Faster Shellsort with Reinforcement Learning

Qingyun Li[1] **Yunjin Li**[2] **Yunqi Li**[1] **Ruoyun Li**[1]
[1]University of Washington [2]North Seattle College
qxli2@cs.washington.edu

Abstract

Shellsort sorts an array through a sequence of insertion-sort passes, where each pass operates on elements separated by a chosen gap and the gaps decrease toward 1. The choice of gaps largely determines its cost, yet classical sequences such as Ciura, Sedgewick, and Tokuda are fixed regardless of the input structure. We investigate whether the gaps should instead adapt to where disorder lies in the array. Because each gap acts at a different length scale, we represent the distribution of disorder using a *disorder spectrum* (the inversion rate measured at logarithmic strides). We then formulate gap selection as a Markov decision process in which a small policy observes the current disorder spectrum and selects the next gap, repeating this process on the updated array after each pass. Notably, on a mixture of seven diverse input families, this adaptive policy reduces comparison counts by 13–16% relative to Sedgewick across array sizes from N=128 through N=16,384. Furthermore, we find that the resulting schedules closely mirror the input's disorder profile. Workloads with disorder concentrated at small scales lead to short, fine-grained sequences, and those with broad-scale disorder produce larger, denser gaps. Conditioning on the initial disorder spectrum is already enough to capture the bulk of the improvement, but sequential adaptation is necessary to achieve the final margin. For stable target distributions, the policy can also be used offline to discover a *fixed schedule specialized to the workload*, which can then be hardcoded to eliminate neural inference and spectrum computation at runtime.[1]

1 Introduction

Shellsort [Shell, 1959] is an extension of insertion sort designed to reduce the number of comparisons required to sort an array. The primary inefficiency of ordinary insertion sort is that elements can only move one position at a time; if a small element is located at the end of a large array, it must be shifted across nearly every other element to reach its correct position. Shellsort overcomes this by performing a sequence of passes using a list of gaps $[h_1, h_2, \ldots, h_k]$, where $h_1 > h_2 > \cdots > h_k = 1$.

In a pass with gap h, the algorithm performs insertion sort on elements separated by h positions. This allows elements to move over long distances early in the sort, which can quickly resolve coarse-scale disorder. As the gaps decrease, the algorithm progressively refines the array until the final pass ($h = 1$), which is just an ordinary insertion sort. Because the array is already nearly sorted by this point, the final pass is highly efficient. The total cost of the algorithm is largely determined by the choice of this gap sequence, yet finding optimal sequences has remained an open problem for over six decades.

Despite the availability of asymptotically faster sorting algorithms, Shellsort remains useful in memory-constrained environments because it is simple, in-place, and requires little auxiliary state: for example, uClibc-ng implements `qsort()` as a Shellsort for embedded Linux,[2] the U-Boot bootloader

[1]Code is available at `https://github.com/cminst/neuroshell`.
[2]`https://github.com/wbx-github/uclibc-ng`, `libc/stdlib/stdlib.c`

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

uses it on billions of ARM devices,[3] and bzip2 relies on it in its Burrows-Wheeler fallback path.[4] In these settings, even a moderate improvement in the gap sequence translates directly to reduced comparisons with zero implementation complexity.

Why adapt the gap sequence? The best-known classical sequences are Ciura's empirically optimized sequence [Ciura, 2001], Sedgewick's theoretically motivated construction [Sedgewick, 1996], and Tokuda's geometric formula [Tokuda, 1992]. All three are fixed, so when applied, every array receives the same sequence of gaps regardless of its structure. However, the disorder in an array can look very different from one input to another. A nearly sorted array may need only local cleanup, while reversed, periodic, or block-structured arrays can contain disorder over much larger distances. Applying the same sequence to all of them can therefore spend passes at scales where little useful work remains to be done.

This suggests a natural way to think about gap selection. Because each gap acts at a different length scale, we summarize where disorder lies across those same scales using a *disorder spectrum*. At each of several logarithmically spaced strides, we measure how often pairs of elements at that separation are out of order (i.e., "inverted"), and together, these inversion rates give the policy a compact fingerprint of how disorder is distributed across a given array. A gap sequence then becomes a schedule over these length scales, and rather than choosing one schedule in advance for every array, we aim to produce a schedule matched to the disorder that is actually present.

Our approach. We formulate gap selection as a Markov decision process (MDP). A small policy observes the current disorder spectrum, selects the next gap, applies the corresponding Shellsort pass, and repeats on the updated array. The spectrum is recomputed after each pass, so the policy can respond not just to how the array looked initially, but also to how its structure changes as sorting proceeds. For the policy, we use a small MLP trained with PPO [Schulman et al., 2017] (PPO is simply the optimizer we choose for this sequential formulation rather than a specialized component of the method).

On a mixture of seven diverse input families, the learned policy substantially reduces comparison counts relative to the strongest classical gap sequences we test. At $N = 256$, it outperforms all seven classical baselines on every input family. Across array sizes from $N = 128$ through $N = 16{,}384$, it remains ahead of both Knuth and Sedgewick, using **13–16% fewer comparisons** than Sedgewick. The gains are smallest on random and many-duplicate inputs, where strong classical sequences already perform well, and much larger on structured families such as nearly-sorted, sawtooth, and interleaved arrays.

A better global sequence accounts for only part of this improvement. When the spectrum features are removed entirely, the learned schedule still improves on Sedgewick by about 4–6%. Supplying the disorder spectrum of the initial array raises the margin to roughly 13–14% across the full range of sizes. Recomputing the spectrum after each pass improves the result again, with the additional gain growing from 0.6 percentage points at $N = 128$ to 2.5 points at $N = 16{,}384$. We also find that the spectrum does not need to be measured densely: four inversion rates spread across the array's length scales recover most of the performance of the full power-of-two spectrum. Together, these experiments show that most of the useful variation in the gap schedule is already present in the array's initial multi-scale structure, with later passes providing additional information as the sort progresses.

Deployment and scope. Running the policy inside every sort gives the best comparison-count results, but the spectrum computation and neural inference add substantial runtime overhead. For workloads with a stable input distribution, we instead use the policy offline to discover a fixed sequence specialized to that workload. We collect complete schedules produced by the trained policy, select among them on held-out data, and then hard-code the resulting sequence in place of the classical gaps. Across $N = 128$ through $N = 16{,}384$, these compiled schedules retain **90.5–95.6%** of the live policy's improvement over Sedgewick. The deployed algorithm is then an ordinary Shellsort with a workload-specific gap sequence, with no disorder-spectrum computation or neural inference at runtime. This setting assumes recurring input structure that can be represented during training; we do not assume that a policy trained on one collection of structures will automatically generalize to arbitrary unseen ones.

[3] `https://github.com/u-boot/u-boot`, `lib/qsort.c`

[4] `https://sourceware.org/bzip2/`, `blocksort.c`

Our contributions are:

- We formulate Shellsort gap selection as a sequential decision problem over the input's disorder spectrum. Because gaps and inversion rates are both indexed by length scale, the formulation gives a direct way to learn spectrum-matched gap schedules rather than applying the same fixed sequence to every array.
- We show that the learned policy outperforms strong classical gap sequences across seven input families and array sizes from $N = 128$ through $N = 16{,}384$. Controlled observation experiments show that conditioning on the initial disorder spectrum accounts for most of the adaptive gain, recomputing the spectrum after each pass improves further, and only a small number of scales are needed to recover most of the full-spectrum performance.
- We show that the policy can also be used offline to discover workload-specific fixed gap sequences. These compiled schedules retain 90.5–95.6% of the live policy's improvement across the tested sizes, while reducing deployment back to an ordinary static Shellsort with no neural component at runtime.

2 Related Work

Shellsort gap sequences. Much of the Shellsort literature has focused on designing better fixed gap sequences. Shell's original sequence $\lfloor N/2^k \rfloor$ [Shell, 1959] gives $O(N^2)$ worst-case complexity, and subsequent work produced sequences with progressively better theoretical guarantees: Hibbard's $2^k - 1$ [Hibbard, 1963], Incerpi and Sedgewick's improved upper bounds [Incerpi and Sedgewick, 1985], Sedgewick's interleaved formula [Sedgewick, 1986, 1996] achieving $O(N^{4/3})$, and Pratt's $2^p 3^q$ sequence [Pratt, 1972] achieving $O(N \log^2 N)$ but requiring too many passes to be practical. Jiang et al. [2000] further proved an $\Omega(pN^{1+1/p})$ average-case lower bound for any p-pass Shellsort, regardless of the increment sequence.

Other sequences have instead been motivated by empirical performance. Tokuda [1992] proposed a geometric formula, while Ciura [2001] searched directly for a high-performing sequence and produced one that remains the strongest classical baseline on random permutations in our experiments. Despite their different origins, all of these sequences make the same choice before seeing the input: once N is fixed, every array receives the same gaps. Knuth [1973] gives a broader treatment of Shellsort and the classical theory behind diminishing-increment sequences.

Machine learning for algorithm design. AlphaDev [Mankowitz et al., 2023] used deep RL to discover faster sorting networks at the assembly level, requiring massive computational resources. Fawzi et al. [2022] applied RL to discover faster matrix multiplication algorithms. Learning-augmented algorithms [Mitzenmacher and Vassilvitskii, 2022, Lykouris and Vassilvitskii, 2021, Dinitz et al., 2021] use ML predictions to improve worst-case algorithms, typically for online problems. Kraska et al. [2018] demonstrated learned index structures that outperform B-trees. Neural combinatorial optimization [Bello et al., 2017, Vinyals et al., 2015, Kool et al., 2019, Khalil et al., 2017] applies RL and attention mechanisms to routing, scheduling, and graph problems. Bengio et al. [2021] survey the broader landscape of ML for combinatorial optimization. Our work is closest in spirit to AlphaDev and Fawzi et al. but differs in two key ways: (1) we operate at a higher abstraction level (gap selection rather than assembly instructions), which makes the MDP compact enough that training completes in minutes on a single CPU rather than requiring days on distributed TPU infrastructure; and (2) our policy is *input-adaptive*, selecting different strategies per input, whereas both AlphaDev and Fawzi et al. discover a single fixed algorithm.

Adaptive sorting. Adaptive sorting algorithms exploit existing order in the input [Estivill-Castro and Wood, 1992]. Timsort [Peters, 2002] detects sorted runs; introsort [Musser, 1997] switches between quicksort and heapsort based on recursion depth. Our approach is complementary in that (1) we adapt the gap sequence rather than the structure of the sorting algorithm itself, and (2) our adaptation is learned rather than hand-designed.

3 MDP Formulation

Choosing a Shellsort gap is inherently sequential. Each pass changes the array, which in turn changes which gaps may be useful next. We therefore formulate gap selection as an episodic MDP $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$.

State. At each step, the policy needs to know where disorder remains in the array, how much of the pass budget remains, and the previous gap in the decreasing schedule. We represent the state as

$$s_t = [\phi(a_t), p_t, g_t] \in \mathbb{R}^{K+2},$$

where $\phi(a_t) \in \mathbb{R}^K$ is the disorder spectrum. Here, $K = \lceil \log_2 N \rceil$ is the number of logarithmically spaced strides; for each stride $s \in \{1, 2, 4, \ldots, N/2\}$, the disorder spectrum measures the fraction of pairs (a_i, a_{i+s}) that are inverted. In this way, small strides capture local disorder, large strides capture coarse structure, and the intermediate values describe how disorder is distributed between the two.

The remaining two features describe the progress of the schedule:

$$p_t = 1 - t/T_{\max}, \qquad g_t = h_{t-1}/N,$$

where p_t is the fraction of the pass budget remaining and g_t is the previous gap normalized by the array length. The resulting state's size grows only logarithmically with N.

Action space. An action selects the next Shellsort gap. Rather than allowing every integer from 1 through $N/2$, we use a smaller candidate set containing all gaps from 1 through 16, a geometric sequence with ratio 1.15 from 16 to $N/2$, and powers of 2 and 3. In our experiments, this gives 27–74 candidate gaps depending on the value of N, compared with $N/2$ possible gaps in the full action space. We mask out every gap greater than or equal to the previous one, in order to enforce a strictly decreasing schedule.

We use this restricted set simply for computational efficiency; as Appendix D shows, repeating the experiment with all $N/2$ possible gaps at the eight tested sizes gives essentially the same policy performance. This suggests that the candidate-set restriction is not responsible for the improvement.

Transition. Choosing gap h_t executes one Shellsort pass at that gap and updates the array in place. The policy then observes the new disorder spectrum before choosing again. An episode ends when the gap reaches 1, the array is sorted, or the maximum pass budget $T_{\max}$ is exhausted.

Reward. The policy is trained to minimize comparisons while avoiding unnecessarily long schedules. If C_t is the number of comparisons made during pass t, the per-step reward is

$$r_t = -\frac{C_t}{N \log_2 N} - \lambda, \tag{1}$$

with $\lambda = 0.005$. The first term directly penalizes comparison cost, while the small step penalty discourages extra passes. If the pass budget is exhausted before the array is sorted, we apply an additional terminal penalty of -1.0. Note that we normalize by a factor of $N \log_2 N$ here to keep the comparison term on a similar scale across array sizes.

Policy and training. The policy π_θ is a two-layer MLP with 128 hidden units per layer, LayerNorm, and ReLU activations, with separate actor and critic heads. We train it with PPO [Schulman et al., 2017] and GAE [Schulman et al., 2016].

Training arrays are sampled uniformly from seven input families: random permutations, nearly-sorted arrays, reversed arrays with local noise, interleaved blocks, pipe-organ arrays, sawtooth arrays, and arrays with many duplicate values. These families span several qualitatively different kinds of structure, from largely unstructured randomness to local disorder, global reversal, periodic and block structure, and repeated values. We include their exact construction and sampling parameters in Appendix B.2; the full optimization configuration can be found in Appendix B.1.

4 Experiments

4.1 Setup

We compare the learned policy against seven classical baseline gap sequences: Shell's original [1959], Hibbard [1963], Pratt [1972], Knuth [1973], Tokuda [1992], Sedgewick [1996], and Ciura [2001]. We also use Sedgewick as the common reference for relative improvements throughout the paper, as it is one of the strongest baselines and consistently strong across the range of array sizes we test. All experiments measure total comparison count (the standard theoretical metric for sorting algorithms). We report mean over 1000 evaluation episodes/arrays per distribution family unless otherwise noted.

4.2 Main Results

We first examine the policy in detail at $N = 256$ (our main setting). Table 1 shows the comparison counts for each of the seven input families as well as the policy's relative reduction in comparisons against the three strongest baseline gap sequences (Knuth, Sedgewick, and Ciura). Averaged across families, the learned policy uses **14.0% fewer comparisons** than Sedgewick and **17.7% fewer** than Ciura. It also beats Sedgewick on every individual family, with the differences statistically significant across the 30 independently trained policies.

Table 1: Per-family results at array size $N = 256$ (mean $\pm$ std over 30 training seeds). Lower is better. RD = random, NS = nearly sorted, RV = reversed, IL = interleaved, PO = pipe organ, ST = sawtooth, and MD = many duplicates. The best per-column value is bolded, and the second-best is underlined.

	Array family							
	RD	NS	RV	IL	PO	ST	MD	**Average**
Shell	3,746	2,232	2,566	6,554	2,177	2,916	3,225	3,345
Hibbard	2,648	1,958	2,174	2,590	2,012	2,183	2,221	2,255
Pratt	5,717	5,367	5,337	5,605	5,553	5,538	5,377	5,499
Knuth	2,498	<u>1,575</u>	1,884	2,121	<u>1,719</u>	<u>1,889</u>	2,041	<u>1,961</u>
Tokuda	2,475	1,802	2,076	2,220	1,965	2,063	2,052	2,093
Sedgewick	2,524	1,644	<u>1,751</u>	<u>2,103</u>	1,785	1,900	2,033	1,963
Ciura	<u>2,449</u>	1,739	1,965	2,258	1,827	2,102	<u>2,019</u>	2,051
Policy (ours)	**2,446** ± 7	**1,086** ± 0	**1,567** ± 35	**1,756** ± 46	**1,618** ± 12	**1,355** ± 19	**1,990** ± 8	**1,688** ± 11
vs. Knuth	-2.1%	-31.0%	-16.8%	-17.2%	-5.8%	-28.3%	-2.5%	-13.9%
vs. Sedgewick	-3.1%	-33.9%	-10.5%	-16.5%	-9.3%	-28.7%	-2.1%	-14.0%
vs. Ciura	-0.1%	-37.5%	-20.3%	-22.2%	-11.4%	-35.5%	-1.4%	-17.7%

We find that the size of the gain depends strongly on the structure of the input. On random permutations, the policy is essentially tied with Ciura, which was itself empirically optimized for random input, and many-duplicate arrays similarly leave little room over the classical schedules. The picture changes on more structured families. Nearly-sorted and sawtooth arrays improve by 33.9% and 28.7% over Sedgewick, respectively, while interleaved, reversed, and pipe-organ inputs also see substantial reductions. In other words, the classical sequences remain difficult to improve on where they already fit the input well, while structured inputs leave considerably more room for the policy to adapt its schedule.

4.3 Scaling Results

Table 2 shows that the policy's advantage persists as the arrays grow. It outperforms both Knuth and Sedgewick at every size from $N = 128$ through $N = 16{,}384$. Knuth has a small edge over Sedgewick through $N = 2{,}048$, but Sedgewick becomes the stronger of the two at the larger sizes. Relative to our common Sedgewick reference, the policy uses **13.5–15.9% fewer comparisons** across the full range, with the margin generally increasing as N grows. Variance across the 30 independently trained policies remains small throughout, and a 500-update training run takes just 7.4 minutes even at $N = 16{,}384$ on our CPU.

The larger action space also does not make the policy substantially harder to optimize. As the candidate set grows from 27 gaps at $N = 128$ to 74 at $N = 16{,}384$, the policies still reach near-final

performance within roughly 150–250 updates (Figure 1a), so the uniform 500-update budget is conservative across all eight sizes. The family-level results show a similarly stable pattern. The magnitude of the gain depends strongly on input structure, but the same families tend to remain easy or difficult to improve on as N grows (Figure 1b).

Table 2: Scaling across array sizes against Knuth and Sedgewick, the two strongest classical baselines on average in our experiments. Entries report mean comparison counts over the seven input families; policy results are mean ± std over 30 training seeds.

N	Sedgewick	Knuth	**Policy (ours)**	vs. Sedge.	vs. Knuth	Train time
128	808	805	**699** ± 2	-13.5%	-13.1%	66s
256	1,963	1,961	**1,688** ± 11	-14.0%	-13.9%	71s
512	4,671	4,631	**4,005** ± 18	-14.3%	-13.5%	76s
1,024	10,864	10,699	**9,327** ± 29	-14.2%	-12.8%	91s
2,048	24,911	24,673	**21,108** ± 56	-15.3%	-14.4%	108s
4,096	56,070	56,130	**47,511** ± 166	-15.3%	-15.4%	174s
8,192	125,924	126,462	**105,944** ± 246	-15.9%	-16.2%	263s
16,384	276,652	293,049	**232,623** $\pm 1{,}136$	-15.9%	-20.6%	444s

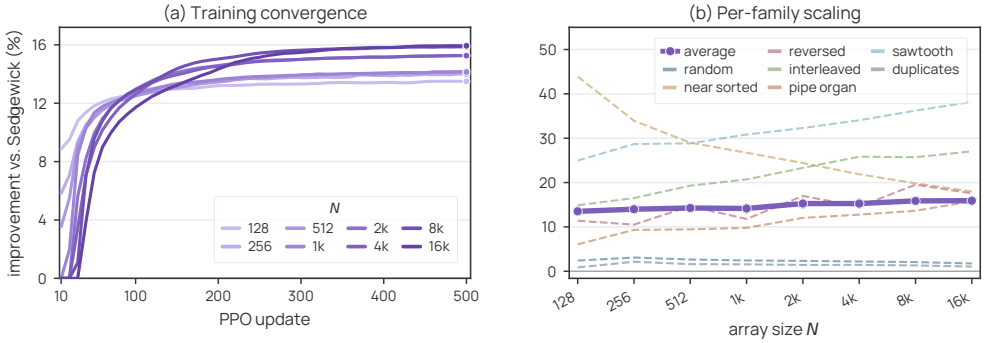

Figure 1: Scaling from $N = 128$ to $N = 16{,}384$. **(a)** Training curves for the policies trained at each size. Despite the candidate action space growing from 27 to 74 gaps, all eight policies converge within roughly 150–250 PPO updates. **(b)** The advantage over Sedgewick remains structured by input family as N grows. Families that benefit strongly from adaptation continue to do so at larger sizes, while random and many-duplicate inputs remain much closer to the classical baseline.

4.4 Input Conditioning and Sequential Feedback

The full policy recomputes the disorder spectrum after every Shellsort pass. We compare it with two simpler variants that remove parts of this information:

1. **No observation**. Here, we zero out the spectrum features $\phi(a_t)$ and keep only p_t (the remaining pass budget) and g_t (the previous gap) as inputs to the policy. This produces the same learned schedule for every input.
2. **Initial spectrum**. We measure the disorder spectrum on the original array *before sorting begins*, and then supply those initial values to the policy at every step, so the resulting schedule is conditioned on how the array started, but not on how each successive pass changed it.

The full policy recomputes the updated spectrum after each pass so that each gap can be conditioned on the array in its current state. The action space, reward, network architecture, and PPO training are otherwise identical across all three settings.

Table 3 shows that most of the improvement appears as soon as the initial spectrum is added. The policy with no array information already improves on Sedgewick by about 4–6%, but using the initial spectrum brings the margin to roughly 13–14% at every size we test. Updating the spectrum after

Table 3: Effect of input conditioning and sequential feedback across array sizes. Entries report mean comparisons across 30 training seeds, with improvement relative to Sedgewick in parentheses. Lower is better.

	Mean comparisons ↓			
N	Sedgewick	No observation	+ Initial spectrum	+ Updated spectrum
128	808	777 (-3.9%)	704 (-12.9%)	**699** (-13.5%)
256	1,963	1,884 (-4.0%)	1,709 (-12.9%)	**1,692** (-13.8%)
512	4,671	4,437 (-5.0%)	4,083 (-12.6%)	**4,012** (-14.1%)
1,024	10,864	10,392 (-4.3%)	9,446 (-13.1%)	**9,326** (-14.2%)
2,048	24,911	23,623 (-5.2%)	21,508 (-13.7%)	**21,108** (-15.3%)
4,096	56,070	52,812 (-5.8%)	48,541 (-13.4%)	**47,511** (-15.3%)
8,192	125,924	117,816 (-6.4%)	108,603 (-13.8%)	**105,928** (-15.9%)
16,384	276,652	260,756 (-5.7%)	239,542 (-13.4%)	**232,529** (-15.9%)

each pass (i.e., the full policy) improves the result again. The difference is small at the shortest arrays and grows with N, from 0.6 percentage points at $N = 128$ to 2.5 points at $N = 16{,}384$. Overall, the starting structure of the array determines most of the useful variation in the schedule, but later passes still reveal information that is worth using, especially on the larger arrays.

4.5 Resolution of the Disorder Spectrum

The full disorder spectrum $\phi(a_t)$ measures inversions at every power-of-two stride from 1 through $N/2$. We next vary how many of those scales are included in the state. When $m = 0$, all spectrum features are removed and this is equivalent to the "No observation" variant from Subsection 4.4. With $m = 1$, we use just the stride-1 inversion rate. For $m \geq 2$, we keep strides 1 and $N/2$ and place the remaining measurements approximately evenly between them on the $\log_2$ scale. The pass-budget and previous-gap features, along with the rest of the policy and training setup, are kept fixed.

Table 4: Effect of varying the resolution of the disorder spectrum. Using just $m = 4$ scales is enough to recover most of the benefit of the full disorder spectrum. "Full" uses the complete power-of-two spectrum from stride 1 through $N/2$. More negative is better.

	Improvement vs. Sedgewick				
m	$N = 256$	$N = 1{,}024$	$N = 4{,}096$	$N = 8{,}192$	$N = 16{,}384$
0	-4.1%	-4.4%	-5.6%	-6.3%	-5.7%
1	-8.9%	-9.3%	-9.4%	-10.4%	-9.2%
2	-11.4%	-11.6%	-12.1%	-12.4%	-11.1%
3	-12.9%	-12.9%	-13.5%	-14.0%	-13.8%
4	-13.7%	-13.7%	-14.7%	-15.3%	-15.1%
5	-13.7%	-13.7%	-14.9%	-15.4%	-15.3%
Full	$\mathbf{-14.0\%}$	$\mathbf{-14.3\%}$	$\mathbf{-15.4\%}$	$\mathbf{-15.9\%}$	$\mathbf{-16.0\%}$

The largest improvements come from the first few spectrum features. A single local inversion rate already performs substantially better than removing the spectrum entirely, and adding measurements at larger scales continues to improve the schedule. By four scales, however, most of the gap to the full spectrum has disappeared, so a small number of inversion rates spread across the array's length scales captures most of the useful information.

5 Compiling Learned Schedules

5.1 Compilation Procedure

For all of the above experiments, we run the policy "live"; i.e., inside every sort, we recompute the disorder spectrum and let the policy select a new gap after each pass. Although this gives the policy the most flexibility, it also carries the cost of spectrum computation and neural inference. For a workload with a stable input distribution, we can instead do this work once offline. We use the trained

policy to discover gap sequences that work well for that distribution, select one of them on held-out data, and then hard-code the resulting sequence in place of a classical sequence such as Sedgewick or Knuth. The deployed algorithm is then just an ordinary Shellsort with a workload-specific fixed gap sequence, and the policy itself is no longer needed.

In our experiments, we compile a separate schedule for each input family. Starting from a single trained policy, we first run it greedily on a set of compile-only arrays from that family and record every distinct gap sequence it produces. We then run each of these candidate gap sequences as an ordinary Shellsort sequence on a separate validation set and keep the one with the lowest mean comparison count.

Once this fixed schedule has been selected, we evaluate it by measuring its comparison count on a third, untouched test set from the same family, using the same set of arrays to evaluate the original policy and the classical baselines.

We repeat this entire procedure independently for each of the 30 trained policies to study what the policies converge to across many runs. Because each policy produces its own compiled schedule, the results in the next section report the mean and standard deviation across these 30 independently compiled schedules.

5.2 Compilation Results

We next examine how much of the policy's improvement remains after compiling a fixed schedule for each input family. Table 5 compares these compiled schedules with the original live policy.

Table 5: Comparison-count performance after compiling the learned policy into a fixed gap sequence. "Gain retained" is the fraction of the live policy's improvement over Sedgewick preserved by the compiled schedule.

N	Sedgewick	Policy (live)	Compiled schedule	Gain retained
128	808	699 ± 2 (-13.5%)	708 ± 4 (-12.3%)	91.3%
256	1,963	1,688 ± 11 (-14.0%)	1,702 ± 19 (-13.3%)	95.0%
512	4,671	4,005 ± 18 (-14.3%)	4,051 ± 31 (-13.3%)	93.1%
1,024	10,864	9,327 ± 29 (-14.2%)	9,426 ± 67 (-13.2%)	93.5%
2,048	24,911	21,108 ± 56 (-15.3%)	21,421 ± 123 (-14.0%)	91.8%
4,096	56,070	47,511 ± 166 (-15.3%)	48,322 ± 187 (-13.8%)	90.5%
8,192	125,924	105,944 ± 246 (-15.9%)	106,814 ± 547 (-15.2%)	95.6%
16,384	276,652	232,623 $\pm 1{,}136$ (-15.9%)	235,040 $\pm 1{,}215$ (-15.0%)	94.5%

As Table 5 shows, the compiled schedules retain the vast majority of the live policy's performance advantage over Sedgewick. Across all tested array sizes, they preserve between 90.5% and 95.6% of the live policy's gain. The performance difference between running the full policy and using the compiled schedule is consistently small, accounting for only a 0.7 to 1.5 percentage point drop in the improvement over Sedgewick. Detailed results for each family, along with the compiled gap sequences, can be found in Appendix C.

This demonstrates that for these stable input families, the benefit discovered by the adaptive policy can be captured in a simple, fixed gap sequence. While the live policy still performs best, a compiled, workload-specific sequence delivers most of the same advantages. The key practical takeaway is that the policy's runtime overhead (the disorder-spectrum computation and neural inference) can be eliminated from the final deployed sort with only a minor loss in performance.

5.3 Compiled Schedules and Disorder Spectra

Compilation produces a fixed gap sequence for each input family, which allows us to directly examine the relationship between the structure of the workload and the fixed schedule discovered for it. We compile per-family gap sequences for each of the 30 trained policies, and then for each family, we take the most commonly selected sequence produced by the 30 policies. Figure 2 shows the initial disorder spectrum for each family, as well as its compiled gap sequence at $N = 256$ overlaid on top.

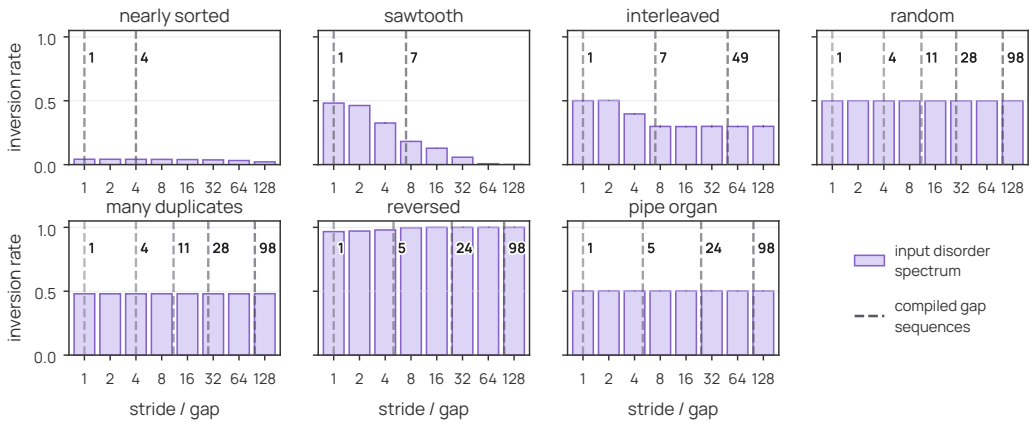

Figure 2: Initial disorder spectra and compiled gap sequences at $N = 256$. Purple bars show the mean inversion rate at each stride, averaged over $5{,}000$ arrays from each family. Gray dashed lines mark the gaps in the compiled schedule. Families with disorder concentrated at small scales compile to short, fine-grained schedules, whereas families with substantial coarse-scale disorder retain much larger gaps.

The resulting schedules closely track the distribution of disorder in each family. For workloads where disorder is concentrated at small scales, such as nearly-sorted and sawtooth inputs, the policy compiles to short, fine-grained schedules like $[4, 1]$ and $[7, 1]$. In contrast, reversed and pipe-organ arrays exhibit substantial disorder across much larger distances, and their compiled schedules begin with correspondingly larger gaps. Random and many-duplicate arrays have flatter spectra and consequently compile to denser schedules that span several scales, while interleaved inputs fall somewhere between these extremes.

These results provide encouraging evidence that the policy has learned to map disorder spectra to appropriate gap sequences.

6 Conclusion

We have shown that formulating Shellsort gap selection as a sequential decision problem conditioned on the disorder spectrum leads to significant performance improvements. Our adaptive policy reduces comparison counts by 13–16% relative to Sedgewick across a diverse set of input families and array sizes. Although classical sequences are highly optimized for random permutations, these results demonstrate that substantial headroom remains on structured inputs that only an adaptive approach can capture.

The effectiveness of the method stems from the policy's ability to map the disorder spectrum to an appropriate gap sequence. This relationship is supported by our ablation studies: most of the improvement is captured by the initial disorder spectrum, with additional gains provided by updating the spectrum after each pass. Furthermore, the fact that only a few logarithmic scales are required to recover most of the benefit suggests that the policy relies on a compact, multi-scale representation of disorder. This mapping is further validated by the qualitative alignment between the disorder spectra and the resulting compiled schedules.

Although the full adaptive policy carries a runtime cost, it serves as a powerful tool for the discovery of optimized schedules. For any stable workload, the policy can be used offline to extract a fixed, workload-specific sequence. As shown in our compilation results, these sequences retain 90–96% of the live policy's advantage while removing all neural inference and spectrum computation from the deployment path.

Our work suggests that machine learning can be effectively used to optimize the hyperparameters of classical algorithms for specialized workloads. By automating the discovery of specialized configurations, we can achieve performance gains that often elude one-size-fits-all theoretical constructions without compromising the simplicity of the deployed algorithm.

Broader impacts. Our approach could improve sorting efficiency in resource-constrained settings, including embedded and edge devices where Shellsort is already used. Because our learned gap sequences can be compiled into ordinary Shellsort, the improvements can be incorporated without the runtime overhead of a neural policy. The primary limitation is that a sequence optimized for one workload may perform poorly if the input distribution changes. We do not identify significant direct societal risks associated with this work.

References

Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2017.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

Marcin Ciura. Best increments for the average case of shellsort. In *Proceedings of the 13th International Symposium on Fundamentals of Computation Theory*, pages 106–117, 2001.

Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Faster matchings via learned duals. In *Advances in Neural Information Processing Systems*, pages 10393–10406, 2021.

Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.

Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610:47–53, 2022.

Thomas N. Hibbard. An empirical study of minimal storage sorting. *Communications of the ACM*, 6 (5):206–213, 1963.

Janet Incerpi and Robert Sedgewick. Improved upper bounds on shellsort. *Journal of Computer and System Sciences*, 31(2):210–224, 1985.

Tao Jiang, Ming Li, and Paul Vitányi. A lower bound on the average-case complexity of shellsort. *Journal of the ACM*, 47(5):905–911, 2000.

Elias B. Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. *Journal of the ACM*, 68(4):1–25, 2021.

Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618:257–263, 2023.

Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Communications of the ACM*, 65(7):33–35, 2022.

David R. Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

Tim Peters. Timsort, 2002. `https://bugs.python.org/file4451/timsort.txt`.

Vaughan R. Pratt. *Shellsort and sorting networks*. PhD thesis, Stanford University, 1972.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Robert Sedgewick. A new upper bound for shellsort. *Journal of Algorithms*, 7(2):159–173, 1986.

Robert Sedgewick. Analysis of shellsort and related algorithms. In *European Symposium on Algorithms*, pages 1–11, 1996.

Donald L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959.

Naoyuki Tokuda. An improved shellsort. In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture*, pages 449–457, 1992.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

A Baseline Gap Sequences

- **Shell** (1959): $\lfloor N/2^k \rfloor$
- **Hibbard** (1963): $2^k - 1$
- **Knuth** (1973): $(3^k - 1)/2$
- **Sedgewick** (1996): Interleaved formula, $O(N^{4/3})$ worst case
- **Tokuda** (1992): $\lceil (9 \cdot (9/4)^{k-1} - 4)/5 \rceil$
- **Ciura** (2001): Empirically optimized: 1, 4, 10, 23, 57, 132, 301, 701, extended by $\times 2.25$
- **Pratt** (1972): All $2^p 3^q < N$, $O(N \log^2 N)$ worst case

B Experimental Details

B.1 Training and evaluation configuration

Unless otherwise stated, we train a separate policy for each of the tested array sizes. Each training array is drawn uniformly from the seven families defined below. The main scaling experiments use 30 independent training runs per array size, and each trained policy is evaluated greedily on the same held-out set of 1000 arrays from each family.

The policy uses a shared two-layer MLP trunk with 128 hidden units per layer, LayerNorm, and ReLU activations, followed by separate actor and critic heads. The full training configuration is given in Table 6.

Table 6: Policy and optimization configuration used in the main experiments.

Setting	Value
Shared policy trunk	2×128, LayerNorm, ReLU
Actor head	$128 \to 128 \to \lvert\mathcal{A}\rvert$, ReLU
Critic head	$128 \to 128 \to 1$, ReLU
Optimizer	Adam
Initial learning rate	3×10^{-4}
Learning-rate schedule	linear decay to $0.3\times$ initial LR
Discount γ	0.99
GAE λ	0.95
PPO clip coefficient	0.2
Entropy coefficient	$0.05 \to 0.005$ linearly
Value-loss coefficient	0.5
PPO epochs per update	4
Minibatch size	256
Parallel environments	32
Rollout length	64
PPO updates	500
Gradient clipping	global norm 0.5
Maximum Shellsort passes ($T_{\max}$)	12
Training-family sampling	uniform over seven families
Evaluation arrays per family	1000

B.2 Input family construction

The main experiments use seven synthetic input families chosen to expose different kinds of array structure rather than to model particular application workloads. Each training array is sampled uniformly from these seven families. Several families also randomize a structural parameter for every new array, so each family covers a range of related inputs rather than one fixed template.

Random. We uniformly shuffle the permutation $[0, 1, \ldots, N-1]$. This gives the standard random-permutation setting used throughout the Shellsort literature.

Nearly sorted. We begin with a sorted permutation, select 5% of its positions uniformly without replacement, and randomly permute only the values at those positions. The rest of the array remains untouched, producing sparse disorder within an otherwise sorted input.

Reversed. We begin with the descending permutation and add a small amount of local noise. Specifically, we perform $0.02N$ random swaps, with each swap restricted to positions at most eight indices apart. The array therefore retains its global reversed structure while varying locally from one sample to another.

Table 7: Parameters used to construct the seven main input families. Set-valued parameters are sampled uniformly for each array.

Family	Parameter setting
Random	–
Nearly sorted	shuffled positions: $0.05N$
Reversed	local swaps: $0.02N$; window: 8
Interleaved	blocks: $\{3, 4, 5, 6, 8\}$
Pipe-organ	peak: $[0.35N,\ 0.65N]$
Sawtooth	teeth: $\{4, 6, 8, 10\}$
Many duplicates	support size: $0.1N$

Interleaved. We divide the sorted value range into several contiguous ascending blocks, randomly permute the block order, and then interleave the blocks round-robin. The number of blocks is sampled from $\{3, 4, 5, 6, 8\}$ for each array. Thus each block remains internally ordered, but values from different parts of the sorted range are mixed throughout the array.

Pipe-organ. We randomly divide the values into a left and right group by shuffling the permutation and cutting it at a peak position sampled between $0.35N$ and $0.65N$. We sort the left group in ascending order and the right group in descending order, producing an array that rises to a peak and then falls.

Sawtooth. We begin with a random permutation, divide it into a sampled number of chunks, sort each chunk independently, and then interleave the sorted chunks round-robin. We sample the number of chunks from $\{4, 6, 8, 10\}$. This produces repeated locally ordered runs whose values remain globally interleaved.

Many duplicates. We draw each of the N entries independently and uniformly from a support of approximately $0.1N$ distinct values. Unlike the other six main families, these arrays are not permutations and contain repeated values throughout.

B.3 Additional input families

The main experiments use the seven input families defined above. To test whether a single policy can cover a broader range of structures, we expand the training mixture with six additional families, giving thirteen in total. These families introduce several patterns that are absent from the original suite, including cyclic shifts, long sorted blocks, heavy-tailed repeated values, and disorder deliberately concentrated at the gaps used by Sedgewick.

Rotated. We begin with the sorted permutation and cyclically rotate it by an offset sampled uniformly between $N/4$ and $3N/4$. The result consists of two individually sorted regions separated by one large wraparound discontinuity.

Table 8: Parameters used to construct the six additional evaluation families.

Family	Parameter setting
Rotated	offset: $[N/4,\ 3N/4)$
Organ-reversed	split: $N/2$
$\sqrt{N}$-sorted	block size: $\lfloor\sqrt{N}\rfloor$
Anti-Shellsort	swap probability: 0.5
Mostly equal	replaced positions: $0.1N$
Zipf	shape: 1.5

Organ-reversed. This reverses the orientation of the pipe-organ family. After randomly dividing the values into two equal groups, we sort the left half in descending order and the right half in ascending order. The array therefore falls toward the center and then rises.

$\sqrt{N}$-sorted. We divide the sorted permutation into consecutive blocks of length approximately $\sqrt{N}$ and randomly shuffle the order of those blocks, while preserving the order within each block. This creates an array that is locally sorted over fairly long ranges but globally misordered.

Anti-Shellsort. We begin with a sorted permutation and deliberately inject disorder at the gaps used by Sedgewick's sequence. For each Sedgewick gap, we visit pairs separated by that gap and swap each pair with probability 0.5. This creates a family whose disorder is concentrated specifically at distances targeted by one of our strongest classical baselines.

Mostly equal. We begin with a constant array and replace 10% of its positions with independently sampled values from 0 through $N-1$. Most entries are therefore identical, with a small number of values scattered around the common background value.

Zipf. We draw all N entries independently from a Zipf distribution with shape parameter 1.5. This produces a heavy-tailed input with many repeated small values and a smaller number of much larger values.

Training on additional input families. We retrain the policy from scratch on a uniform mixture of all thirteen families, keeping the policy architecture and training setup unchanged. We repeat this at $N = 256$, $4{,}096$, and $16{,}384$, with 30 independent training seeds at each size. Table 9 reports performance on both the original seven families and the six additions.

Table 9: Performance when training jointly on all thirteen input families. Entries report mean comparisons $\pm$ standard deviation over 30 training seeds, with improvement relative to Sedgewick in parentheses. Lower is better.

	$N = 256$		$N = 4{,}096$		$N = 16{,}384$	
Family	Sedgewick	Policy	Sedgewick	Policy	Sedgewick	Policy
Original seven families						
Random	2,522	2,448 ± 3 (-2.9%)	70,514	69,025 ± 264 (-2.1%)	345,043	339,108 $\pm 1{,}232$ (-1.7%)
Nearly sorted	1,651	1,095 ± 0 (-33.7%)	57,715	45,109 ± 187 (-21.8%)	293,456	241,601 $\pm 1{,}557$ (-17.7%)
Reversed	1,751	1,550 ± 25 (-11.5%)	46,950	39,994 ± 573 (-14.8%)	228,608	192,554 $\pm 4{,}055$ (-15.8%)
Interleaved	2,095	1,790 ± 57 (-14.5%)	58,265	44,321 $\pm 1{,}268$ (-23.9%)	288,044	215,407 $\pm 4{,}087$ (-25.2%)
Pipe organ	1,785	1,622 ± 11 (-9.2%)	46,664	40,805 ± 530 (-12.6%)	227,728	192,278 ± 793 (-15.6%)
Sawtooth	1,894	1,363 ± 36 (-28.0%)	49,493	33,033 ± 311 (-33.3%)	239,415	150,043 $\pm 1{,}934$ (-37.3%)
Many duplicates	2,035	1,984 ± 6 (-2.5%)	62,409	61,564 ± 303 (-1.4%)	312,486	309,124 $\pm 1{,}059$ (-1.1%)
Six additional families						
Rotated	1,667	1,131 ± 31 (-32.1%)	45,523	32,103 $\pm 1{,}033$ (-29.5%)	222,654	156,124 $\pm 4{,}582$ (-29.9%)
Organ reversed	1,749	1,569 ± 5 (-10.3%)	46,230	39,863 ± 347 (-13.8%)	225,172	187,791 $\pm 1{,}587$ (-16.6%)
$\sqrt{N}$-sorted	2,084	895 ± 14 (-57.1%)	64,862	30,119 $\pm 1{,}036$ (-53.6%)	322,767	165,106 $\pm 5{,}423$ (-48.8%)
Anti-Shellsort	2,214	2,446 ± 3 $(+10.5\%)$	67,098	68,826 ± 275 $(+2.6\%)$	331,989	338,307 $\pm 1{,}240$ $(+1.9\%)$
Mostly equal	1,263	894 ± 1 (-29.3%)	36,253	24,024 ± 81 (-33.7%)	179,477	118,144 ± 543 (-34.2%)
Zipf	1,838	1,823 ± 4 (-0.8%)	45,686	43,395 ± 280 (-5.0%)	215,281	198,639 $\pm 1{,}494$ (-7.7%)

The expanded policy beats Sedgewick on 12 of the 13 families at every array size we test. Several of the new families leave substantial room for specialization: the gains reach roughly 30% on rotated inputs, 29–34% on mostly-equal arrays, and 49–57% on the $\sqrt{N}$-sorted family. The policy also retains strong performance on the original seven families after the training mixture is expanded.

The only exception is Anti-Shellsort, which is deliberately constructed by placing disorder at Sedgewick's own gap distances and therefore gives that baseline an unusually favorable input. Even in this case, the jointly trained policy closes most of the gap as the arrays grow, from $+10.5\%$ at $N = 256$ to only $+1.9\%$ at $N = 16{,}384$. Nevertheless, we find that overall, the same policy can be trained jointly across a substantially broader collection of input structures without losing the improvements seen in the original suite.

C Compiled Schedule Details

Section 5 reports the average performance of the compiled schedules across all seven input families. Here we give the corresponding family-level results and show the fixed gap sequences produced by the compilation procedure.

Per-family results. Table 10 compares the live policy with the compiled schedules at three representative array sizes. Compilation is nearly lossless on several families. For random, nearly-sorted, reversed, and many-duplicate inputs, the compiled schedules closely match the live policy throughout

the tested range. The largest differences appear on interleaved inputs at the larger array sizes and on sawtooth inputs at $N = 256$.

Table 10: Per-family performance of the live policy and compiled fixed schedules at three array sizes. Entries are percentage changes in comparison count relative to Sedgewick; more negative is better. Compiled results use the independently selected schedule from each of the 30 trained policies, as described in Section 5.1.

	$N = 256$		$N = 4{,}096$		$N = 16{,}384$	
Family	Live	Compiled	Live	Compiled	Live	Compiled
Random	-3.1%	-3.0%	-2.2%	-2.2%	-1.8%	-1.8%
Nearly sorted	-33.9%	-33.9%	-21.9%	-21.9%	-18.0%	-18.0%
Reversed	-10.5%	-10.5%	-14.6%	-14.6%	-17.6%	-17.6%
Interleaved	-16.5%	-15.8%	-25.8%	-17.3%	-27.0%	-21.8%
Pipe organ	-9.3%	-8.7%	-12.8%	-12.5%	-15.8%	-15.8%
Sawtooth	-28.7%	-25.1%	-34.0%	-32.8%	-38.2%	-37.3%
Many duplicates	-2.1%	-2.2%	-1.4%	-1.5%	-1.1%	-1.1%

Compiled gap sequences. Each of the 30 trained policies produces its own set of candidate schedules, so the compilation procedure does not necessarily return the same sequence across training runs. Tables 11 and 12 show the most common compiled sequence for each family and report how many of the 30 runs selected that exact sequence. These representative sequences are shown only to illustrate what compilation produces; all performance results above use each run's independently selected schedule rather than the modal sequence.

Table 11: Most common compiled fixed gap sequences for $N = 256$ and $N = 4{,}096$. Parentheses give the number of training runs, out of 30, for which that exact sequence was selected.

Family	$N = 256$	$N = 4{,}096$
Random	$[98, 28, 11, 4, 1]$ $(12/30)$	$[1853, 697, 262, 130, 56, 21, 10, 4, 1]$ $(2/30)$
Nearly sorted	$[4, 1]$ $(27/30)$	$[81, 28, 13, 4, 1]$ $(12/30)$
Reversed	$[98, 24, 5, 1]$ $(17/30)$	$[1611, 262, 49, 11, 4, 1]$ $(10/30)$
Interleaved	$[49, 7, 1]$ $(22/30)$	$[697, 113, 37, 11, 4, 1]$ $(11/30)$
Pipe organ	$[98, 24, 5, 1]$ $(12/30)$	$[2048, 527, 113, 24, 4, 1]$ $(2/30)$
Sawtooth	$[7, 1]$ $(16/30)$	$[37, 11, 4, 1]$ $(22/30)$
Many duplicates	$[98, 28, 11, 4, 1]$ $(12/30)$	$[1611, 697, 301, 113, 49, 28, 11, 4, 1]$ $(2/30)$

Table 12: Most common compiled fixed gap sequences for $N = 16{,}384$. Parentheses give the number of training runs, out of 30, for which that exact sequence was selected.

Family	$N = 16{,}384$
Random	$[3241, 1401, 606, 262, 130, 49, 28, 11, 4, 1]$ $(2/30)$
Nearly sorted	$[301, 113, 49, 28, 11, 4, 1]$ $(5/30)$
Reversed	$[6561, 1401, 346, 65, 15, 4, 1]$ $(4/30)$
Interleaved	$[3241, 527, 49, 11, 4, 1]$ $(10/30)$
Pipe organ	$[4096, 697, 130, 27, 5, 1]$ $(2/30)$
Sawtooth	$[49, 11, 4, 1]$ $(18/30)$
Many duplicates	$[3241, 1401, 606, 262, 130, 49, 28, 11, 4, 1]$ $(2/30)$

The amount of agreement across training runs varies substantially by family. At $N = 256$, nearly-sorted inputs compile to $[4, 1]$ in 27 of the 30 runs, and interleaved inputs compile to $[49, 7, 1]$ in 22. Other families have several different schedules that perform similarly, particularly at the larger array sizes. This variation does not prevent successful compilation: the performance results in Table 10 remain stable even when different training runs settle on different fixed sequences.

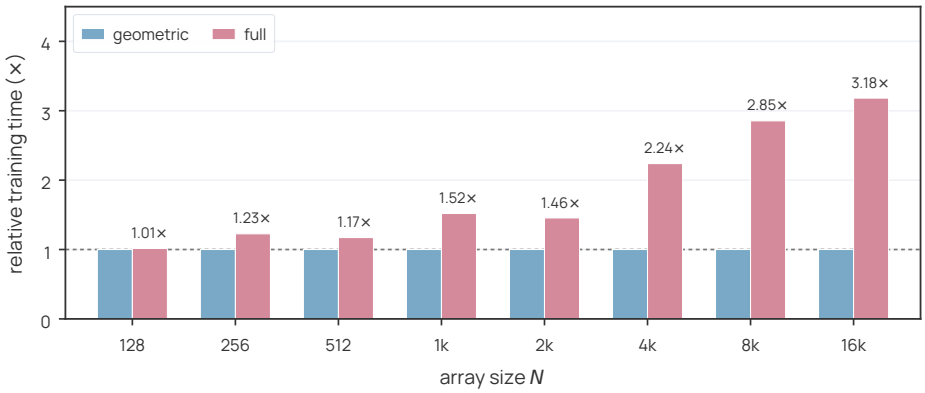

Figure 3: Training cost of the geometric and full gap action spaces across array sizes. The difference is negligible at the smallest sizes but grows quickly with N: the full action space takes $2.24\times$ as long at $N = 4{,}096$, $2.85\times$ at $N = 8{,}192$, and $3.18\times$ at $N = 16{,}384$.

D Action Space Discretization

Shellsort permits any integer gap from 1 through $N/2$. Our main experiments use a much smaller candidate set containing all gaps through 16, a geometric sequence with ratio 1.15, and powers of 2 and 3 (as described in Section 3). Since these candidates are chosen before training, one natural question is whether that restriction itself might be making the scheduling problem unusually easy. We repeat the main experiment with the full $N/2$-gap action space at every array size, keeping the training distribution, policy architecture, reward, 500-update training budget, and evaluation protocol unchanged.

Table 13: Restricted geometric and full integer gap action spaces across all eight array sizes (mean ± std over 30 training seeds). The two action spaces give similar comparison counts throughout the tested range, but the cost of training with every possible gap grows rapidly with N. At $N = 16{,}384$, the full action space contains $8{,}192$ gaps instead of 74 and takes $3.18\times$ as long to train.

N	Action space	Gaps	Policy ($\pm\sigma$)	vs. Sedgewick	Train time
128	Geometric	27	699 ± 3	-13.5%	66s
	Full	64	694 ± 2	-14.1%	67s
256	Geometric	34	1,690 ± 13	-13.9%	71s
	Full	128	1,682 ± 8	-14.3%	87s
512	Geometric	41	4,001 ± 18	-14.3%	76s
	Full	256	3,993 ± 26	-14.5%	89s
1,024	Geometric	47	9,324 ± 32	-14.2%	91s
	Full	512	9,220 ± 32	-15.1%	138s
2,048	Geometric	54	21,102 ± 58	-15.3%	108s
	Full	1,024	21,034 ± 55	-15.6%	157s
4,096	Geometric	60	47,465 ± 142	-15.3%	174s
	Full	2,048	47,419 ± 158	-15.4%	389s
8,192	Geometric	67	105,868 ± 248	-15.9%	263s
	Full	4,096	106,090 ± 401	-15.8%	751s
16,384	Geometric	74	232,671 ± 878	-15.9%	444s
	Full	8,192	236,216 $\pm 1{,}310$	-14.6%	1,413s

Table 13 shows that opening the action space to every possible gap changes the comparison count surprisingly little. Through $N = 4{,}096$, the full action space is usually slightly better, but the

difference never exceeds 0.9 percentage points relative to Sedgewick. For example, at $N = 256$ the margin changes from -13.9% to -14.3%, and at $N = 4{,}096$ from -15.3% to -15.4%. The policy reaches essentially the same improvement even when it is free to choose from every valid Shellsort gap.

On the other hand, Figure 3 shows that the computational difference becomes much larger as the arrays grow. At $N = 128$, the full action space contains only 64 gaps and takes about the same time to train as the 27-gap geometric set. At $N = 4{,}096$, the comparison is $2{,}048$ gaps against 60, and training takes $2.24\times$ as long. By $N = 16{,}384$, the full space has grown to $8{,}192$ gaps against only 74 in the geometric set, increasing training time from 444 to $1{,}413$ seconds ($3.18\times$).

Under the same 500-update budget, the full action space also stops improving the result at the two largest sizes. At $N = 8{,}192$ the two policies are nearly tied, and at $N = 16{,}384$ the geometric set is ahead by 1.3 percentage points relative to Sedgewick. We use the geometric candidate set mainly to keep the learning problem small as N grows. The full-space experiments show that restricting the available gaps is not what produces the gain over the classical sequences.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The paper reports 13–16% fewer comparisons than Sedgewick across the tested sizes and seven-family mixture, isolates the effects of initial-spectrum conditioning and sequential feedback, and shows that compiled fixed schedules retain 90.5–95.6% of the live policy's gain. We show that our claims hold in the evaluated settings.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper discusses the cost of running the policy inside every sort, the need for representative stable workloads when compiling fixed schedules, and the limits of the synthetic evaluation. Appendix B.3 includes an input family on which the policy does not beat Sedgewick.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.

- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The contribution is empirical; no new theorems or proofs are claimed.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Section 3 defines the MDP. The appendix documents the optimizer, synthetic input generators, baseline gap sequences, and evaluation settings. Training and evaluation scripts are publicly available at `https://github.com/cminst/neuroshell`.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case

of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.

- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The training and evaluation scripts, `requirements.txt`, and license are available at `https://github.com/cminst/neuroshell`. The datasets are synthetic and can be regenerated using the supplied code.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: The appendix specifies the PPO configuration, array-family generators, and evaluation protocol. Results use 30 training seeds and held-out arrays; compilation uses separate candidate-generation, validation, and test sets (Section 5.1).

   Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: The results report means and standard deviations across 30 independent training seeds, using 1,000 held-out evaluation arrays per family. Table captions identify the reported variability.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Experiments ran entirely on an AMD Ryzen 9 9950X CPU (16 cores, 32 threads) with 123 GiB of system RAM; no GPUs were used. Table 2 reports 500-update training times from 66 to 444 seconds across the main array sizes.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

Answer: [Yes]

Justification: The work studies a classical sorting algorithm using synthetic arrays. It involves no human participants, personal data, or sensitive datasets.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: The broader-impacts paragraph discusses possible efficiency benefits for embedded and edge devices, and notes that a workload-specific schedule may degrade if the input distribution changes.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The work releases sorting code, not a high-risk model or sensitive dataset requiring misuse safeguards.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.

- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The public repository pins NumPy 2.2.6 (BSD-3-Clause), Numba 0.61.2 (BSD-2-Clause), and PyTorch 2.12.0 (BSD-3-Clause). The classical gap sequences and PPO method are credited to their original publications.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: Our public repository at `https://github.com/cminst/neuroshell` contains the training and evaluation scripts, installation requirements, and a LICENSE file for the released code.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: No crowdsourcing or human-subject studies were conducted.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: No human subjects were involved, so IRB review does not apply.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [N/A]

Justification: No LLM is part of the method or experimental evaluation. The learned scheduling policy is a small MLP trained with PPO.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.