ≤ δ)
    (hεR : 0 ≤ εR) (hεC : 0 ≤ εC)
    (hδm : 64 * δ ≤ (45 * (M : K) - 1) / (80 * M ^ 2 * (8 * M + 1)))
    (hεm : 64 * εC ≤ (45 * (M : K) - 1) / (80 * M ^ 2 * (8 * M + 1))) (q
    : Finset ℕ × Finset ℕ) :
    StepApprox M N δ εR εC (range M, range (M + 1)) q ↔ q = (range M,
    range (M + 2)) := by
  obtain ⟨x, y, hxy, hy1⟩ := (cert_handoff (K := K) hM).1
  obtain ⟨hge, hx1, -⟩ := (cert_handoff hM).2 x y hxy
  have hM' : (2 : K) ≤ M := by exact_mod_cast hM
  have hM0 : (0 : K) < M := by linarith
  obtain ⟨v, hv_def⟩ : ∃ v : K, v = 1 / ((M : K) * (8 * M + 1)) := ⟨_,
    rfl⟩
  obtain ⟨m, hm_def⟩ : ∃ m : K, m = (45 * (M : K) - 1) / (80 * M ^ 2 * (8
    * M + 1)) := ⟨_, rfl⟩
  rw [← hv_def] at hge hx1
  rw [← hm_def] at hδm hεm
  have hv0 : 0 < v := by rw [hv_def]; positivity
```

```
  have hm0 : 0 < m := by rw [hm_def]; exact div_pos (by linarith) (by
    positivity)
  have hmv : 80 * (M : K) * m = (45 * M - 1) * v := by rw [hm_def,
    hv_def]; field_simp
  have hmlev : m ≤ v := by
    refine le_of_mul_le_mul_left ?_ (by linarith : (0 : K) < 80 * (M :
    K))
    linarith only [hmv, mul_nonneg hM0.le hv0.le, hv0]
  have hcol1 : ∀ z : ℕ → K, colPay M (range M) z 1 = z 1 - z 0 := fun z
    => by
    rw [colPay_succ (show 0 + 1 < M by omega)]
    simp [show (1 : ℕ) < M by omega, show 0 < M by omega]
  have hu : ∀ j ∈ range (M + 1), v ≤ colPay M (range M) x j := fun j hj =
    > hge j (mem_range.1 hj)
  have hw : ∀ i ∈ range M, rowPay M (range (M + 1)) y i ≤ v := fun i hi =
    > by
    have := hxy.2.2 i hi 1 (mem_range.2 (by omega))
    rwa [hcol1 x, hx1] at this
  refine corridor_step (d := 1) hM (by omega) hδ hεR hεC ⟨x, y, hxy⟩ (fun
    x' y' h j hjN hne => ?_) q
  obtain ⟨hlow, -⟩ := approx_bounds hxy.1 hxy.2.1 hu hw h
  -- the tight columns `M` and `1` pay within `δ / weight` of the value
  obtain ⟨A, hA_def⟩ : ∃ t : K, t = colPay M (range M) x' M := ⟨_, rfl⟩
  obtain ⟨B, hB_def⟩ : ∃ t : K, t = colPay M (range M) x' 1 := ⟨_, rfl⟩
  have hA_low : v - δ ≤ A := by rw [hA_def]; exact hlow M (mem_range.2
    (by omega))
  have hB_high : (B - v) * (5 * (M : K) - 2) ≤ δ * (20 * M ^ 2) := by
    have h1 := approx_cs_le hxy.1 hxy.2.1 hu hw h (mem_range.2 (show 1 <
    M + 1 by omega)) hδ
      (div_pos (by linarith) (by positivity)) hy1
    rw [← hB_def, ← sub_le_iff_le_add', div_div_eq_mul_div,
      le_div_iff₀ (by linarith : (0 : K) < 5 * (M : K) - 2)] at h1
    exact h1
  -- the new column pays `-(9/16) A + B / (80 M)`, which is at most `-m +
     δ`
  have hAid : 15 * (M : K) * A = 5 * x' 0 - 3 * x' 1 := by
    rw [hA_def, show colPay M (range M) x' M = pay M (x' 0) (x' 1) 4
    from colPay_add (by omega) x' 0]
    exact pay_four (by omega) _ _
  have hBid : B = x' 1 - x' 0 := by rw [hB_def, hcol1]
  have hnewid : 240 * (M : K) * colPay M (range M) x' (M + 1) = -9 * (15
    * M * A) + 3 * B := by
    rw [show colPay M (range M) x' (M + 1) = pay M (x' 0) (x' 1) 5 from
    colPay_add (by omega) x' 1,
      pay, s5, s6, hAid, hBid]
    field_simp
    ring
  have hδv64 : 64 * δ ≤ v := hδm.trans hmlev
  have ⟨hnew, hBA⟩ := handoff_new hM hδ hv0 hmv hδv64 hA_low hB_high
    hnewid
  -- by alternation, it beats every later corridor column
  have hg45 : g (x' 0) (x' 1) 4 = 5 * x' 0 - 3 * x' 1 ∧
      g (x' 0) (x' 1) 5 = -(8 * x' 0 - 5 * x' 1) := by
    obtain ⟨-, f4, f5, f6⟩ := fib_vals
    constructor <;> simp [g, f4, f5, f6] <;> ring
  have hA0 : 0 < A := by linarith
  obtain ⟨hneg, -, hquant⟩ := alternation (M := M) (m := 4) (a := x' 0)
    (b := x' 1) (by omega)
    (by omega) (by rw [hg45.1, ← hAid]; exact mul_pos (by linarith) hA0)
    (by rw [hg45.1, hg45.2]; linarith only [hAid, hBid, hBA])
  have hL : colPay M (range M) x' (M + 1) = pay M (x' 0) (x' 1) 5 :=
    colPay_add (by omega) x' 1
  rcases lt_or_gt_of_ne hne with hlt | hgt
  · -- a column of the subgame, which pays at least `v - δ`
```

```
    linarith only [hlow j (mem_range.2 (by omega : j < M + 1)), hnew,
    hδm, hεm, hmlev, hm0]
  · -- a later corridor column
    obtain ⟨t, rfl⟩ : ∃ t, j = M + t := ⟨j - M, by omega⟩
    rw [hL, colPay_add (by omega) x' t]
    exact later_beats hm0 hδm hεm (by rw [← hL]; linarith)
      (hquant (t + 4) (by omega) : pay M (x' 0) (x' 1) 5 * (5 / 8) ≤ _)

/-- The margin at `C_{M+e+2}`, with `K = e + 5`: the new column pays
    `-V F(K)^2 / (F(K+1) F(K+2))`. -/
def corrMargin (M e : ℕ) : K :=
  (fib (e + 5) : K) ^ 2 / (corrMass M e * ((fib (e + 4) : K) + fib (e +
    5))
    * ((fib (e + 4) : K) + 2 * fib (e + 5)))

/-- Three consecutive corridor payoffs: `F(K+1) F(K+2) u_{K+1} = F(K-1)
    F(K) u_{K-1}
    - F(K) F(K+1) u_K`, by `pay_g` and the recurrence for `g`. -/
theorem pay_three (hM : 0 < M) (a b : K) (e : ℕ) {p q : K}
    (hp : p = (fib (e + 4) : K)) (hq : q = (fib (e + 5) : K)) :
    (p + q) * (p + 2 * q) * pay M a b (e + 6)
      = p * q * pay M a b (e + 4) - q * (p + q) * pay M a b (e + 5) := by
  have hf6 : ((fib (e + 6) : K)) = p + q := by
    rw [hp, hq, show e + 6 = e + 4 + 2 from rfl]; exact_mod_cast
    fib_add_two
  have hf7 : ((fib (e + 7) : K)) = p + 2 * q := by
    rw [show e + 7 = e + 5 + 2 from rfl, fib_add_two]
    push_cast
    rw [show e + 5 + 1 = e + 6 from rfl, hf6, ← hq]
    ring
  have hMne : (M : K) ≠ 0 := by positivity
  have hp0 : p ≠ 0 := by rw [hp]; simp
  have hq0 : q ≠ 0 := by rw [hq]; simp
  have hpq0 : p + q ≠ 0 := by rw [← hf6]; simp
  have hpq0' : p + 2 * q ≠ 0 := by rw [← hf7]; simp
  have hpq2 : p + q * 2 ≠ 0 := by rw [show p + q * 2 = p + 2 * q by
    ring]; exact hpq0'
  have e4 : (M : K) * p * q * pay M a b (e + 4) = g a b (e + 4) := by
    rw [pay_g hM (by omega), show e + 4 + 1 = e + 5 from rfl, ← hp, ←
    hq]
    field_simp
  have e5 : (M : K) * q * (p + q) * pay M a b (e + 5) = g a b (e + 5) :=
    by
    rw [pay_g hM (by omega), show e + 5 + 1 = e + 6 from rfl, hf6, ← hq]
    field_simp
  have e6 : (M : K) * ((p + q) * (p + 2 * q)) * pay M a b (e + 6) = g a
    b (e + 6) := by
    rw [pay_g hM (by omega), show e + 6 + 1 = e + 7 from rfl, hf6, hf7]
    field_simp
  refine mul_left_cancel₀ hMne ?_
  linear_combination e6 - e4 + e5 + g_rec a b (e + 4)

set_option maxHeartbeats 400000 in
/-- The arithmetic at `C_{M+e+2}`: from the identity of `pay_three` and
    the bounds on the two
    frontier columns, the new column pays at most `-m + 2 δ`, and `F(K)
    u_K` is at most
    `-2 F(K+2) u_{K+1}`, which is the alternation hypothesis there. -/
theorem corridor_new {p q V m δ u v w : K} (hp : 3 ≤ p) (hq : 5 ≤ q)
    (hpq : p ≤ q)
    (h85 : 8 * p ≤ 5 * q) (hδ : 0 ≤ δ) (hV : 0 < V) (hδm : 64 * δ ≤ m)
    (hmV : m * ((p + q) * (p + 2 * q)) = V * q ^ 2) (hu : u ≤ V + 6 * δ)
    (hv : V - δ ≤ v)
    (hid : (p + q) * (p + 2 * q) * w = p * q * u - q * (p + q) * v) :
    w ≤ -m + 2 * δ ∧ q * v ≤ -2 * (p + 2 * q) * w := by
```

```
  have hprod : (0 : K) < (p + q) * (p + 2 * q) :=
    mul_pos (by linarith) (by linarith)
  constructor
  · refine le_of_mul_le_mul_left ?_ hprod
    have h1 : p * q * u ≤ p * q * (V + 6 * δ) :=
    mul_le_mul_of_nonneg_left hu (by positivity)
    have h2 : q * (p + q) * (V - δ) ≤ q * (p + q) * v :=
      mul_le_mul_of_nonneg_left hv (by positivity)
    have h3 : δ * (q * (7 * p + q)) ≤ δ * (2 * ((p + q) * (p + 2 * q)))
    :=
      mul_le_mul_of_nonneg_left (by nlinarith) hδ
    linarith only [hid, hmV, h1, h2, h3]
  · -- it suffices that `2 p u ≤ (p + q) v`, which needs `V (q - p) ≥ δ
    (13 p + q)`
    have key : 2 * p * u ≤ (p + q) * v := by
      have h2 : 64 * δ * ((p + q) * (p + 2 * q)) ≤ V * q ^ 2 :=
        (mul_le_mul_of_nonneg_right hδm hprod.le).trans_eq hmV
      -- `64 (q - p) (p + q) (p + 2 q) ≥ 48 q^3 ≥ q^2 (13 p + q)`, as
    `8 p ≤ 5 q`
      have h4 : q ^ 2 * (13 * p + q) ≤ 64 * ((q - p) * ((p + q) * (p + 2
    * q))) := by
        have c1 : 24 * q * ((p + q) * (p + 2 * q)) ≤ 64 * (q - p) * ((p +
     q) * (p + 2 * q)) :=
          mul_le_mul_of_nonneg_right (by linarith) hprod.le
        have c2 : 48 * q ^ 3 ≤ 24 * q * ((p + q) * (p + 2 * q)) := by
          nlinarith [mul_nonneg (by linarith : (0 : K) ≤ q) (sq_nonneg
    p),
            mul_nonneg (by linarith : (0 : K) ≤ p) (sq_nonneg q)]
        have c3 : q ^ 2 * (13 * p + q) ≤ 48 * q ^ 3 := by
          nlinarith [mul_nonneg (sq_nonneg q) (by linarith : (0 : K) ≤
    47 * q - 13 * p)]
        linarith
      have h3 : δ * (13 * p + q) ≤ V * (q - p) := by
        refine le_of_mul_le_mul_right ?_ (by linarith : (0 : K) < 64 *
    ((p + q) * (p + 2 * q)))
        linarith only [mul_le_mul_of_nonneg_right h2 (by linarith : (0 :
    K) ≤ 13 * p + q),
          mul_le_mul_of_nonneg_left h4 hV.le]
      linarith only [h3, mul_le_mul_of_nonneg_left hu (by linarith : (0
    : K) ≤ 2 * p),
        mul_le_mul_of_nonneg_left hv (by linarith : (0 : K) ≤ p + q)]
    refine le_of_mul_le_mul_left ?_ (show (0 : K) < p + q by linarith)
    linarith only [hid, mul_le_mul_of_nonneg_left key (by linarith : (0
    : K) ≤ q)]

set_option maxHeartbeats 800000 in
/-- At `C_{M+e+2}`, with approximate oracles: the new column pays about
    `-corrMargin M e`, so it
    is the only answer within tolerance. -/
theorem corridor_deep_approx (hM : 2 ≤ M) (e : ℕ) (hN : M + e + 2 < N)
    {δ εR εC : K}
    (hδ : 0 ≤ δ) (hεR : 0 ≤ εR) (hεC : 0 ≤ εC) (hδm : 64 * δ ≤
    corrMargin M e)
    (hεm : 64 * εC ≤ corrMargin M e)
    (r : Finset ℕ × Finset ℕ) :
    StepApprox M N δ εR εC (range M, range (M + e + 2)) r ↔ r = (range
    M, range (M + e + 3)) := by
  obtain ⟨x, y, hxy, hya⟩ := (cert_corridor (K := K) hM e).1
  obtain ⟨hge, h4, h5, -⟩ := (cert_corridor hM e).2 x y hxy
  obtain ⟨p, hp_def⟩ : ∃ t : K, t = (fib (e + 4) : K) := ⟨_, rfl⟩
  obtain ⟨q, hq_def⟩ : ∃ t : K, t = (fib (e + 5) : K) := ⟨_, rfl⟩
  obtain ⟨V, hV_def⟩ : ∃ t : K, t = 1 / corrMass M e := ⟨_, rfl⟩
  obtain ⟨m, hm_def⟩ : ∃ t : K, t = corrMargin M e := ⟨_, rfl⟩
  rw [← hV_def] at hge h4 h5
  rw [← hm_def] at hδm hεm
```

```
  have hM0 : (0 : K) < M := by exact_mod_cast (by omega : 0 < M)
  -- the Fibonacci ratios at the frontier
  have hp3 : (3 : K) ≤ p := by rw [hp_def]; exact_mod_cast (fib_mono (by
    omega : 4 ≤ e + 4))
  have hq5 : (5 : K) ≤ q := by rw [hq_def]; exact_mod_cast (fib_mono (by
    omega : 5 ≤ e + 5))
  have hpq : p ≤ q := by rw [hp_def, hq_def]; exact_mod_cast
    fib_le_fib_succ
  have h85 : 8 * p ≤ 5 * q := by
    rw [hp_def, hq_def]; exact_mod_cast (fib_ratio (by omega : 4 ≤ e +
    4)).2
  have hp0 : (0 : K) < p := by linarith
  have hq0 : (0 : K) < q := by linarith
  have hprod : (0 : K) < (p + q) * (p + 2 * q) := by nlinarith
  have hV0 : 0 < V := by rw [hV_def]; exact one_div_pos.2 (corrMass_pos
    hM e)
  have hmass : (corrMass M e : K) ≠ 0 := (corrMass_pos hM e).ne'
  have hpq1 : p + q ≠ 0 := by positivity
  have hpq2 : p + 2 * q ≠ 0 := by positivity
  have hmV : m * ((p + q) * (p + 2 * q)) = V * q ^ 2 := by
    rw [hm_def, hV_def, corrMargin, ← hp_def, ← hq_def]
    field_simp
  have hm0 : 0 < m := by nlinarith [mul_pos hV0 (mul_pos hq0 hq0)]
  have h2q : 2 * q ^ 2 ≤ (p + q) * (p + 2 * q) := by nlinarith
  have hmV' : m ≤ V := by nlinarith [mul_le_mul_of_nonneg_left h2q
    hm0.le]
  have hcol : ∀ (z : ℕ → K) t, colPay M (range M) z (M + t) = pay M (z
    0) (z 1) (t + 4) :=
    fun z t => colPay_add (by omega) z t
  have hu : ∀ j ∈ range (M + e + 2), V ≤ colPay M (range M) x j :=
    fun j hj => hge j (mem_range.1 hj)
  have hw : ∀ i ∈ range M, rowPay M (range (M + e + 2)) y i ≤ V := fun i
    hi => by
    have := hxy.2.2 i hi (M + e) (mem_range.2 (by omega))
    rwa [hcol x e, h4] at this
  refine corridor_step (d := e + 2) hM (by omega) hδ hεR hεC ⟨x, y, hxy⟩
    (fun x' y' h j hjN hne => ?_) r
  obtain ⟨hlow, -⟩ := approx_bounds hxy.1 hxy.2.1 hu hw h
  -- the two frontier columns pay between `V - δ` and `V + 6 δ`
  have hA : pay M (x' 0) (x' 1) (e + 4) ≤ V + 6 * δ := by
    have h1 := approx_cs_le hxy.1 hxy.2.1 hu hw h
      (mem_range.2 (show M + e < M + e + 2 by omega)) hδ (by norm_num)
    hya
    rwa [hcol x' e, show δ / ((1 : K) / 6) = 6 * δ by ring] at h1
  have hB' : V - δ ≤ pay M (x' 0) (x' 1) (e + 5) := by
    have := hlow (M + e + 1) (mem_range.2 (by omega))
    rwa [show M + e + 1 = M + (e + 1) from rfl, hcol x' (e + 1),
      show e + 1 + 4 = e + 5 from rfl] at this
  -- the new column pays at most `-m + 2 δ`
  have hid := pay_three (M := M) (by omega) (x' 0) (x' 1) e hp_def hq_def
  obtain ⟨hnew, hkey⟩ := corridor_new hp3 hq5 hpq h85 hδ hV0 hδm hmV hA
    hB' hid
  have e5 : (M : K) * q * (p + q) * pay M (x' 0) (x' 1) (e + 5) = g (x'
    0) (x' 1) (e + 5) := by
    rw [pay_g (show 0 < M by omega) (by omega), show e + 5 + 1 = e + 6
    from rfl,
      show ((fib (e + 6) : K)) = p + q by
        rw [hp_def, hq_def, show e + 6 = e + 4 + 2 from rfl];
    exact_mod_cast fib_add_two, ← hq_def]
    have : (M : K) ≠ 0 := by positivity
    have : q ≠ 0 := hq0.ne'
    have : p + q ≠ 0 := by linarith
    field_simp
  have e6 : (M : K) * ((p + q) * (p + 2 * q)) * pay M (x' 0) (x' 1) (e +
    6)
```

```
      = g (x' 0) (x' 1) (e + 6) := by
    rw [pay_g (show 0 < M by omega) (by omega), show e + 6 + 1 = e + 7
    from rfl,
      show ((fib (e + 6) : K)) = p + q by
        rw [hp_def, hq_def, show e + 6 = e + 4 + 2 from rfl];
    exact_mod_cast fib_add_two,
      show ((fib (e + 7) : K)) = p + 2 * q by
        rw [show e + 7 = e + 5 + 2 from rfl, fib_add_two]
        push_cast
        rw [show e + 5 + 1 = e + 6 from rfl, show ((fib (e + 6) : K)) =
    p + q by
          rw [hp_def, hq_def, show e + 6 = e + 4 + 2 from rfl];
    exact_mod_cast fib_add_two,
          ← hq_def]
        ring]
    have : (M : K) ≠ 0 := by positivity
    have : p + q ≠ 0 := by linarith
    have : p + 2 * q ≠ 0 := by linarith
    field_simp
  obtain ⟨hneg, -, hquant⟩ := alternation (M := M) (m := e + 5) (a := x'
    0) (b := x' 1)
    (by omega) (by omega)
    (by rw [← e5]
        exact mul_pos (mul_pos (mul_pos hM0 hq0) (by linarith)) (by
    linarith))
    (by rw [← e5, show e + 5 + 1 = e + 6 from rfl, ← e6]
        nlinarith [mul_le_mul_of_nonneg_left hkey
          (mul_nonneg hM0.le (by linarith : (0 : K) ≤ p + q))])
  have hL : colPay M (range M) x' (M + (e + 2)) = pay M (x' 0) (x' 1) (e
    + 6) := hcol x' (e + 2)
  rcases lt_or_gt_of_ne hne with hlt | hgt
  · -- a column of the subgame, which pays at least `V - δ`
    rw [hL]
    linarith only [hlow j (mem_range.2 hlt), hnew, hδm, hεm, hmV', hm0]
  · -- a later corridor column
    obtain ⟨t, rfl⟩ : ∃ t, j = M + t := ⟨j - M, by omega⟩
    rw [hL, hcol x' t]
    exact later_beats hm0 hδm hεm (by linarith)
      (hquant (t + 4) (by omega) : pay M (x' 0) (x' 1) (e + 6) * (5 / 8)
    ≤ _)

set_option maxHeartbeats 400000 in
/-- Staircase right for every `N`: at `S_k` with `k < M ≤ N`, accuracies
    below a fixed fraction
    of the value `v = 2 / (k (k + 1))` leave `T_k` as the only
    approximate successor. -/
theorem staircase_right_gen (hM : 2 ≤ M) (hMN : M ≤ N) (hk : 0 < k)
    (hkM : k < M)
    {δ εR εC : K} (hδ : 0 ≤ δ) (hεR : 0 ≤ εR) (hεC : 0 ≤ εC)
    (h1 : 64 * δ ≤ 2 / ((k : K) * (k + 1))) (h2R : 64 * εR ≤ 2 / ((k :
    K) * (k + 1)))
    (h2C : 64 * εC ≤ 2 / ((k : K) * (k + 1))) (q : Finset ℕ × Finset ℕ)
    :
    StepApprox M N δ εR εC (range k, range k) q ↔ q = (range k, range (k
    + 1)) := by
  have hk' : (1 : K) ≤ k := by exact_mod_cast hk
  have hkM' : (k : K) + 1 ≤ M := by exact_mod_cast hkM
  have hM0 : (0 : K) < M := by linarith
  have h3M : (0 : K) < 3 * M := by linarith
  have hv0 : (0 : K) < 2 / ((k : K) * (k + 1)) := by positivity
  have hkv : (k : K) * (2 / ((k : K) * (k + 1))) = 2 / ((k : K) + 1) :=
    by field_simp
  have hvW : 2 / ((k : K) * (k + 1)) ≤ 2 / ((k : K) + 1) :=
    div_le_div_of_nonneg_left (by norm_num) (by positivity) (by
    nlinarith)
```

```
  -- `1 / (3 M) ≤ W / 6` and `k δ ≤ W / 64`, where `W = k v = 2 / (k +
    1)`
  have hcW : (1 : K) / (3 * M) ≤ 2 / ((k : K) + 1) / 6 := by
    rw [show (2 : K) / ((k : K) + 1) / 6 = 1 / (3 * ((k : K) + 1)) by
    field_simp; norm_num]
    exact div_le_div_of_nonneg_left (by norm_num) (by positivity) (by
    linarith)
  have hkδ : (k : K) * δ ≤ 2 / ((k : K) + 1) / 64 := by
    rw [le_div_iff₀ (by norm_num : (0 : K) < 64), ← hkv]
    nlinarith [mul_le_mul_of_nonneg_left h1 (by linarith : (0 : K) ≤ (k
    : K))]
  refine staircase_right_approx (K := K) (b := 1 / (3 * (M : K))) hk hkM
    (by omega) hδ hεR hεC
    (by positivity)
    (by linarith) ?_ ?_ q
  · rw [mul_sub, hkv]
    linarith
  · intro x' hx' j hkj hjN
    rcases lt_or_ge j M with hjM | hjM
    · obtain ⟨t, rfl⟩ : ∃ t, j = t + 1 := ⟨j - 1, by omega⟩
      rw [colPay_succ hjM]
      simp only [if_neg (show ¬ t + 1 < k by omega), if_neg (show ¬ t <
    k by omega), sub_zero,
        neg_le, neg_zero]
      positivity
    · have hsum : Σ i ∈ range (min k 2), x' i ≤ 1 := by
        rw [← hx'.2]
        exact sum_le_sum_of_subset_of_nonneg
          (fun i hi => mem_range.2 ((mem_range.1 hi).trans_le
    (Nat.min_le_left k 2)))
        fun i hi _ => hx'.1 i hi
      have hc := corr_ge_mix hM hk hjM hx'.1 hsum
      rw [show -(1 / (3 * (M : K))) = -1 / (3 * M) by ring, div_le_iff₀
    h3M]
      linarith

set_option maxHeartbeats 400000 in
/-- Staircase down for every `N`: at `T_k` the same accuracies leave
    `S_{k+1}` as the only
    approximate successor. -/
theorem staircase_down_gen (hM : 2 ≤ M) (hMN : M ≤ N) (hk : 0 < k) (hkM
    : k < M)
    {δ εR εC : K} (hδ : 0 ≤ δ) (hεR : 0 ≤ εR) (hεC : 0 ≤ εC)
    (h1 : 64 * δ ≤ 2 / ((k : K) * (k + 1))) (h2R : 64 * εR ≤ 2 / ((k :
    K) * (k + 1)))
    (h2C : 64 * εC ≤ 2 / ((k : K) * (k + 1))) (q : Finset ℕ × Finset ℕ)
    :
    StepApprox M N δ εR εC (range k, range (k + 1)) q ↔ q = (range (k +
    1), range (k + 1)) := by
  have hk' : (1 : K) ≤ k := by exact_mod_cast hk
  have hkM' : (k : K) + 1 ≤ M := by exact_mod_cast hkM
  have hM0 : (0 : K) < M := by linarith
  have h3M : (0 : K) < 3 * M := by linarith
  have hv0 : (0 : K) < 2 / ((k : K) * (k + 1)) := by positivity
  have hkv : (k : K) * (2 / ((k : K) * (k + 1))) = 2 / ((k : K) + 1) :=
    by field_simp
  have hvW : 2 / ((k : K) * (k + 1)) ≤ 2 / ((k : K) + 1) :=
    div_le_div_of_nonneg_left (by norm_num) (by positivity) (by
    nlinarith)
  have hkδ : (k : K) * δ ≤ 2 / ((k : K) + 1) / 64 := by
    rw [le_div_iff₀ (by norm_num : (0 : K) < 64), ← hkv]
    nlinarith [mul_le_mul_of_nonneg_left h1 (by linarith : (0 : K) ≤ (k
    : K))]
  refine staircase_down_approx (K := K) (b := 3 * (2 / ((k : K) * (k +
    1))) / 4) hk hkM
```

```
    (by omega) hδ hεR hεC
    (by linarith) ?_ ?_ q
  · rw [mul_sub, hkv]
    linarith
  · -- the weights decay, `x' i ≤ (k - i) (v + δ)`, so the corridor pays
    at least `-3 v / 4`
    intro x' hx' hlow j hkj hjN
    rcases lt_or_ge j M with hjM | hjM
    · obtain ⟨t, rfl⟩ : ∃ t, j = t + 1 := ⟨j - 1, by omega⟩
      rw [colPay_succ hjM]
      simp only [if_neg (show ¬ t + 1 < k by omega), if_neg (show ¬ t <
    k by omega), sub_zero,
        neg_le, neg_zero]
      positivity
    · have htel : ∀ d, d < k → x' (k - 1 - d) ≤ ((d : K) + 1) * (2 / ((k
    : K) * (k + 1)) + δ) := by
        intro d
        induction d with
        | zero =>
          intro _
          have h6 := hlow k (mem_range.2 (by omega))
          obtain ⟨t, rfl⟩ : ∃ t, k = t + 1 := ⟨k - 1, by omega⟩
          rw [colPay_succ (by omega : t + 1 < M)] at h6
          simp only [if_neg (show ¬ t + 1 < t + 1 by omega), if_pos
    (show t < t + 1 by omega),
            zero_sub] at h6
          rw [show t + 1 - 1 - 0 = t by omega]
          push_cast at h6 ⊢
          linarith
        | succ d ih =>
          intro hd
          have h6 := hlow (k - 1 - d) (mem_range.2 (by omega))
          rw [show k - 1 - d = (k - 1 - (d + 1)) + 1 by omega,
            colPay_succ (by omega : (k - 1 - (d + 1)) + 1 < M)] at h6
          simp only [if_pos (show k - 1 - (d + 1) + 1 < k by omega),
            if_pos (show k - 1 - (d + 1) < k by omega)] at h6
          have h7 := ih (by omega)
          rw [show k - 1 - d = (k - 1 - (d + 1)) + 1 by omega] at h7
          push_cast
          linarith
      have hx1 : Σ i ∈ range (min k 2), x' i
          ≤ (2 * (k : K) - 1) * (2 / ((k : K) * (k + 1)) + δ) := by
        have hx0 : x' 0 ≤ (k : K) * (2 / ((k : K) * (k + 1)) + δ) := by
          have h8 := htel (k - 1) (by omega)
          rw [show k - 1 - (k - 1) = 0 by omega,
            show ((k - 1 : ℕ) : K) + 1 = (k : K) by rw [Nat.cast_sub (by
    omega)]; push_cast; ring]
            at h8
          exact h8
        rcases lt_or_ge k 2 with hk2 | hk2
        · obtain rfl : k = 1 := by omega
          rw [show min 1 2 = 1 by omega, sum_range_one]
          push_cast at hx0 ⊢
          linarith
        · rw [show min k 2 = 2 by omega, sum_range_succ, sum_range_one]
          have h9 := htel (k - 2) (by omega)
          rw [show k - 1 - (k - 2) = 1 by omega,
            show ((k - 2 : ℕ) : K) + 1 = (k : K) - 1 by
              rw [Nat.cast_sub (by omega)]; push_cast; ring] at h9
          linarith
      have hc := corr_ge_mix hM hk hjM hx'.1 hx1
      have hkey : (2 * (k : K) - 1) * (2 / ((k : K) * (k + 1)) + δ)
          ≤ 3 * M * (3 * (2 / ((k : K) * (k + 1))) / 4) := by
        have h10 : δ ≤ 2 / ((k : K) * (k + 1)) / 64 := by linarith
        nlinarith [hv0, hkM', hk']
```

```
      nlinarith [hc, hkey, h3M]

/-- The quantity `min (|v_t|, m_t)` of Table 1 at iteration `t`, which
    is what `δ` and `εC` must
    stay below a fixed fraction of: the value `2 / (k (k + 1))` at the
    staircase states `S_k` and
    `T_k`, and the margin in the corridor, where the new column pays its
    negative. The branches
    record which of the two is smaller, and `margin_eq_min` proves that
    they do. -/
def margin (M t : ℕ) : K :=
  if t + 2 < 2 * M then 2 / (((t / 2 + 1 : ℕ) : K) * (((t / 2 + 1 : ℕ) :
    K) + 1))
  else if t = 2 * M - 2 then 2 / (15 * (M : K) ^ 2 * ((M : K) + 1))
  else if t = 2 * M - 1 then (45 * (M : K) - 1) / (80 * (M : K) ^ 2 * (8
    * (M : K) + 1))
  else corrMargin M (t - 2 * M)

/-- The magnitude `|v_t|` of the subgame value at iteration `t`, as in
    Table 1: `2 / (k (k + 1))`
    at `S_k` and `T_k`, `2 / (M (M + 1))` at `C_M`, `1 / (M (8 M + 1))`
    at `C_{M+1}`, and
    `V_k = 1 / corrMass` at the deeper corridor states. -/
def tableValue (M t : ℕ) : K :=
  if t + 2 < 2 * M then 2 / (((t / 2 + 1 : ℕ) : K) * (((t / 2 + 1 : ℕ) :
    K) + 1))
  else if t = 2 * M - 2 then 2 / ((M : K) * ((M : K) + 1))
  else if t = 2 * M - 1 then 1 / ((M : K) * (8 * (M : K) + 1))
  else 1 / corrMass M (t - 2 * M)

/-- The margin `m_t = d_t - |v_t|` of Table 1 at iteration `t`, the
    payoff of the new strategy
    in its player's favor (a new column pays `-m_t`, a new row `m_t`):
    `2 / (k + 1)` at `S_k` and
    `T_k`, `2 / (15 M^2 (M + 1))` at `C_M`, `(45 M - 1) / (80 M^2 (8 M +
    1))` at `C_{M+1}`, and
    `corrMargin` at the deeper corridor states. `gap_run` ties it to the
    run. -/
def tableMargin (M t : ℕ) : K :=
  if t + 2 < 2 * M then 2 / (((t / 2 + 1 : ℕ) : K) + 1)
  else if t = 2 * M - 2 then 2 / (15 * (M : K) ^ 2 * ((M : K) + 1))
  else if t = 2 * M - 1 then (45 * (M : K) - 1) / (80 * (M : K) ^ 2 * (8
    * (M : K) + 1))
  else corrMargin M (t - 2 * M)

set_option maxHeartbeats 400000 in
/-- **`margin` is the minimum of the two columns of Table 1.** The value
    is the smaller one at
    the staircase states, where `2 / (k (k + 1)) ≤ 2 / (k + 1)`, and the
    margin is the smaller one
    at every corridor state, where `M ≤ 15 M^2`, `(45 M - 1) M ≤ 80
    M^2`, and
    `F(K)^2 ≤ F(K+1) F(K+2)`. -/
theorem margin_eq_min (hM : 2 ≤ M) (t : ℕ) :
    margin M t = min (tableValue (K := K) M t) (tableMargin M t) := by
  have hM2 : (2 : K) ≤ M := by exact_mod_cast hM
  have hM0 : (0 : K) < M := by linarith
  rw [margin, tableValue, tableMargin]
  split_ifs with h1 h2 h3
  · -- the staircase states, where the value is the smaller of the two
    refine (min_eq_left ?_).symm
    have hk : (1 : K) ≤ ((t / 2 + 1 : ℕ) : K) := by exact_mod_cast (show
    1 ≤ t / 2 + 1 by omega)
    rw [div_le_div_iff₀ (mul_pos (by linarith) (by linarith)) (by
    linarith)]
```

```
    nlinarith
  · -- `C_M`
    refine (min_eq_right ?_).symm
    rw [div_le_div_iff₀ (mul_pos (mul_pos (by norm_num) (pow_pos hM0 2))
    (by linarith))
      (mul_pos hM0 (by linarith))]
    nlinarith
  · -- `C_{M+1}`
    refine (min_eq_right ?_).symm
    rw [div_le_div_iff₀ (mul_pos (mul_pos (by norm_num) (pow_pos hM0 2))
    (by linarith))
      (mul_pos hM0 (by linarith))]
    nlinarith [mul_nonneg (mul_nonneg hM0.le (by linarith : (0 : K) ≤ 8 *
     (M : K) + 1))
      (by linarith : (0 : K) ≤ 35 * (M : K) + 1)]
  · -- the deeper corridor states
    refine (min_eq_right ?_).symm
    have hp : (1 : K) ≤ (fib (t - 2 * M + 4) : K) := by exact_mod_cast
    fib_pos.2 (by omega)
    have hq : (1 : K) ≤ (fib (t - 2 * M + 5) : K) := by exact_mod_cast
    fib_pos.2 (by omega)
    have hmass : (0 : K) < corrMass M (t - 2 * M) := corrMass_pos hM _
    have hsq : (fib (t - 2 * M + 5) : K) ^ 2
        ≤ ((fib (t - 2 * M + 4) : K) + fib (t - 2 * M + 5))
          * ((fib (t - 2 * M + 4) : K) + 2 * fib (t - 2 * M + 5)) := by
    nlinarith
    rw [corrMargin,
      div_le_div_iff₀ (mul_pos (mul_pos hmass (by linarith)) (by
    linarith)) hmass]
    linarith [mul_le_mul_of_nonneg_left hsq hmass.le]

/-- `v` is the value of the subgame on `R × C` at the equilibrium `(x,
    y)`: every column of `C`
    pays at least `v` against `x`, and every row of `R` pays at most `v`
    against `y`. -/
def IsValue (M : ℕ) (R C : Finset ℕ) (x y : ℕ → K) (v : K) : Prop :=
  (∀ j ∈ C, v ≤ colPay M R x j) ∧ ∀ i ∈ R, rowPay M C y i ≤ v

/-- At `S_k`, every equilibrium has the value `2 / (k (k + 1))`, which
    is Table 1's first row:
    the certificate makes every column of the subgame pay `V` and every
    row pay `V`, and the row
    weights `(i + 1) V` sum to `1`. -/
theorem value_S_eq (hk : 0 < k) (hkM : k ≤ M) (x y : ℕ → K)
    (hxy : IsEq M (range k) (range k) x y) :
    ∃ v : K, IsValue M (range k) (range k) x y v ∧ |v| = 2 / ((k : K) *
    ((k : K) + 1)) := by
  obtain ⟨V, hV0, hx, hcol, hrow⟩ := (cert_S (K := K) hk hkM).2 x y hxy
  have hk' : (0 : K) < k := by exact_mod_cast hk
  have hsum : Σ i ∈ range k, ((i : K) + 1) * V = 1 := by
    have h : Σ i ∈ range k, ((i : K) + 1) * V = Σ i ∈ range k, x i :=
      sum_congr rfl fun i hi => (hx i (mem_range.1 hi)).symm
    rw [h, hxy.1.2]
  rw [← sum_mul, sum_range_cast_succ] at hsum
  refine ⟨V, ⟨fun j hj => (hcol j (mem_range.1 hj)).ge, fun i hi => (hrow
    i (mem_range.1 hi)).le⟩,
    ?_⟩
  rw [abs_of_pos hV0, eq_div_iff (by nlinarith)]
  linarith

/-- At `T_k`, every equilibrium has the value `-2 / (k (k + 1))`, Table
    1's second row. The
    columns are bounded through `IsEq`, since row `0` pays exactly the
    value. -/
theorem value_T_eq (hk : 0 < k) (hkM : k < M) (x y : ℕ → K)
```

```
    (hxy : IsEq M (range k) (range (k + 1)) x y) :
    ∃ v : K, IsValue M (range k) (range (k + 1)) x y v ∧ |v| = 2 / ((k :
    K) * ((k : K) + 1)) := by
  obtain ⟨V, hV0, hx, hrow⟩ := (cert_T (K := K) hk hkM).2 x y hxy
  have hk' : (0 : K) < k := by exact_mod_cast hk
  have hsum : Σ i ∈ range k, ((i : K) - k) * V = 1 := by
    have h : Σ i ∈ range k, ((i : K) - k) * V = Σ i ∈ range k, x i :=
      sum_congr rfl fun i hi => (hx i (mem_range.1 hi)).symm
    rw [h, hxy.1.2]
  rw [← sum_mul, sum_range_cast_sub] at hsum
  refine ⟨V, ⟨fun j hj => ?_, fun i hi => (hrow i (mem_range.1 hi)).le⟩,
    ?_⟩
  · have h := hxy.2.2 0 (mem_range.2 hk) j hj
    rwa [hrow 0 hk] at h
  · rw [abs_of_neg hV0, eq_div_iff (by nlinarith)]
    linarith

set_option maxHeartbeats 400000 in
/-- **Table 1 gives the value of the run.** At every state before the
    last, every equilibrium of
    that subgame has a value, and its magnitude is `tableValue M t`.
    With `margin_eq_min`, this
    ties `margin`, and hence the thresholds of `double_oracle_accuracy`,
    to the table. -/
theorem value_run (hM : 2 ≤ M) {t : ℕ} (ht : t < M + N - 2) (x y : ℕ →
    K)
    (hxy : IsEq M (run M N t).1 (run M N t).2 x y) :
    ∃ v : K, IsValue M (run M N t).1 (run M N t).2 x y v ∧ |v| =
    tableValue M t := by
  have hM2 : (2 : K) ≤ M := by exact_mod_cast hM
  have hM0 : (0 : K) < M := by linarith
  by_cases ht2 : t + 2 < 2 * M
  · -- the staircase states
    rcases Nat.even_or_odd' t with ⟨k, rfl | rfl⟩
    · -- `S_{k+1}`
      have e1 : run M N (2 * k) = (range (k + 1), range (k + 1)) := by
        simp only [run, if_pos ht2, show 2 * k / 2 = k by omega, show (2
    * k + 1) / 2 = k by omega]
      have htab : tableValue (K := K) M (2 * k) = 2 / (((k : K) + 1) *
    (((k : K) + 1) + 1)) := by
        rw [tableValue, if_pos ht2, show 2 * k / 2 = k by omega]
        push_cast
        ring
      rw [e1] at hxy ⊢
      obtain ⟨v, hv, habs⟩ := value_S_eq (K := K) (k := k + 1) (by omega)
    (by omega) x y hxy
      refine ⟨v, hv, ?_⟩
      rw [htab, habs]
      push_cast
      ring
    · -- `T_{k+1}`
      have e1 : run M N (2 * k + 1) = (range (k + 1), range (k + 1 + 1))
    := by
        simp only [run, if_pos ht2, show (2 * k + 1) / 2 = k by omega,
          show (2 * k + 1 + 1) / 2 = k + 1 by omega]
      have htab : tableValue (K := K) M (2 * k + 1)
          = 2 / (((k : K) + 1) * (((k : K) + 1) + 1)) := by
        rw [tableValue, if_pos ht2, show (2 * k + 1) / 2 = k by omega]
        push_cast
        ring
      rw [e1] at hxy ⊢
      obtain ⟨v, hv, habs⟩ := value_T_eq (K := K) (k := k + 1) (by omega)
    (by omega) x y hxy
      refine ⟨v, hv, ?_⟩
      rw [htab, habs]
```

```
      push_cast
      ring
  · -- the corridor states, where the row player holds every row
    have hR : (run M N t).1 = range M := by simp only [run, if_neg ht2]
    have hC : (run M N t).2 = range (t + 2 - M) := by
      simp only [run, if_neg ht2, min_eq_left (show t + 2 - M ≤ N by
    omega)]
    rw [hR, hC] at hxy ⊢
    rcases eq_or_lt_of_le (show 2 * M - 2 ≤ t by omega) with heq | hlt
    · -- `C_M = S_M`
      rw [show t + 2 - M = M by omega] at hxy ⊢
      have htab : tableValue (K := K) M t = 2 / ((M : K) * ((M : K) +
    1)) := by
        rw [tableValue, if_neg ht2, if_pos (show t = 2 * M - 2 by omega)]
      obtain ⟨v, hv, habs⟩ := value_S_eq (K := K) (k := M) (by omega)
    le_rfl x y hxy
      exact ⟨v, hv, by rw [htab, habs]⟩
    · rcases eq_or_lt_of_le (show 2 * M - 1 ≤ t by omega) with heq2 |
    hlt2
      · -- `C_{M+1}`, where column `M` pays exactly the value
        rw [show t + 2 - M = M + 1 by omega] at hxy ⊢
        have htab : tableValue (K := K) M t = 1 / ((M : K) * (8 * (M :
    K) + 1)) := by
          rw [tableValue, if_neg ht2, if_neg (show ¬ t = 2 * M - 2 by
    omega),
            if_pos (show t = 2 * M - 1 by omega)]
        obtain ⟨hcol, -, hpay, -⟩ := (cert_handoff (K := K) hM).2 x y hxy
        refine ⟨1 / ((M : K) * (8 * M + 1)),
          ⟨fun j hj => hcol j (mem_range.1 hj), fun i hi => ?_⟩, ?_⟩
        · have h := hxy.2.2 i hi M (mem_range.2 (by omega))
          rwa [show colPay M (range M) x M = pay M (x 0) (x 1) 4 from
    colPay_add hM x 0,
            hpay] at h
        · rw [htab, abs_of_pos (one_div_pos.2 (mul_pos hM0 (by
    linarith)))]
      · -- the deeper corridor states
        obtain ⟨e, he⟩ : ∃ e, t = 2 * M + e := ⟨t - 2 * M, by omega⟩
        rw [show t + 2 - M = M + e + 2 by omega] at hxy ⊢
        have htab : tableValue (K := K) M t = 1 / corrMass M e := by
          rw [tableValue, if_neg ht2, if_neg (show ¬ t = 2 * M - 2 by
    omega),
            if_neg (show ¬ t = 2 * M - 1 by omega), show t - 2 * M = e
    by omega]
        obtain ⟨hcol, hpay, -⟩ := (cert_corridor (K := K) hM e).2 x y hxy
        refine ⟨1 / corrMass M e, ⟨fun j hj => hcol j (mem_range.1 hj),
    fun i hi => ?_⟩, ?_⟩
        · have h := hxy.2.2 i hi (M + e) (mem_range.2 (by omega))
          rwa [show colPay M (range M) x (M + e) = pay M (x 0) (x 1) (e +
     4) from
            colPay_add hM x e, hpay] at h
        · rw [htab, abs_of_pos (one_div_pos.2 (corrMass_pos hM e))]

/-- A best response against payoffs bounded by an attained `v` pays `v`.
    -/
theorem br_row_eq {C : Finset ℕ} {y : ℕ → K} {v : K} {i : ℕ}
    (hi' : ∀ i' < M, rowPay M C y i' ≤ rowPay M C y i) (hi : i < M)
    (hle : ∀ i' < M, rowPay M C y i' ≤ v) (hex : ∃ i₀ < M, v ≤ rowPay M
    C y i₀) :
    rowPay M C y i = v := by
  obtain ⟨i₀, hi₀, hv⟩ := hex
  exact le_antisymm (hle i hi) (hv.trans (hi' i₀ hi₀))

theorem br_col_eq {R : Finset ℕ} {x : ℕ → K} {u : K} {j : ℕ}
    (hj' : ∀ j' < N, colPay M R x j ≤ colPay M R x j') (hj : j < N)
```

```
    (hge : ∀ j' < N, u ≤ colPay M R x j') (hex : ∃ j₀ < N, colPay M R x
    j₀ ≤ u) :
    colPay M R x j = u := by
  obtain ⟨j₀, hj₀, hu⟩ := hex
  exact le_antisymm ((hj' j₀ hj₀).trans hu) (hge j hj)

/-- At `S_k` with `k < M`, a best-response row pays the value `W = 2 /
    (k (k + 1))` and a
    best-response column pays `-k W`. -/
theorem gap_S (hM : 2 ≤ M) (hk : 0 < k) (hkM : k < M) (hkN : k < N) {x y
    : ℕ → K}
    (h : IsEq M (range k) (range k) x y) {i j : ℕ} (hi : i < M)
    (hi' : ∀ i' < M, rowPay M (range k) y i' ≤ rowPay M (range k) y i)
    (hj' : ∀ j' < N, colPay M (range k) x j ≤ colPay M (range k) x j') :
    rowPay M (range k) y i - colPay M (range k) x j = ((k : K) + 1) * (2
    / (k * (k + 1))) := by
  obtain ⟨hx, -⟩ := pin_S hk hkM.le h
  obtain ⟨V, -, hxV, -, hr⟩ := (cert_S (K := K) hk hkM.le).2 x y h
  have hVW : V = 2 / (k * (k + 1)) := by
    have h0 := hxV 0 hk
    rw [hx 0 hk] at h0
    simpa using h0.symm
  have hW : (0 : K) < 2 / (k * (k + 1)) := by positivity
  have hri : rowPay M (range k) y i = 2 / (k * (k + 1)) := by
    refine br_row_eq hi' hi (fun i' hi'M => ?_) ⟨0, by omega, by rw [hr
    0 hk, hVW]⟩
    by_cases h1 : i' < k
    · rw [hr i' h1, hVW]
    · rw [rowPay_stair hkM.le, if_neg h1, if_neg (by omega), sub_zero];
    exact hW.le
  have hck : colPay M (range k) x k = -((k : K) * (2 / (k * (k + 1)))) :=
     by
    obtain ⟨m, rfl⟩ : ∃ m, k = m + 1 := ⟨k - 1, by omega⟩
    rw [colPay_succ hkM, if_neg (by omega), if_pos (by omega), hx m (by
    omega)]
    push_cast
    ring
  have hcj : colPay M (range k) x j = colPay M (range k) x k := by
    rcases eq_or_ne j k with rfl | hne
    · rfl
    · exact absurd (hj' k hkN) (not_le.2 (cert_S_out hM hk hkM h hne))
  rw [hri, hcj, hck]
  ring

/-- At `T_k`, a best-response row pays `k W`, with `W = 2 / (k (k +
    1))`, and a best-response
    column pays the value `-W`. -/
theorem gap_T (hk : 0 < k) (hkM : k < M) (hkN : k < N) {x y : ℕ → K}
    (h : IsEq M (range k) (range (k + 1)) x y) {i j : ℕ} (hi : i < M)
    (hi' : ∀ i' < M, rowPay M (range (k + 1)) y i' ≤ rowPay M (range (k +
     1)) y i) (hj : j < N)
    (hj' : ∀ j' < N, colPay M (range k) x j ≤ colPay M (range k) x j') :
    rowPay M (range (k + 1)) y i - colPay M (range k) x j
      = ((k : K) + 1) * (2 / (k * (k + 1))) := by
  obtain ⟨hx, hy⟩ := pin_T hk hkM h
  have hW : (0 : K) < 2 / (k * (k + 1)) := by positivity
  have hk1 : (1 : K) ≤ k := by exact_mod_cast hk
  have hrow : ∀ i' < k, rowPay M (range (k + 1)) y i' = -(2 / (k * (k +
    1))) := fun i' h1 => by
    rw [rowPay_stair (by omega), if_pos (by omega), if_pos (by omega),
    hy i' (by omega),
      hy (i' + 1) (by omega)]
    push_cast
    ring
  have hri : rowPay M (range (k + 1)) y i = k * (2 / (k * (k + 1))) := by
```

```
    have hrk : rowPay M (range (k + 1)) y k = k * (2 / (k * (k + 1))) :=
    by
      rw [rowPay_stair (by omega), if_pos (by omega), if_neg (by omega),
    sub_zero, hy k (by omega)]
    refine br_row_eq hi' hi (fun i' hi'M => ?_) ⟨k, hkM, hrk.ge⟩
    rcases lt_trichotomy i' k with h1 | rfl | h1
    · rw [hrow i' h1]; nlinarith
    · exact hrk.le
    · rw [rowPay_stair (by omega), if_neg (by omega), if_neg (by omega),
    sub_zero]; positivity
  have hck : colPay M (range k) x k = -(2 / (k * (k + 1))) := by
    obtain ⟨m, rfl⟩ : ∃ m, k = m + 1 := ⟨k - 1, by omega⟩
    rw [colPay_succ hkM, if_neg (by omega), if_pos (by omega), hx m (by
    omega)]
    push_cast
    ring
  have hcj : colPay M (range k) x j = -(2 / (k * (k + 1))) := by
    refine br_col_eq hj' hj (fun j' hj'N => ?_) ⟨k, hkN, hck.le⟩
    rcases le_or_gt j' k with h1 | h1
    · have := h.2.2 0 (mem_range.2 hk) j' (mem_range.2 (by omega))
      rwa [hrow 0 hk] at this
    · rw [← hck]; exact (cert_T_out hk hkM h h1).le
  rw [hri, hcj]
  ring

/-- At a corridor state, a best-response row pays the value `V`
    certified by `x`. -/
theorem br_row_corr {C : Finset ℕ} {x y : ℕ → K} {V : K} {i j₀ : ℕ}
    (h : IsEq M (range M) C x y) (hu : ∀ j ∈ C, V ≤ colPay M (range M) x
    j) (hj₀ : j₀ ∈ C)
    (hV : colPay M (range M) x j₀ = V) (hi : i < M)
    (hi' : ∀ i' < M, rowPay M C y i' ≤ rowPay M C y i) (hM : 0 < M) :
    rowPay M C y i = V := by
  obtain ⟨i₀, hi₀, hv⟩ := exists_row_ge h.1 h.2.1 hu
    (nonempty_range_iff.2 hM.ne')
  exact br_row_eq hi' hi (fun i' h' => by rw [← hV]; exact h.2.2 i'
    (mem_range.2 h') j₀ hj₀)
    ⟨i₀, mem_range.1 hi₀, hv⟩

/-- At `C_M = S_M`, with `M < N`, a best-response row pays the value `W =
     2 / (M (M + 1))`, and a
    best-response column pays `-W / (15 M)`. -/
theorem gap_CM (hM : 2 ≤ M) (hMN : M < N) {x y : ℕ → K} (h : IsEq M
    (range M) (range M) x y)
    {i j : ℕ} (hi : i < M) (hj : j < N)
    (hj' : ∀ j' < N, colPay M (range M) x j ≤ colPay M (range M) x j') :
    rowPay M (range M) y i - colPay M (range M) x j
      = 2 / (M * (M + 1)) + 2 / (M * (M + 1)) / (15 * M) := by
  obtain ⟨hx, -⟩ := pin_S (by omega) le_rfl h
  obtain ⟨V, -, hxV, hc, hr⟩ := (cert_S (K := K) (by omega) le_rfl).2 x y
    h
  have hVW : V = 2 / (M * (M + 1)) := by
    have h0 := hxV 0 (by omega)
    rw [hx 0 (by omega)] at h0
    simpa using h0.symm
  have hW : (0 : K) < 2 / (M * (M + 1)) := by positivity
  have hM0 : (0 : K) < M := by exact_mod_cast (show 0 < M by omega)
  have hri : rowPay M (range M) y i = 2 / (M * (M + 1)) := by rw [hr i
    hi, hVW]
  -- the corridor column `M` pays `-W / (15 M)`, less than every later
    one
  have hcol : ∀ t, colPay M (range M) x (M + t) = pay M (x 0) (x 1) (t +
    4) :=
    fun t => colPay_add hM x t
```

```
  have h01 : x 0 = 2 / (M * (M + 1)) ∧ x 1 = 2 * (2 / (M * (M + 1))) :=
    by
    rw [hx 0 (by omega), hx 1 (by omega)]; push_cast; constructor <;>
    ring
  have hp4 : pay M (x 0) (x 1) 4 = -(2 / (M * (M + 1)) / (15 * M)) := by
    have h15 := pay_four (M := M) (by omega) (x 0) (x 1)
    rw [neg_div', eq_div_iff (by positivity)]
    linarith [h01.1, h01.2]
  obtain ⟨e3, e4⟩ := g_three (x 0) (x 1)
  obtain ⟨-, hmin, -⟩ := alternation (M := M) (m := 3) (a := x 0) (b := x
    1) (by omega) le_rfl
    (by rw [e3, h01.1, h01.2]; linarith) (by rw [e3, e4, h01.1, h01.2];
    linarith)
  have hcj : colPay M (range M) x j = -(2 / (M * (M + 1)) / (15 * M)) :=
    by
    have hcolM : colPay M (range M) x M = pay M (x 0) (x 1) 4 := by
      simpa using hcol 0
    refine br_col_eq hj' hj (fun j' hj'N => ?_) ⟨M, hMN, by rw [hcolM];
    exact hp4.le⟩
    rcases lt_or_ge j' M with h1 | h1
    · rw [hc j' h1, hVW]
      have : (0 : K) < 2 / (M * (M + 1)) / (15 * M) := by positivity
      linarith
    · obtain ⟨t, rfl⟩ : ∃ t, j' = M + t := ⟨j' - M, by omega⟩
      rw [hcol, ← hp4]
      rcases Nat.eq_zero_or_pos t with rfl | ht
      · exact le_rfl
      · exact (hmin (t + 4) (by omega)).le
  rw [hri, hcj]
  ring

/-- At `C_{M+1}`, with `M + 1 < N`, a best-response row pays the value
    `V = 1 / (M (8 M + 1))`,
    and a best-response column pays `-(45 M - 1) V / (80 M)`. -/
theorem gap_H (hM : 2 ≤ M) (hMN : M + 1 < N) {x y : ℕ → K}
    (h : IsEq M (range M) (range (M + 1)) x y) {i j : ℕ} (hi : i < M)
    (hi' : ∀ i' < M, rowPay M (range (M + 1)) y i' ≤ rowPay M (range (M +
     1)) y i)
    (hj : j < N) (hj' : ∀ j' < N, colPay M (range M) x j ≤ colPay M
    (range M) x j') :
    rowPay M (range (M + 1)) y i - colPay M (range M) x j
      = 1 / (M * (8 * M + 1)) + (45 * M - 1) * (1 / (M * (8 * M + 1))) /
    (80 * M) := by
  obtain ⟨hge, h1, -, -⟩ := (cert_handoff (K := K) hM).2 x y h
  obtain ⟨hx, -⟩ := pin_H hM h
  obtain ⟨V, hV_def⟩ : ∃ V : K, V = 1 / ((M : K) * (8 * M + 1)) := ⟨_,
    rfl⟩
  rw [← hV_def] at hge h1 hx ⊢
  have hM0 : (0 : K) < M := by exact_mod_cast (show 0 < M by omega)
  have hM2 : (2 : K) ≤ M := by exact_mod_cast hM
  have hV0 : 0 < V := by rw [hV_def]; positivity
  have hcol1 : colPay M (range M) x 1 = V := by
    rw [colPay_succ (show 0 + 1 < M by omega), if_pos (by omega), if_pos
    (by omega)]; exact h1
  have hri : rowPay M (range (M + 1)) y i = V :=
    br_row_corr h (fun j hj => hge j (mem_range.1 hj)) (mem_range.2 (by
    omega)) hcol1 hi hi'
      (by omega)
  -- the new column pays `-(45 M - 1) V / (80 M)`, less than every later
    one
  have hcol : ∀ t, colPay M (range M) x (M + t) = pay M (x 0) (x 1) (t +
    4) :=
    fun t => colPay_add hM x t
  have hx0 := hx 0 (by omega)
  have hx1 := hx 1 (by omega)
```

```
  push_cast at hx0 hx1
  have hp5 : pay M (x 0) (x 1) 5 = -((45 * M - 1) * V / (80 * M)) := by
    rw [pay, s5, s6, hx0, hx1]
    field_simp
    ring
  have hg4 : g (x 0) (x 1) 4 = 15 * M * V := by
    obtain ⟨-, f4, f5, -⟩ := fib_vals
    simp only [g, f4, f5, hx0, hx1]; push_cast; ring
  have hg5 : g (x 0) (x 1) 5 = -((45 * M - 1) * V / 2) := by
    obtain ⟨-, -, f5, f6⟩ := fib_vals
    simp only [g, f5, f6, hx0, hx1]; push_cast; ring
  obtain ⟨-, hmin, -⟩ := alternation (M := M) (m := 4) (a := x 0) (b := x
    1) (by omega)
    (by omega) (by rw [hg4]; positivity) (by rw [hg4, hg5]; nlinarith)
  have hcj : colPay M (range M) x j = -((45 * M - 1) * V / (80 * M)) :=
    by
    refine br_col_eq hj' hj (fun j' hj'N => ?_)
      ⟨M + 1, hMN, by rw [hcol 1]; exact hp5.le⟩
    rcases lt_or_ge j' (M + 1) with h1 | h1
    · have := hge j' h1
      have : (0 : K) < (45 * M - 1) * V / (80 * M) := div_pos (mul_pos
    (by linarith) hV0)
        (by linarith)
      linarith
    · obtain ⟨t, rfl⟩ : ∃ t, j' = M + t := ⟨j' - M, by omega⟩
      rw [hcol, ← hp5]
      rcases eq_or_lt_of_le (show 1 ≤ t by omega) with rfl | ht
      · exact le_rfl
      · exact (hmin (t + 4) (by omega)).le
  rw [hri, hcj]
  ring

/-- At `C_{M+e+2}`, with `M + e + 2 < N`, a best-response row pays the
    value `V = 1 / corrMass`,
    and a best-response column pays `-corrMargin M e`. -/
theorem gap_C (hM : 2 ≤ M) (e : ℕ) (hN : M + e + 2 < N) {x y : ℕ → K}
    (h : IsEq M (range M) (range (M + e + 2)) x y) {i j : ℕ} (hi : i < M)
    (hi' : ∀ i' < M, rowPay M (range (M + e + 2)) y i' ≤ rowPay M (range
    (M + e + 2)) y i)
    (hj : j < N) (hj' : ∀ j' < N, colPay M (range M) x j ≤ colPay M
    (range M) x j') :
    rowPay M (range (M + e + 2)) y i - colPay M (range M) x j
      = 1 / corrMass M e + corrMargin M e := by
  obtain ⟨hge, h4, h5, -⟩ := (cert_corridor (K := K) hM e).2 x y h
  have hV0 : (0 : K) < 1 / corrMass M e := one_div_pos.2 (corrMass_pos
    hM e)
  have hcol : ∀ t, colPay M (range M) x (M + t) = pay M (x 0) (x 1) (t +
    4) :=
    fun t => colPay_add hM x t
  have hri : rowPay M (range (M + e + 2)) y i = 1 / corrMass M e :=
    br_row_corr (j₀ := M + e) h (fun j hj => hge j (mem_range.1 hj))
    (mem_range.2 (by omega))
      (by rw [hcol e]; exact h4) hi hi' (by omega)
  obtain ⟨hneg, hmin, -⟩ := pay_above (by omega) hV0 (by omega) h4 h5
  -- the new column pays `-corrMargin M e`, by the identity of
    `pay_three`
  have hp6 : pay M (x 0) (x 1) (e + 6) = -corrMargin M e := by
    have hid := pay_three (M := M) (by omega) (x 0) (x 1) e rfl rfl
    rw [h4, show e + 5 = e + 4 + 1 from rfl, h5] at hid
    have hp : (0 : K) < (fib (e + 4) : K) := by exact_mod_cast fib_pos.2
    (by omega)
    have hq : (0 : K) < (fib (e + 5) : K) := by exact_mod_cast fib_pos.2
    (by omega)
    have hmass := (corrMass_pos (K := K) hM e).ne'
    rw [corrMargin]
```

```
    field_simp at hid ⊢
    linear_combination hid
  have hcj : colPay M (range M) x j = -corrMargin M e := by
    refine br_col_eq hj' hj (fun j' hj'N => ?_)
      ⟨M + (e + 2), by omega, by rw [hcol]; exact (show e + 2 + 4 = e +
    6 from rfl) ▷ hp6.le⟩
    rcases lt_or_ge j' (M + e + 2) with h1 | h1
    · have := hge j' h1
      have hneg' : pay M (x 0) (x 1) (e + 6) < 0 := hneg
      linarith
    · obtain ⟨t, rfl⟩ : ∃ t, j' = M + t := ⟨j' - M, by omega⟩
      rw [hcol, ← hp6]
      rcases eq_or_lt_of_le (show e + 2 ≤ t by omega) with rfl | ht
      · exact le_rfl
      · exact (hmin (t + 4) (by omega)).le
  rw [hri, hcj]
  ring

set_option maxHeartbeats 400000 in
/-- **Table 1 gives the duality gap of the run.** At every state before
    the last, for every
    equilibrium of the subgame and every pair of best responses `i` and
    `j`, the gap
    `w_i - u_j` is `|v_t| + m_t`, the value plus the margin of Table 1.
    -/
theorem gap_run (hM : 2 ≤ M) (hMN : M ≤ N) {t : ℕ} (ht : t < M + N - 2)
    {x y : ℕ → K}
    (h : IsEq M (run M N t).1 (run M N t).2 x y) {i j : ℕ} (hi : i < M)
    (hi' : ∀ i' < M, rowPay M (run M N t).2 y i' ≤ rowPay M (run M N
    t).2 y i) (hj : j < N)
    (hj' : ∀ j' < N, colPay M (run M N t).1 x j ≤ colPay M (run M N t).1
    x j') :
    rowPay M (run M N t).2 y i - colPay M (run M N t).1 x j
      = tableValue M t + tableMargin M t := by
  have hM0 : (0 : K) < M := by exact_mod_cast (show 0 < M by omega)
  by_cases ht2 : t + 2 < 2 * M
  · rcases Nat.even_or_odd' t with ⟨k, rfl | rfl⟩
    · rw [run_even ht2] at h hi' hj' ⊢
      rw [gap_S hM (by omega) (by omega) (by omega) h hi hi' hj',
    tableValue, tableMargin,
        if_pos ht2, if_pos ht2, show 2 * k / 2 = k by omega]
      push_cast
      field_simp
      ring
    · rw [run_odd ht2] at h hi' hj' ⊢
      rw [gap_T (by omega) (by omega) (by omega) h hi hi' hj hj',
    tableValue, tableMargin,
        if_pos ht2, if_pos ht2, show (2 * k + 1) / 2 = k by omega]
      push_cast
      field_simp
      ring
  obtain ⟨d, hrun, hd⟩ := run_corr (N := N) hMN ht2
  rw [hrun] at h hi' hj' ⊢
  dsimp only at h hi' hj' ⊢
  rw [min_eq_left (by omega)] at hd
  rcases d with _ | _ | e
  · rw [Nat.add_zero]
    have hv : tableValue (K := K) M t = 2 / (M * (M + 1)) := by
      rw [tableValue, if_neg ht2, if_pos (show t = 2 * M - 2 by omega)]
    have hm : tableMargin (K := K) M t = 2 / (15 * M ^ 2 * (M + 1)) := by
      rw [tableMargin, if_neg ht2, if_pos (show t = 2 * M - 2 by omega)]
    rw [hv, hm, gap_CM hM (by omega) h hi hj hj']
    field_simp
  · have hv : tableValue (K := K) M t = 1 / (M * (8 * M + 1)) := by
      rw [tableValue, if_neg ht2, if_neg (show ¬ t = 2 * M - 2 by omega),
```

```
        if_pos (show t = 2 * M - 1 by omega)]
    have hm : tableMargin (K := K) M t = (45 * M - 1) / (80 * M ^ 2 * (8
    * M + 1)) := by
      rw [tableMargin, if_neg ht2, if_neg (show ¬ t = 2 * M - 2 by
    omega),
        if_pos (show t = 2 * M - 1 by omega)]
    rw [hv, hm, gap_H hM (by omega) h hi hi' hj hj']
    field_simp
  · rw [show M + (e + 1 + 1) = M + e + 2 from rfl] at h hi' ⊢
    have hv : tableValue (K := K) M t = 1 / corrMass M e := by
      rw [tableValue, if_neg ht2, if_neg (show ¬ t = 2 * M - 2 by omega),
        if_neg (show ¬ t = 2 * M - 1 by omega), show t - 2 * M = e by
    omega]
    have hm : tableMargin (K := K) M t = corrMargin M e := by
      rw [tableMargin, if_neg ht2, if_neg (show ¬ t = 2 * M - 2 by
    omega),
        if_neg (show ¬ t = 2 * M - 1 by omega), show t - 2 * M = e by
    omega]
    rw [hv, hm, gap_C hM e (by omega) h hi hi' hj hj']

/-- `F(n) ≥ n` from `n = 5` on. -/
theorem le_fib_five (hn : 5 ≤ n) : n ≤ fib n := by
  induction n, hn using Nat.le_induction with
  | base => decide
  | succ n hn ih =>
    have h1 : fib (n + 1) = fib (n - 1) + fib n := by
      rw [show n + 1 = n - 1 + 2 by omega, fib_add_two, show n - 1 + 1 =
    n by omega]
    have h2 : 0 < fib (n - 1) := fib_pos.2 (by omega)
    omega

set_option maxHeartbeats 400000 in
/-- **The gap decays at least like `4 / (t + 1)`.** Table 1's gap `|v_t|
    + m_t` after `t`
    iterations is at most `4 / (t + 1)`, with equality at the states
    `T_k`. -/
theorem table_gap_le (hM : 2 ≤ M) {t : ℕ} :
    tableValue (K := K) M t + tableMargin M t ≤ 4 / (t + 1) := by
  have hM2 : (2 : K) ≤ M := by exact_mod_cast hM
  have hM0 : (0 : K) < M := by linarith
  have hM3 : (0 : K) < (M : K) ^ 3 := pow_pos hM0 3
  by_cases ht2 : t + 2 < 2 * M
  · -- the staircase states, where the gap is `2 / (k + 1)`
    rw [tableValue, tableMargin, if_pos ht2, if_pos ht2]
    have hk : (0 : K) ≤ ((t / 2 : ℕ) : K) := Nat.cast_nonneg _
    have e : 2 / ((((t / 2 + 1 : ℕ) : K)) * (((t / 2 + 1 : ℕ) : K) + 1))
        + 2 / (((t / 2 + 1 : ℕ) : K) + 1) = 2 / (((t / 2 : ℕ) : K) + 1)
    := by
      push_cast; field_simp; ring
    have ht : (t : K) ≤ 2 * ((t / 2 : ℕ) : K) + 1 := by
      exact_mod_cast (show t ≤ 2 * (t / 2) + 1 by omega)
    rw [e, div_le_div_iff₀ (by positivity) (by positivity)]
    linarith
  rw [tableValue, tableMargin, if_neg ht2, if_neg ht2]
  split_ifs with h1 h2
  · -- `C_M`, at `t = 2 M - 2`
    rw [h1, show ((2 * M - 2 : ℕ) : K) = 2 * M - 2 by
      rw [Nat.cast_sub (by omega)]; push_cast; ring]
    have e : 2 / ((M : K) * (M + 1)) + 2 / (15 * (M : K) ^ 2 * (M + 1))
        = 2 * (15 * M + 1) / (15 * (M : K) ^ 2 * (M + 1)) := by
      field_simp
    have hden : (0 : K) < 15 * (M : K) ^ 2 * (M + 1) :=
      mul_pos (mul_pos (by norm_num) (pow_pos hM0 2)) (by linarith)
    rw [e, div_le_div_iff₀ hden (by linarith)]
    linarith [hM3]
```

```
  · -- `C_{M+1}`, at `t = 2 M - 1`
    rw [h2, show ((2 * M - 1 : ℕ) : K) = 2 * M - 1 by
      rw [Nat.cast_sub (by omega)]; push_cast; ring]
    have e : 1 / ((M : K) * (8 * M + 1)) + (45 * M - 1) / (80 * (M : K)
    ^ 2 * (8 * M + 1))
        = (125 * M - 1) / (80 * (M : K) ^ 2 * (8 * M + 1)) := by
      field_simp; ring
    have hden : (0 : K) < 80 * (M : K) ^ 2 * (8 * M + 1) :=
      mul_pos (mul_pos (by norm_num) (pow_pos hM0 2)) (by linarith)
    rw [e, div_le_div_iff₀ hden (by linarith)]
    linarith [hM3]
  · -- deeper, where the gap is at most twice the value, and the mass is
    at least `M F(K)`
    obtain ⟨e, rfl⟩ : ∃ e, t = 2 * M + e := ⟨t - 2 * M, by omega⟩
    rw [show 2 * M + e - 2 * M = e by omega]
    obtain ⟨p, hp_def⟩ : ∃ p : K, p = fib (e + 4) := ⟨_, rfl⟩
    obtain ⟨q, hq_def⟩ : ∃ q : K, q = fib (e + 5) := ⟨_, rfl⟩
    have hp : (0 : K) ≤ p := by rw [hp_def]; positivity
    have hq : (e : K) + 5 ≤ q := by rw [hq_def]; exact_mod_cast
    le_fib_five (by omega)
    have he : (0 : K) ≤ e := Nat.cast_nonneg e
    have hmass := corrMass_pos (K := K) hM e
    -- `corrMargin ≤ 1 / corrMass`, since `F(K)^2 ≤ F(K+1) F(K+2)`
    have hq0 : (0 : K) ≤ q := by linarith
    have hsq : q ^ 2 ≤ (p + q) * (p + 2 * q) := by
      linarith [mul_nonneg hp hp, mul_nonneg hp hq0, mul_nonneg hq0 hq0]
    have hmV : corrMargin (K := K) M e ≤ 1 / corrMass M e := by
      rw [corrMargin, ← hp_def, ← hq_def,
        div_le_div_iff₀ (mul_pos (mul_pos hmass (by linarith)) (by
    linarith)) hmass]
      linarith [mul_le_mul_of_nonneg_left hsq hmass.le]
    -- `corrMass ≥ M F(K)`, and `2 M F(K) ≥ 2 M + e + 1`
    have h1 : (1 : K) ≤ (M : K) * (p + q) ^ 2 - q ^ 2 := by
      have h2 : 2 * (p + q) ^ 2 ≤ (M : K) * (p + q) ^ 2 :=
        mul_le_mul_of_nonneg_right hM2 (sq_nonneg _)
      have h3 : (5 : K) * 5 ≤ q * q := mul_le_mul (by linarith) (by
    linarith) (by norm_num)
        (by linarith)
      linarith [mul_nonneg hp hp, mul_nonneg hp (show (0 : K) ≤ q by
    linarith)]
    have hMq : (M : K) * q ≤ corrMass M e := by
      have h3 := le_mul_of_one_le_right (mul_nonneg hM0.le (by linarith
    : (0 : K) ≤ q)) h1
      have h4 : (0 : K) ≤ ((M : K) - 1) * ((M : K) - 2) := mul_nonneg
    (by linarith) (by linarith)
      simp only [corrMass, ← hp_def, ← hq_def]
      linarith
    have hbig : ((2 * M + e : ℕ) : K) + 1 ≤ 2 * corrMass M e := by
      push_cast
      linarith [mul_le_mul_of_nonneg_left hq hM0.le,
    mul_le_mul_of_nonneg_right hM2 he]
    calc 1 / corrMass M e + corrMargin M e ≤ 2 / corrMass M e := by
          rw [show (2 : K) / corrMass M e = 1 / corrMass M e + 1 /
    corrMass M e by ring]
          linarith
      _ ≤ 4 / (((2 * M + e : ℕ) : K) + 1) := by
          rw [div_le_div_iff₀ hmass (by positivity)]
          linarith

/-- So after `t` iterations the duality gap of the run is at most `4 /
    (t + 1)`, for every
    equilibrium and every pair of best responses, and double oracle
    finds a `τ`-equilibrium of
    `A` within `4 / τ` iterations. -/
```

```
theorem gap_run_le (hM : 2 ≤ M) (hMN : M ≤ N) {t : ℕ} (ht : t < M + N -
    2) {x y : ℕ → K}
    (h : IsEq M (run M N t).1 (run M N t).2 x y) {i j : ℕ} (hi : i < M)
    (hi' : ∀ i' < M, rowPay M (run M N t).2 y i' ≤ rowPay M (run M N
    t).2 y i) (hj : j < N)
    (hj' : ∀ j' < N, colPay M (run M N t).1 x j ≤ colPay M (run M N t).1
    x j') :
    rowPay M (run M N t).2 y i - colPay M (run M N t).1 x j ≤ 4 / (t +
    1) := by
  rw [gap_run hM hMN ht h hi hi' hj hj']
  exact table_gap_le hM

set_option maxHeartbeats 400000 in
/-- **Each state of the run has the next one as its unique approximate
    successor**, as soon as
    `δ` and `εC` stay below a fixed fraction of `margin` at every state
    before the last, and
    `εR` below a fixed fraction of it at the staircase states, which are
    the only ones where a
    row is added. -/
theorem double_oracle_accuracy_step (hM : 2 ≤ M) (hMN : M ≤ N) {δ εR εC
    : K} (hδ : 0 ≤ δ)
    (hεR : 0 ≤ εR) (hεC : 0 ≤ εC) (h1 : ∀ t < M + N - 2, 64 * δ ≤ margin
    M t)
    (h2R : ∀ t < M + N - 2, t + 2 < 2 * M → 64 * εR ≤ margin M t)
    (h2C : ∀ t < M + N - 2, 64 * εC ≤ margin M t) (t : ℕ) (q : Finset ℕ
    × Finset ℕ) :
    StepApprox M N δ εR εC (run M N t) q ↔ q = run M N (t + 1) := by
  by_cases ht : t + 2 < 2 * M
  · -- the staircase phase, where `margin` is the value
    rcases Nat.even_or_odd' t with ⟨k, rfl | rfl⟩
    · have e1 : run M N (2 * k) = (range (k + 1), range (k + 1)) := by
        simp only [run, if_pos ht, show 2 * k / 2 = k by omega, show (2 *
     k + 1) / 2 = k by omega]
      have e2 : run M N (2 * k + 1) = (range (k + 1), range (k + 1 + 1))
    := by
        simp only [run, if_pos (show 2 * k + 1 + 2 < 2 * M by omega),
          show (2 * k + 1) / 2 = k by omega, show (2 * k + 1 + 1) / 2 =
    k + 1 by omega]
      have hmar : margin M (2 * k) = 2 / (((k : K) + 1) * (((k : K) + 1)
    + 1)) := by
        rw [margin, if_pos ht, show 2 * k / 2 = k by omega]
        push_cast
        ring
      rw [e1, e2]
      refine staircase_right_gen hM hMN (by omega) (by omega) hδ hεR hεC
    ?_ ?_ ?_ q
      · have h3 := h1 (2 * k) (by omega); rw [hmar] at h3; push_cast;
    exact h3
      · have h3 := h2R (2 * k) (by omega) (by omega); rw [hmar] at h3;
    push_cast; exact h3
      · have h3 := h2C (2 * k) (by omega); rw [hmar] at h3; push_cast;
    exact h3
    · have e1 : run M N (2 * k + 1) = (range (k + 1), range (k + 1 + 1))
    := by
        simp only [run, if_pos ht, show (2 * k + 1) / 2 = k by omega,
          show (2 * k + 1 + 1) / 2 = k + 1 by omega]
      have e2 : run M N (2 * k + 1 + 1) = (range (k + 1 + 1), range (k +
    1 + 1)) := by
        simp only [run]
        split_ifs with h
        · rw [show (2 * k + 1 + 1) / 2 = k + 1 by omega,
            show (2 * k + 1 + 1 + 1) / 2 = k + 1 by omega]
        · rw [show 2 * k + 1 + 1 + 2 - M = M by omega, min_eq_left hMN,
    show M = k + 1 + 1 by omega]
```

```
      have hmar : margin M (2 * k + 1) = 2 / (((k : K) + 1) * (((k : K) +
    1) + 1)) := by
        rw [margin, if_pos ht, show (2 * k + 1) / 2 = k by omega]
        push_cast
        ring
      rw [e1, e2]
      refine staircase_down_gen hM hMN (by omega) (by omega) hδ hεR hεC
    ?_ ?_ ?_ q
      · have h3 := h1 (2 * k + 1) (by omega); rw [hmar] at h3; push_cast;
    exact h3
      · have h3 := h2R (2 * k + 1) (by omega) (by omega); rw [hmar] at
    h3; push_cast; exact h3
      · have h3 := h2C (2 * k + 1) (by omega); rw [hmar] at h3;
    push_cast; exact h3
  · -- the corridor phase
    have hrun : ∀ t, ¬ t + 2 < 2 * M → run M N t = (range M, range (min
    (t + 2 - M) N)) :=
      fun t ht => if_neg ht
    rw [hrun t ht, hrun (t + 1) (by omega)]
    rcases le_or_gt N (t + 2 - M) with hN | hN
    · -- the full game, which is its own successor
      rw [min_eq_right hN, min_eq_right (by omega)]
      obtain ⟨d, rfl⟩ : ∃ d, N = M + d := ⟨N - M, by omega⟩
      obtain ⟨x, y, hxy⟩ : ∃ x y : ℕ → K, IsEq M (range M) (range (M +
    d)) x y := by
        rcases d with _ | _ | e
        · exact (cert_S (by omega) le_rfl).1
        · exact cert_handoff_ex hM
        · exact cert_corridor_ex hM e
      exact stepApprox_iff (by omega) (by omega) hεR hεC ⟨x, y,
    hxy.approx hδ⟩
        (fun x y h i _ hi => insert_eq_of_mem (mem_range.2 hi))
        (fun x y h j _ hj => insert_eq_of_mem (mem_range.2 hj)) q
    · -- a corridor state before the last
      obtain ⟨d, hd⟩ : ∃ d, t + 2 - M = M + d := ⟨t + 2 - 2 * M, by omega⟩
      rw [min_eq_left hN.le, min_eq_left (show t + 1 + 2 - M ≤ N by
    omega),
        show t + 1 + 2 - M = M + d + 1 by omega, hd]
      have hmarg : t < M + N - 2 := by omega
      rcases d with _ | _ | e
      · have hmar : margin M t = 2 / (15 * (M : K) ^ 2 * ((M : K) + 1)) :=
     by
          rw [margin, if_neg ht, if_pos (show t = 2 * M - 2 by omega)]
        exact corridor_M_approx hM (by omega) hδ hεR hεC (hmar ▷ h1 t
    hmarg) (hmar ▷ h2C t hmarg) q
      · have hmar : margin M t
            = (45 * (M : K) - 1) / (80 * (M : K) ^ 2 * (8 * (M : K) +
    1)) := by
          rw [margin, if_neg ht, if_neg (show ¬ t = 2 * M - 2 by omega),
            if_pos (show t = 2 * M - 1 by omega)]
        exact corridor_handoff_approx hM (by omega) hδ hεR hεC (hmar ▷ h1
    t hmarg)
          (hmar ▷ h2C t hmarg) q
      · have hmar : margin (K := K) M t = corrMargin M e := by
          rw [margin, if_neg ht, if_neg (show ¬ t = 2 * M - 2 by omega),
            if_neg (show ¬ t = 2 * M - 1 by omega), show t - 2 * M = e
    by omega]
        exact corridor_deep_approx hM e (by omega) hδ hεR hεC (hmar ▷ h1
    t hmarg)
          (hmar ▷ h2C t hmarg) q

/-- **Accuracy guarantee.** For `2 ≤ M ≤ N`, if `δ` and `εC` stay below
    a fixed fraction of
    `margin`, the smaller of the value and the margin of Table 1, at
    every state before the last,
```

```
    and `εR` below a fixed fraction of
    `2 / (M (M - 1))`, the least value at a staircase state, then the
    runs of double oracle from
    `({0}, {0})` with `δ`-equilibria and approximate best responses are
    exactly `run M N`, so they
    take `M + N - 2` iterations. -/
theorem double_oracle_accuracy (hM : 2 ≤ M) (hMN : M ≤ N) {δ εR εC : K}
    (hδ : 0 ≤ δ)
    (hεR : 0 ≤ εR) (hεC : 0 ≤ εC) (h1 : ∀ t < M + N - 2, 64 * δ ≤ margin
    M t)
    (h2R : 64 * εR ≤ 2 / ((M : K) * ((M : K) - 1)))
    (h2C : ∀ t < M + N - 2, 64 * εC ≤ margin M t) (s : ℕ → Finset ℕ ×
    Finset ℕ) :
    (s 0 = ({0}, {0}) ∧ ∀ t, StepApprox M N δ εR εC (s t) (s (t + 1))) ↔
    s = run M N := by
  -- the row tolerance is needed only at the staircase states, where
    `margin` is the value
  have h2S : ∀ t < M + N - 2, t + 2 < 2 * M → 64 * εR ≤ margin M t := by
    intro t _ ht
    have hM2 : (2 : K) ≤ M := by exact_mod_cast hM
    have hk : ((t / 2 + 1 : ℕ) : K) + 1 ≤ M := by
      exact_mod_cast (show t / 2 + 1 + 1 ≤ M by omega)
    have hk1 : (1 : K) ≤ ((t / 2 + 1 : ℕ) : K) := by
      exact_mod_cast (show 1 ≤ t / 2 + 1 by omega)
    rw [margin, if_pos ht]
    refine h2R.trans ?_
    rw [div_le_div_iff₀ (by nlinarith) (by nlinarith)]
    nlinarith
  have h0 : run M N 0 = ({0}, {0}) := by simp [run, show 2 < 2 * M by
    omega]
  refine ⟨fun ⟨hs0, hs⟩ => funext fun t => ?_, ?_⟩
  · induction t with
    | zero => rw [hs0, h0]
    | succ t ih =>
      exact (double_oracle_accuracy_step hM hMN hδ hεR hεC h1 h2S h2C t
    _).1 (ih ▷ hs t)
  · rintro rfl
    exact ⟨h0, fun t => (double_oracle_accuracy_step hM hMN hδ hεR hεC h1
    h2S h2C t _).2 rfl⟩

/-- `margin` is positive at every state. -/
theorem margin_pos (hM : 2 ≤ M) (t : ℕ) : (0 : K) < margin M t := by
  have hM2 : (2 : K) ≤ M := by exact_mod_cast hM
  rw [margin]
  split_ifs with h1 h2 h3
  · positivity
  · positivity
  · exact div_pos (by linarith) (by positivity)
  · rw [corrMargin]
    have hp : (0 : K) < (fib (t - 2 * M + 4) : K) := by exact_mod_cast
    fib_pos.2 (by omega)
    have hq : (0 : K) < (fib (t - 2 * M + 5) : K) := by exact_mod_cast
    fib_pos.2 (by omega)
    exact div_pos (by positivity)
      (by have := corrMass_pos (K := K) hM (t - 2 * M); positivity)

set_option maxHeartbeats 400000 in
/-- The corridor margin at `C_{M+e+2}` is at least `2 / (9 M^2
    F(K+1)^3)`, where `F(K+1) =
    fib (e + 6)`: the mass is at most `(3/2) M^2 F(K) F(K+1)^2`, since
    `F(K) F(K+1)^2 ≥ 1`, and
    `F(K+2) ≤ 3 F(K)`. -/
theorem corrMargin_ge (hM : 2 ≤ M) (e : ℕ) :
    2 / (9 * (M : K) ^ 2 * (fib (e + 6) : K) ^ 3) ≤ corrMargin M e := by
  obtain ⟨p, hp_def⟩ : ∃ t : K, t = (fib (e + 4) : K) := ⟨_, rfl⟩
```

```
  obtain ⟨q, hq_def⟩ : ∃ t : K, t = (fib (e + 5) : K) := ⟨_, rfl⟩
  have hM2 : (2 : K) ≤ M := by exact_mod_cast hM
  have hp1 : (1 : K) ≤ p := by rw [hp_def]; exact_mod_cast fib_pos.2 (by
    omega)
  have hpq : p ≤ q := by rw [hp_def, hq_def]; exact_mod_cast fib_mono
    (by omega)
  have hq1 : (1 : K) ≤ q := hp1.trans hpq
  have hf6 : (fib (e + 6) : K) = p + q := by
    rw [hp_def, hq_def, show e + 6 = e + 4 + 2 from rfl]
    exact_mod_cast fib_add_two
  have hmass : (0 : K) < corrMass M e := corrMass_pos hM e
  -- `1 / V_k ≤ M^2 F(K) F(K+1)^2 + M^2 / 2 ≤ (3/2) M^2 F(K) F(K+1)^2`
  have hA : corrMass M e ≤ 3 / 2 * (M : K) ^ 2 * q * (p + q) ^ 2 := by
    have h4 : (4 : K) ≤ (p + q) ^ 2 := by nlinarith
    have h1 : (1 : K) * 4 ≤ q * (p + q) ^ 2 := mul_le_mul hq1 h4 (by
    norm_num) (by linarith)
    have h2 : (M : K) ^ 2 * 1 ≤ (M : K) ^ 2 * (q * (p + q) ^ 2) :=
      mul_le_mul_of_nonneg_left (by linarith) (by positivity)
    have h3 : (0 : K) ≤ (M : K) * q ^ 3 :=
      mul_nonneg (by linarith) (pow_nonneg (by linarith) 3)
    simp only [corrMass, ← hp_def, ← hq_def]
    linarith [h2, h3]
  -- `F(K+2) ≤ 3 F(K)`, and the two bounds multiply
  have hB : (p + q) * (p + 2 * q) ≤ (p + q) * (3 * q) :=
    mul_le_mul_of_nonneg_left (by linarith) (by linarith)
  have hC : corrMass M e * ((p + q) * (p + 2 * q))
      ≤ 3 / 2 * (M : K) ^ 2 * q * (p + q) ^ 2 * ((p + q) * (3 * q)) :=
    mul_le_mul hA hB (mul_nonneg (by linarith) (by linarith))
      (mul_nonneg (mul_nonneg (by positivity) (by linarith)) (sq_nonneg
    _))
  rw [corrMargin, ← hp_def, ← hq_def, hf6, div_le_div_iff₀
    (mul_pos (by positivity) (pow_pos (by linarith : (0 : K) < p + q) 3))
    (mul_pos (mul_pos hmass (by linarith)) (by linarith))]
  linarith [hC]

set_option maxHeartbeats 400000 in
/-- **`margin` is explicit.** At every state before the last it is at
    least
    `1 / (8 M^3 F(N - M + 3)^3)`, so the thresholds of
    `double_oracle_approx` are explicit. -/
theorem margin_ge (hM : 2 ≤ M) (hMN : M ≤ N) {t : ℕ} (ht : t < M + N -
    2) :
    1 / (8 * (M : K) ^ 3 * (fib (N - M + 3) : K) ^ 3) ≤ margin M t := by
  have hM2 : (2 : K) ≤ M := by exact_mod_cast hM
  have hR : (2 : K) ≤ (fib (N - M + 3) : K) := by
    exact_mod_cast (fib_vals.1 ▸ fib_mono (show 3 ≤ N - M + 3 by omega)
    : 2 ≤ fib (N - M + 3))
  have hR3 : (8 : K) ≤ (fib (N - M + 3) : K) ^ 3 := by
    have h := pow_le_pow_left₀ (by norm_num : (0 : K) ≤ 2) hR 3
    norm_num at h
    linarith
  have hcube : (0 : K) ≤ (M : K) ^ 3 := by positivity
  have hsq : (M : K) ^ 2 ≤ (M : K) ^ 3 := by nlinarith [sq_nonneg (M :
    K)]
  have hden : (0 : K) < 8 * (M : K) ^ 3 * (fib (N - M + 3) : K) ^ 3 := by
    exact mul_pos (mul_pos (by norm_num) (pow_pos (by linarith) 3)) (by
    linarith)
  -- `128 M^3 ≤ 16 M^3 F(N - M + 3)^3`, the slack that covers the first
    three families
  have hbig : 128 * (M : K) ^ 3 ≤ 16 * (M : K) ^ 3 * (fib (N - M + 3) :
    K) ^ 3 := by
    linarith [mul_le_mul_of_nonneg_left hR3 (show (0 : K) ≤ 16 * (M : K)
    ^ 3 by positivity)]
  have key : ∀ a b : K, 0 < b → b ≤ a * (8 * (M : K) ^ 3 * (fib (N - M +
     3) : K) ^ 3) →
```

```
      1 / (8 * (M : K) ^ 3 * (fib (N - M + 3) : K) ^ 3) ≤ a / b := fun a
    b hb h => by
    rw [div_le_div_iff₀ hden hb]; linarith
  rw [margin]
  split_ifs with h1 h2 h3
  · -- the staircase states, where the margin is the value `2 / (k (k +
    1))` with `k + 1 ≤ M`
    have hk : ((t / 2 + 1 : ℕ) : K) + 1 ≤ M := by exact_mod_cast (show t
    / 2 + 1 + 1 ≤ M by omega)
    have hk1 : (1 : K) ≤ ((t / 2 + 1 : ℕ) : K) := by exact_mod_cast
    (show 1 ≤ t / 2 + 1 by omega)
    refine key _ _ (mul_pos (by linarith) (by linarith)) ?_
    have hkk : ((t / 2 + 1 : ℕ) : K) * (((t / 2 + 1 : ℕ) : K) + 1) ≤ (M
    : K) * M :=
      mul_le_mul (by linarith) hk (by linarith) (by linarith)
    linarith
  · -- `C_M`, of margin `2 / (15 M^2 (M + 1))`
    refine key _ _ (by positivity) ?_
    linarith
  · -- `C_{M+1}`, of margin `(45 M - 1) / (80 M^2 (8 M + 1))`
    refine key _ _ (by positivity) ?_
    have hM4 : 2 * (M : K) ^ 3 ≤ (M : K) ^ 4 := by nlinarith [hcube]
    have h45 : (45 * (M : K) - 1) * (8 * (M : K) ^ 3 * 8)
        ≤ (45 * (M : K) - 1) * (8 * (M : K) ^ 3 * (fib (N - M + 3) : K)
    ^ 3) :=
      mul_le_mul_of_nonneg_left
        (by linarith [mul_le_mul_of_nonneg_left hR3 (show (0 : K) ≤ 8 *
    (M : K) ^ 3 by positivity)])
        (by linarith)
    linarith
  · -- the deeper corridor states, where `F(K+1) = fib (t - 2 M + 6) ≤
    fib (N - M + 3)`
    refine le_trans (key _ _ ?_ ?_) (corrMargin_ge hM (t - 2 * M))
    · exact mul_pos (by positivity) (pow_pos (by
        exact_mod_cast fib_pos.2 (show 0 < t - 2 * M + 6 by omega)) 3)
    · have hX3 : (fib (t - 2 * M + 6) : K) ^ 3 ≤ (fib (N - M + 3) : K) ^
    3 :=
        pow_le_pow_left₀ (by positivity) (by exact_mod_cast fib_mono (by
    omega)) 3
      linarith [mul_le_mul_of_nonneg_left hX3 (show (0 : K) ≤ 9 * (M :
    K) ^ 2 by positivity),
        mul_le_mul_of_nonneg_right (show 9 * (M : K) ^ 2 ≤ 16 * (M : K)
    ^ 3 by linarith)
          (show (0 : K) ≤ (fib (N - M + 3) : K) ^ 3 by positivity)]

/-- **Approximate oracles.** For `2 ≤ M ≤ N`, there are thresholds `δ₀, ε
    ₀ > 0` below which the
    runs of the approximate algorithm from `({0}, {0})` are exactly `run
    M N`: apply
    `double_oracle_accuracy` at the least value of `margin`. The case `M
    > N` is
    `double_oracle_approx_swap`. -/
theorem double_oracle_approx (hM : 2 ≤ M) (hMN : M ≤ N) :
    ∃ δ₀ > (0 : K), ∃ ε₀ > (0 : K), ∀ δ εR εC : K, 0 ≤ δ → δ < δ₀ → 0 ≤ ε
    R → εR < ε₀ →
      0 ≤ εC → εC < ε₀ →
      ∀ s : ℕ → Finset ℕ × Finset ℕ,
        (s 0 = ({0}, {0}) ∧ ∀ t, StepApprox M N δ εR εC (s t) (s (t +
    1))) ↔ s = run M N := by
  obtain ⟨μ, hμ0, hμ⟩ : ∃ μ : K, 0 < μ ∧ ∀ t < M + N - 2, μ ≤ margin M t
    :=
    ⟨1 / (8 * (M : K) ^ 3 * (fib (N - M + 3) : K) ^ 3),
      one_div_pos.2 (by
        have : (2 : K) ≤ M := by exact_mod_cast hM
        exact mul_pos (mul_pos (by norm_num) (pow_pos (by linarith) 3))
```

```
          (pow_pos (by exact_mod_cast fib_pos.2 (by omega)) 3)),
      fun t ht => margin_ge hM hMN ht⟩
  have hμS : μ ≤ 2 / ((M : K) * ((M : K) - 1)) := by
    have h := hμ (2 * M - 4) (by omega)
    rw [margin, if_pos (by omega : 2 * M - 4 + 2 < 2 * M),
      show (2 * M - 4) / 2 + 1 = M - 1 by omega] at h
    have hMc : ((M - 1 : ℕ) : K) = (M : K) - 1 := by
      have : (1 : ℕ) ≤ M := by omega
      push_cast [this]; ring
    rw [hMc, show (M : K) - 1 + 1 = (M : K) by ring, mul_comm] at h
    exact h
  exact ⟨μ / 64, by positivity, μ / 64, by positivity,
    fun δ εR εC hδ hδ0 hεR hεR0 hεC hεC0 s =>
    double_oracle_accuracy hM hMN hδ hεR hεC
      (fun t ht => by rw [← le_div_iff₀' (by norm_num : (0 : K) < 64)]
    at *; linarith [hμ t ht])
      (by rw [← le_div_iff₀' (by norm_num : (0 : K) < 64)] at *;
    linarith)
      (fun t ht => by rw [← le_div_iff₀' (by norm_num : (0 : K) < 64)]
    at *; linarith [hμ t ht]) s⟩

/-! ### Necessity

  Large tolerances let some run stop early. The player that moves at a
    state may stay put once
  its tolerance reaches the duality gap there (`sharp_gap`). The row
    player may add a row too
  many at `S_{M-1}` (`sharp_row`), or too few at `T_{M-1}` (`sharp_col`).
  `double_oracle_accuracy_sharp` collects the three cases. -/

/-- An approximate successor, from a row and a column answering within
    `εR` and `εC` against a
`δ`-equilibrium. -/
theorem stepApprox_mk {R C R' C' : Finset ℕ} {δ εR εC : K} {x y : ℕ → K}
    {i j : ℕ}
    (h : IsEqApprox M R C δ x y) (hi : i < M)
    (hi' : ∀ i' < M, rowPay M C y i' ≤ rowPay M C y i + εR) (hj : j < N)
    (hj' : ∀ j' < N, colPay M R x j ≤ colPay M R x j' + εC) (hR : insert
    i R = R')
    (hC : insert j C = C') : StepApprox M N δ εR εC (R, C) (R', C') :=
  ⟨x, y, i, j, h, hi, hi', hj, hj', by rw [hR, hC]⟩

/-- If `εR` reaches the value at `S_{M-1}`, the row player may add row
    `M - 1` there, and the run
    reaches `S_M` an iteration early. -/
theorem sharp_row (hM : 2 ≤ M) (hMN : M ≤ N) {δ εR εC : K} (hδ : 0 ≤ δ)
    (hεR : 0 ≤ εR)
    (hεC : 0 ≤ εC)
    (h : 2 / ((M : K) * (M - 1)) ≤ εR) :
    StepApprox M N δ εR εC (range (M - 1), range (M - 1)) (range M, range
    M) := by
  obtain ⟨t, rfl⟩ : ∃ t, M = t + 2 := ⟨M - 2, by omega⟩
  rw [show t + 2 - 1 = t + 1 from rfl]
  obtain ⟨x, y, hxy, -, hw, -⟩ :=
    value_S (K := K) (M := t + 2) (k := t + 1) (by omega) (by omega)
  have hv : (2 : K) / (((t : K) + 1) * ((t : K) + 1 + 1)) ≤ εR := by
    push_cast at h
    rw [show ((t : K) + 1) * ((t : K) + 1 + 1) = ((t : K) + 2) * ((t :
    K) + 2 - 1) by ring]
    exact h
  -- row `M - 1` pays `0`, within `εR` of the value, and column `M - 1`
    is the best response
  have hzero : ∀ i, t + 1 ≤ i → rowPay (t + 2) (range (t + 1)) y i = 0
    := fun i hi => by
    rw [rowPay_stair (by omega)]
```

```
    simp [show ¬ i < t + 1 by omega, show ¬ i + 1 < t + 1 by omega]
  refine stepApprox_mk (x := x) (y := y) (i := t + 1) (j := t + 1)
    (hxy.approx hδ) (by omega) ?_
    (by omega) ?_ range_add_one.symm range_add_one.symm
  · intro i' hi'
    rw [hzero (t + 1) le_rfl]
    rcases lt_or_ge i' (t + 1) with h' | h'
    · have := hw i' (mem_range.2 h')
      push_cast at this hv ⊢
      linarith
    · rw [hzero i' h']; exact le_add_of_nonneg_right hεR
  · intro j' _
    rcases eq_or_ne j' (t + 1) with rfl | hne
    · exact le_add_of_nonneg_right hεC
    · exact le_add_of_le_of_nonneg
        (cert_S_out (by omega) (by omega) (by omega) hxy hne).le hεC

/-- If `δ` and `εR` reach a joint threshold at `T_{M-1}`, the row player
    may add nothing there,
    and the algorithm stops there. -/
theorem sharp_col (hM : 2 ≤ M) (hMN : M ≤ N) {δ εR εC : K} (hεR : 0 ≤ ε
    R) (hεC : 0 ≤ εC)
    (hMε : (M : K) * εR ≤ 1)
    (h : 4 / ((M : K) ^ 2 - 1) ≤ δ + 2 * εR / (M + 1)) :
    StepApprox M N δ εR εC (range (M - 1), range M) (range (M - 1), range
    M) := by
  obtain ⟨m, rfl⟩ : ∃ m, M = m + 1 := ⟨M - 1, by omega⟩
  rw [show m + 1 - 1 = m from rfl]
  obtain ⟨x, y, hxy, hu, hw⟩ := value_T (K := K) (M := m + 1) (k := m)
    (by omega) (by omega)
  have hm : (1 : K) ≤ m := by exact_mod_cast (show 1 ≤ m by omega)
  -- the column player plays `y'`, against which row `M - 1` beats the
    subgame rows by only `εR`
  set c : K := 2 * (1 - ((m : K) + 1) * εR) / (((m : K) + 1) * ((m : K) +
     2)) with hc_def
  have hc : 0 ≤ c := by
    have : ((m : K) + 1) * εR ≤ 1 := by push_cast at hMε; linarith
    rw [hc_def]; positivity
  set y' : ℕ → K := fun j => c * (((m : K) + 1) - j) + εR with hy'_def
  have hmix : IsMix (range (m + 1)) y' := by
    refine ⟨fun j hj => ?_, ?_⟩
    · have hj' : (j : K) ≤ m := by exact_mod_cast Nat.lt_succ_iff.1
    (mem_range.1 hj)
      simp only [hy'_def]
      nlinarith
    · simp only [hy'_def]
      rw [sum_add_distrib, ← mul_sum, sum_const, card_range,
    nsmul_eq_mul,
        show ∑ j ∈ range (m + 1), (((m : K) + 1) - j) = ((m : K) + 1) *
    ((m : K) + 2) / 2 by
          have := sum_range_cast_rev (K := K) (m + 1); push_cast at this
    ⊢; linarith, hc_def]
      push_cast
      field_simp
      ring
  have hrow : ∀ i < m, rowPay (m + 1) (range (m + 1)) y' i = c := fun i
    hi => by
    rw [rowPay_stair (by omega)]
    simp only [hy'_def, if_pos (show i < m + 1 by omega), if_pos (show i
    + 1 < m + 1 by omega)]
    push_cast
    ring
  have hrowM : rowPay (m + 1) (range (m + 1)) y' m = c + εR := by
    rw [rowPay_stair (by omega)]
```

```
    simp only [hy'_def, if_pos (show m < m + 1 by omega), if_neg (show ¬
    m + 1 < m + 1 by omega)]
    ring
  have hkey : c ≤ -(2 / ((m : K) * ((m : K) + 1))) + δ := by
    have he : c + 2 / ((m : K) * ((m : K) + 1))
        = 4 / ((m : K) * ((m : K) + 2)) - 2 * εR / ((m : K) + 2) := by
      rw [hc_def]; field_simp; ring
    push_cast at h
    rw [show ((m : K) + 1) ^ 2 - 1 = (m : K) * ((m : K) + 2) by ring,
      show (m : K) + 1 + 1 = (m : K) + 2 by ring] at h
    linarith
  -- against `x` some column of the subgame is tight, and those after it
    pay more
  obtain ⟨j, hj, hjle⟩ := exists_col_le hxy.2.1 hxy.1 hw
    (nonempty_range_iff.2 (by omega))
  refine stepApprox_mk (x := x) (y := y') (i := 0) (j := j) ⟨hxy.1,
    hmix, fun i hi j' hj' => ?_⟩
    (by omega) ?_ ((mem_range.1 hj).trans_le hMN) ?_
    (insert_eq_of_mem (mem_range.2 (by omega))) (insert_eq_of_mem hj)
  · rw [hrow i (mem_range.1 hi)]
    linarith [hu j' hj']
  · intro i' hi'
    rw [hrow 0 (by omega)]
    rcases lt_or_ge i' m with h' | h'
    · rw [hrow i' h']; linarith
    · obtain rfl : i' = m := by omega
      rw [hrowM]
  · intro j' _
    refine le_add_of_le_of_nonneg ?_ hεC
    rcases lt_or_ge j' (m + 1) with h' | h'
    · linarith [hu j' (mem_range.2 h')]
    · linarith [hu m (mem_range.2 (by omega)), cert_T_out (by omega) (by
    omega) hxy
        (show m < j' by omega)]

/-- The state of a run that follows `run M N` for `n` iterations and
    then follows `f`. -/
def early (M N n : ℕ) (f : ℕ → Finset ℕ × Finset ℕ) (t : ℕ) : Finset ℕ
    × Finset ℕ :=
  if t < n then run M N t else f t

/-- Such a run is an approximate run as soon as `f` starts with an
    approximate successor of
    `run M N (n - 1)` and keeps stepping. -/
theorem early_steps (hM : 2 ≤ M) (hMN : M ≤ N) {δ εR εC : K} (hδ : 0 ≤ δ
    ) (hεR : 0 ≤ εR)
    (hεC : 0 ≤ εC) {n : ℕ}
    (hn : 0 < n) {f : ℕ → Finset ℕ × Finset ℕ}
    (hjoin : StepApprox M N δ εR εC (run M N (n - 1)) (f n))
    (hf : ∀ t, StepApprox M N δ εR εC (f t) (f (t + 1))) (t : ℕ) :
    StepApprox M N δ εR εC (early M N n f t) (early M N n f (t + 1)) := by
  simp only [early]
  by_cases ht : t + 1 < n
  · rw [if_pos (by omega), if_pos ht]
    exact ((double_oracle hM hMN t _).2 rfl).approx hδ hεR hεC
  · by_cases ht' : t < n
    · rw [if_pos ht', if_neg ht, show t = n - 1 by omega,
    Nat.sub_add_cancel hn]
      exact hjoin
    · rw [if_neg ht', if_neg (by omega)]
      exact hf t

/-- The states of the run around `S_{M-1}`. -/
theorem run_last_two (hM : 2 ≤ M) (hMN : M ≤ N) :
    run M N (2 * M - 4) = (range (M - 1), range (M - 1)) ∧
```

```
      run M N (2 * M - 3) = (range (M - 1), range M) ∧
      run M N (2 * M - 2) = (range M, range M) := by
  obtain ⟨t, rfl⟩ : ∃ t, M = t + 2 := ⟨M - 2, by omega⟩
  rw [show t + 2 - 1 = t + 1 from rfl]
  refine ⟨?_, ?_, ?_⟩
  · rw [show 2 * (t + 2) - 4 = 2 * t by omega, run, if_pos (by omega),
      show 2 * t / 2 = t by omega, show (2 * t + 1) / 2 = t by omega]
  · rw [show 2 * (t + 2) - 3 = 2 * t + 1 by omega, run, if_pos (by omega),
      show (2 * t + 1) / 2 = t by omega, show (2 * t + 1 + 1) / 2 = t +
    1 by omega]
  · rw [show 2 * (t + 2) - 2 = 2 * t + 2 by omega, run, if_neg (by omega),
      show 2 * t + 2 + 2 - (t + 2) = t + 2 by omega, min_eq_left hMN]

/-- Rows are added only at the states `T_k`, which come at the odd
    iterations of the staircase
    phase. Everywhere else a step keeps the rows, and at `T_k` it keeps
    the columns. -/
theorem run_rows {t : ℕ} (h : ¬ (t % 2 = 1 ∧ t + 2 < 2 * M)) :
    (run M N (t + 1)).1 = (run M N t).1 := by
  by_cases ht : t + 2 < 2 * M
  · obtain ⟨k, rfl⟩ : ∃ k, t = 2 * k := ⟨t / 2, by omega⟩
    rw [run_even ht, run_odd (by omega)]
  · simp only [run, if_neg ht, if_neg (show ¬ t + 1 + 2 < 2 * M by
    omega)]

theorem run_cols (hMN : M ≤ N) {t : ℕ} (h : t % 2 = 1 ∧ t + 2 < 2 * M) :
    (run M N (t + 1)).2 = (run M N t).2 := by
  obtain ⟨k, rfl⟩ : ∃ k, t = 2 * k + 1 := ⟨t / 2, by omega⟩
  rw [run_odd h.2]
  simp only [run]
  split_ifs with h1
  · rw [show (2 * k + 1 + 1 + 1) / 2 = k + 1 by omega]
  · rw [min_eq_left (by omega), show 2 * k + 1 + 1 + 2 - M = k + 1 + 1 by
    omega]

/-- **The gap is necessary.** If the tolerance of the player that moves
    at `run M N t` reaches
    its duality gap `|v_t| + m_t`, the approximate algorithm may add
    nothing there. That player
    picks a strategy of the subgame that pays the value, and the other
    its exact best response,
    which lies in the subgame. -/
theorem sharp_gap (hM : 2 ≤ M) (hMN : M ≤ N) {δ εR εC : K} (hδ : 0 ≤ δ)
    (hεR : 0 ≤ εR)
    (hεC : 0 ≤ εC) {t : ℕ} (ht : t < M + N - 2)
    (h : tableValue M t + tableMargin M t ≤ if t % 2 = 1 ∧ t + 2 < 2 * M
    then εR else εC) :
    StepApprox M N δ εR εC (run M N t) (run M N t) := by
  obtain ⟨x, y, i, j, hxy, hi, hi', hj, hj', hq⟩ :=
    (double_oracle (K := K) hM hMN t (run M N (t + 1))).2 rfl
  have hgap := gap_run hM hMN ht hxy hi hi' hj hj'
  obtain ⟨hx, hy, he⟩ := hxy
  have hRne : (run M N t).1.Nonempty := by
    by_contra h0
    have h' := hx.2
    rw [not_nonempty_iff_eq_empty.1 h0, sum_empty] at h'
    exact zero_ne_one h'
  have hCne : (run M N t).2.Nonempty := by
    by_contra h0
    have h' := hy.2
    rw [not_nonempty_iff_eq_empty.1 h0, sum_empty] at h'
    exact zero_ne_one h'
  -- both averages are the value
  have havg : Σ i' ∈ (run M N t).1, x i' * rowPay M (run M N t).2 y i'
```

```
      = Σ j' ∈ (run M N t).2, y j' * colPay M (run M N t).1 x j' :=
    sum_swap x y
  have hRM : (run M N t).1 ⊆ range M := by
    simp only [run]; split_ifs <;> exact range_subset_range.2 (by omega)
  have hCN : (run M N t).2 ⊆ range N := by
    simp only [run]; split_ifs <;> exact range_subset_range.2 (by omega)
  rw [show run M N t = ((run M N t).1, (run M N t).2) from rfl]
  by_cases hrow : t % 2 = 1 ∧ t + 2 < 2 * M
  · rw [if_pos hrow] at h
    -- the column player's best response lies in the subgame
    have hjC : j ∈ (run M N t).2 := by
      have := congrArg Prod.snd hq
      rw [run_cols hMN hrow] at this
      exact insert_eq_self.1 this.symm
    -- the row player picks a row of the subgame that pays what `j` pays
    obtain ⟨i₀, hi₀, hi₀'⟩ := exists_max_image (run M N t).1 (rowPay M
    (run M N t).2 y) hRne
    have hval : rowPay M (run M N t).2 y i₀ = colPay M (run M N t).1 x j
    := by
      refine le_antisymm (he i₀ hi₀ j hjC) ?_
      calc colPay M (run M N t).1 x j
          ≤ Σ j' ∈ (run M N t).2, y j' * colPay M (run M N t).1 x j' :=
            mix_ge hy fun j' hj'C => hj' j' (mem_range.1 (hCN hj'C))
        _ = Σ i' ∈ (run M N t).1, x i' * rowPay M (run M N t).2 y i' :=
    havg.symm
        _ ≤ rowPay M (run M N t).2 y i₀ := mix_le hx hi₀'
    exact stepApprox_mk ⟨hx, hy, fun a ha b hb => by linarith [he a ha b
    hb]⟩
      (mem_range.1 (hRM hi₀)) (fun i' h' => by linarith [hi' i' h']) hj
      (fun j' h' => by linarith [hj' j' h']) (insert_eq_of_mem hi₀)
    (insert_eq_of_mem hjC)
  · rw [if_neg hrow] at h
    -- the row player's best response lies in the subgame
    have hiR : i ∈ (run M N t).1 := by
      have := congrArg Prod.fst hq
      rw [run_rows hrow] at this
      exact insert_eq_self.1 this.symm
    -- the column player picks a column of the subgame that pays what
    `i` pays
    obtain ⟨j₀, hj₀, hj₀'⟩ := exists_min_image (run M N t).2 (colPay M
    (run M N t).1 x) hCne
    have hval : colPay M (run M N t).1 x j₀ = rowPay M (run M N t).2 y i
    := by
      refine le_antisymm ?_ (he i hiR j₀ hj₀)
      calc colPay M (run M N t).1 x j₀
          ≤ Σ j' ∈ (run M N t).2, y j' * colPay M (run M N t).1 x j' :=
    mix_ge hy hj₀'
        _ = Σ i' ∈ (run M N t).1, x i' * rowPay M (run M N t).2 y i' :=
    havg.symm
        _ ≤ rowPay M (run M N t).2 y i :=
          mix_le hx fun i' hi'R => hi' i' (mem_range.1 (hRM hi'R))
    exact stepApprox_mk ⟨hx, hy, fun a ha b hb => by linarith [he a ha b
    hb]⟩ hi
      (fun i' h' => by linarith [hi' i' h']) (mem_range.1 (hCN hj₀))
      (fun j' h' => by linarith [hj' j' h']) (insert_eq_of_mem hiR)
    (insert_eq_of_mem hj₀)

/-- **Necessity.** Some run from `({0}, {0})` has stopped by iteration
    `M + N - 3`, so it is not
    `run M N`, which reaches the full game only at `M + N - 2`, in each
    of three cases: the
    tolerance of the player that moves at some state before the last
    reaches the duality gap
    `|v_t| + m_t` there (the row player moves exactly at the odd
    iterations of the staircase
```

```
    phase), `εR` reaches the value at `S_{M-1}`, or `δ` and `εR` reach a
    joint threshold at
    `T_{M-1}`. -/
theorem double_oracle_accuracy_sharp (hM : 2 ≤ M) (hMN : M ≤ N) {δ εR εC
    : K} (hδ : 0 ≤ δ)
    (hεR : 0 ≤ εR) (hεC : 0 ≤ εC)
    (h : (∃ t < M + N - 2, tableValue M t + tableMargin M t ≤
          if t % 2 = 1 ∧ t + 2 < 2 * M then εR else εC) ∨
      2 / ((M : K) * (M - 1)) ≤ εR ∨
      ((M : K) * εR ≤ 1 ∧ 4 / ((M : K) ^ 2 - 1) ≤ δ + 2 * εR / (M + 1)))
    :
    ∃ s : ℕ → Finset ℕ × Finset ℕ, s 0 = ({0}, {0}) ∧
      (∀ t, StepApprox M N δ εR εC (s t) (s (t + 1))) ∧ s (M + N - 3) = s
    (M + N - 2) ∧
      s ≠ run M N := by
  rcases h with ⟨t, ht, hd⟩ | h
  · -- the moving player may stay put at `run M N t`
    have hstop := sharp_gap hM hMN hδ hεR hεC ht hd
    refine ⟨early M N (t + 1) (fun _ => run M N t), ?_,
      early_steps hM hMN hδ hεR hεC (by omega) (by rwa
    [Nat.add_sub_cancel]) (fun _ => hstop),
      ?_, fun hs => ?_⟩
    · simp only [early, if_pos (show 0 < t + 1 by omega)]
      simp [run, show 2 < 2 * M by omega]
    · simp only [early, if_neg (show ¬ M + N - 2 < t + 1 by omega)]
      split_ifs with h1
      · rw [show M + N - 3 = t by omega]
      · rfl
    · have h1 := congrFun hs (t + 1)
      simp only [early, if_neg (lt_irrefl _)] at h1
      have h2 := run_card hM hMN t
      have h3 := run_card hM hMN (t + 1)
      rw [← h1] at h3
      omega
  obtain ⟨e1, e2, e3⟩ := run_last_two hM hMN
  -- either the run jumps to `S_M` at `S_{M-1}`, or it stops at `T_{M-1}`
  obtain ⟨f, hjoin, hf, hstop⟩ : ∃ f : ℕ → Finset ℕ × Finset ℕ,
      StepApprox M N δ εR εC (run M N (2 * M - 4)) (f (2 * M - 3)) ∧
      (∀ t, StepApprox M N δ εR εC (f t) (f (t + 1))) ∧ f (M + N - 3) = f
    (M + N - 2) := by
    rcases h with h | ⟨hMε, h⟩
    · refine ⟨fun t => run M N (t + 1), ?_,
        fun t => ((double_oracle hM hMN (t + 1) _).2 rfl).approx hδ hεR
      hεC, ?_⟩
      · dsimp only
        rw [e1, show 2 * M - 3 + 1 = 2 * M - 2 by omega, e3]
        exact sharp_row hM hMN hδ hεR hεC h
      · dsimp only
        rw [run_last hMN (by omega), run_last hMN (by omega)]
    · refine ⟨fun _ => (range (M - 1), range M), ?_,
        fun _ => sharp_col hM hMN hεR hεC hMε h, rfl⟩
      have hstep := ((double_oracle hM hMN (2 * M - 4) _).2 rfl).approx
        (δ := δ) (εR := εR) (εC := εC) hδ hεR hεC
      rwa [show 2 * M - 4 + 1 = 2 * M - 3 by omega, e2] at hstep
  refine ⟨early M N (2 * M - 3) f, ?_,
    early_steps hM hMN hδ hεR hεC (by omega) (by rwa [show 2 * M - 3 - 1 =
     2 * M - 4 by omega]) hf,
    ?_, fun hs => ?_⟩
  · simp only [early, if_pos (show 0 < 2 * M - 3 by omega)]
    simp [run, show 2 < 2 * M by omega]
  · simp only [early, if_neg (show ¬ M + N - 3 < 2 * M - 3 by omega),
      if_neg (show ¬ M + N - 2 < 2 * M - 3 by omega)]
    exact hstop
  · have hEq : run M N (M + N - 3) = run M N (M + N - 2) := by
      rw [← hs]
```

```
      simp only [early, if_neg (show ¬ M + N - 3 < 2 * M - 3 by omega),
        if_neg (show ¬ M + N - 2 < 2 * M - 3 by omega)]
      exact hstop
    have h1 := run_card hM hMN (M + N - 3)
    have h2 := run_card hM hMN (M + N - 2)
    rw [hEq] at h1
    omega

/-! ## Gap and shape

Everything above is about one matrix. This section is about all of them.

Along a run that adds one strategy per iteration, a step that adds a
    column leaves the row best
response inside the subgame, so the new column beats every column
    already present by at least
the duality gap, against the equilibrium of that subgame.
`gap_sep` turns one such comparison into a gap of `d` in a single row,
    and `card_le_of_sep`
counts, so `gap_shape` bounds the number of columns by `k ^ |R|`
    whenever `2 < k * d`. A matrix
whose entries lie in `[-1, 1]` therefore cannot hold the gap away from
    zero once it has many
more columns than rows. `gap_shape` takes the comparisons as a
    hypothesis, and `gap_shape_run`, in
the last section, derives the separation from a run and counts it with
    `card_le_of_sep`.
-/

section Gap

variable {B : ℕ → ℕ → K} {R C S : Finset ℕ} {d : K}

/-- A column that an average over `S` ranks `d` below another differs
    from it by at least `d`
in some row of `S`. -/
theorem gap_sep {x : ℕ → K} (hx : IsMix S x) {j j' : ℕ}
    (h : Σ i ∈ S, x i * B i j' + d ≤ Σ i ∈ S, x i * B i j) :
    ∃ i ∈ S, d ≤ B i j - B i j' := by
  by_contra hcon
  have hsmall : ∀ i ∈ S, B i j - B i j' < d := fun i hi =>
    not_le.1 fun hge => hcon ⟨i, hi, hge⟩
  obtain ⟨i₀, hi₀, hpos⟩ : ∃ i ∈ S, (0 : K) < x i := by
    by_contra h0
    have hle : Σ i ∈ S, x i ≤ 0 :=
      sum_nonpos fun i hi => not_lt.1 fun hgt => h0 ⟨i, hi, hgt⟩
    rw [hx.2] at hle
    linarith
  have key : Σ i ∈ S, x i * (B i j - B i j') < Σ i ∈ S, x i * d :=
    sum_lt_sum (fun i hi => mul_le_mul_of_nonneg_left (hsmall i hi).le
    (hx.1 i hi))
      ⟨i₀, hi₀, mul_lt_mul_of_pos_left (hsmall i₀ hi₀) hpos⟩
  rw [← sum_mul, hx.2, one_mul] at key
  simp only [mul_sub, sum_sub_distrib] at key
  linarith

/-- Columns with entries in `[-1, 1]` that are pairwise at sup-norm
    distance at least `d` are few:
they number at most `k ^ |R|` for any `k` with `2 < k * d`. -/
theorem card_le_of_sep {k : ℕ} (hk : 2 < k * d)
    (hrange : ∀ i ∈ R, ∀ j ∈ C, |B i j| ≤ 1)
    (hsep : ∀ j ∈ C, ∀ j' ∈ C, j ≠ j' → ∃ i ∈ R, d ≤ |B i j - B i j'|)
    :
    C.card ≤ k ^ R.card := by
  classical
```

```
  set p : ℕ → ℕ → ℕ → Prop := fun i j m => -1 + (m : K) * d ≤ B i j
    with hp
  set cell : ℕ → ℕ → ℕ := fun i j => Nat.findGreatest (p i j) k with
    hcell
  have hlow : ∀ i ∈ R, ∀ j ∈ C, -1 + (cell i j : K) * d ≤ B i j := by
    intro i hi j hj
    have hb := (abs_le.1 (hrange i hi j hj)).1
    exact Nat.findGreatest_spec (P := p i j) (Nat.zero_le k) (by simp
    [hp]; linarith)
  have hlt : ∀ i ∈ R, ∀ j ∈ C, cell i j < k := by
    intro i hi j hj
    refine lt_of_le_of_ne (Nat.findGreatest_le k) fun hEq => ?_
    have := hlow i hi j hj
    rw [hEq] at this
    have h1 := (abs_le.1 (hrange i hi j hj)).2
    linarith
  have hup : ∀ i ∈ R, ∀ j ∈ C, B i j < -1 + ((cell i j : K) + 1) * d :=
    by
    intro i hi j hj
    by_contra hcon
    have hge : -1 + ((cell i j : K) + 1) * d ≤ B i j := not_lt.1 hcon
    have hmem : p i j (cell i j + 1) := by simpa [hp, add_mul] using hge
    have hle : cell i j + 1 ≤ cell i j := by
      simpa [hcell] using Nat.le_findGreatest (hlt i hi j hj) hmem
    omega
  have hclose : ∀ i ∈ R, ∀ j ∈ C, ∀ j' ∈ C, cell i j = cell i j' →
      |B i j - B i j'| < d := by
    intro i hi j hj j' hj' hEq
    have l1 := hlow i hi j hj
    have u1 := hup i hi j hj
    have l2 := hlow i hi j' hj'
    have u2 := hup i hi j' hj'
    rw [hEq] at l1 u1
    rw [abs_lt]
    constructor <;> linarith
  have hcard : C.card ≤ (Fintype.piFinset fun _ : {a // a ∈ R} => range
    k).card := by
    apply card_le_card_of_injOn (fun j => fun i : {a // a ∈ R} => cell
    (i : ℕ) j)
    · intro j hj
      simp only [mem_coe, Fintype.mem_piFinset, mem_range]
      intro i
      exact hlt (i : ℕ) i.2 j hj
    · intro j hj j' hj' hEq
      by_contra hne
      obtain ⟨i, hi, hd'⟩ := hsep j hj j' hj' hne
      have := hclose i hi j hj j' hj' (congrFun hEq ⟨i, hi⟩)
      linarith
  have hcount : (Fintype.piFinset fun _ : {a // a ∈ R} => range k).card =
     k ^ R.card := by
    simp [Fintype.card_piFinset]
  omega

/-- The shape bound. If any two of the columns `C` are separated by `d`
    at some subgame
equilibrium over rows of `R`, and the entries lie in `[-1, 1]`, then
    `|C| ≤ k ^ |R|` for every
`k` with `2 < k * d`. -/
theorem gap_shape {k : ℕ} (hk : 2 < k * d)
    (hrange : ∀ i ∈ R, ∀ j ∈ C, |B i j| ≤ 1)
    (hbeat : ∀ j ∈ C, ∀ j' ∈ C, j ≠ j' → ∃ (S : Finset ℕ) (x : ℕ → K),
    S ⊆ R ∧ IsMix S x ∧
      (Σ i ∈ S, x i * B i j' + d ≤ Σ i ∈ S, x i * B i j ∨
        Σ i ∈ S, x i * B i j + d ≤ Σ i ∈ S, x i * B i j')) :
    C.card ≤ k ^ R.card := by
```

```
  refine card_le_of_sep hk hrange fun j hj j' hj' hne => ?_
  obtain ⟨S, x, hSR, hx, h⟩ := hbeat j hj j' hj' hne
  rcases h with h | h
  · obtain ⟨i, hi, hle⟩ := gap_sep hx h
    exact ⟨i, hSR hi, hle.trans (le_abs_self _)⟩
  · obtain ⟨i, hi, hle⟩ := gap_sep hx h
    exact ⟨i, hSR hi, by rw [abs_sub_comm]; exact hle.trans (le_abs_self
    _)⟩

end Gap

/-! ## A large gap in the square case

The bigger-number game with a tilt: the larger index wins, ties go to
    the row player, and the
prize `1 - k η` shrinks with the winner's index `k`. Without the tilt,
    both players move on the
same iteration and the run is half as long. With it, every run takes `2
    M - 2` iterations on the
`M × M` matrix (`tilt_worst_case`), and the duality gap at every state
    before the last is at
least `2 (1 - (M - 1) η)` (`tilt_gap`), which `η` brings as close as one
    likes to `2`, the most
that entries in `[-1, 1]` allow.
-/

section Tilt

variable {η : K} {m : ℕ}

/-- The tilted bigger-number game. -/
def tilt (η : K) (i j : ℕ) : K := if j ≤ i then 1 - i * η else -(1 - j *
    η)

/-- The payoff of column `j` against `x` on the rows `R`. -/
def colPayT (η : K) (R : Finset ℕ) (x : ℕ → K) (j : ℕ) : K := Σ i ∈ R,
    x i * tilt η i j

/-- The payoff of row `i` against `y` on the columns `C`. -/
def rowPayT (η : K) (C : Finset ℕ) (y : ℕ → K) (i : ℕ) : K := Σ j ∈ C,
    tilt η i j * y j

/-- `(x, y)` is an equilibrium of the subgame on `R × C`. -/
def IsEqT (η : K) (R C : Finset ℕ) (x y : ℕ → K) : Prop :=
  IsMix R x ∧ IsMix C y ∧ ∀ i ∈ R, ∀ j ∈ C, rowPayT η C y i ≤ colPayT η
    R x j

/-- A successor of the state `p`, as in `Step`. -/
def StepT (K : Type*) [Field K] [LinearOrder K] [IsStrictOrderedRing K]
    (M : ℕ) (η : K)
    (p q : Finset ℕ × Finset ℕ) : Prop :=
  ∃ (x y : ℕ → K) (i j : ℕ), IsEqT η p.1 p.2 x y ∧
    i < M ∧ (∀ i' < M, rowPayT η p.2 y i' ≤ rowPayT η p.2 y i) ∧
    j < M ∧ (∀ j' < M, colPayT η p.1 x j ≤ colPayT η p.1 x j') ∧
    q = (insert i p.1, insert j p.2)

/-- The pure strategy at `m`. -/
def dirac (m : ℕ) : ℕ → K := fun i => if i = m then 1 else 0

theorem isMix_dirac (h : m < k) : IsMix (range k) (dirac m : ℕ → K) :=
    by
  refine ⟨fun i _ => ?_, ?_⟩
  · unfold dirac; split <;> norm_num
  · simp [dirac, mem_range.2 h]
```

```
theorem rowPayT_dirac (h : m < k) (i : ℕ) :
    rowPayT η (range k) (dirac m) i = tilt η i m := by
  simp [rowPayT, dirac, mul_ite, mem_range.2 h]

theorem colPayT_dirac (h : m < k) (j : ℕ) :
    colPayT η (range k) (dirac m) j = tilt η m j := by
  simp [colPayT, dirac, ite_mul, mem_range.2 h]

/-- Against columns of index at most `i`, row `i` collects its own
    prize. -/
theorem rowPayT_const {y : ℕ → K} (hy : IsMix (range k) y) (h : k ≤ i +
    1) :
    rowPayT η (range k) y i = 1 - i * η := by
  have e : ∀ j ∈ range k, tilt η i j * y j = (1 - (i : K) * η) * y j :=
    fun j hj => by
    rw [tilt, if_pos (by simp only [mem_range] at hj; omega)]
  rw [rowPayT, sum_congr rfl e, ← mul_sum, hy.2, mul_one]

/-- Against rows of index below `j`, column `j` pays minus its own
    prize. -/
theorem colPayT_const {x : ℕ → K} (hx : IsMix (range k) x) (h : k ≤ j) :
    colPayT η (range k) x j = -(1 - j * η) := by
  have e : ∀ i ∈ range k, x i * tilt η i j = x i * -(1 - (j : K) * η) :=
    fun i hi => by
    rw [tilt, if_neg (by simp only [mem_range] at hi; omega)]
  rw [colPayT, sum_congr rfl e, ← sum_mul, hx.2, one_mul]

/-- Every prize along the matrix is positive. -/
theorem prize_pos (hη : 0 < η) (hηM : ((M : K) - 1) * η < 1) (hi : i <
    M) :
    0 < 1 - (i : K) * η := by
  have h1 : (i : K) ≤ (M : K) - 1 := by
    have : (i : K) + 1 ≤ (M : K) := by exact_mod_cast hi
    linarith
  have := mul_le_mul_of_nonneg_right h1 hη.le
  linarith

/-- The prizes strictly decrease. -/
theorem prize_lt (hη : 0 < η) (h : i < j) : 1 - (j : K) * η < 1 - (i :
    K) * η := by
  have : (i : K) < (j : K) := by exact_mod_cast h
  nlinarith

/-- A unique successor, as in `step_iff`. -/
theorem stepT_iff (hM : 0 < M) {R C R' C' : Finset ℕ}
    (hE : ∃ x y : ℕ → K, IsEqT η R C x y)
    (hR : ∀ x y : ℕ → K, IsEqT η R C x y → ∀ i, (∀ i' < M, rowPayT η C
    y i' ≤ rowPayT η C y i) →
      i < M → insert i R = R')
    (hC : ∀ x y : ℕ → K, IsEqT η R C x y → ∀ j, (∀ j' < M, colPayT η R
    x j ≤ colPayT η R x j') →
      j < M → insert j C = C')
    (q : Finset ℕ × Finset ℕ) : StepT K M η (R, C) q ↔ q = (R', C') :=
    by
  refine ⟨fun ⟨x, y, i, j, h, hi, hi', hj, hj', hq⟩ => ?_, fun hq => ?_⟩
  · rw [hq, hR x y h i hi' hi, hC x y h j hj' hj]
  obtain ⟨x, y, h⟩ := hE
  obtain ⟨i, hi, hi'⟩ := exists_max_image (range M) (rowPayT η C y)
    (nonempty_range_iff.2 hM.ne')
  obtain ⟨j, hj, hj'⟩ := exists_min_image (range M) (colPayT η R x)
    (nonempty_range_iff.2 hM.ne')
  simp only [mem_range] at hi hj hi' hj'
  exact ⟨x, y, i, j, h, hi, hi', hj, hj', by rw [hq, hR x y h i hi' hi,
    hC x y h j hj' hj]⟩
```

```
/-- The pure pair at `k` is an equilibrium of the square subgame on
    `[k+1] × [k+1]`. -/
theorem isEqT_square (hDk : (0 : K) < 1 - (k : K) * η) :
    IsEqT η (range (k + 1)) (range (k + 1)) (dirac k) (dirac k) := by
  refine ⟨isMix_dirac (by omega), isMix_dirac (by omega), fun i hi j hj =
    > ?_⟩
  simp only [mem_range] at hi hj
  have hc : tilt η k j = 1 - (k : K) * η := by rw [tilt, if_pos (by
    omega)]
  rw [rowPayT_dirac (by omega), colPayT_dirac (by omega), hc, tilt]
  split_ifs with h
  · have : i = k := by omega
    subst this; exact le_refl _
  · linarith

/-- At the square state, the row player is pinned to index `k`, so every
    column pays what it
pays against that pure strategy. -/
theorem colPayT_pin (hDk : (0 : K) < 1 - (k : K) * η) {x y : ℕ → K}
    (h : IsEqT η (range (k + 1)) (range (k + 1)) x y) (j : ℕ) :
    colPayT η (range (k + 1)) x j = tilt η k j := by
  have hck : colPayT η (range (k + 1)) x k = (2 * x k - 1) * (1 - (k :
    K) * η) := by
    rw [colPayT, sum_range_succ]
    have e : ∀ i ∈ range k, x i * tilt η i k = x i * -(1 - (k : K) * η)
    := fun i hi => by
      rw [tilt, if_neg (by simp only [mem_range] at hi; omega)]
    have hs : Σ i ∈ range k, x i = 1 - x k := by
      have := h.1.2; rw [sum_range_succ] at this; linarith
    rw [sum_congr rfl e, ← sum_mul, hs, tilt, if_pos (le_refl k)]; ring
  have hxk : x k = 1 := by
    have h1 := h.2.2 k (mem_range.2 (by omega)) k (mem_range.2 (by
    omega))
    rw [rowPayT_const h.2.1 (by omega), hck] at h1
    have h2 : x k ≤ 1 := by
      have := single_le_sum (f := x) (fun i hi => h.1.1 i hi)
        (mem_range.2 (show k < k + 1 by omega))
      rw [h.1.2] at this; exact this
    nlinarith
  have hs : Σ i ∈ range k, x i = 0 := by
    have := h.1.2; rw [sum_range_succ, hxk] at this; linarith
  have hz := (sum_eq_zero_iff_of_nonneg fun i hi =>
    h.1.1 i (mem_range.2 (by simp only [mem_range] at hi; omega))).1 hs
  rw [colPayT, sum_range_succ, hxk, one_mul, sum_congr rfl (fun i hi =>
    by
    rw [hz i hi, zero_mul]), sum_const_zero, zero_add]

/-- At the oblong state, the column player is pinned to index `k + 1`. -/
theorem rowPayT_pin (hDk : (0 : K) < 1 - (k : K) * η)
    (hDk1 : (0 : K) < 1 - ((k : K) + 1) * η) {x y : ℕ → K}
    (h : IsEqT η (range (k + 1)) (range (k + 2)) x y) (i : ℕ) :
    rowPayT η (range (k + 2)) y i = tilt η i (k + 1) := by
  have hrk : rowPayT η (range (k + 2)) y k
      = (1 - (k : K) * η) * (1 - y (k + 1)) + -(1 - ((k : K) + 1) * η) *
    y (k + 1) := by
    rw [rowPayT, sum_range_succ]
    have e : ∀ j ∈ range (k + 1), tilt η k j * y j = (1 - (k : K) * η) *
    y j := fun j hj => by
      rw [tilt, if_pos (by simp only [mem_range] at hj; omega)]
    have hs : Σ j ∈ range (k + 1), y j = 1 - y (k + 1) := by
      have := h.2.1.2; rw [sum_range_succ] at this; linarith
    rw [sum_congr rfl e, ← mul_sum, hs, tilt, if_neg (by omega)]
    push_cast; ring
  have hy1 : y (k + 1) = 1 := by
```

```
    have h1 := h.2.2 k (mem_range.2 (by omega)) (k + 1) (mem_range.2 (by
    omega))
    rw [hrk, colPayT_const h.1 (show k + 1 ≤ k + 1 by omega)] at h1
    have h2 : y (k + 1) ≤ 1 := by
      have := single_le_sum (f := y) (fun j hj => h.2.1.1 j hj)
        (mem_range.2 (show k + 1 < k + 2 by omega))
      rw [h.2.1.2] at this; exact this
    push_cast at h1
    nlinarith
  have hs : Σ j ∈ range (k + 1), y j = 0 := by
    have := h.2.1.2; rw [sum_range_succ, hy1] at this; linarith
  have hz := (sum_eq_zero_iff_of_nonneg fun j hj =>
    h.2.1.1 j (mem_range.2 (by simp only [mem_range] at hj; omega))).1 hs
  rw [rowPayT, sum_range_succ, hy1, mul_one, sum_congr rfl (fun j hj =>
    by
    rw [hz j hj, mul_zero]), sum_const_zero, zero_add]

/-- At `(range (k+1), range (k+1))`, no row is added, and column `k + 1`
    is the only best
response. -/
theorem tiltStep_even (hη : 0 < η) (hηM : ((M : K) - 1) * η < 1) (hk : k +
     1 < M)
    (q : Finset ℕ × Finset ℕ) :
    StepT K M η (range (k + 1), range (k + 1)) q ↔ q = (range (k + 1),
    range (k + 2)) := by
  have hDk : (0 : K) < 1 - (k : K) * η := prize_pos hη hηM (M := M) (by
    omega)
  have hDk1 : (0 : K) < 1 - ((k : K) + 1) * η := by
    have := prize_pos hη hηM (M := M) (i := k + 1) (by omega)
    push_cast at this; linarith
  refine stepT_iff (by omega) ⟨_, _, isEqT_square hDk⟩ (fun x y h i hi' _
    => ?_)
    (fun x y h j hj' _ => ?_) q
  · refine insert_eq_self.2 (mem_range.2 ?_)
    by_contra hcon
    have h1 := hi' k (by omega)
    rw [rowPayT_const h.2.1 (by omega), rowPayT_const h.2.1 (show k + 1 ≤
     i + 1 by omega)] at h1
    have := prize_lt (η := η) hη (show k < i by omega)
    linarith
  · have hj1 : j = k + 1 := by
      by_contra hcon
      have hck1 : tilt η k (k + 1) = -(1 - ((k : K) + 1) * η) := by
        rw [tilt, if_neg (by omega)]; push_cast; ring
      have h1 := hj' (k + 1) (by omega)
      rw [colPayT_pin hDk h, colPayT_pin hDk h, hck1] at h1
      rcases lt_or_gt_of_ne hcon with hlt | hgt
      · rw [tilt, if_pos (by omega)] at h1; linarith
      · rw [tilt, if_neg (by omega)] at h1
        have := prize_lt (η := η) hη (show k + 1 < j by omega)
        push_cast at this h1; linarith
    subst hj1
    exact Finset.range_add_one.symm

/-- At `(range (k+1), range (k+2))`, no column is added, and row `k + 1`
    is the only best
response. -/
theorem tiltStep_odd (hη : 0 < η) (hηM : ((M : K) - 1) * η < 1) (hk : k +
     1 < M)
    (q : Finset ℕ × Finset ℕ) :
    StepT K M η (range (k + 1), range (k + 2)) q ↔ q = (range (k + 2),
    range (k + 2)) := by
  have hDk : (0 : K) < 1 - (k : K) * η := prize_pos hη hηM (M := M) (by
    omega)
  have hDk1 : (0 : K) < 1 - ((k : K) + 1) * η := by
```

```
    have := prize_pos hη hηM (M := M) (i := k + 1) (by omega)
    push_cast at this; linarith
  have hE : IsEqT η (range (k + 1)) (range (k + 2)) (dirac k) (dirac (k +
     1)) := by
    refine ⟨isMix_dirac (by omega), isMix_dirac (by omega), fun i hi j
    hj => ?_⟩
    simp only [mem_range] at hi hj
    have hr : tilt η i (k + 1) = -(1 - ((k : K) + 1) * η) := by
      rw [tilt, if_neg (by omega)]; push_cast; ring
    rw [rowPayT_dirac (by omega), colPayT_dirac (by omega), hr, tilt]
    split_ifs with h
    · linarith
    · have : j = k + 1 := by omega
      subst this; push_cast; linarith
  refine stepT_iff (by omega) ⟨_, _, hE⟩ (fun x y h i hi' _ => ?_) (fun x
    y h j hj' _ => ?_) q
  · have hik : i = k + 1 := by
      by_contra hcon
      have h1 := hi' (k + 1) (by omega)
      rw [rowPayT_pin hDk hDk1 h, rowPayT_pin hDk hDk1 h, tilt,
        if_pos (le_refl (k + 1))] at h1
      push_cast at h1
      rcases lt_or_gt_of_ne hcon with hlt | hgt
      · rw [tilt, if_neg (by omega)] at h1; push_cast at h1; linarith
      · rw [tilt, if_pos (by omega)] at h1
        have := prize_lt (η := η) hη (show k + 1 < i by omega)
        push_cast at this; linarith
    subst hik
    exact Finset.range_add_one.symm
  · refine insert_eq_self.2 (mem_range.2 ?_)
    by_contra hcon
    have h1 := hj' (k + 1) (by omega)
    rw [colPayT_const h.1 (show k + 1 ≤ j by omega),
      colPayT_const h.1 (show k + 1 ≤ k + 1 by omega)] at h1
    have := prize_lt (η := η) hη (show k + 1 < j by omega)
    linarith

/-- At the full game, every best response is already inside, so the run
    stops. -/
theorem tiltStep_full (hη : 0 < η) (hηM : ((M : K) - 1) * η < 1) (hM : 0
    < M)
    (q : Finset ℕ × Finset ℕ) :
    StepT K M η (range M, range M) q ↔ q = (range M, range M) := by
  obtain ⟨k, rfl⟩ : ∃ k, M = k + 1 := ⟨M - 1, by omega⟩
  exact stepT_iff hM ⟨_, _, isEqT_square (prize_pos hη hηM (by omega))⟩
    (fun _ _ _ i _ hi => insert_eq_of_mem (mem_range.2 hi))
    (fun _ _ _ j _ hj => insert_eq_of_mem (mem_range.2 hj)) q

/-- Each state of the run has the next one as its unique successor. -/
theorem tiltStep (hM : 2 ≤ M) (hη : 0 < η) (hηM : ((M : K) - 1) * η < 1)
    (t : ℕ)
    (q : Finset ℕ × Finset ℕ) : StepT K M η (run M M t) q ↔ q = run M M
    (t + 1) := by
  by_cases ht : t + 2 < 2 * M
  · rcases Nat.even_or_odd' t with ⟨k, rfl | rfl⟩
    · have e1 : run M M (2 * k) = (range (k + 1), range (k + 1)) := by
        simp only [run, if_pos ht, show 2 * k / 2 = k by omega, show (2 *
     k + 1) / 2 = k by omega]
      have e2 : run M M (2 * k + 1) = (range (k + 1), range (k + 1 + 1))
    := by
        simp only [run, if_pos (show 2 * k + 1 + 2 < 2 * M by omega),
          show (2 * k + 1) / 2 = k by omega, show (2 * k + 1 + 1) / 2 =
    k + 1 by omega]
      rw [e1, e2]; exact tiltStep_even hη hηM (by omega) q
```

```
    · have e1 : run M M (2 * k + 1) = (range (k + 1), range (k + 1 + 1))
    := by
        simp only [run, if_pos ht, show (2 * k + 1) / 2 = k by omega,
          show (2 * k + 1 + 1) / 2 = k + 1 by omega]
      have e2 : run M M (2 * k + 1 + 1) = (range (k + 1 + 1), range (k +
    1 + 1)) := by
        simp only [run]
        split_ifs with h
        · rw [show (2 * k + 1 + 1) / 2 = k + 1 by omega,
            show (2 * k + 1 + 1 + 1) / 2 = k + 1 by omega]
        · rw [show 2 * k + 1 + 1 + 2 - M = M by omega, min_self, show M =
    k + 1 + 1 by omega]
      rw [e1, e2]; exact tiltStep_odd hη hηM (by omega) q
  have hrun : ∀ t, ¬ t + 2 < 2 * M → run M M t = (range M, range M) :=
    fun t ht => by
    rw [run, if_neg ht, min_eq_right (by omega)]
  rw [hrun t ht, hrun (t + 1) (by omega)]
  exact tiltStep_full hη hηM (by omega) q

/-- **Large gap, square case.** Every run of double oracle on the tilted
    bigger-number game,
from `({0}, {0})`, is `run M M`, and so takes `M + M - 2` iterations by
    `run_card`. -/
theorem tilt_worst_case (hM : 2 ≤ M) (hη : 0 < η) (hηM : ((M : K) - 1) *
    η < 1)
    (s : ℕ → Finset ℕ × Finset ℕ) :
    (s 0 = ({0}, {0}) ∧ ∀ t, StepT K M η (s t) (s (t + 1))) ↔ s = run M
    M := by
  have h0 : run M M 0 = ({0}, {0}) := by simp [run, show 2 < 2 * M by
    omega]
  rw [← h0]
  exact runs_iff (tiltStep hM hη hηM) s

/-- At every state before the last, the duality gap is at least `2 (1 -
    (M - 1) η)`. -/
theorem tilt_gap (hη : 0 < η) (hηM : ((M : K) - 1) * η < 1) {t : ℕ}
    (ht : t + 2 < 2 * M) {x y : ℕ → K} (h : IsEqT η (run M M t).1 (run
    M M t).2 x y)
    {i j : ℕ} (hi' : ∀ i' < M, rowPayT η (run M M t).2 y i' ≤ rowPayT η
    (run M M t).2 y i)
    (hj' : ∀ j' < M, colPayT η (run M M t).1 x j ≤ colPayT η (run M M
    t).1 x j') :
    2 * (1 - ((M : K) - 1) * η) ≤
      rowPayT η (run M M t).2 y i - colPayT η (run M M t).1 x j := by
  have hMcast : ∀ n : ℕ, n < M → (n : K) ≤ (M : K) - 1 := fun n hn => by
    have : (n : K) + 1 ≤ (M : K) := by exact_mod_cast hn
    linarith
  rcases Nat.even_or_odd' t with ⟨k, rfl | rfl⟩
  · have e1 : run M M (2 * k) = (range (k + 1), range (k + 1)) := by
      simp only [run, if_pos ht, show 2 * k / 2 = k by omega, show (2 *
    k + 1) / 2 = k by omega]
    rw [e1] at h hi' hj' ⊢
    have hDk : (0 : K) < 1 - (k : K) * η := prize_pos hη hηM (M := M)
    (by omega)
    have hck1 : tilt η k (k + 1) = -(1 - ((k : K) + 1) * η) := by
      rw [tilt, if_neg (by omega)]; push_cast; ring
    have h1 := hi' k (by omega)
    rw [rowPayT_const h.2.1 (by omega)] at h1
    have h2 := hj' (k + 1) (by omega)
    rw [colPayT_pin hDk h, colPayT_pin hDk h, hck1] at h2
    rw [colPayT_pin hDk h]
    have h3 := mul_le_mul_of_nonneg_right (hMcast k (by omega)) hη.le
    have h4 := mul_le_mul_of_nonneg_right (hMcast (k + 1) (by omega))
    hη.le
    push_cast at h4
```

```
    linarith
  · have e1 : run M M (2 * k + 1) = (range (k + 1), range (k + 1 + 1)) :=
    by
      simp only [run, if_pos ht, show (2 * k + 1) / 2 = k by omega,
        show (2 * k + 1 + 1) / 2 = k + 1 by omega]
    rw [e1] at h hi' hj' ⊢
    have hDk : (0 : K) < 1 - (k : K) * η := prize_pos hη hηM (M := M)
    (by omega)
    have hDk1 : (0 : K) < 1 - ((k : K) + 1) * η := by
      have := prize_pos hη hηM (M := M) (i := k + 1) (by omega)
      push_cast at this; linarith
    have h1 := hi' (k + 1) (by omega)
    rw [rowPayT_pin hDk hDk1 h, rowPayT_pin hDk hDk1 h, tilt, if_pos
    (le_refl (k + 1))] at h1
    have h2 := hj' (k + 1) (by omega)
    rw [colPayT_const h.1 (show k + 1 ≤ k + 1 by omega)] at h2
    rw [rowPayT_pin hDk hDk1 h]
    have h4 := mul_le_mul_of_nonneg_right (hMcast (k + 1) (by omega))
    hη.le
    push_cast at h1 h2 h4
    linarith

end Tilt

/-! ## Every matrix

  The results so far are about one matrix, or assume what a run
    supplies. This section runs
  double oracle on an arbitrary matrix `B` (`StepB`, which is `Step` on
    `A` by `stepB_A`). Bartl's
  induction proves Farkas' lemma over any ordered field (`farkas`),
    which yields the minimax
  theorem (`exists_isEqB`), so runs exist from every state
    (`exists_runB`). Then: some run from
  some pure state stops early on every matrix except in the `2 × 2` case
    (`any_start`,
  `any_start_two`), no run of `M + N - 2` iterations has larger subgames
    than `run M N`
  (`run_sizes`), every matrix close enough to `A` has the same runs
    (`double_oracle_perturbed`),
  and a run of `M + N - 2` iterations whose duality gap stays at least
    `γ` separates the columns
  and rows that `card_le_of_sep` counts (`gap_shape_run`).
-/

/-- Pairing is linear in its left argument. -/
theorem dot_sub_left {δ : Type*} (D : Finset δ) (α β : K) (u v w : δ →
    K) :
    Σ d ∈ D, (α * u d - β * v d) * w d
      = α * Σ d ∈ D, u d * w d - β * Σ d ∈ D, v d * w d := by
  rw [mul_sum, mul_sum, ← sum_sub_distrib]
  exact sum_congr rfl fun d _ => by ring

/-- ... and in its right argument. -/
theorem dot_sub_right {δ : Type*} (D : Finset δ) (α β : K) (u w z : δ →
    K) :
    Σ d ∈ D, u d * (α * w d - β * z d)
      = α * Σ d ∈ D, u d * w d - β * Σ d ∈ D, u d * z d := by
  rw [mul_sum, mul_sum, ← sum_sub_distrib]
  exact sum_congr rfl fun d _ => by ring

/-- **Farkas' lemma**, by Bartl's very short induction on the number of
    vectors, which needs
    nothing but an ordered field: if every `y` that pairs nonnegatively
    with each `a i`, `i ∈ I`,
```

```
    also pairs nonnegatively with `b`, then `b` is a nonnegative
    combination of the `a i`. If the
    vectors other than `a m` already force this, `a m` gets weight `0`.
    Otherwise some `y` pairs
    nonnegatively with them and negatively with `b`, hence with `a m`,
    and projecting every vector
    along `a m` onto the hyperplane of `y` leaves one vector fewer. -/
theorem farkas {ι δ : Type*} [DecidableEq ι] (D : Finset δ) (I : Finset ι)
    (a : ι → δ → K)
    (b : δ → K)
    (h : ∀ y : δ → K, (∀ i ∈ I, 0 ≤ Σ d ∈ D, a i d * y d) → 0 ≤ Σ d ∈
    D, b d * y d) :
    ∃ x : ι → K, (∀ i ∈ I, 0 ≤ x i) ∧ ∀ d ∈ D, Σ i ∈ I, x i * a i d = b
    d := by
  induction I using Finset.induction_on generalizing a b with
  | empty =>
    -- `b` pairs nonnegatively with `-b`, so it vanishes on `D`
    have h1 := h (fun d => -b d) (by simp)
    simp only [mul_neg, sum_neg_distrib, neg_nonneg] at h1
    refine ⟨0, by simp, fun d hd => ?_⟩
    have := (sum_eq_zero_iff_of_nonneg fun d _ => mul_self_nonneg (b
    d)).1
      (le_antisymm h1 (sum_nonneg fun d _ => mul_self_nonneg _)) d hd
    simpa using (mul_self_eq_zero.1 this).symm
  | insert m I hm ih =>
    have hupd : ∀ (x : ι → K) (c : K), ∀ i ∈ I, Function.update x m c i =
     x i :=
      fun x c i hi => Function.update_of_ne (ne_of_mem_of_not_mem hi hm)
    c x
    have hnn : ∀ (x : ι → K) (c : K), 0 ≤ c → (∀ i ∈ I, 0 ≤ x i) →
        ∀ i ∈ insert m I, 0 ≤ Function.update x m c i := fun x c hc hx i
    hi => by
      rcases eq_or_ne i m with rfl | hne
      · simpa using hc
      · rw [Function.update_of_ne hne]; exact hx i
    (mem_of_mem_insert_of_ne hi hne)
    by_cases h4 : ∀ y : δ → K, (∀ i ∈ I, 0 ≤ Σ d ∈ D, a i d * y d) → 0
    ≤ Σ d ∈ D, b d * y d
    · -- the other vectors already force it
      obtain ⟨x, hx, hxb⟩ := ih a b h4
      refine ⟨Function.update x m 0, hnn x 0 le_rfl hx, fun d hd => ?_⟩
      rw [sum_insert hm, Function.update_self, zero_mul, zero_add,
        sum_congr rfl fun i hi => by rw [hupd x 0 i hi], hxb d hd]
    push Not at h4
    obtain ⟨y, hy, hby⟩ := h4
    set am := Σ d ∈ D, a m d * y d with ham
    set bm := Σ d ∈ D, b d * y d with hbm
    have hm0 : am < 0 := not_le.1 fun hc => (not_le.2 hby) (h y fun i hi
    => by
      rcases mem_insert.1 hi with rfl | hi
      exacts [hc, hy i hi])
    -- project along `a m`: `(a m · w) y - (a m · y) w` is orthogonal to
    `a m`, and pairs with
    -- the other vectors as `w` pairs with their projections
    obtain ⟨x, hx, hxb⟩ := ih (fun i d => (Σ d ∈ D, a i d * y d) * a m d
    - am * a i d)
      (fun d => bm * a m d - am * b d) fun w hw => by
       have h1 := h (fun d => (Σ d ∈ D, a m d * w d) * y d - am * w d)
    fun i hi => by
          rw [dot_sub_right]
          rcases mem_insert.1 hi with rfl | hi
          · rw [← ham, mul_comm, sub_self]
          · have := hw i hi
            rw [dot_sub_left] at this
            linarith
```

```
        rw [dot_sub_right] at h1
        rw [dot_sub_left]
        linarith
    -- lift the combination, with a positive weight on `a m`
    set S := Σ i ∈ I, x i * Σ d ∈ D, a i d * y d with hS_def
    have hS : 0 ≤ S := sum_nonneg fun i hi => mul_nonneg (hx i hi) (hy i
    hi)
    have hcm : 0 ≤ (bm - S) / am := div_nonneg_of_nonpos (by linarith)
    hm0.le
    refine ⟨Function.update x m ((bm - S) / am), hnn x _ hcm hx, fun d
    hd => ?_⟩
    rw [sum_insert hm, Function.update_self, sum_congr rfl fun i hi =>
    by rw [hupd x _ i hi]]
    have h' := hxb d hd
    have e : Σ i ∈ I, x i * ((Σ d ∈ D, a i d * y d) * a m d - am * a i
    d)
        = S * a m d - am * Σ i ∈ I, x i * a i d := by
      rw [hS_def, sum_mul, mul_sum, ← sum_sub_distrib]
      exact sum_congr rfl fun i _ => by ring
    rw [e] at h'
    have ham0 : am ≠ 0 := hm0.ne
    field_simp
    linear_combination -h'

/-- Farkas' lemma for inequalities, by adding a slack variable for each
    coordinate: if every
    `y ≥ 0` that pairs nonnegatively with each `a i` pairs nonnegatively
    with `b`, then some
    `x ≥ 0` has `Σ x i a i ≤ b`. -/
theorem farkas_le {ι δ : Type*} [DecidableEq ι] [DecidableEq δ] (D :
    Finset δ) (I : Finset ι)
    (a : ι → δ → K) (b : δ → K)
    (h : ∀ y : δ → K, (∀ d ∈ D, 0 ≤ y d) → (∀ i ∈ I, 0 ≤ Σ d ∈ D, a i
    d * y d) →
      0 ≤ Σ d ∈ D, b d * y d) :
    ∃ x : ι → K, (∀ i ∈ I, 0 ≤ x i) ∧ ∀ d ∈ D, Σ i ∈ I, x i * a i d ≤
    b d := by
  obtain ⟨x, hx, hxb⟩ := farkas D (I.disjSum D) (Sum.elim a fun e d => if
    d = e then 1 else 0) b
    fun y hy => by
      refine h y (fun d hd => ?_) fun i hi => ?_
      · simpa [ite_mul, sum_ite_eq', hd] using hy (Sum.inr d)
    (inr_mem_disjSum.2 hd)
      · simpa using hy (Sum.inl i) (inl_mem_disjSum.2 hi)
  refine ⟨fun i => x (Sum.inl i), fun i hi => hx _ (inl_mem_disjSum.2
    hi), fun d hd => ?_⟩
  have h' := hxb d hd
  rw [sum_disjSum] at h'
  simp only [Sum.elim_inl, Sum.elim_inr, mul_ite, mul_one, mul_zero,
    sum_ite_eq, if_pos hd] at h'
  linarith [hx _ (inr_mem_disjSum.2 hd)]

section Every

variable (B : ℕ → ℕ → K)

/-- The payoff of column `j` against `x` on the rows `R`, in the matrix
    `B`. -/
def colPayB (R : Finset ℕ) (x : ℕ → K) (j : ℕ) : K := Σ i ∈ R, x i * B
    i j

/-- The payoff of row `i` against `y` on the columns `C`, in the matrix
    `B`. -/
def rowPayB (C : Finset ℕ) (y : ℕ → K) (i : ℕ) : K := Σ j ∈ C, B i j *
    y j
```

```
/-- `(x, y)` is an equilibrium of the subgame of `B` on `R × C`. -/
def IsEqB (R C : Finset ℕ) (x y : ℕ → K) : Prop :=
  IsMix R x ∧ IsMix C y ∧ ∀ i ∈ R, ∀ j ∈ C, rowPayB B C y i ≤ colPayB B
    R x j

/-- `q` is a successor of the state `p` in the `M × N` matrix `B`, as in
    `Step`. -/
def StepB (M N : ℕ) (p q : Finset ℕ × Finset ℕ) : Prop :=
  ∃ (x y : ℕ → K) (i j : ℕ), IsEqB B p.1 p.2 x y ∧
    i < M ∧ (∀ i' < M, rowPayB B p.2 y i' ≤ rowPayB B p.2 y i) ∧
    j < N ∧ (∀ j' < N, colPayB B p.1 x j ≤ colPayB B p.1 x j') ∧ q =
    (insert i p.1, insert j p.2)

/-- On the matrix of the construction, this is `Step`. -/
theorem stepB_A : StepB (A (K := K) M) M N = Step K M N := rfl

/-- Adding a constant to every entry adds it to every payoff, so it
    changes no equilibrium. -/
theorem isEqB_shift {R C : Finset ℕ} {x y : ℕ → K} (c : K)
    (h : IsEqB (fun i j => B i j + c) R C x y) : IsEqB B R C x y := by
  refine ⟨h.1, h.2.1, fun i hi j hj => ?_⟩
  have e1 : rowPayB (fun i j => B i j + c) C y i = rowPayB B C y i + c :=
    by
    simp only [rowPayB, add_mul, sum_add_distrib, ← mul_sum, h.2.1.2,
    mul_one]
  have e2 : colPayB (fun i j => B i j + c) R x j = colPayB B R x j + c :=
    by
    simp only [colPayB, mul_add, sum_add_distrib, ← sum_mul, h.1.2,
    one_mul]
  have := h.2.2 i hi j hj
  rw [e1, e2] at this
  linarith

/-- **Minimax theorem.** Every subgame of every matrix has an
    equilibrium, over every ordered
    field. After a shift makes the entries positive, Farkas' lemma
    solves the row player's and
    the column player's linear programs together with the comparison of
    their objectives. -/
theorem exists_isEqB {R C : Finset ℕ} (hR : R.Nonempty) (hC :
    C.Nonempty) :
    ∃ x y : ℕ → K, IsEqB B R C x y := by
  -- shift the entries to be positive
  set c : K := 1 + ∑ i ∈ R, ∑ j ∈ C, |B i j| with hc
  set P : ℕ → ℕ → K := fun i j => B i j + c with hP
  have hpos : ∀ i ∈ R, ∀ j ∈ C, 0 < P i j := fun i hi j hj => by
    have h1 : |B i j| ≤ ∑ j ∈ C, |B i j| :=
      single_le_sum (f := fun j => |B i j|) (fun _ _ => abs_nonneg _) hj
    have h2 : ∑ j ∈ C, |B i j| ≤ ∑ i ∈ R, ∑ j ∈ C, |B i j| :=
      single_le_sum (f := fun i => ∑ j ∈ C, |B i j|)
        (fun _ _ => sum_nonneg fun _ _ => abs_nonneg _) hi
    simp only [hP, hc]
    linarith [neg_abs_le (B i j)]
  suffices ∃ x y : ℕ → K, IsEqB P R C x y from
    let ⟨x, y, h⟩ := this; ⟨x, y, isEqB_shift B c h⟩
  -- variables `p` on `R` and `q` on `C`, with constraints `pP ≥ 1` on
    `C`, `Pq ≤ 1` on `R`,
  -- and `∑ p ≤ ∑ q`
  obtain ⟨z, hz, hzb⟩ := farkas_le (C.disjSum (R.disjSum (univ : Finset
    Unit))) (R.disjSum C)
      (Sum.elim (fun i => Sum.elim (fun j => -P i j) (Sum.elim (fun _ =>
    0) (fun _ => 1)))
        (fun j => Sum.elim (fun _ => 0) (Sum.elim (fun i => P i j) (fun
    _ => -1))))
```

```
   (Sum.elim (fun _ => -1) (Sum.elim (fun _ => 1) (fun _ => 0))) fun
w hw hwa => by
-- a certificate of infeasibility would contradict weak duality
by_contra hwb
push Not at hwb
set α : ℕ → K := fun j => w (Sum.inl j)
set β : ℕ → K := fun i => w (Sum.inr (Sum.inl i))
set γ : K := w (Sum.inr (Sum.inr ()))
have hα : ∀ j ∈ C, 0 ≤ α j := fun j hj => hw _ (inl_mem_disjSum.2
hj)
have hβ : ∀ i ∈ R, 0 ≤ β i := fun i hi => hw _ (inr_mem_disjSum.2
(inl_mem_disjSum.2 hi))
have hγ : 0 ≤ γ := hw _ (inr_mem_disjSum.2 (inr_mem_disjSum.2
(mem_univ _)))
have hrowα : ∀ i ∈ R, Σ j ∈ C, P i j * α j ≤ γ := fun i hi => by
  have h := hwa (Sum.inl i) (inl_mem_disjSum.2 hi)
  simp only [sum_disjSum, Sum.elim_inl, Sum.elim_inr, neg_mul,
sum_neg_distrib, zero_mul,
    sum_const_zero, one_mul, univ_unique, sum_singleton] at h
  linarith
have hcolβ : ∀ j ∈ C, γ ≤ Σ i ∈ R, β i * P i j := fun j hj => by
  have h := hwa (Sum.inr j) (inr_mem_disjSum.2 hj)
  simp only [sum_disjSum, Sum.elim_inl, Sum.elim_inr, zero_mul,
sum_const_zero, zero_add,
    neg_mul, one_mul, univ_unique, sum_singleton] at h
  have e : Σ i ∈ R, P i j * w (Sum.inr (Sum.inl i)) = Σ i ∈ R, β i *
 P i j :=
    sum_congr rfl fun _ _ => by ring
  linarith
have hb : Σ i ∈ R, β i < Σ j ∈ C, α j := by
  simp only [sum_disjSum, Sum.elim_inl, Sum.elim_inr, neg_mul,
one_mul, sum_neg_distrib,
    zero_mul, univ_unique, sum_singleton] at hwb
  linarith
-- weak duality: 'γ Σ α ≤ β P α ≤ γ Σ β'
have hdual : γ * Σ j ∈ C, α j ≤ γ * Σ i ∈ R, β i := by
  calc γ * Σ j ∈ C, α j = Σ j ∈ C, α j * γ := by
        rw [mul_sum]; exact sum_congr rfl fun _ _ => by ring
    _ ≤ Σ j ∈ C, α j * Σ i ∈ R, β i * P i j :=
        sum_le_sum fun j hj => mul_le_mul_of_nonneg_left (hcolβ j
hj) (hα j hj)
    _ = Σ i ∈ R, β i * Σ j ∈ C, P i j * α j := by
        simp only [mul_sum]; rw [sum_comm]
        exact sum_congr rfl fun _ _ => sum_congr rfl fun _ _ => by
ring
    _ ≤ Σ i ∈ R, β i * γ :=
        sum_le_sum fun i hi => mul_le_mul_of_nonneg_left (hrowα i
hi) (hβ i hi)
    _ = γ * Σ i ∈ R, β i := by rw [mul_sum]; exact sum_congr rfl fun
_ _ => by ring
have hγ0 : γ = 0 := by
  by_contra hne
  have := le_of_mul_le_mul_left hdual (lt_of_le_of_ne hγ (Ne.symm
hne))
  linarith
-- so 'P α ≤ 0' on a row of positive entries, which forces 'α = 0'
obtain ⟨i₀, hi₀⟩ := hR
have hα0 : ∀ j ∈ C, α j = 0 := by
  have h := hrowα i₀ hi₀
  rw [hγ0] at h
  have hnn : ∀ j ∈ C, 0 ≤ P i₀ j * α j := fun j hj => mul_nonneg
(hpos i₀ hi₀ j hj).le (hα j hj)
  have hz := (sum_eq_zero_iff_of_nonneg hnn).1 (le_antisymm h
(sum_nonneg hnn))
```

```
      exact fun j hj => (mul_eq_zero.1 (hz j hj)).resolve_left (hpos i₀
    hi₀ j hj).ne'
    rw [sum_eq_zero hα0] at hb
    linarith [sum_nonneg hβ]
  -- the programs are solved: normalize
  set p : ℕ → K := fun i => z (Sum.inl i)
  set q : ℕ → K := fun j => z (Sum.inr j)
  have hp : ∀ i ∈ R, 0 ≤ p i := fun i hi => hz _ (inl_mem_disjSum.2 hi)
  have hq : ∀ j ∈ C, 0 ≤ q j := fun j hj => hz _ (inr_mem_disjSum.2 hj)
  have hcol : ∀ j ∈ C, 1 ≤ Σ i ∈ R, p i * P i j := fun j hj => by
    have h := hzb (Sum.inl j) (inl_mem_disjSum.2 hj)
    simp only [sum_disjSum, Sum.elim_inl, Sum.elim_inr, mul_neg,
    mul_zero, sum_const_zero,
      add_zero, sum_neg_distrib] at h
    linarith
  have hrow : ∀ i ∈ R, Σ j ∈ C, P i j * q j ≤ 1 := fun i hi => by
    have h := hzb (Sum.inr (Sum.inl i)) (inr_mem_disjSum.2
    (inl_mem_disjSum.2 hi))
    simp only [sum_disjSum, Sum.elim_inl, Sum.elim_inr, mul_zero,
    sum_const_zero,
      zero_add] at h
    simpa only [mul_comm] using h
  have hsum : Σ i ∈ R, p i ≤ Σ j ∈ C, q j := by
    have h := hzb (Sum.inr (Sum.inr ())) (inr_mem_disjSum.2
    (inr_mem_disjSum.2 (mem_univ _)))
    simp only [sum_disjSum, Sum.elim_inl, Sum.elim_inr, mul_one,
    mul_neg, sum_neg_distrib] at h
    linarith
  -- weak duality closes the comparison
  have hdual : Σ j ∈ C, q j ≤ Σ i ∈ R, p i := by
    calc Σ j ∈ C, q j ≤ Σ j ∈ C, q j * Σ i ∈ R, p i * P i j :=
          sum_le_sum fun j hj => le_mul_of_one_le_right (hq j hj) (hcol
    j hj)
      _ = Σ i ∈ R, p i * Σ j ∈ C, P i j * q j := by
          simp only [mul_sum]; rw [sum_comm]
          exact sum_congr rfl fun _ _ => sum_congr rfl fun _ _ => by ring
      _ ≤ Σ i ∈ R, p i := sum_le_sum fun i hi => mul_le_of_le_one_right
    (hp i hi) (hrow i hi)
  obtain ⟨j₀, hj₀⟩ := hC
  have hS : 0 < Σ i ∈ R, p i := by
    rcases (sum_nonneg hp).lt_or_eq with h | h
    · exact h
    · have hz0 := (sum_eq_zero_iff_of_nonneg hp).1 h.symm
      have := hcol j₀ hj₀
      rw [sum_eq_zero fun i hi => by rw [hz0 i hi, zero_mul]] at this
      linarith
  set S := Σ i ∈ R, p i
  refine ⟨fun i => p i / S, fun j => q j / S, ⟨fun i hi => div_nonneg (hp
    i hi) hS.le,
    by rw [← sum_div, div_self hS.ne']⟩, ⟨fun j hj => div_nonneg (hq j
    hj) hS.le,
    by rw [← sum_div, le_antisymm hdual hsum, div_self hS.ne']⟩, fun i
    hi j hj => ?_⟩
  have e1 : rowPayB P C (fun j => q j / S) i = (Σ j ∈ C, P i j * q j) /
    S := by
    rw [rowPayB, sum_div]; exact sum_congr rfl fun _ _ => by ring
  have e2 : colPayB P R (fun i => p i / S) j = (Σ i ∈ R, p i * P i j) /
    S := by
    rw [colPayB, sum_div]; exact sum_congr rfl fun _ _ => by ring
  rw [e1, e2]
  exact div_le_div_of_nonneg_right ((hrow i hi).trans (hcol j hj)) hS.le

/-- A successor exists at every state whose two sets are nonempty. -/
theorem exists_stepB (hM : 0 < M) (hN : 0 < N) {p : Finset ℕ × Finset ℕ
    } (hp1 : p.1.Nonempty)
```

```
    (hp2 : p.2.Nonempty) : ∃ q, StepB B M N p q := by
  obtain ⟨x, y, h⟩ := exists_isEqB B hp1 hp2
  obtain ⟨i, hi, hi'⟩ := exists_max_image (range M) (rowPayB B p.2 y)
    (nonempty_range_iff.2 hM.ne')
  obtain ⟨j, hj, hj'⟩ := exists_min_image (range N) (colPayB B p.1 x)
    (nonempty_range_iff.2 hN.ne')
  simp only [mem_range] at hi hj hi' hj'
  exact ⟨_, x, y, i, j, h, hi, hi', hj, hj', rfl⟩

/-- So a run starts from every such state. -/
theorem exists_runB (hM : 0 < M) (hN : 0 < N) {p : Finset ℕ × Finset ℕ}
    (hp1 : p.1.Nonempty)
    (hp2 : p.2.Nonempty) :
    ∃ s : ℕ → Finset ℕ × Finset ℕ, s 0 = p ∧ ∀ t, StepB B M N (s t) (s
    (t + 1)) := by
  have hstep : ∀ q : {q : Finset ℕ × Finset ℕ // q.1.Nonempty ∧
    q.2.Nonempty},
      ∃ q' : {q : Finset ℕ × Finset ℕ // q.1.Nonempty ∧ q.2.Nonempty},
    StepB B M N q.1 q'.1 :=
    fun q => by
      obtain ⟨q', hq'⟩ := exists_stepB B hM hN q.2.1 q.2.2
      have hne : q'.1.Nonempty ∧ q'.2.Nonempty := by
        obtain ⟨x, y, i, j, -, -, -, -, -, hq⟩ := hq'
        rw [hq]; exact ⟨insert_nonempty _ _, insert_nonempty _ _⟩
      exact ⟨⟨q', hne⟩, hq'⟩
  choose f hf using hstep
  refine ⟨fun t => (f^[t] ⟨p, hp1, hp2⟩).1, rfl, fun t => ?_⟩
  simp only [Function.iterate_succ_apply']
  exact hf _

/-- A step adds the two best responses, so the sets only grow, stay
    inside the matrix, and gain
    at least one strategy unless the state repeats. -/
theorem stepB_grow {p q : Finset ℕ × Finset ℕ} (h : StepB B M N p q) :
    p.1 ⊆ q.1 ∧ p.2 ⊆ q.2 ∧ (p.1 ⊆ range M → q.1 ⊆ range M) ∧ (p.2 ⊆
    range N → q.2 ⊆ range N) ∧
      (p ≠ q → p.1.card + p.2.card + 1 ≤ q.1.card + q.2.card) := by
  obtain ⟨x, y, i, j, -, hi, -, hj, -, rfl⟩ := h
  refine ⟨subset_insert _ _, subset_insert _ _, fun h => insert_subset
    (mem_range.2 hi) h,
    fun h => insert_subset (mem_range.2 hj) h, fun hne => ?_⟩
  by_cases hi' : i ∈ p.1
  · by_cases hj' : j ∈ p.2
    · exact absurd (Prod.ext (insert_eq_of_mem hi').symm
    (insert_eq_of_mem hj').symm) hne
    · rw [card_insert_of_notMem hj']; linarith [card_le_card
    (subset_insert i p.1)]
  · rw [card_insert_of_notMem hi']; linarith [card_le_card (subset_insert
    j p.2)]

/-- Along a run, the sets only grow and stay inside the matrix. -/
theorem runB_sub {s : ℕ → Finset ℕ × Finset ℕ} (hs : ∀ t, StepB B M N
    (s t) (s (t + 1)))
    (h0 : (s 0).1 ⊆ range M ∧ (s 0).2 ⊆ range N) (t : ℕ) :
    ((s 0).1 ⊆ (s t).1 ∧ (s 0).2 ⊆ (s t).2) ∧ (s t).1 ⊆ range M ∧ (s
    t).2 ⊆ range N := by
  induction t with
  | zero => exact ⟨⟨subset_refl _, subset_refl _⟩, h0⟩
  | succ t ih =>
    obtain ⟨h1, h2, h3, h4, -⟩ := stepB_grow B (hs t)
    exact ⟨⟨ih.1.1.trans h1, ih.1.2.trans h2⟩, h3 ih.2.1, h4 ih.2.2⟩

/-- A run that does not stop during `T` iterations from `a` adds at
    least `T` strategies. -/
```

```
theorem runB_card {s : ℕ → Finset ℕ × Finset ℕ} (hs : ∀ t, StepB B M N
    (s t) (s (t + 1)))
    {a T : ℕ} (hne : ∀ t, a ≤ t → t < a + T → s t ≠ s (t + 1)) :
    (s a).1.card + (s a).2.card + T ≤ (s (a + T)).1.card + (s (a +
    T)).2.card := by
  induction T with
  | zero => simp
  | succ T ih =>
    have h1 := ih fun t h1 h2 => hne t h1 (by omega)
    have h2 := (stepB_grow B (hs (a + T))).2.2.2.2 (hne (a + T) (by
    omega) (by omega))
    rw [show a + (T + 1) = a + T + 1 by omega]
    omega

/-- The pure profile `(i, j)` is the equilibrium of the one-entry
    subgame. -/
theorem isEqB_single (i j : ℕ) : IsEqB B {i} {j} (dirac i) (dirac j) :=
    by
  refine ⟨⟨fun a _ => ?_, by simp [dirac]⟩, ⟨fun a _ => ?_, by simp
    [dirac]⟩, fun a ha b hb => ?_⟩
  · unfold dirac; split <;> norm_num
  · unfold dirac; split <;> norm_num
  · rw [mem_singleton] at ha hb
    subst ha hb
    simp [rowPayB, colPayB, dirac]

/-- **Any start, negative result.** For `M, N ≥ 2` with `(M, N) ≠ (2,
    2)`, on every `M × N`
    matrix some run from some pure initial state `({i}, {j})` stops
    before iteration
    `M + N - 2`. Fix a best row against each column and a best column
    against each row. At most
    `N` initial states have their own row as that best row, and at most
    `M` their own column as
    that best column, which leaves one with neither, from which a first
    step adds two strategies. -/
theorem any_start (hM : 2 ≤ M) (hN : 2 ≤ N) (h22 : (M, N) ≠ (2, 2)) :
    ∃ i < M, ∃ j < N, ∃ s : ℕ → Finset ℕ × Finset ℕ, s 0 = ({i}, {j}) ∧
      (∀ t, StepB B M N (s t) (s (t + 1))) ∧ ∃ t < M + N - 2, s t = s (t
    + 1) := by
  classical
  -- a best row against each column, and a best column against each row
  have hr : ∀ j, ∃ i, i < M ∧ ∀ i' < M, B i' j ≤ B i j := fun j => by
    obtain ⟨i, hi, hi'⟩ :=
      exists_max_image (range M) (fun i => B i j) (nonempty_range_iff.2
    (by omega))
    exact ⟨i, mem_range.1 hi, fun i' h => hi' i' (mem_range.2 h)⟩
  have hc : ∀ i, ∃ j, j < N ∧ ∀ j' < N, B i j ≤ B i j' := fun i => by
    obtain ⟨j, hj, hj'⟩ :=
      exists_min_image (range N) (fun j => B i j) (nonempty_range_iff.2
    (by omega))
    exact ⟨j, mem_range.1 hj, fun j' h => hj' j' (mem_range.2 h)⟩
  choose r hrM hr' using hr
  choose c hcN hc' using hc
  -- the pure states whose own row or column is that best response are
    fewer than `M N`
  have h5 : 5 ≤ M + N := by
    by_contra h
    exact h22 (Prod.ext (by simp; omega) (by simp; omega))
  have hcard : ((range N).image (fun j => (r j, j)) ∪ (range M).image
    (fun i => (i, c i))).card
      < (range M ×ˢ range N).card := by
    rw [card_product, card_range, card_range]
    refine lt_of_le_of_lt ((card_union_le _ _).trans (Nat.add_le_add
```

```
      (card_image_le.trans (card_range N).le) (card_image_le.trans
    (card_range M).le))) ?_
    obtain ⟨a, rfl⟩ : ∃ a, M = a + 2 := ⟨M - 2, by omega⟩
    obtain ⟨b, rfl⟩ : ∃ b, N = b + 2 := ⟨N - 2, by omega⟩
    nlinarith [Nat.zero_le (a * b)]
  obtain ⟨⟨i, j⟩, hij, hbad⟩ := exists_mem_notMem_of_card_lt_card hcard
  rw [mem_product, mem_range, mem_range] at hij
  have hri : r j ≠ i := fun h =>
    hbad (mem_union_left _ (mem_image.2 ⟨j, mem_range.2 hij.2, by rw
    [h]⟩))
  have hcj : c i ≠ j := fun h =>
    hbad (mem_union_right _ (mem_image.2 ⟨i, mem_range.2 hij.1, by rw
    [h]⟩))
  -- the first step adds both best responses
  have hrow : ∀ a, rowPayB B {j} (dirac j) a = B a j := fun a => by simp
    [rowPayB, dirac]
  have hcol : ∀ b, colPayB B {i} (dirac i) b = B i b := fun b => by simp
    [colPayB, dirac]
  have hstep : StepB B M N ({i}, {j}) (insert (r j) {i}, insert (c i)
    {j}) :=
    ⟨dirac i, dirac j, r j, c i, isEqB_single B i j, hrM j,
      fun a ha => by rw [hrow, hrow]; exact hr' j a ha, hcN i,
      fun b hb => by rw [hcol, hcol]; exact hc' i b hb, rfl⟩
  obtain ⟨s', hs'0, hs'⟩ := exists_runB B (M := M) (N := N) (by omega)
    (by omega)
    (p := (insert (r j) {i}, insert (c i) {j})) (insert_nonempty _ _)
    (insert_nonempty _ _)
  refine ⟨i, hij.1, j, hij.2, fun t => if t = 0 then ({i}, {j}) else s'
    (t - 1), rfl,
    fun t => ?_, ?_⟩
  · rcases t with _ | t
    · simp only [if_neg (Nat.succ_ne_zero 0), Nat.sub_self, hs'0]
      exact hstep
    · simp only [if_neg (Nat.succ_ne_zero _), Nat.add_sub_cancel]
      exact hs' t
  · -- otherwise `M + N - 3` more steps would add `M + N - 3` more
    strategies to the four
    by_contra hcon
    push Not at hcon
    set s : ℕ → Finset ℕ × Finset ℕ := fun t => if t = 0 then ({i},
    {j}) else s' (t - 1)
    have hs : ∀ t, StepB B M N (s t) (s (t + 1)) := fun t => by
      rcases t with _ | t
      · simp only [s, if_pos rfl, if_neg (Nat.succ_ne_zero 0),
    Nat.sub_self, hs'0]
        exact hstep
      · simp only [s, if_neg (Nat.succ_ne_zero _), Nat.add_sub_cancel]
        exact hs' t
    have hsub := runB_sub B hs (by simp [s]; omega)
    have h4 : (s 1).1.card + (s 1).2.card = 4 := by
      simp only [s, if_neg (Nat.succ_ne_zero 0), Nat.sub_self, hs'0]
      rw [card_insert_of_notMem (by simpa using hri),
    card_insert_of_notMem (by simpa using hcj),
        card_singleton, card_singleton]
    have hgrow := runB_card B hs (a := 1) (T := M + N - 3) fun t h1 h2 =
    > hcon t (by omega)
    have hR := card_le_card (hsub (1 + (M + N - 3))).2.1
    have hC := card_le_card (hsub (1 + (M + N - 3))).2.2
    rw [card_range] at hR hC
    omega

/-- For `a + b = a' + b'`, the product `a b` is at most `a' b'` when
    `a'` and `b'` are both on the
    same side of `a`: `a' b' - a b = (a' - a) (b' - a)`. -/
theorem mul_le_mul_of_sum {a b a' b' : ℕ} (hs : a + b = a' + b')
```

```
    (h : (a ≤ a' ∧ a ≤ b') ∨ (a' ≤ a ∧ b' ≤ a)) : a * b ≤ a' * b' := by
  have key : (a' * b' : ℤ) - a * b = (a' - a) * (b' - a) := by
    have : (b : ℤ) = a' + b' - a := by omega
    rw [this]; ring
  rcases h with ⟨h1, h2⟩ | ⟨h1, h2⟩
  · have : (0 : ℤ) ≤ (a' - a) * (b' - a) := mul_nonneg (by omega) (by
    omega)
    zify; linarith
  · have : (0 : ℤ) ≤ (a' - a) * (b' - a) :=
    mul_nonneg_of_nonpos_of_nonpos (by omega) (by omega)
    zify; linarith

/-- **Intermediate sizes.** Along any run of `M + N - 2` iterations, on
    any `M × N` matrix, the
    subgame after `t` iterations has at most as many entries as that of
    `run M N`. Such a run
    adds exactly one strategy per iteration, and `run M N` keeps the
    subgame as square as the
    shape allows. -/
theorem run_sizes (hM : 2 ≤ M) (hMN : M ≤ N) {s : ℕ → Finset ℕ ×
    Finset ℕ}
    (h0 : (s 0).1 ⊆ range M ∧ (s 0).2 ⊆ range N) (hs : ∀ t, StepB B M N
    (s t) (s (t + 1)))
    (hlong : ∀ t < M + N - 2, s t ≠ s (t + 1)) (t : ℕ) :
    (s t).1.card * (s t).2.card ≤ (run M N t).1.card * (run M N
    t).2.card := by
  have hsub := runB_sub B hs h0
  have hRM : ∀ t, (s t).1.card ≤ M := fun t => (card_le_card (hsub
    t).2.1).trans (card_range M).le
  have hCN : ∀ t, (s t).2.card ≤ N := fun t => (card_le_card (hsub
    t).2.2).trans (card_range N).le
  -- the initial sets are nonempty, since the first subgame has an
    equilibrium
  have hne : ∀ t, 1 ≤ (s t).1.card ∧ 1 ≤ (s t).2.card := fun t => by
    obtain ⟨x, y, i, j, ⟨hx, hy, -⟩, -⟩ := hs 0
    have h1 : (s 0).1.Nonempty := by
      by_contra h
      have h' := hx.2
      rw [not_nonempty_iff_eq_empty.1 h, sum_empty] at h'
      exact zero_ne_one h'
    have h2 : (s 0).2.Nonempty := by
      by_contra h
      have h' := hy.2
      rw [not_nonempty_iff_eq_empty.1 h, sum_empty] at h'
      exact zero_ne_one h'
    exact ⟨card_pos.2 (h1.mono (hsub t).1.1), card_pos.2 (h2.mono (hsub
    t).1.2)⟩
  rcases le_or_gt t (M + N - 2) with ht | ht
  · -- exactly `t + 2` strategies after `t` iterations
    have hlow := runB_card B hs (a := 0) (T := t) (fun t' _ h => hlong
    t' (by omega))
    have hup := runB_card B hs (a := t) (T := M + N - 2 - t) (fun t' h1
    h2 => hlong t' (by omega))
    rw [show t + (M + N - 2 - t) = M + N - 2 by omega] at hup
    rw [zero_add] at hlow
    have e0 := hne 0
    have hRe := hRM (M + N - 2)
    have hCe := hCN (M + N - 2)
    have hsum : (s t).1.card + (s t).2.card = t + 2 := by omega
    have hrun := run_card hM hMN t
    rw [min_eq_left ht] at hrun
    obtain ⟨ha1, hb1⟩ := hne t
    have ha := hRM t
    refine mul_le_mul_of_sum (by omega) ?_
    simp only [run] at hrun ⊢
```

```
    split_ifs at hrun ⊢ with h
    · simp only [card_range] at hrun ⊢
      omega
    · simp only [card_range] at hrun ⊢
      rw [min_eq_left (by omega)] at hrun ⊢
      omega
  · -- after the last iteration, `run M N` is the full game
    rw [run_last hMN (by omega), card_range, card_range]
    exact Nat.mul_le_mul (hRM t) (hCN t)

/-- Moving every entry by at most `ρ` moves a payoff against a mixed
    strategy by at most `ρ`. -/
theorem mix_close {S : Finset ℕ} {w f g : ℕ → K} {ρ : K} (hw : IsMix S
    w)
    (h : ∀ i ∈ S, |f i - g i| ≤ ρ) : |Σ i ∈ S, w i * f i - Σ i ∈ S, w i
    * g i| ≤ ρ := by
  rw [← sum_sub_distrib]
  calc |Σ i ∈ S, (w i * f i - w i * g i)| ≤ Σ i ∈ S, |w i * f i - w i *
    g i| :=
        abs_sum_le_sum_abs _ _
    _ = Σ i ∈ S, w i * |f i - g i| := sum_congr rfl fun i hi => by
        rw [← mul_sub, abs_mul, abs_of_nonneg (hw.1 i hi)]
    _ ≤ Σ i ∈ S, w i * ρ := sum_le_sum fun i hi =>
    mul_le_mul_of_nonneg_left (h i hi) (hw.1 i hi)
    _ = ρ := by rw [← sum_mul, hw.2, one_mul]

/-- **Perturbed matrices.** For `2 ≤ M ≤ N`, there is a radius `ρ > 0`
    such that, on any matrix
    whose entries are within `ρ` of those of `A`, the runs of double
    oracle from `({0}, {0})` are
    exactly `run M N`. Each payoff moves by less than `ρ`, so a run on
    the perturbed matrix is a
    run of the approximate algorithm on `A` with all three tolerances `2
    ρ`, and
    `double_oracle_accuracy` applies. A run exists by `exists_runB`. The
    case `M > N` is
    `double_oracle_perturbed_swap`. -/
theorem double_oracle_perturbed (hM : 2 ≤ M) (hMN : M ≤ N) :
    ∃ ρ > (0 : K), ∀ B : ℕ → ℕ → K, (∀ i < M, ∀ j < N, |B i j - A M i
    j| < ρ) →
      ∀ s : ℕ → Finset ℕ × Finset ℕ,
        (s 0 = ({0}, {0}) ∧ ∀ t, StepB B M N (s t) (s (t + 1))) ↔ s =
    run M N := by
  obtain ⟨μ, hμ0, hμ⟩ : ∃ μ : K, 0 < μ ∧ ∀ t < M + N - 2, μ ≤ margin M t
    :=
    ⟨1 / (8 * (M : K) ^ 3 * (fib (N - M + 3) : K) ^ 3),
      one_div_pos.2 (by
        have : (2 : K) ≤ M := by exact_mod_cast hM
        exact mul_pos (mul_pos (by norm_num) (pow_pos (by linarith) 3))
          (pow_pos (by exact_mod_cast fib_pos.2 (by omega)) 3)),
      fun t ht => margin_ge hM hMN ht⟩
  have hμS : μ ≤ 2 / ((M : K) * ((M : K) - 1)) := by
    have h := hμ (2 * M - 4) (by omega)
    rw [margin, if_pos (by omega : 2 * M - 4 + 2 < 2 * M),
      show (2 * M - 4) / 2 + 1 = M - 1 by omega] at h
    have hMc : ((M - 1 : ℕ) : K) = (M : K) - 1 := by
      have : (1 : ℕ) ≤ M := by omega
      push_cast [this]; ring
    rw [hMc, show (M : K) - 1 + 1 = (M : K) by ring, mul_comm] at h
    exact h
  refine ⟨μ / 128, by positivity, fun B hB => ?_⟩
  suffices key : ∀ s : ℕ → Finset ℕ × Finset ℕ, s 0 = ({0}, {0}) →
      (∀ t, StepB B M N (s t) (s (t + 1))) → s = run M N by
    intro s
    refine ⟨fun ⟨hs0, hs⟩ => key s hs0 hs, fun hs => ?_⟩
```

```
    obtain ⟨s', hs'0, hs'⟩ := exists_runB B (M := M) (N := N) (by omega)
    (by omega)
      (p := ({0}, {0})) (singleton_nonempty _) (singleton_nonempty _)
    rw [hs, ← key s' hs'0 hs']
    exact ⟨hs'0, hs'⟩
  intro s hs0 hs
  have hsub := runB_sub B hs (by rw [hs0]; simp; omega)
  -- a run on `B` is an approximate run on `A`
  have happrox : ∀ t, StepApprox M N (μ / 64) (μ / 64) (μ / 64) (s t) (s
    (t + 1)) := fun t => by
    obtain ⟨x, y, i, j, ⟨hx, hy, he⟩, hi, hi', hj, hj', hq⟩ := hs t
    obtain ⟨-, hR, hC⟩ := hsub t
    have hrow : ∀ a < M, |rowPay M (s t).2 y a - rowPayB B (s t).2 y a| ≤
     μ / 128 := fun a ha => by
      have h := mix_close (f := fun b => A M a b) (g := fun b => B a b)
    (ρ := μ / 128) hy
        fun b hb => by
          rw [abs_sub_comm]; exact (hB a ha b (mem_range.1 (hC hb))).le
      simpa only [rowPay, rowPayB, mul_comm] using h
    have hcol : ∀ b < N, |colPay M (s t).1 x b - colPayB B (s t).1 x b| ≤
     μ / 128 := fun b hb => by
      have h := mix_close (f := fun a => A M a b) (g := fun a => B a b)
    (ρ := μ / 128) hx
        fun a ha => by
          rw [abs_sub_comm]; exact (hB a (mem_range.1 (hR ha)) b hb).le
      simpa only [colPay, colPayB] using h
    refine ⟨x, y, i, j, ⟨hx, hy, fun a ha b hb => ?_⟩, hi, fun a ha =>
    ?_, hj, fun b hb => ?_, hq⟩
    · have h1 := abs_le.1 (hrow a (mem_range.1 (hR ha)))
      have h2 := abs_le.1 (hcol b (mem_range.1 (hC hb)))
      linarith [he a ha b hb]
    · have h1 := abs_le.1 (hrow a ha)
      have h2 := abs_le.1 (hrow i hi)
      linarith [hi' a ha]
    · have h1 := abs_le.1 (hcol b hb)
      have h2 := abs_le.1 (hcol j hj)
      linarith [hj' b hb]
  exact (double_oracle_accuracy hM hMN (δ := μ / 64) (εR := μ / 64) (εC :=
     μ / 64)
    (by positivity) (by positivity) (by positivity) (fun t ht => by
    linarith [hμ t ht])
    (by linarith) (fun t ht => by linarith [hμ t ht]) s).1 ⟨hs0, happrox⟩

/-- A step, with the equilibrium and the two best responses it uses,
    whose duality gap
    `w_i - u_j` is at least `γ`. -/
def GapStep (M N : ℕ) (γ : K) (p q : Finset ℕ × Finset ℕ) : Prop :=
  ∃ (x y : ℕ → K) (i j : ℕ), IsEqB B p.1 p.2 x y ∧
    i < M ∧ (∀ i' < M, rowPayB B p.2 y i' ≤ rowPayB B p.2 y i) ∧
    j < N ∧ (∀ j' < N, colPayB B p.1 x j ≤ colPayB B p.1 x j') ∧
    q = (insert i p.1, insert j p.2) ∧ γ ≤ rowPayB B p.2 y i - colPayB B
    p.1 x j

/-- Along a run of `M + N - 2` iterations, every iteration adds exactly
    one strategy, the initial
    sets are singletons, and the final state is the full game. -/
theorem long_run (h3 : 3 ≤ M + N) {γ : K} {s : ℕ → Finset ℕ × Finset ℕ}
    (h0 : (s 0).1 ⊆ range M ∧ (s 0).2 ⊆ range N)
    (hs : ∀ t < M + N - 2, GapStep B M N γ (s t) (s (t + 1)))
    (hlong : ∀ t < M + N - 2, s t ≠ s (t + 1)) :
    (∀ t ≤ M + N - 2, (s t).1.card + (s t).2.card = t + 2) ∧
      (∀ t ≤ M + N - 2, (s 0).1 ⊆ (s t).1 ∧ (s 0).2 ⊆ (s t).2 ∧ (∀ t' ≤
    t, (s t').1 ⊆ (s t).1 ∧
        (s t').2 ⊆ (s t).2) ∧ (s t).1 ⊆ range M ∧ (s t).2 ⊆ range N) ∧
      ((s 0).1.Nonempty ∧ (s 0).2.Nonempty) := by
```

```
  -- the step relation up to `M + N - 2`, as a `StepB` relation
  have hstep : ∀ t < M + N - 2, StepB B M N (s t) (s (t + 1)) := fun t
    ht => by
    obtain ⟨x, y, i, j, hxy, hi, hi', hj, hj', hq, -⟩ := hs t ht
    exact ⟨x, y, i, j, hxy, hi, hi', hj, hj', hq⟩
  have hsub : ∀ t ≤ M + N - 2, (∀ t' ≤ t, (s t').1 ⊆ (s t).1 ∧ (s t').2
    ⊆ (s t).2) ∧
      (s t).1 ⊆ range M ∧ (s t).2 ⊆ range N := by
    intro t
    induction t with
    | zero =>
      intro _
      exact ⟨fun t' h => by rw [Nat.le_zero.1 h]; exact ⟨subset_refl _,
    subset_refl _⟩, h0⟩
    | succ t ih =>
      intro ht
      obtain ⟨hmono, hR, hC⟩ := ih (by omega)
      obtain ⟨h1, h2, h3, h4, -⟩ := stepB_grow B (hstep t (by omega))
      refine ⟨fun t' h => ?_, h3 hR, h4 hC⟩
      rcases eq_or_lt_of_le h with rfl | h'
      · exact ⟨subset_refl _, subset_refl _⟩
      · obtain ⟨a, b⟩ := hmono t' (by omega); exact ⟨a.trans h1, b.trans
    h2⟩
  have hne : (s 0).1.Nonempty ∧ (s 0).2.Nonempty := by
    obtain ⟨x, y, i, j, ⟨hx, hy, -⟩, -⟩ := hs 0 (by omega)
    constructor
    · by_contra h'
      have := hx.2
      rw [not_nonempty_iff_eq_empty.1 h', sum_empty] at this
      exact zero_ne_one this
    · by_contra h'
      have := hy.2
      rw [not_nonempty_iff_eq_empty.1 h', sum_empty] at this
      exact zero_ne_one this
  -- at least one strategy per iteration, and at most `M + N` in all
  have hcount : ∀ a T, a + T ≤ M + N - 2 →
      (s a).1.card + (s a).2.card + T ≤ (s (a + T)).1.card + (s (a +
    T)).2.card := by
    intro a T
    induction T with
    | zero => intro _; simp
    | succ T ih =>
      intro h
      have h1 := ih (by omega)
      have h2 := (stepB_grow B (hstep (a + T) (by omega))).2.2.2.2
    (hlong (a + T) (by omega))
      rw [show a + (T + 1) = a + T + 1 by omega]
      omega
  have hfull := hsub (M + N - 2) le_rfl
  have hRe := card_le_card hfull.2.1
  have hCe := card_le_card hfull.2.2
  rw [card_range] at hRe hCe
  refine ⟨fun t ht => ?_, fun t ht => ⟨((hsub t ht).1 0 (Nat.zero_le
    _)).1,
     ((hsub t ht).1 0 (Nat.zero_le _)).2, (hsub t ht).1, (hsub t ht).2⟩,
    hne⟩
  have hlow := hcount 0 t (by omega)
  have hup := hcount t (M + N - 2 - t) (by omega)
  rw [zero_add] at hlow
  rw [show t + (M + N - 2 - t) = M + N - 2 by omega] at hup
  have := card_pos.2 hne.1
  have := card_pos.2 hne.2
  omega
```

```
/-- **Large gap, negative result.** If a run of double oracle on an `M × N` matrix takes
    `M + N - 2` iterations, and the duality gap of the equilibrium it uses is at least `γ` at every
    state before the last, then any two columns differ by at least `γ` in some row. At the iteration
    that adds the later of the two, the row best response lies in the subgame, so it pays at most
    what the earlier column pays, and `gap_sep` finds the row. -/
theorem gap_run_cols (hM : 1 ≤ M) (hN : 1 ≤ N) {γ : K} {s : ℕ → Finset ℕ × Finset ℕ}
    (h0 : (s 0).1 ⊆ range M ∧ (s 0).2 ⊆ range N)
    (hs : ∀ t < M + N - 2, GapStep B M N γ (s t) (s (t + 1)))
    (hlong : ∀ t < M + N - 2, s t ≠ s (t + 1)) :
    ∀ j ∈ range N, ∀ j' ∈ range N, j ≠ j' → ∃ i ∈ range M, γ ≤ |B i j - B i j'| := by
  classical
  -- with a single column there is nothing to separate
  rcases lt_or_ge (M + N) 3 with h2 | h3
  · intro j hj j' hj' hne'
    rw [mem_range] at hj hj'
    omega
  obtain ⟨hcard, hsub, hne⟩ := long_run B h3 h0 hs hlong
  -- the final state holds every column
  have hfull : (s (M + N - 2)).2 = range N := by
    obtain ⟨-, -, -, hR, hC⟩ := hsub (M + N - 2) le_rfl
    have h1 := card_le_card hR
    have h2 := card_le_card hC
    rw [card_range] at h1 h2
    have := hcard (M + N - 2) le_rfl
    exact eq_of_subset_of_card_le hC (by rw [card_range]; omega)
  -- the iteration at which a column enters
  have hin : ∀ c ∈ range N, ∃ t, c ∈ (s t).2 := fun c hc => ⟨M + N - 2,
    hfull ▸ hc⟩
  -- each iteration adds exactly one strategy, so the two columns enter
    at different iterations
  have hone : ∀ t < M + N - 2, ∀ x y i j, IsEqB B (s t).1 (s t).2 x y →
      s (t + 1) = (insert i (s t).1, insert j (s t).2) → j ∉ (s t).2 →
    i ∈ (s t).1 := by
    intro t ht x y i j _ hq hj
    by_contra hi
    have h1 := hcard t (by omega)
    have h2 := hcard (t + 1) (by omega)
    rw [hq, card_insert_of_notMem hi, card_insert_of_notMem hj] at h2
    omega
  -- the later column is added at an iteration at which the earlier one
    is present
  have key : ∀ j (hj : j ∈ range N) j' (hj' : j' ∈ range N), Nat.find
    (hin j hj) <
      Nat.find (hin j' hj') → ∃ i ∈ range M, γ ≤ |B i j - B i j'| := by
    intro j hj j' hj' hlt
    set τ := Nat.find (hin j' hj')
    have hτ : τ ≤ M + N - 2 := Nat.find_min' _ (hfull ▸ hj')
    obtain ⟨t, ht⟩ : ∃ t, τ = t + 1 := ⟨τ - 1, by omega⟩
    have hj't : j' ∉ (s t).2 := Nat.find_min (hin j' hj') (by omega)
    have hj't1 : j' ∈ (s (t + 1)).2 := ht ▸ Nat.find_spec (hin j' hj')
    have hjt : j ∈ (s t).2 := by
      have := Nat.find_spec (hin j hj)
      exact ((hsub t (by omega)).2.2.1 _ (by omega)).2 this
    obtain ⟨x, y, i, c, hxy, hi, hi', hc, hc', hq, hgap⟩ := hs t (by
    omega)
    have hcj : c = j' := by
      rw [hq] at hj't1
      rcases mem_insert.1 hj't1 with h | h
      · exact h.symm
```

```
      · exact absurd h hj't
    subst hcj
    have hiR := hone t (by omega) x y i c hxy hq hj't
    have hle := hxy.2.2 i hiR j hjt
    obtain ⟨r, hr, hsep⟩ := gap_sep (B := B) (S := (s t).1) (d := γ)
    hxy.1 (j := j) (j' := c) (by
      simp only [colPayB] at hle hgap ⊢; linarith)
    exact ⟨r, (hsub t (by omega)).2.2.2.1 hr, hsep.trans (le_abs_self _)⟩
  intro j hj j' hj' hne'
  rcases lt_trichotomy (Nat.find (hin j hj)) (Nat.find (hin j' hj'))
    with h | h | h
  · exact key j hj j' hj' h
  · -- two columns cannot enter together: at the start there is one, and
    later one per iteration
    exfalso
    set τ := Nat.find (hin j hj)
    rcases Nat.eq_zero_or_pos τ with hτ0 | hτ0
    · have h1 : j ∈ (s 0).2 := hτ0 ▸ Nat.find_spec (hin j hj)
      have h2 : j' ∈ (s 0).2 := (h ▸ hτ0) ▸ Nat.find_spec (hin j' hj')
      have hsing := hcard 0 (by omega)
      have := card_pos.2 hne.1
      have hle : ({j, j'} : Finset ℕ).card ≤ (s 0).2.card :=
        card_le_card (insert_subset h1 (singleton_subset_iff.2 h2))
      rw [card_pair hne'] at hle
      omega
    · obtain ⟨t, ht⟩ : ∃ t, τ = t + 1 := ⟨τ - 1, by omega⟩
      have hτ : τ ≤ M + N - 2 := Nat.find_min' _ (hfull ▸ hj)
      obtain ⟨x, y, i, c, -, -, -, -, -, hq, -⟩ := hs t (by omega)
      have hj1 : j ∈ (s (t + 1)).2 := ht ▸ Nat.find_spec (hin j hj)
      have hj'1 : j' ∈ (s (t + 1)).2 := (h ▸ ht) ▸ Nat.find_spec (hin j'
    hj')
      have hj0 : j ∉ (s t).2 := Nat.find_min (hin j hj) (by omega)
      have hj'0 : j' ∉ (s t).2 := Nat.find_min (hin j' hj') (by rw [←
    h]; omega)
      rw [hq] at hj1 hj'1
      rcases mem_insert.1 hj1 with h1 | h1
      · rcases mem_insert.1 hj'1 with h2 | h2
        · exact hne' (h1.trans h2.symm)
        · exact hj'0 h2
      · exact hj0 h1
  · obtain ⟨r, hr, hsep⟩ := key j' hj' j hj h
    exact ⟨r, hr, by rwa [abs_sub_comm]⟩

/-- Exchanging the players: a step on `B` is a step on its negative
    transpose, with the two sets
    swapped and the same duality gap. -/
theorem gapStep_swap {γ : K} {p q : Finset ℕ × Finset ℕ} (h : GapStep B
    M N γ p q) :
    GapStep (fun a b => -B b a) N M γ p.swap q.swap := by
  obtain ⟨x, y, i, j, ⟨hx, hy, he⟩, hi, hi', hj, hj', rfl, hgap⟩ := h
  have hr : ∀ a, rowPayB (fun a b => -B b a) p.1 x a = -colPayB B p.1 x
    a := fun a => by
    simp only [rowPayB, colPayB, ← sum_neg_distrib]
    exact sum_congr rfl fun _ _ => by ring
  have hc : ∀ b, colPayB (fun a b => -B b a) p.2 y b = -rowPayB B p.2 y
    b := fun b => by
    simp only [rowPayB, colPayB, ← sum_neg_distrib]
    exact sum_congr rfl fun _ _ => by ring
  refine ⟨y, x, j, i, ⟨hy, hx, fun a ha b hb => ?_⟩, hj, fun a ha => ?_,
    hi, fun b hb => ?_,
    rfl, ?_⟩
  · simp only [Prod.fst_swap, Prod.snd_swap, hr, hc]
    linarith [he b hb a ha]
  · simp only [Prod.snd_swap, hr]
    linarith [hj' a ha]
```

```
  · simp only [Prod.fst_swap, hc]
    linarith [hi' b hb]
  · simp only [Prod.fst_swap, Prod.snd_swap, hr, hc]
    linarith

/-- The rows, by exchanging the players: any two rows differ by at least
    `γ` in some column. -/
theorem gap_run_rows (hM : 1 ≤ M) (hN : 1 ≤ N) {γ : K} {s : ℕ → Finset
    ℕ × Finset ℕ}
    (h0 : (s 0).1 ⊆ range M ∧ (s 0).2 ⊆ range N)
    (hs : ∀ t < M + N - 2, GapStep B M N γ (s t) (s (t + 1)))
    (hlong : ∀ t < M + N - 2, s t ≠ s (t + 1)) :
    ∀ i ∈ range M, ∀ i' ∈ range M, i ≠ i' → ∃ j ∈ range N, γ ≤ |B i j
    - B i' j| := by
  intro i hi i' hi' hne
  obtain ⟨j, hj, hsep⟩ := gap_run_cols (fun a b => -B b a) (M := N) (N :=
    M) hN hM
    (s := fun t => (s t).swap) ⟨h0.2, h0.1⟩
    (fun t ht => gapStep_swap B (hs t (by omega)))
    (fun t ht h => hlong t (by omega) (by simpa using congrArg Prod.swap
    h)) i hi i' hi' hne
  refine ⟨j, hj, ?_⟩
  rw [show -B i j - -B i' j = -(B i j - B i' j) by ring, abs_neg] at hsep
  exact hsep

/-- **Large gap, negative result, the shape bound.** On an `M × N`
    matrix with entries in
    `[-1, 1]`, a run of `M + N - 2` iterations whose duality gap stays
    at least `γ` separates any
    two columns, and any two rows, by `γ`. So `N ≤ k ^ M` and `M ≤ k ^
    N` for every integer `k`
    with `k γ > 2`, which is the bound `(1 + 2 / γ)^M` when `k` is the
    integer part of
    `2 / γ`, plus one. -/
theorem gap_shape_run (hM : 1 ≤ M) (hN : 1 ≤ N) {γ : K} {k : ℕ} (hk : 2
    < k * γ)
    (hB : ∀ i < M, ∀ j < N, |B i j| ≤ 1) {s : ℕ → Finset ℕ × Finset ℕ}
    (h0 : (s 0).1 ⊆ range M ∧ (s 0).2 ⊆ range N)
    (hs : ∀ t < M + N - 2, GapStep B M N γ (s t) (s (t + 1)))
    (hlong : ∀ t < M + N - 2, s t ≠ s (t + 1)) :
    N ≤ k ^ M ∧ M ≤ k ^ N := by
  constructor
  · have := card_le_of_sep (B := B) (R := range M) (C := range N) hk
      (fun i hi j hj => hB i (mem_range.1 hi) j (mem_range.1 hj))
      (gap_run_cols B hM hN h0 hs hlong)
    simpa using this
  · have := card_le_of_sep (B := fun a b => -B b a) (R := range N) (C :=
    range M) hk
      (fun j hj i hi => by rw [abs_neg]; exact hB i (mem_range.1 hi) j
    (mem_range.1 hj))
      (fun i hi i' hi' hne => by
        obtain ⟨j, hj, hsep⟩ := gap_run_rows B hM hN h0 hs hlong i hi i'
    hi' hne
        exact ⟨j, hj, by
          rw [show -B i j - -B i' j = -(B i j - B i' j) by ring,
    abs_neg]; exact hsep⟩)
    simpa using this

end Every

/-- The `2 × 2` matrix of the construction is `[[1, -1], [0, 1]]`. -/
theorem A_two : (A 2 0 0 : K) = 1 ∧ (A 2 0 1 : K) = -1 ∧ (A 2 1 0 : K) =
    0 ∧ (A 2 1 1 : K) = 1 := by
  simp [A]
```

```
/-- From a pure state of the `2 × 2` matrix, the best responses are row
    `j` and column `1 - i`. -/
theorem step_two_pure (hi : i < 2) (hj : j < 2) {q : Finset ℕ × Finset ℕ
    }
    (h : Step K 2 2 ({i}, {j}) q) : q = (insert j {i}, insert (1 - i)
    {j}) := by
  obtain ⟨x, y, a, b, ⟨hx, hy, -⟩, ha, ha', hb, hb', rfl⟩ := h
  have hx1 : x i = 1 := by simpa using hx.2
  have hy1 : y j = 1 := by simpa using hy.2
  have hrow : ∀ a' : ℕ, rowPay 2 {j} y a' = A 2 a' j := fun a' => by
    simp [rowPay, hy1]
  have hcol : ∀ b' : ℕ, colPay 2 {i} x b' = A 2 i b' := fun b' => by
    simp [colPay, hx1]
  simp only [hrow, hcol] at ha' hb'
  obtain ⟨e00, e01, e10, e11⟩ := A_two (K := K)
  have haj : a = j := by
    interval_cases a <;> interval_cases j
    all_goals first
      | rfl
      | (have := ha' 0 (by norm_num); have := ha' 1 (by norm_num);
    simp_all; linarith)
  have hbi : b = 1 - i := by
    interval_cases b <;> interval_cases i
    all_goals first
      | rfl
      | (have := hb' 0 (by norm_num); have := hb' 1 (by norm_num);
    simp_all; linarith)
  rw [haj, hbi]

/-- From `({k}, [2])`, the column player is pinned to column `1 - k`,
    and row `1 - k` is added. -/
theorem step_two_row (hk : k < 2) {q : Finset ℕ × Finset ℕ} (h : Step K
    2 2 ({k}, range 2) q) :
    q = (range 2, range 2) := by
  obtain ⟨x, y, a, b, ⟨hx, hy, he⟩, ha, ha', hb, -, rfl⟩ := h
  have hx1 : x k = 1 := by simpa using hx.2
  obtain ⟨e00, e01, e10, e11⟩ := A_two (K := K)
  have hy01 : y 0 + y 1 = 1 := by simpa [sum_range_succ] using hy.2
  have hy0 := hy.1 0 (by simp)
  have hy1 := hy.1 1 (by simp)
  have hrow : ∀ a' : ℕ, rowPay 2 (range 2) y a' = A 2 a' 0 * y 0 + A 2
    a' 1 * y 1 := fun a' => by
    simp [rowPay, sum_range_succ]
  have hcol : ∀ b' : ℕ, colPay 2 {k} x b' = A 2 k b' := fun b' => by
    simp [colPay, hx1]
  have hbr := he k (mem_singleton_self k)
  simp only [hrow, hcol] at hbr ha'
  have haR : a = 1 - k := by
    interval_cases k
    · have h1 := hbr 1 (by simp)
      simp only [e00, e01] at h1
      interval_cases a
      · have := ha' 1 (by norm_num); simp only [e00, e01, e10, e11] at
    this; linarith
      · rfl
    · have h1 := hbr 0 (by simp)
      simp only [e10, e11] at h1
      interval_cases a
      · rfl
      · have := ha' 0 (by norm_num); simp only [e00, e01, e10, e11] at
    this; linarith
  subst haR
  refine Prod.ext ?_ (insert_eq_of_mem (mem_range.2 hb))
  show insert (1 - k) ({k} : Finset ℕ) = range 2
  interval_cases k <;> decide
```

```
/-- From `([2], {l})`, the row player is pinned to row `l`, and column
    `1 - l` is added. -/
theorem step_two_col (hl : l < 2) {q : Finset ℕ × Finset ℕ} (h : Step K
    2 2 (range 2, {l}) q) :
    q = (range 2, range 2) := by
  obtain ⟨x, y, a, b, ⟨hx, hy, he⟩, ha, -, hb, hb', rfl⟩ := h
  have hy1 : y l = 1 := by simpa using hy.2
  obtain ⟨e00, e01, e10, e11⟩ := A_two (K := K)
  have hx01 : x 0 + x 1 = 1 := by simpa [sum_range_succ] using hx.2
  have hx0 := hx.1 0 (by simp)
  have hx1 := hx.1 1 (by simp)
  have hcol : ∀ b' : ℕ, colPay 2 (range 2) x b' = x 0 * A 2 0 b' + x 1 *
    A 2 1 b' := fun b' => by
    simp [colPay, sum_range_succ]
  have hrow : ∀ a' : ℕ, rowPay 2 {l} y a' = A 2 a' l := fun a' => by
    simp [rowPay, hy1]
  simp only [hrow, hcol] at he hb'
  have hbC : b = 1 - l := by
    interval_cases l
    · have h1 := he 0 (by simp) 0 (by simp)
      simp only [e00, e10] at h1
      interval_cases b
      · have := hb' 1 (by norm_num); simp only [e00, e01, e10, e11] at
    this; linarith
      · rfl
    · have h1 := he 1 (by simp) 1 (by simp)
      simp only [e01, e11] at h1
      interval_cases b
      · rfl
      · have := hb' 0 (by norm_num); simp only [e00, e01, e10, e11] at
    this; linarith
  subst hbC
  refine Prod.ext (insert_eq_of_mem (mem_range.2 ha)) ?_
  show insert (1 - l) ({l} : Finset ℕ) = range 2
  interval_cases l <;> decide

/-- **Any start, `M = N = 2`.** On the `2 × 2` matrix of the
    construction, runs start from every
    pure state `({i}, {j})`, and every one of them adds a strategy on
    each of its first two
    iterations, reaching the full game. So the hypothesis `(M, N) ≠ (2,
    2)` of `any_start` cannot
    be dropped. -/
theorem any_start_two (hi : i < 2) (hj : j < 2) :
    (∃ s : ℕ → Finset ℕ × Finset ℕ, s 0 = ({i}, {j}) ∧ ∀ t, Step K 2 2
    (s t) (s (t + 1))) ∧
      ∀ s : ℕ → Finset ℕ × Finset ℕ, s 0 = ({i}, {j}) → (∀ t, Step K 2
    2 (s t) (s (t + 1))) →
        s 0 ≠ s 1 ∧ s 1 ≠ s 2 ∧ s 2 = (range 2, range 2) := by
  refine ⟨?_, fun s hs0 hs => ?_⟩
  · obtain ⟨s, hs0, hs⟩ := exists_runB (A (K := K) 2) (M := 2) (N := 2)
    (by norm_num)
      (by norm_num) (p := ({i}, {j})) (singleton_nonempty _)
    (singleton_nonempty _)
    exact ⟨s, hs0, hs⟩
  have h1 := step_two_pure hi hj (hs0 ▷ hs 0)
  have h2 : s 2 = (range 2, range 2) := by
    have hs1 := hs 1
    rw [h1] at hs1
    interval_cases i <;> interval_cases j
    · rw [show ((insert 0 {0}, insert (1 - 0) {0}) : Finset ℕ × Finset ℕ
    ) = ({0}, range 2) by
        decide] at hs1
      exact step_two_row (by norm_num) hs1
```

```
    · rw [show ((insert 1 {0}, insert (1 - 0) {1}) : Finset ℕ × Finset ℕ
    ) = (range 2, {1}) by
        decide] at hs1
      exact step_two_col (by norm_num) hs1
    · rw [show ((insert 0 {1}, insert (1 - 1) {0}) : Finset ℕ × Finset ℕ
    ) = (range 2, {0}) by
        decide] at hs1
      exact step_two_col (by norm_num) hs1
    · rw [show ((insert 1 {1}, insert (1 - 1) {1}) : Finset ℕ × Finset ℕ
    ) = ({1}, range 2) by
        decide] at hs1
      exact step_two_row (by norm_num) hs1
  -- two, three, then four strategies
  have c0 : (s 0).1.card + (s 0).2.card = 2 := by rw [hs0]; simp
  have c1 : (s 1).1.card + (s 1).2.card = 3 := by
    rw [h1]; interval_cases i <;> interval_cases j <;> decide
  have c2 : (s 2).1.card + (s 2).2.card = 4 := by rw [h2]; simp
  exact ⟨fun h => by rw [h] at c0; omega, fun h => by rw [h] at c1;
    omega, h2⟩

/-! ## Infinite games

  The matrix is defined for all `i` and `j`, so it also describes the
    games in which `N`, or `M`,
  or both are infinite. A best response there ranges over infinitely
    many strategies, so a step
  needs one that is attained. It is, because every large enough finite
    game takes its best
  responses among the strategies of the next state (`stepOn_iff`). So
    the runs are those of the
  finite games, continued forever (`double_oracle_cols_inf`,
    `double_oracle_rows_inf`,
  `double_oracle_inf`). -/

section Infinite

variable (B : ℕ → ℕ → K)

/-- `q` is a successor of `p` in the game of `B` whose rows are the `i`
    with `P i` and whose
    columns are the `j` with `Q j`, finitely or infinitely many. -/
def StepOn (P Q : ℕ → Prop) (p q : Finset ℕ × Finset ℕ) : Prop :=
  ∃ (x y : ℕ → K) (i j : ℕ), IsEqB B p.1 p.2 x y ∧
    P i ∧ (∀ i', P i' → rowPayB B p.2 y i' ≤ rowPayB B p.2 y i) ∧
    Q j ∧ (∀ j', Q j' → colPayB B p.1 x j ≤ colPayB B p.1 x j') ∧ q =
    (insert i p.1, insert j p.2)

/-- With finitely many rows and columns, this is `StepB`. -/
theorem stepOn_lt : StepOn B (· < M) (· < N) = StepB B M N := rfl

/-- **Transfer.** Let an increasing family of finite games exhaust the
    rows `P` and the columns
    `Q`. If `p` has the unique successor `p'` in every large enough game
    of the family, it has the
    unique successor `p'` in the game on all of them. Best responses are
    attained there, since
    every finite game takes them among the strategies of `p'`. -/
theorem stepOn_iff {P Q : ℕ → Prop} {m m' : ℕ → ℕ} {n₀ : ℕ} {p p' :
    Finset ℕ × Finset ℕ}
    (hm : Monotone m) (hm' : Monotone m') (hP : ∀ i, P i ↔ ∃ n, i < m n)
    (hQ : ∀ j, Q j ↔ ∃ n, j < m' n) (hp' : p'.1 ⊆ range (m n₀) ∧ p'.2 ⊆
    range (m' n₀))
    (h : ∀ n ≥ n₀, ∀ q, StepB B (m n) (m' n) p q ↔ q = p') (q : Finset ℕ
     × Finset ℕ) :
    StepOn B P Q p q ↔ q = p' := by
```

```
  constructor
  · rintro ⟨x, y, i, j, hxy, hi, hi', hj, hj', rfl⟩
    obtain ⟨n1, hn1⟩ := (hP i).1 hi
    obtain ⟨n2, hn2⟩ := (hQ j).1 hj
    set n := max (max n1 n2) n₀
    exact (h n (le_max_right _ _) _).1 ⟨x, y, i, j, hxy,
      hn1.trans_le (hm ((le_max_left _ _).trans (le_max_left _ _))),
      fun i' hi'' => hi' i' ((hP i').2 ⟨n, hi''⟩),
      hn2.trans_le (hm' ((le_max_right _ _).trans (le_max_left _ _))),
      fun j' hj'' => hj' j' ((hQ j').2 ⟨n, hj''⟩), rfl⟩
  rintro rfl
  obtain ⟨x, y, i₀, j₀, hxy, hi₀, -, hj₀, -, hq₀⟩ := (h n₀ le_rfl q).2 rfl
  -- in every finite game, the best responses lie in `q`
  have key : ∀ n ≥ n₀, ∀ i < m n, ∀ j < m' n,
      (∀ i' < m n, rowPayB B p.2 y i' ≤ rowPayB B p.2 y i) →
      (∀ j' < m' n, colPayB B p.1 x j ≤ colPayB B p.1 x j') → i ∈ q.1 ∧
    j ∈ q.2 := by
    intro n hn i hi j hj hi' hj'
    rw [← (h n hn _).1 ⟨x, y, i, j, hxy, hi, hi', hj, hj', rfl⟩]
    exact ⟨mem_insert_self _ _, mem_insert_self _ _⟩
  have hbr : ∀ n ≥ n₀, (∃ i < m n, (∀ i' < m n, rowPayB B p.2 y i' ≤
    rowPayB B p.2 y i) ∧
      i ∈ q.1) ∧ ∃ j < m' n, (∀ j' < m' n, colPayB B p.1 x j ≤ colPayB B
    p.1 x j') ∧ j ∈ q.2 := by
    intro n hn
    obtain ⟨i, hi, hi'⟩ := exists_max_image (range (m n)) (rowPayB B p.2
    y)
      ⟨i₀, mem_range.2 (hi₀.trans_le (hm hn))⟩
    obtain ⟨j, hj, hj'⟩ := exists_min_image (range (m' n)) (colPayB B p.1
    x)
      ⟨j₀, mem_range.2 (hj₀.trans_le (hm' hn))⟩
    simp only [mem_range] at hi hj hi' hj'
    obtain ⟨h1, h2⟩ := key n hn i hi j hj hi' hj'
    exact ⟨⟨i, hi, hi', h1⟩, ⟨j, hj, hj', h2⟩⟩
  -- so the best responses among the strategies of `q` are best
    responses in the whole game
  obtain ⟨i, hi, hi'⟩ := exists_max_image q.1 (rowPayB B p.2 y)
    (by rw [hq₀]; exact insert_nonempty _ _)
  obtain ⟨j, hj, hj'⟩ := exists_min_image q.2 (colPayB B p.1 x)
    (by rw [hq₀]; exact insert_nonempty _ _)
  have hrow : ∀ i', P i' → rowPayB B p.2 y i' ≤ rowPayB B p.2 y i :=
    fun i' h' => by
    obtain ⟨n, hn⟩ := (hP i').1 h'
    obtain ⟨⟨a, -, ha, haq⟩, -⟩ := hbr (max n n₀) (le_max_right _ _)
    exact (ha i' (hn.trans_le (hm (le_max_left _ _)))).trans (hi' a haq)
  have hcol : ∀ j', Q j' → colPayB B p.1 x j ≤ colPayB B p.1 x j' :=
    fun j' h' => by
    obtain ⟨n, hn⟩ := (hQ j').1 h'
    obtain ⟨-, ⟨b, -, hb, hbq⟩⟩ := hbr (max n n₀) (le_max_right _ _)
    exact (hj' b hbq).trans (hb j' (hn.trans_le (hm' (le_max_left _ _))))
  have hiM : i < m n₀ := mem_range.1 (hp'.1 hi)
  have hjN : j < m' n₀ := mem_range.1 (hp'.2 hj)
  refine ⟨x, y, i, j, hxy, (hP i).2 ⟨n₀, hiM⟩, hrow, (hQ j).2 ⟨n₀, hjN⟩,
    hcol, ?_⟩
  exact ((h n₀ le_rfl _).1 ⟨x, y, i, j, hxy, hiM, fun i' h' => hrow i'
    ((hP i').2 ⟨n₀, h'⟩), hjN,
    fun j' h' => hcol j' ((hQ j').2 ⟨n₀, h'⟩), rfl⟩).symm

/-- Exchanging the players: a step on `B` is a step on its negative
    transpose, with the two sets
    swapped. -/
theorem stepOn_swap {P Q : ℕ → Prop} {p q : Finset ℕ × Finset ℕ} (h :
    StepOn B P Q p q) :
    StepOn (fun a b => -B b a) Q P p.swap q.swap := by
  obtain ⟨x, y, i, j, ⟨hx, hy, he⟩, hi, hi', hj, hj', rfl⟩ := h
```

```
  have hr : ∀ a, rowPayB (fun a b => -B b a) p.1 x a = -colPayB B p.1 x
    a := fun a => by
    simp only [rowPayB, colPayB, ← sum_neg_distrib]
    exact sum_congr rfl fun _ _ => by ring
  have hc : ∀ b, colPayB (fun a b => -B b a) p.2 y b = -rowPayB B p.2 y
    b := fun b => by
    simp only [rowPayB, colPayB, ← sum_neg_distrib]
    exact sum_congr rfl fun _ _ => by ring
  refine ⟨y, x, j, i, ⟨hy, hx, fun a ha b hb => ?_⟩, hj, fun a ha => ?_,
    hi, fun b hb => ?_, rfl⟩
  · simp only [Prod.fst_swap, Prod.snd_swap, hr, hc]
    linarith [he b hb a ha]
  · simp only [Prod.snd_swap, hr]
    linarith [hj' a ha]
  · simp only [Prod.fst_swap, hc]
    linarith [hi' b hb]

/-- ... and back, since the negative transpose of the negative transpose
    is `B`. -/
theorem stepOn_swap_iff {P Q : ℕ → Prop} {p q : Finset ℕ × Finset ℕ} :
    StepOn (fun a b => -B b a) Q P p.swap q.swap ↔ StepOn B P Q p q :=
    by
  refine ⟨fun h => ?_, stepOn_swap B⟩
  have := stepOn_swap _ h
  simp only [neg_neg, Prod.swap_swap] at this
  exact this

/-- The steps of a finite game see only its own entries. -/
theorem stepB_congr {B' : ℕ → ℕ → K} (hB : ∀ i < M, ∀ j < N, B i j =
    B' i j)
    {p q : Finset ℕ × Finset ℕ} (hp : p.1 ⊆ range M ∧ p.2 ⊆ range N) (h
    : StepB B M N p q) :
    StepB B' M N p q := by
  obtain ⟨x, y, i, j, ⟨hx, hy, he⟩, hi, hi', hj, hj', hq⟩ := h
  have hr : ∀ a < M, rowPayB B p.2 y a = rowPayB B' p.2 y a := fun a ha =
    >
    sum_congr rfl fun b hb => by rw [hB a ha b (mem_range.1 (hp.2 hb))]
  have hc : ∀ b < N, colPayB B p.1 x b = colPayB B' p.1 x b := fun b hb =
    >
    sum_congr rfl fun a ha => by rw [hB a (mem_range.1 (hp.1 ha)) b hb]
  refine ⟨x, y, i, j, ⟨hx, hy, fun a ha b hb => ?_⟩, hi, fun a ha => ?_,
    hj, fun b hb => ?_, hq⟩
  · rw [← hr a (mem_range.1 (hp.1 ha)), ← hc b (mem_range.1 (hp.2
    hb))]; exact he a ha b hb
  · rw [← hr a ha, ← hr i hi]; exact hi' a ha
  · rw [← hc b hb, ← hc j hj]; exact hj' b hb

end Infinite

/-- The run when `N` is infinite: `S_1, T_1, ..., S_M = C_M, C_{M+1}, ...
    `, which never stops. -/
def runInf (M t : ℕ) : Finset ℕ × Finset ℕ :=
  if t + 2 < 2 * M then (range (t / 2 + 1), range ((t + 1) / 2 + 1))
  else (range M, range (t + 2 - M))

theorem run_eq_runInf (h : t + 2 ≤ M + N) : run M N t = runInf M t := by
  simp only [run, runInf]
  split_ifs
  · rfl
  · rw [min_eq_left (by omega)]

theorem runInf_card (hM : 2 ≤ M) (t : ℕ) :
    (runInf M t).1.card + (runInf M t).2.card = t + 2 := by
  simp only [runInf]; split_ifs <;> simp <;> omega
```

```
/-- **Infinitely many columns.** For `M ≥ 2` and `N` infinite, the runs
    of double oracle from
    `({0}, {0})` are exactly `runInf M`, which adds a strategy at every
    iteration
    (`runInf_card`). -/
theorem double_oracle_cols_inf (hM : 2 ≤ M) (s : ℕ → Finset ℕ × Finset
    ℕ) :
    (s 0 = ({0}, {0}) ∧ ∀ t, StepOn (A (K := K) M) (· < M) (fun _ =>
    True) (s t) (s (t + 1))) ↔
      s = runInf M := by
  have h0 : runInf M 0 = ({0}, {0}) := by simp [runInf, show 2 < 2 * M
    by omega]
  rw [← h0]
  refine runs_iff (fun t q => ?_) s
  refine stepOn_iff (A M) (m := fun _ => M) (m' := fun n => n) (n₀ := M +
    t + 1)
    (fun _ _ _ => le_rfl) (fun _ _ h => h) (fun i => ⟨fun h => ⟨0, h⟩,
    fun ⟨_, h⟩ => h⟩)
    (fun j => ⟨fun _ => ⟨j + 1, by omega⟩, fun _ => trivial⟩) ?_ (fun n
    hn q => ?_) q
  · simp only [runInf]
    split_ifs <;> exact ⟨range_subset_range.2 (by omega),
    range_subset_range.2 (by omega)⟩
  · rw [stepB_A, ← run_eq_runInf (N := n) (t := t) (by omega),
      ← run_eq_runInf (N := n) (t := t + 1) (by omega)]
    exact double_oracle hM (by omega) t q

/-- **Infinitely many rows.** For `N ≥ 2` and `M` infinite, the matrix
    is the negative transpose,
    and the runs are those of `runInf N` with the players exchanged. -/
theorem double_oracle_rows_inf (hN : 2 ≤ N) (s : ℕ → Finset ℕ × Finset
    ℕ) :
    (s 0 = ({0}, {0}) ∧
      ∀ t, StepOn (fun i j => -A (K := K) N j i) (fun _ => True) (· < N)
    (s t) (s (t + 1))) ↔
      s = fun t => (runInf N t).swap := by
  have := double_oracle_cols_inf (K := K) hN (fun t => (s t).swap)
  simp only [← stepOn_swap_iff (A N) (P := (· < N)) (Q := fun _ =>
    True), Prod.swap_swap] at this
  constructor
  · rintro ⟨h0, hs⟩
    have := this.1 ⟨by rw [h0]; rfl, hs⟩
    funext t
    rw [← congrFun this t, Prod.swap_swap]
  · rintro rfl
    obtain ⟨h0, hs⟩ := this.2 (funext fun t => Prod.swap_swap _)
    exact ⟨by simpa using congrArg Prod.swap h0, hs⟩

/-- The infinite staircase, which the matrix is when `M` and `N` are
    both infinite. -/
def stair (i j : ℕ) : K := (if i = j then 1 else 0) - (if i + 1 = j then
    1 else 0)

/-- The run on the infinite staircase: `S_1, T_1, S_2, T_2, ...`, which
    never stops. -/
def runStair (t : ℕ) : Finset ℕ × Finset ℕ := (range (t / 2 + 1), range
    ((t + 1) / 2 + 1))

theorem runStair_card (t : ℕ) : (runStair t).1.card + (runStair
    t).2.card = t + 2 := by
  simp [runStair]; omega

/-- **Infinitely many rows and columns.** The runs of double oracle on
    the infinite staircase
```

```
    from `({0}, {0})` are exactly `runStair`, which adds a strategy at
    every iteration
    (`runStair_card`). -/
theorem double_oracle_inf (s : ℕ → Finset ℕ × Finset ℕ) :
    (s 0 = ({0}, {0}) ∧
      ∀ t, StepOn (stair (K := K)) (fun _ => True) (fun _ => True) (s t)
    (s (t + 1))) ↔
      s = runStair := by
  have h0 : runStair 0 = ({0}, {0}) := rfl
  rw [← h0]
  refine runs_iff (fun t q => ?_) s
  have hsub : ∀ t n, t + 2 < n → (runStair t).1 ⊆ range n ∧ (runStair
    t).2 ⊆ range n :=
    fun t n h => ⟨range_subset_range.2 (by omega), range_subset_range.2
    (by omega)⟩
  refine stepOn_iff stair (m := fun n => n) (m' := fun n => n) (n₀ := t +
    4)
    (fun _ _ h => h) (fun _ _ h => h) (fun i => ⟨fun _ => ⟨i + 1, by
    omega⟩, fun _ => trivial⟩)
    (fun j => ⟨fun _ => ⟨j + 1, by omega⟩, fun _ => trivial⟩) (hsub (t +
    1) (t + 4) (by omega))
    (fun n hn q => ?_) q
  -- the `n × n` game of the staircase is that of `A n`
  have hB : ∀ i < n, ∀ j < n, (A n i j : K) = stair i j := fun i _ j hj =
    > by
    simp [A, stair, hj]
  have hrun : ∀ t', t' + 2 < 2 * n → run n n t' = runStair t' := fun t'
    h => by
    simp only [run, runStair, if_pos h]
  rw [← hrun t (by omega), ← hrun (t + 1) (by omega)]
  have e := double_oracle (K := K) (M := n) (N := n) (by omega) le_rfl t
    q
  rw [← stepB_A] at e
  rw [← e, hrun t (by omega)]
  exact ⟨stepB_congr _ (fun i hi j hj => (hB i hi j hj).symm) (hsub t n
    (by omega)),
    stepB_congr _ hB (hsub t n (by omega))⟩

/-! ## Fewer columns than rows

  For `N < M`, the matrix is the negative transpose of the `N × M` one,
    which exchanges the
  players. A step on it is a step on the `N × M` matrix with the two
    sets swapped
  (`stepOn_swap_iff`), and an approximate step is one with the two
    tolerances swapped too
  (`stepApproxB_swap_iff`). So the main theorem, both of its
    corollaries, and the gap bound hold
  at every shape. -/

section Swap

variable (B : ℕ → ℕ → K)

/-- `q` is an approximate successor of the state `p` in the `M × N`
    matrix `B`, as in
    `StepApprox`. -/
def StepApproxB (M N : ℕ) (δ εR εC : K) (p q : Finset ℕ × Finset ℕ) :
    Prop :=
  ∃ (x y : ℕ → K) (i j : ℕ),
    (IsMix p.1 x ∧ IsMix p.2 y ∧ ∀ a ∈ p.1, ∀ b ∈ p.2, rowPayB B p.2 y a
    ≤ colPayB B p.1 x b + δ) ∧
    i < M ∧ (∀ i' < M, rowPayB B p.2 y i' ≤ rowPayB B p.2 y i + εR) ∧
    j < N ∧ (∀ j' < N, colPayB B p.1 x j ≤ colPayB B p.1 x j' + εC) ∧
    q = (insert i p.1, insert j p.2)
```

```
/-- On the matrix of the construction, this is `StepApprox`. -/
theorem stepApproxB_A {δ εR εC : K} :
    StepApproxB (A (K := K) M) M N δ εR εC = StepApprox M N δ εR εC := rfl

/-- Exchanging the players: an approximate step on `B` is one on its
    negative transpose, with the
    two sets swapped and the two tolerances swapped. -/
theorem stepApproxB_swap {δ εR εC : K} {p q : Finset ℕ × Finset ℕ}
    (h : StepApproxB B M N δ εR εC p q) :
    StepApproxB (fun a b => -B b a) N M δ εC εR p.swap q.swap := by
  obtain ⟨x, y, i, j, ⟨hx, hy, he⟩, hi, hi', hj, hj', rfl⟩ := h
  have hr : ∀ a, rowPayB (fun a b => -B b a) p.1 x a = -colPayB B p.1 x
    a := fun a => by
    simp only [rowPayB, colPayB, ← sum_neg_distrib]
    exact sum_congr rfl fun _ _ => by ring
  have hc : ∀ b, colPayB (fun a b => -B b a) p.2 y b = -rowPayB B p.2 y
    b := fun b => by
    simp only [rowPayB, colPayB, ← sum_neg_distrib]
    exact sum_congr rfl fun _ _ => by ring
  refine ⟨y, x, j, i, ⟨hy, hx, fun a ha b hb => ?_⟩, hj, fun a ha => ?_,
    hi, fun b hb => ?_, rfl⟩
  · simp only [Prod.fst_swap, Prod.snd_swap, hr, hc]
    linarith [he b hb a ha]
  · simp only [Prod.snd_swap, hr]
    linarith [hj' a ha]
  · simp only [Prod.fst_swap, hc]
    linarith [hi' b hb]

/-- ... and back, since the negative transpose of the negative transpose
    is `B`. -/
theorem stepApproxB_swap_iff {δ εR εC : K} {p q : Finset ℕ × Finset ℕ} :
    StepApproxB (fun a b => -B b a) N M δ εC εR p.swap q.swap ↔
    StepApproxB B M N δ εR εC p q := by
  refine ⟨fun h => ?_, stepApproxB_swap B⟩
  have := stepApproxB_swap _ h
  simp only [neg_neg, Prod.swap_swap] at this
  exact this

end Swap

/-- **Main theorem, fewer columns than rows.** For `2 ≤ N < M`, the `M ×
    N` matrix is the negative
    transpose of the `N × M` one, and the runs of double oracle from
    `({0}, {0})` are exactly
    those of `run N M` with the players exchanged. -/
theorem double_oracle_swap (hN : 2 ≤ N) (hNM : N < M) (s : ℕ → Finset ℕ
    × Finset ℕ) :
    (s 0 = ({0}, {0}) ∧ ∀ t, StepB (fun i j => -A (K := K) N j i) M N (s
    t) (s (t + 1))) ↔
      s = fun t => (run N M t).swap := by
  have := double_oracle_worst_case (K := K) hN hNM.le (fun t => (s
    t).swap)
  rw [← stepB_A] at this
  simp only [← stepOn_lt, ← stepOn_swap_iff (A N) (P := (· < N)) (Q :=
    (· < M)),
    Prod.swap_swap] at this
  constructor
  · rintro ⟨h0, hs⟩
    have := this.1 ⟨by rw [h0]; rfl, hs⟩
    funext t
    rw [← congrFun this t, Prod.swap_swap]
  · rintro rfl
    obtain ⟨h0, hs⟩ := this.2 (funext fun t => Prod.swap_swap _)
    exact ⟨by simpa using congrArg Prod.swap h0, hs⟩
```

```
/-- **Perturbed matrices, fewer columns than rows.** For `2 ≤ N < M`,
    there is a radius `ρ > 0`
    such that, on any `M × N` matrix whose entries are within `ρ` of
    those of the negative
    transpose, the runs of double oracle from `({0}, {0})` are those of
    `run N M` with the players
    exchanged. -/
theorem double_oracle_perturbed_swap (hN : 2 ≤ N) (hNM : N < M) :
    ∃ ρ > (0 : K), ∀ B : ℕ → ℕ → K, (∀ i < M, ∀ j < N, |B i j - -A N j
    i| < ρ) →
      ∀ s : ℕ → Finset ℕ × Finset ℕ,
        (s 0 = ({0}, {0}) ∧ ∀ t, StepB B M N (s t) (s (t + 1))) ↔
          s = fun t => (run N M t).swap := by
  obtain ⟨ρ, hρ, h⟩ := double_oracle_perturbed (K := K) hN hNM.le
  refine ⟨ρ, hρ, fun B hB s => ?_⟩
  have := h (fun a b => -B b a) (fun a ha b hb => by
      have e := hB b hb a ha
      rwa [show B b a - -A N a b = -(-B b a - A N a b) by ring, abs_neg]
    at e)
    (fun t => (s t).swap)
  simp only [← stepOn_lt, stepOn_swap_iff B (P := (· < M)) (Q := (· <
    N))] at this
  constructor
  · rintro ⟨h0, hs⟩
    have := this.1 ⟨by rw [h0]; rfl, hs⟩
    funext t
    rw [← congrFun this t, Prod.swap_swap]
  · rintro rfl
    obtain ⟨h0, hs⟩ := this.2 (funext fun t => Prod.swap_swap _)
    exact ⟨by simpa using congrArg Prod.swap h0, hs⟩

/-- **Approximate oracles, fewer columns than rows.** For `2 ≤ N < M`,
    there are thresholds
    `δ₀, ε₀ > 0` below which the runs of the approximate algorithm on the
    negative transpose, from
    `({0}, {0})`, are those of `run N M` with the players exchanged. The
    players' tolerances trade
    places, and `double_oracle_approx` bounds both by the same `ε₀`. -/
theorem double_oracle_approx_swap (hN : 2 ≤ N) (hNM : N < M) :
    ∃ δ₀ > (0 : K), ∃ ε₀ > (0 : K), ∀ δ εR εC : K, 0 ≤ δ → δ < δ₀ → 0 ≤ ε
    R → εR < ε₀ →
      0 ≤ εC → εC < ε₀ →
      ∀ s : ℕ → Finset ℕ × Finset ℕ,
        (s 0 = ({0}, {0}) ∧
          ∀ t, StepApproxB (fun i j => -A (K := K) N j i) M N δ εR εC (s
    t) (s (t + 1))) ↔
          s = fun t => (run N M t).swap := by
  obtain ⟨δ₀, hδ₀, ε₀, hε₀, h⟩ := double_oracle_approx (K := K) hN hNM.le
  refine ⟨δ₀, hδ₀, ε₀, hε₀, fun δ εR εC hδ hδ0 hεR hεR0 hεC hεC0 s => ?_⟩
  have := h δ εC εR hδ hδ0 hεC hεC0 hεR hεR0 (fun t => (s t).swap)
  simp only [← stepApproxB_A, ← stepApproxB_swap_iff (A N),
    Prod.swap_swap] at this
  constructor
  · rintro ⟨h0, hs⟩
    have := this.1 ⟨by rw [h0]; rfl, hs⟩
    funext t
    rw [← congrFun this t, Prod.swap_swap]
  · rintro rfl
    obtain ⟨h0, hs⟩ := this.2 (funext fun t => Prod.swap_swap _)
    exact ⟨by simpa using congrArg Prod.swap h0, hs⟩

/-- For `2 ≤ N < M`, the gap bound of `gap_run_le` holds along the runs
    of `double_oracle_swap` -
```

```
    too: exchanging the players exchanges the two best-response payoffs
    and negates both, which
    leaves the duality gap as it is. -/
theorem gap_run_le_swap (hN : 2 ≤ N) (hNM : N < M) {t : ℕ} (ht : t < M +
    N - 2) {x y : ℕ → K}
    (h : IsEqB (fun a b => -A N b a) (run N M t).2 (run N M t).1 x y) {i
    j : ℕ} (hi : i < M)
    (hi' : ∀ i' < M, rowPayB (fun a b => -A N b a) (run N M t).1 y i' ≤
      rowPayB (fun a b => -A N b a) (run N M t).1 y i) (hj : j < N)
    (hj' : ∀ j' < N, colPayB (fun a b => -A N b a) (run N M t).2 x j ≤
      colPayB (fun a b => -A N b a) (run N M t).2 x j') :
    rowPayB (fun a b => -A N b a) (run N M t).1 y i -
      colPayB (fun a b => -A N b a) (run N M t).2 x j ≤ 4 / (t + 1) := by
  have hr : ∀ (R : Finset ℕ) (w : ℕ → K) a,
      rowPayB (fun a b => -A N b a) R w a = -colPay N R w a := fun R w a
    => by
    simp only [rowPayB, colPay, ← sum_neg_distrib]
    exact sum_congr rfl fun _ _ => by ring
  have hc : ∀ (C : Finset ℕ) (w : ℕ → K) b,
      colPayB (fun a b => -A N b a) C w b = -rowPay N C w b := fun C w b
    => by
    simp only [colPayB, rowPay, ← sum_neg_distrib]
    exact sum_congr rfl fun _ _ => by ring
  obtain ⟨hx, hy, he⟩ := h
  simp only [hr, hc] at hi' hj' he ⊢
  have := gap_run_le (K := K) hN hNM.le (by omega) (x := y) (y := x)
    ⟨hy, hx, fun a ha b hb => by linarith [he b hb a ha]⟩ hj
    (fun a ha => by linarith [hj' a ha]) hi (fun b hb => by linarith
    [hi' b hb])
  linarith

/-! ## The corridor and the golden ratio

  The corridor column of index `n` pays a negative amount exactly when
    the row player's ratio
  `x 1 / x 0` lies on one side of `F(n + 1) / F(n)` (`pay_neg_iff`), a
    convergent of the golden
  ratio `φ`, whose partial quotients are all `1` (`goldenRatio_cf`,
    `goldenRatio_convs`). So these
  denominators are the least that a continued fraction allows
    (`goldenRatio_dens_le`). The
  equilibrium ratio at `C_k` (`ratio_C`) tends to `φ`, alternating
    around it (`ratio_golden`).
  By d'Ocagne's identity, no two corridor columns are parallel
    (`det_corr`), and staircase column
  `1` is `M` times the corridor column of index `1` (`stair_one`). Along
    a geometric sequence
  instead, the first two corridor columns would do all that the others
    do (`geom_corr`). -/

/-- In rows `0` and `1`, the corridor columns of indices `a < b` meet in
    a matrix of determinant
    `(-1)^(b + 1) F(b - a) / (M^2 F(a) F(a + 1) F(b) F(b + 1))`, which
    is not zero. So no two
    corridor columns are parallel. -/
theorem det_corr (hM : 0 < M) {a b : ℕ} (ha : 0 < a) (hab : a < b) :
    (s M a * s M (b + 1) - s M b * s M (a + 1) : K) =
      (-1) ^ (b + 1) * fib (b - a) / (M ^ 2 * fib a * fib (a + 1) * fib
    b * fib (b + 1)) ∧
      (s M a * s M (b + 1) - s M b * s M (a + 1) : K) ≠ 0 := by
  have hF : ∀ n, 0 < n → (fib n : K) ≠ 0 := fun n hn => by
    exact_mod_cast (fib_pos.2 hn).ne'
  have hM' : (M : K) ≠ 0 := by exact_mod_cast hM.ne'
  have ha' := hF a ha
  have ha1 := hF (a + 1) (by omega)
```

```
  have hb := hF b (by omega)
  have hb1 := hF (b + 1) (by omega)
  have hd := docagne (K := K) hab.le
  have hsq : ((-1 : K) ^ a) ^ 2 = 1 := by rw [← pow_mul, pow_mul']; simp
  have e : (s M a * s M (b + 1) - s M b * s M (a + 1) : K) =
      (-1) ^ (b + 1) * fib (b - a) / (M ^ 2 * fib a * fib (a + 1) * fib
    b * fib (b + 1)) := by
    simp only [s]
    rw [eq_div_iff (by positivity)]
    field_simp
    rw [pow_succ, pow_succ]
    linear_combination -((-1 : K) ^ a * (-1) ^ b) * hd - (-1) ^ b * fib
    (b - a) * hsq
  refine ⟨e, ?_⟩
  rw [e]
  exact div_ne_zero (mul_ne_zero (pow_ne_zero _ (by norm_num)) (hF _ (by
    omega))) (by positivity)

/-- In rows `0` and `1`, staircase column `1` is `M` times the corridor
    column of index `1`. So
    `det_corr` covers the two tight columns of the handoff too, of
    indices `1` and `4`. -/
theorem stair_one (hM : 2 ≤ M) : (A M 0 1 : K) = M * s M 1 ∧ (A M 1 1 :
    K) = M * s M 2 := by
  have : (M : K) ≠ 0 := by exact_mod_cast (show M ≠ 0 by omega)
  constructor <;> simp [A, s, show 1 < M by omega] <;> field_simp

/-- Entries `c r^n` along a geometric sequence would instead make every
    two corridor columns
    proportional. If they still alternated and shrank, `-1 ≤ r ≤ 0`,
    then against every `(a, b)`
    the better of the columns of indices `m` and `m + 1` would pay at
    most what every later one
    pays, so the first two corridor columns would suffice for the column
    player. -/
theorem geom_corr {c r : K} (hr : r ≤ 0) (hr1 : -1 ≤ r) (a b : K) {m n
    : ℕ} (hmn : m ≤ n) :
    c * r ^ m * (c * r ^ (n + 1)) - c * r ^ n * (c * r ^ (m + 1)) = 0 ∧
      min (a * (c * r ^ m) + b * (c * r ^ (m + 1)))
        (a * (c * r ^ (m + 1)) + b * (c * r ^ (m + 1 + 1)))
        ≤ a * (c * r ^ n) + b * (c * r ^ (n + 1)) := by
  refine ⟨by ring, ?_⟩
  have hp : ∀ k, a * (c * r ^ k) + b * (c * r ^ (k + 1)) = c * (a + r *
    b) * r ^ k :=
    fun k => by ring
  simp only [hp]
  -- a payoff `X ≤ 0` only grows along the sequence
  have key : ∀ X : K, X ≤ 0 → ∀ d, X ≤ X * r ^ d := fun X hX d => by
    have h1 : |r ^ d| ≤ 1 := by
      rw [abs_pow]; exact pow_le_one₀ (abs_nonneg r) (abs_le.2 ⟨hr1, by
    linarith⟩)
    nlinarith [le_abs_self (r ^ d)]
  set w := c * (a + r * b)
  obtain ⟨d, rfl⟩ : ∃ d, n = m + d := ⟨n - m, by omega⟩
  rcases le_or_gt (w * r ^ m) 0 with h | h
  · calc _ ≤ w * r ^ m := min_le_left _ _
      _ ≤ w * r ^ m * r ^ d := key _ h d
      _ = w * r ^ (m + d) := by ring
  · have h1 : w * r ^ (m + 1) ≤ 0 := by
      rw [pow_succ, ← mul_assoc]; exact mul_nonpos_of_nonneg_of_nonpos
    h.le hr
    rcases d with _ | d
    · exact (min_le_right _ _).trans (h1.trans h.le)
    · calc _ ≤ w * r ^ (m + 1) := min_le_right _ _
        _ ≤ w * r ^ (m + 1) * r ^ d := key _ h1 d
```

```
        _ = w * r ^ (m + (d + 1)) := by ring

/-- Against `x 0 = a > 0` and `x 1 = b`, the corridor column of index `n
    ≥ 1` pays a negative
    amount exactly when the ratio `b / a` lies above `F(n + 1) / F(n)`
    for even `n`, and below it
    for odd `n`. -/
theorem pay_neg_iff (hM : 0 < M) (hn : 0 < n) {a b : K} (ha : 0 < a) :
    pay M a b n < 0 ↔ (Even n ∧ (fib (n + 1) : K) / fib n < b / a) ∨
      (Odd n ∧ b / a < (fib (n + 1) : K) / fib n) := by
  have hF : (0 : K) < fib n := by exact_mod_cast fib_pos.2 hn
  have hF1 : (0 : K) < fib (n + 1) := by exact_mod_cast fib_pos.2 (by
    omega)
  have hM' : (0 : K) < M := by exact_mod_cast hM
  rw [pay_g hM hn, div_lt_iff₀ (by positivity), zero_mul, g,
    div_lt_div_iff₀ hF ha,
    div_lt_div_iff₀ ha hF]
  rcases Nat.even_or_odd n with he | ho
  · rw [he.neg_one_pow, one_mul]
    simp only [he, true_and, Nat.not_odd_iff_even.2 he, false_and,
    or_false]
    constructor <;> intro <;> linarith
  · rw [ho.neg_one_pow]
    simp only [ho, true_and, Nat.not_even_iff_odd.2 ho, false_and,
    false_or]
    constructor <;> intro <;> linarith

/-- At `C_{M+e+2}`, with `K = e + 5`, every equilibrium has the ratio
    `x 1 / x 0 = F(K + 1)^2 / (F(K - 1) F(K + 2))`. -/
theorem ratio_C (hM : 2 ≤ M) (e : ℕ) {x y : ℕ → K}
    (h : IsEq M (range M) (range (M + e + 2)) x y) :
    x 1 / x 0 = (fib (e + 6) : K) ^ 2 / (fib (e + 4) * fib (e + 7)) := by
  obtain ⟨hx0, hx, -⟩ := pin_C hM e h rfl rfl rfl
  rw [hx0, hx 1 le_rfl (by omega)]
  have f6 : (fib (e + 6) : K) = fib (e + 4) + fib (e + 5) := by
    exact_mod_cast fib_add_two
  have f7 : (fib (e + 7) : K) = fib (e + 5) + fib (e + 6) := by
    exact_mod_cast fib_add_two
  have hp : (0 : K) < fib (e + 4) := by exact_mod_cast fib_pos.2 (by
    omega)
  have hq : (0 : K) < fib (e + 5) := by exact_mod_cast fib_pos.2 (by
    omega)
  have hM' : (0 : K) < M := by exact_mod_cast (show 0 < M by omega)
  have hC : (corrMass M e : K) ≠ 0 := (corrMass_pos hM e).ne'
  rw [f7, f6]
  field_simp
  ring

section Golden

open Real Filter

/-- **The continued fraction of the golden ratio.** Every partial
    quotient of `φ` is `1`. -/
theorem goldenRatio_cf :
    (GenContFract.of goldenRatio).h = 1 ∧
      ∀ n, (GenContFract.of goldenRatio).s.get? n = some ⟨1, 1⟩ := by
  have h1 : Int.floor goldenRatio = 1 := by
    rw [Int.floor_eq_iff]
    constructor <;> push_cast <;> linarith [one_lt_goldenRatio,
    goldenRatio_lt_two]
  have hfr : Int.fract goldenRatio = goldenRatio - 1 := by
    rw [Int.fract, h1]; simp
  have hinv : (Int.fract goldenRatio)⁻¹ = goldenRatio := by
```

```
    rw [hfr]
    exact inv_eq_of_mul_eq_one_left (by linear_combination
    goldenRatio_sq)
  refine ⟨by rw [GenContFract.of_h_eq_floor, h1]; simp, fun n => ?_⟩
  induction n with
  | zero =>
    have := GenContFract.of_s_head (v := goldenRatio)
      (by rw [hfr]; linarith [one_lt_goldenRatio])
    rw [hinv, h1, Int.cast_one] at this
    exact this
  | succ n ih => rw [GenContFract.of_s_succ, hinv, ih]

/-- So the convergents of `φ` are `F(n + 2) / F(n + 1)`. -/
theorem goldenRatio_convs (n : ℕ) :
    (GenContFract.of goldenRatio).nums n = fib (n + 2) ∧
      (GenContFract.of goldenRatio).dens n = fib (n + 1) ∧
      (GenContFract.of goldenRatio).convs n = fib (n + 2) / fib (n + 1)
    := by
  obtain ⟨hh, hs⟩ := goldenRatio_cf
  have key : ∀ n, (GenContFract.of goldenRatio).nums n = fib (n + 2) ∧
      (GenContFract.of goldenRatio).dens n = fib (n + 1) := by
    intro n
    induction n using Nat.strong_induction_on with
    | _ n ih =>
      rcases n with _ | _ | n
      · rw [GenContFract.zeroth_num_eq_h, GenContFract.zeroth_den_eq_one,
    hh]
        simp
      · rw [GenContFract.first_num_eq (hs 0), GenContFract.first_den_eq
    (hs 0), hh]
        norm_num
      · obtain ⟨a0, b0⟩ := ih n (by omega)
        obtain ⟨a1, b1⟩ := ih (n + 1) (by omega)
        rw [GenContFract.nums_recurrence (hs (n + 1)) a0 a1,
          GenContFract.dens_recurrence (hs (n + 1)) b0 b1]
        constructor <;> push_cast [fib_add_two] <;> ring
  obtain ⟨h1, h2⟩ := key n
  exact ⟨h1, h2, by rw [GenContFract.conv_eq_num_div_den, h1, h2]⟩

/-- Their denominators are the least that a continued fraction allows:
    those of every real
    number are at least `F(n + 1)` too. -/
theorem goldenRatio_dens_le (n : ℕ) {v : ℝ}
    (hv : n = 0 ∨ ¬(GenContFract.of v).TerminatedAt (n - 1)) :
    (GenContFract.of goldenRatio).dens n ≤ (GenContFract.of v).dens n :=
    by
  rw [(goldenRatio_convs n).2.1]
  exact GenContFract.succ_nth_fib_le_of_nth_den hv

/-- The equilibrium ratio of `ratio_C`, which is `F(n + 2)^2 / (F(n) F(n
    + 3))` for `n = K - 1`,
    tends to `φ`, alternating around it: above for even `n` and below
    for odd `n`. -/
theorem ratio_golden :
    Tendsto (fun n => (fib (n + 2) : ℝ) ^ 2 / (fib n * fib (n + 3)))
    atTop (nhds goldenRatio) ∧
      ∀ n, 0 < n →
        0 < (-1) ^ n * ((fib (n + 2) : ℝ) ^ 2 / (fib n * fib (n + 3)) -
    goldenRatio) := by
  constructor
  · -- the product of three ratios of consecutive Fibonacci numbers
    have hT := tendsto_fib_succ_div_fib_atTop
    have h := ((hT.comp (tendsto_add_atTop_nat 1)).mul hT).div
      (hT.comp (tendsto_add_atTop_nat 2)) goldenRatio_ne_zero
```

```
    rw [show goldenRatio * goldenRatio / goldenRatio = goldenRatio by
    field_simp] at h
    refine h.congr' ?_
    filter_upwards [eventually_ge_atTop 1] with n hn
    have hF : ∀ m, 0 < m → (fib m : ℝ) ≠ 0 := fun m hm => by
    exact_mod_cast (fib_pos.2 hm).ne'
    have := hF n hn
    have := hF (n + 1) (by omega)
    have := hF (n + 2) (by omega)
    have := hF (n + 3) (by omega)
    simp only [Function.comp, Pi.div_apply]
    field_simp
  intro n hn
  have ha : (1 : ℝ) ≤ fib n := by exact_mod_cast fib_pos.2 hn
  have hb : (1 : ℝ) ≤ fib (n + 1) := by exact_mod_cast fib_pos.2 (by
    omega)
  have f2 : (fib (n + 2) : ℝ) = fib n + fib (n + 1) := by exact_mod_cast
    fib_add_two
  have f3 : (fib (n + 3) : ℝ) = fib n + 2 * fib (n + 1) := by
    have : (fib (n + 3) : ℝ) = fib (n + 1) + fib (n + 2) := by
    exact_mod_cast fib_add_two
    rw [this, f2]; ring
  -- Cassini: `b^2 - a (a + b) = (-1)^n` for `a = F(n)` and `b = F(n +
    1)`
  have hσ : (fib (n + 1) : ℝ) ^ 2 - fib n * (fib n + fib (n + 1)) = (-1)
    ^ n := by
    have := cassini (K := ℝ) n
    rw [pow_succ (-1 : ℝ), f2] at this
    linarith
  rw [f2, f3]
  generalize (fib n : ℝ) = a at *
  generalize (fib (n + 1) : ℝ) = b at *
  have hσ2 : ((-1 : ℝ) ^ n) ^ 2 = 1 := by rw [← pow_mul, mul_comm,
    pow_mul]; simp
  set r := (a + b) ^ 2 / (a * (a + 2 * b)) with hr
  -- '(r - φ) (r - ψ) = r^2 - r - 1 = (-1)^n (a^2 + 3 a b + b^2) / (a (a
    + 2 b))^2'
  have hq : (r - goldenRatio) * (r - goldenConj)
      = (-1) ^ n * (a ^ 2 + 3 * a * b + b ^ 2) / (a * (a + 2 * b)) ^ 2 :=
     by
    have e1 : (r - goldenRatio) * (r - goldenConj) = r ^ 2 - r - 1 := by
      linear_combination (-r) * goldenRatio_add_goldenConj +
    goldenRatio_mul_goldenConj
    rw [e1, hr, ← hσ]
    field_simp
    ring
  have hψ : 0 < r - goldenConj := by
    have : 0 < r := by positivity
    linarith [goldenConj_neg]
  have key : 0 < (-1) ^ n * (r - goldenRatio) * (r - goldenConj) := by
    rw [mul_assoc, hq, mul_div_assoc', ← mul_assoc, ← sq, hσ2, one_mul]
    positivity
  exact (mul_pos_iff_of_pos_right hψ).1 key

end Golden

/-! ## Anytime double oracle

  In anytime double oracle (ADO), each player plays a least-exploitable
    distribution on its
  population, an equilibrium strategy of the game in which the opponent
    keeps all of its
  strategies. The opponent's best response against it is added, a novel
    one whenever there is
```

```
  one. On our matrix, ADO follows `run M N` until every row is present.
    Its row distribution is
  then optimal in the full game, so only the columns tight against it
    can still enter, and among
  the corridor columns these are the last two (`ado_iterations`). -/

/-- `C'` is `C` with a novel best response added if there is one, and
    `C` otherwise, where `br`
    says which `j < N` are best responses. -/
def AddNovel (N : ℕ) (br : ℕ → Prop) (C C' : Finset ℕ) : Prop :=
  (∃ j < N, br j ∧ j ∉ C ∧ C' = insert j C) ∨ ((∀ j < N, br j → j ∈ C) ∧
    C' = C)

/-- `q` is a successor of the state `p` in ADO: `x` is a
    least-exploitable distribution of the
    rows `p.1`, `y` one of the columns `p.2`, and each population gains
    a novel best response
    against the other's distribution if there is one. -/
def StepADO (K : Type*) [Field K] [LinearOrder K] [IsStrictOrderedRing
    K] (M N : ℕ)
    (p q : Finset ℕ × Finset ℕ) : Prop :=
  ∃ x y x' y' : ℕ → K, IsEq M p.1 (range N) x y' ∧ IsEq M (range M) p.2
    x' y ∧
    AddNovel M (fun i => ∀ i' < M, rowPay M p.2 y i' ≤ rowPay M p.2 y i)
    p.1 q.1 ∧
    AddNovel N (fun j => ∀ j' < N, colPay M p.1 x j ≤ colPay M p.1 x j')
    p.2 q.2

theorem AddNovel.subset {br : ℕ → Prop} {C C' : Finset ℕ} (h : AddNovel
    N br C C') :
    C ⊆ C' := by
  rcases h with ⟨j, -, -, -, rfl⟩ | ⟨-, rfl⟩
  exacts [subset_insert _ _, subset_refl _]

/-- When the only novel best response is `j`, it is added. -/
theorem AddNovel.eq {br : ℕ → Prop} {C C' : Finset ℕ} (h : AddNovel N
    br C C') {j : ℕ}
    (hj : j < N) (hbr : br j) (hjC : j ∉ C) (hone : ∀ j' < N, br j' →
    j' ∉ C → j' = j) :
    C' = insert j C := by
  rcases h with ⟨j', hj', hbr', hj'C, rfl⟩ | ⟨hall, -⟩
  · rw [hone j' hj' hbr' hj'C]
  · exact absurd (hall j hj hbr) hjC

/-- When there is no novel best response, nothing is added. -/
theorem AddNovel.eq_self {br : ℕ → Prop} {C C' : Finset ℕ} (h :
    AddNovel N br C C')
    (hall : ∀ j < N, br j → j ∈ C) : C' = C := by
  rcases h with ⟨j, hj, hbr, hjC, -⟩ | ⟨-, rfl⟩
  exacts [absurd (hall j hj hbr) hjC, rfl]

/-- ADO steps exist from every state with nonempty populations, by the
    minimax theorem. -/
theorem exists_stepADO (hM : 0 < M) (hN : 0 < N) {p : Finset ℕ × Finset
    ℕ} (hp1 : p.1.Nonempty)
    (hp2 : p.2.Nonempty) : ∃ q, StepADO K M N p q := by
  obtain ⟨x, y', hx⟩ := exists_isEqB (A (K := K) M) hp1
    (nonempty_range_iff.2 hN.ne')
  obtain ⟨x', y, hy⟩ := exists_isEqB (A (K := K) M) (nonempty_range_iff.2
    hM.ne') hp2
  have hnovel : ∀ (n : ℕ) (br : ℕ → Prop) (C : Finset ℕ), ∃ C',
    AddNovel n br C C' := by
    intro n br C
    by_cases h : ∃ j < n, br j ∧ j ∉ C
    · obtain ⟨j, hj, hbr, hjC⟩ := h
```

```
      exact ⟨_, Or.inl ⟨j, hj, hbr, hjC, rfl⟩⟩
    · push Not at h
      exact ⟨C, Or.inr ⟨h, rfl⟩⟩
  obtain ⟨R', hR'⟩ := hnovel M (fun i => ∀ i' < M, rowPay M p.2 y i' ≤
    rowPay M p.2 y i) p.1
  obtain ⟨C', hC'⟩ := hnovel N (fun j => ∀ j' < N, colPay M p.1 x j ≤
    colPay M p.1 x j') p.2
  exact ⟨(R', C'), x, y, x', y', hx, hy, hR', hC'⟩

/-- So an ADO run starts from every such state. -/
theorem exists_runADO (hM : 0 < M) (hN : 0 < N) {p : Finset ℕ × Finset ℕ
    } (hp1 : p.1.Nonempty)
    (hp2 : p.2.Nonempty) :
    ∃ s : ℕ → Finset ℕ × Finset ℕ, s 0 = p ∧ ∀ t, StepADO K M N (s t)
    (s (t + 1)) := by
  have hstep : ∀ q : {q : Finset ℕ × Finset ℕ // q.1.Nonempty ∧
    q.2.Nonempty},
      ∃ q' : {q : Finset ℕ × Finset ℕ // q.1.Nonempty ∧ q.2.Nonempty},
    StepADO K M N q.1 q'.1 :=
    fun q => by
      obtain ⟨q', hq'⟩ := exists_stepADO (K := K) hM hN q.2.1 q.2.2
      obtain ⟨x, y, x', y', -, -, h1, h2⟩ := id hq'
      exact ⟨⟨q', q.2.1.mono (AddNovel.subset h1), q.2.2.mono
    (AddNovel.subset h2)⟩, hq'⟩
  choose f hf using hstep
  refine ⟨fun t => (f^[t] ⟨p, hp1, hp2⟩).1, rfl, fun t => ?_⟩
  simp only [Function.iterate_succ_apply']
  exact hf _

/-- At the rows `[k]`, `0 < k < M ≤ N`, a least-exploitable distribution
    is the equilibrium
    strategy of `T_k`, against which column `k` is a best response and
    no later column is. -/
theorem ado_cols (hk : 0 < k) (hkM : k < M) (hMN : M ≤ N) {x y' : ℕ → K}
    (h : IsEq M (range k) (range N) x y') :
    (∀ j' < N, colPay M (range k) x k ≤ colPay M (range k) x j') ∧
      ∀ j < N, (∀ j' < N, colPay M (range k) x j ≤ colPay M (range k) x
    j') → j ≤ k := by
  obtain ⟨x₀, y₀, h₀⟩ := (cert_T (K := K) hk hkM).1
  obtain ⟨-, hy₀⟩ := pin_T hk hkM h₀
  set W : K := 2 / (k * (k + 1)) with hW_def
  have hW : 0 < W := by positivity
  have hrow : ∀ i < k, rowPay M (range (k + 1)) y₀ i = -W := fun i hi =>
    by
    rw [rowPay_stair (by omega), if_pos (by omega), if_pos (by omega),
    hy₀ i (by omega),
      hy₀ (i + 1) (by omega)]
    push_cast
    ring
  -- `x₀` holds every column to `-W`, the value of `T_k`, and by weak
    duality so does `x`
  have hge₀ : ∀ j ∈ range N, -W ≤ colPay M (range k) x₀ j := fun j hj =>
    by
    rcases le_or_gt j k with hjk | hjk
    · rw [← hrow 0 hk]; exact h₀.2.2 0 (mem_range.2 hk) j (mem_range.2
    (by omega))
    · have h1 := h₀.2.2 0 (mem_range.2 hk) k (mem_range.2 (by omega))
      rw [hrow 0 hk] at h1
      exact h1.trans (cert_T_out hk hkM h₀ hjk).le
  obtain ⟨i₀, hi₀, hv⟩ := exists_row_ge h₀.1 h.2.1 hge₀
    (nonempty_range_iff.2 hk.ne')
  have hge : ∀ j ∈ range N, -W ≤ colPay M (range k) x j := fun j hj =>
    hv.trans (h.2.2 i₀ hi₀ j hj)
  -- so `x` is an equilibrium strategy of `T_k`
```

```
  have hT : IsEq M (range k) (range (k + 1)) x y₀ := ⟨h.1, h₀.2.1, fun i
    hi j hj => by
    rw [hrow i (mem_range.1 hi)]; exact hge j (mem_range.2 (by simp at
    hj; omega))⟩
  obtain ⟨hx, -⟩ := pin_T hk hkM hT
  have hck : colPay M (range k) x k = -W := by
    obtain ⟨m, rfl⟩ : ∃ m, k = m + 1 := ⟨k - 1, by omega⟩
    rw [colPay_succ hkM, if_neg (by omega), if_pos (by omega), hx m (by
    omega), hW_def]
    push_cast
    ring
  refine ⟨fun j' hj' => hck ▷ hge j' (mem_range.2 hj'), fun j hj hbr =>
    ?_⟩
  by_contra hjk
  exact absurd (hbr k (by omega)) (not_le.2 (cert_T_out hk hkM hT (by
    omega)))

/-- At the columns `[c]`, `0 < c ≤ M`, a least-exploitable distribution
    is the equilibrium
    strategy of `S_c`, against which the best-response rows are exactly
    those of `[c]`. -/
theorem ado_rows (hc : 0 < c) (hcM : c ≤ M) {x' y : ℕ → K} (h : IsEq M
    (range M) (range c) x' y)
    {i : ℕ} (hi : i < M) :
    (∀ i' < M, rowPay M (range c) y i' ≤ rowPay M (range c) y i) ↔ i <
    c := by
  obtain ⟨x₀, y₀, h₀⟩ := (cert_S (K := K) hc hcM).1
  obtain ⟨-, hy₀⟩ := pin_S hc hcM h₀
  set W : K := 2 / (c * (c + 1)) with hW_def
  have hW : 0 < W := by positivity
  -- against the equilibrium strategy of `S_c`, the rows of `[c]` pay
    `W` and the others `0`
  have hpay : ∀ z : ℕ → K, (∀ j < c, z j = ((c : K) - j) * W) → ∀ a < M,
      rowPay M (range c) z a = if a < c then W else 0 := fun z hz a ha =
    > by
    rw [rowPay_stair hcM]
    split_ifs with h1 h2
    · rw [hz a h1, hz (a + 1) h2]; push_cast; ring
    · rw [hz a h1, sub_zero, show (c : K) - a = 1 by
        have : (a : K) + 1 = c := by exact_mod_cast (show a + 1 = c by
    omega)
        linarith]
      ring
    · omega
    · simp
  -- by weak duality, `y` holds every row to `W` too, so it is an
    equilibrium strategy of `S_c`
  obtain ⟨j₀, hj₀, hv⟩ := exists_col_le h₀.2.1 h.1 (fun a ha => by
    rw [hpay y₀ hy₀ a (mem_range.1 ha)]; split_ifs <;> linarith)
    (nonempty_range_iff.2 hc.ne')
  have hS : IsEq M (range c) (range c) x₀ y := ⟨h₀.1, h.2.1, fun a ha j
    hj => by
    have h1 := (h.2.2 a (mem_range.2 ((mem_range.1 ha).trans_le hcM)) j₀
    hj₀).trans hv
    have h2 := h₀.2.2 0 (mem_range.2 hc) j hj
    rw [hpay y₀ hy₀ 0 (by omega), if_pos hc] at h2
    exact h1.trans h2⟩
  obtain ⟨-, hy⟩ := pin_S hc hcM hS
  have hr := hpay y hy
  constructor
  · intro hbr
    by_contra hic
    have := hbr 0 (by omega)
    rw [hr 0 (by omega), if_pos hc, hr i hi, if_neg hic] at this
    linarith
```

```
  · intro hic i' hi'
    rw [hr i' hi', hr i hi, if_pos hic]
    split_ifs <;> linarith

/-- In the full game, a corridor column is a best response against an
    equilibrium strategy
    exactly when it is one of the last two. -/
theorem ado_full_br (hM : 2 ≤ M) (hMN : M ≤ N) {x y' : ℕ → K}
    (h : IsEq M (range M) (range N) x y') {j : ℕ} (hjM : M ≤ j) (hjN : j
    < N) :
    (∀ j' < N, colPay M (range M) x j ≤ colPay M (range M) x j') ↔ N ≤
    j + 2 := by
  obtain ⟨d, rfl⟩ : ∃ d, N = M + d := ⟨N - M, by omega⟩
  obtain ⟨t, rfl⟩ : ∃ t, j = M + t := ⟨j - M, by omega⟩
  have hcol : ∀ t, colPay M (range M) x (M + t) = pay M (x 0) (x 1) (t +
    4) :=
    fun t => colPay_add hM x t
  rcases d with _ | _ | e
  · omega
  · -- `C_{M+1}`: column `M` is tight
    obtain ⟨hge, -, h2, -⟩ := (cert_handoff (K := K) hM).2 x y' h
    obtain rfl : t = 0 := by omega
    refine ⟨fun _ => by omega, fun _ j' hj' => ?_⟩
    rw [hcol, h2]
    exact hge j' hj'
  · -- `C_{M+e+2}`: the frontier columns are tight, and the lower ones
    pay more
    obtain ⟨hge, h4, h5, -⟩ := (cert_corridor (K := K) hM e).2 x y'
      (show IsEq M (range M) (range (M + e + 2)) x y' from h)
    have hV : (0 : K) < 1 / corrMass M e := one_div_pos.2 (corrMass_pos
    hM e)
    constructor
    · intro hbr
      by_contra hlt
      have h1 := hbr (M + e) (by omega)
      rw [hcol, hcol, h4] at h1
      linarith [pay_below (by omega) hV (by omega) h4 h5 (n := t + 4)
    (by omega) (by omega)]
    · intro hle j' hj'
      have hpay : colPay M (range M) x (M + t) = 1 / corrMass M e := by
        rcases (show t = e ∨ t = e + 1 by omega) with rfl | rfl
        · rw [hcol, h4]
        · rw [hcol, h5]
      rw [hpay]
      exact hge j' (by omega)

/-- ADO at `S_k`, `0 < k < M ≤ N`: the only novel best response is
    column `k`, and no row is
    added, so every successor is `T_k`, as in double oracle. -/
theorem ado_S (hk : 0 < k) (hkM : k < M) (hMN : M ≤ N) {q : Finset ℕ ×
    Finset ℕ}
    (h : StepADO K M N (range k, range k) q) : q = (range k, range (k +
    1)) := by
  obtain ⟨x, y, x', y', hx, hy, hR, hC⟩ := h
  obtain ⟨hk', hbr⟩ := ado_cols hk hkM hMN hx
  have e1 := hR.eq_self fun i hi hbr => mem_range.2 ((ado_rows hk hkM.le
    hy hi).1 hbr)
  have e2 := hC.eq (j := k) (by omega) hk' (by simp) fun j' hj' hbr'
    hj'C => by
    have := hbr j' hj' hbr'
    simp at hj'C
    omega
  rw [range_add_one]
  exact Prod.ext e1 e2
```

```
/-- ADO at `T_k`, `0 < k < M ≤ N`: the only novel best response is row
    `k`, and no column is
    added, so every successor is `S_{k+1}`, as in double oracle. -/
theorem ado_T (hk : 0 < k) (hkM : k < M) (hMN : M ≤ N) {q : Finset ℕ ×
    Finset ℕ}
    (h : StepADO K M N (range k, range (k + 1)) q) : q = (range (k + 1),
    range (k + 1)) := by
  obtain ⟨x, y, x', y', hx, hy, hR, hC⟩ := h
  obtain ⟨-, hbr⟩ := ado_cols hk hkM hMN hx
  have e1 := hR.eq (j := k) hkM ((ado_rows (by omega) (by omega) hy
    hkM).2 (by omega)) (by simp)
    fun i hi hbr' hiR => by
      have := (ado_rows (by omega) (by omega) hy hi).1 hbr'
      simp at hiR
      omega
  have e2 := hC.eq_self fun j hj hbr' => mem_range.2 (by have := hbr j
    hj hbr'; omega)
  rw [← range_add_one] at e1
  exact Prod.ext e1 e2

/-- The columns ADO ends with: the staircase and the last two corridor
    columns. -/
def adoCols (M N : ℕ) : Finset ℕ := insert (N - 2) (insert (N - 1)
    (range M))

theorem adoCols_card (hM : 2 ≤ M) (hMN : M ≤ N) :
    (adoCols M N).card = M + min (N - M) 2 := by
  simp only [adoCols]
  rcases (show N = M ∨ N = M + 1 ∨ M + 2 ≤ N by omega) with rfl | rfl |
    h
  · rw [insert_eq_of_mem (mem_insert_of_mem (mem_range.2 (by omega))),
      insert_eq_of_mem (mem_range.2 (by omega))]
    simp
  · rw [insert_eq_of_mem (mem_insert_of_mem (mem_range.2 (by omega))),
      card_insert_of_notMem (by simp), card_range]
    omega
  · rw [card_insert_of_notMem (by simp; omega), card_insert_of_notMem (by
    simp; omega),
      card_range]
    omega

/-- ADO once every row is present, with the columns between `[M]` and
    `adoCols M N`: the rows
    stay, and a missing one of the last two columns is added, until none
    is missing. -/
theorem ado_full (hM : 2 ≤ M) (hMN : M ≤ N) {C : Finset ℕ} (hC : range
    M ⊆ C)
    (hCF : C ⊆ adoCols M N) {q : Finset ℕ × Finset ℕ} (h : StepADO K M
    N (range M, C) q) :
    q.1 = range M ∧ C ⊆ q.2 ∧ q.2 ⊆ adoCols M N ∧ (C ≠ adoCols M N →
    q.2.card = C.card + 1) ∧
      (C = adoCols M N → q.2 = C) := by
  obtain ⟨x, y, x', y', hx, hy, hR, hC'⟩ := h
  have e1 := hR.eq_self fun i hi _ => mem_range.2 hi
  -- the best responses against `x` lie in `adoCols M N`, and every
    corridor column there is one
  have hbrF : ∀ j < N, (∀ j' < N, colPay M (range M) x j ≤ colPay M
    (range M) x j') →
      j ∈ adoCols M N := fun j hj hbr => by
    simp only [adoCols, mem_insert, mem_range]
    rcases lt_or_ge j M with hjM | hjM
    · exact Or.inr (Or.inr hjM)
    · have := (ado_full_br hM hMN hx hjM hj).1 hbr
      omega
  have hbrF' : ∀ j ∈ adoCols M N, M ≤ j → ∀ j' < N,
```

```
      colPay M (range M) x j ≤ colPay M (range M) x j' := fun j hj hjM =
    > by
    simp only [adoCols, mem_insert, mem_range] at hj
    exact (ado_full_br hM hMN hx hjM (by omega)).2 (by omega)
  refine ⟨e1, hC'.subset, ?_, fun hne => ?_, fun heq => hC'.eq_self fun
    j hj hbr => ?_⟩
  · rcases hC' with ⟨j, hj, hbr, -, hq⟩ | ⟨-, hq⟩
    · rw [hq]; exact insert_subset (hbrF j hj hbr) hCF
    · rw [hq]; exact hCF
  · -- some column of `adoCols M N` is missing, and it is a novel best
    response
    obtain ⟨j, hjF, hjC⟩ := not_subset.1 fun h' => hne (subset_antisymm
    hCF h')
    have hjM : M ≤ j := by
      by_contra h'
      exact hjC (hC (mem_range.2 (by omega)))
    have hjN : j < N := by simp only [adoCols, mem_insert, mem_range] at
    hjF; omega
    rcases hC' with ⟨j', -, -, hj'C, hq⟩ | ⟨hall, -⟩
    · rw [hq]; exact card_insert_of_notMem hj'C
    · exact absurd (hall j hjN (hbrF' j hjF hjM)) hjC
  · rw [heq]; exact hbrF j hj hbr

/-- **ADO on our matrix.** For `2 ≤ M ≤ N`, ADO runs from `({0}, {0})`,
    and every such run
    follows `run M N` until the rows run out at iteration `2 M - 2`,
    then adds the missing ones of
    the last two columns, one per iteration, and stops at `adoCols M N`.
    So it takes
    `min (M + N - 2) (2 M)` iterations, the worst case only while `N ≤ M
    + 2`, and never adds the
    other corridor columns. -/
theorem ado_iterations (hM : 2 ≤ M) (hMN : M ≤ N) :
    (∃ s : ℕ → Finset ℕ × Finset ℕ, s 0 = ({0}, {0}) ∧ ∀ t, StepADO K
    M N (s t) (s (t + 1))) ∧
      ∀ s : ℕ → Finset ℕ × Finset ℕ, s 0 = ({0}, {0}) → (∀ t, StepADO
    K M N (s t) (s (t + 1))) →
        (∀ t ≤ 2 * M - 2, s t = run M N t) ∧ (∀ t, (s t).2 ⊆ adoCols M
    N) ∧
        (∀ t < min (M + N - 2) (2 * M), s t ≠ s (t + 1)) ∧
        ∀ t, min (M + N - 2) (2 * M) ≤ t → s t = (range M, adoCols M N)
    := by
  refine ⟨exists_runADO (by omega) (by omega) (singleton_nonempty _)
    (singleton_nonempty _),
    fun s hs0 hs => ?_⟩
  -- the staircase phase
  have hstair : ∀ t ≤ 2 * M - 2, s t = run M N t := by
    intro t
    induction t with
    | zero => intro _; rw [hs0]; simp [run, show 2 < 2 * M by omega]
    | succ t ih =>
      intro ht
      have hst := hs t
      rw [ih (by omega)] at hst
      rcases Nat.even_or_odd' t with ⟨k, rfl | rfl⟩
      · rw [run_even (by omega)] at hst
        rw [ado_S (by omega) (by omega) hMN hst, ← run_odd (M := M) (N
    := N) (by omega)]
      · rw [run_odd (by omega)] at hst
        rw [ado_T (by omega) (by omega) hMN hst]
        by_cases h : 2 * k + 1 + 1 + 2 < 2 * M
        · rw [show 2 * k + 1 + 1 = 2 * (k + 1) by ring, run_even (M := M)
    (N := N) (by omega)]
        · rw [run, if_neg h, show 2 * k + 1 + 1 + 2 - M = M by omega,
    min_eq_left hMN,
```

```
            show k + 1 + 1 = M by omega]
  set t₀ := 2 * M - 2 with ht₀
  set F := adoCols M N
  have hF : range M ⊆ F := fun j hj => mem_insert_of_mem
    (mem_insert_of_mem hj)
  have hs₀ : s t₀ = (range M, range M) := by
    rw [hstair t₀ le_rfl, run, if_neg (by omega), show t₀ + 2 - M = M by
    omega, min_eq_left hMN]
  -- the corridor phase: one of the last two columns per iteration,
    until both are present
  have hcorr : ∀ u, (s (t₀ + u)).1 = range M ∧ range M ⊆ (s (t₀ + u)).2 ∧
      (s (t₀ + u)).2 ⊆ F ∧ (s (t₀ + u)).2.card = M + min u (min (N - M)
    2) := by
    intro u
    induction u with
    | zero => rw [add_zero, hs₀]; exact ⟨rfl, subset_refl _, hF, by simp⟩
    | succ u ih =>
      rw [← add_assoc]
      obtain ⟨h1, h2, h3, h4⟩ := ih
      have hst := hs (t₀ + u)
      rw [show s (t₀ + u) = (range M, (s (t₀ + u)).2) from Prod.ext h1
    rfl] at hst
      obtain ⟨e1, e2, e3, e4, e5⟩ := ado_full hM hMN h2 h3 hst
      refine ⟨e1, h2.trans e2, e3, ?_⟩
      by_cases hne : (s (t₀ + u)).2 = F
      · rw [e5 hne, h4]
        rw [hne, adoCols_card hM hMN] at h4
        omega
      · rw [e4 hne, h4]
        have hlt : (s (t₀ + u)).2.card < F.card := card_lt_card
    (ssubset_of_subset_of_ne h3 hne)
        rw [adoCols_card hM hMN, h4] at hlt
        omega
  refine ⟨hstair, fun t => ?_, fun t ht heq => ?_, fun t ht => ?_⟩
  · rcases le_or_gt t t₀ with h | h
    · rw [hstair t h]
      refine subset_trans ?_ hF
      simp only [run]
      split_ifs <;> exact range_subset_range.2 (by omega)
    · obtain ⟨u, rfl⟩ : ∃ u, t = t₀ + u := ⟨t - t₀, by omega⟩
      exact (hcorr u).2.2.1
  · rcases lt_or_ge t t₀ with h | h
    · have c1 := run_card (M := M) (N := N) hM hMN t
      have c2 := run_card (M := M) (N := N) hM hMN (t + 1)
      rw [← hstair t h.le, heq] at c1
      rw [← hstair (t + 1) (by omega)] at c2
      omega
    · obtain ⟨u, rfl⟩ : ∃ u, t = t₀ + u := ⟨t - t₀, by omega⟩
      have c1 := (hcorr u).2.2.2
      have c2 := (hcorr (u + 1)).2.2.2
      rw [← add_assoc, ← heq] at c2
      omega
  · obtain ⟨u, rfl⟩ : ∃ u, t = t₀ + u := ⟨t - t₀, by omega⟩
    obtain ⟨h1, -, h3, h4⟩ := hcorr u
    refine Prod.ext h1 (eq_of_subset_of_card_le h3 ?_)
    rw [h4, adoCols_card hM hMN]
    omega

#print axioms double_oracle_worst_case
#print axioms double_oracle_approx
#print axioms double_oracle_accuracy
#print axioms double_oracle_accuracy_sharp
#print axioms margin_ge
#print axioms gap_shape
#print axioms tilt_worst_case
```

```
#print axioms tilt_gap
#print axioms eq_unique
#print axioms gap_run
#print axioms gap_run_le
#print axioms exists_isEqB
#print axioms any_start
#print axioms any_start_two
#print axioms run_sizes
#print axioms double_oracle_perturbed
#print axioms gap_shape_run
#print axioms nonzero_card
#print axioms double_oracle_cols_inf
#print axioms double_oracle_rows_inf
#print axioms double_oracle_inf
#print axioms double_oracle_swap
#print axioms double_oracle_approx_swap
#print axioms double_oracle_perturbed_swap
#print axioms gap_run_le_swap
#print axioms det_corr
#print axioms geom_corr
#print axioms ratio_golden
#print axioms goldenRatio_dens_le
#print axioms ado_iterations
```