ONLINE TOKENIZATION WITH OPTIMAL DELAY IN LINEAR TIME

Anonymous authors
Paper under double-blind review

ABSTRACT

Tokenization is an essential preprocessing step in almost all large language models (LLMs). One of the most popular frameworks is BPE tokenization, which consists of special token segmentation, pretokenization, and byte-pair encoding (BPE). However, existing algorithms are sequential, with each stage only processing complete output from the previous stage. In order to reduce latency and memory footprints, recent works (Jiang and Gong, ICML 2026) (Li, Yang and Mamouras, ASPLOS 2026) (Mamouras, Li and Yang, PLDI 2026) have initiated the study of online algorithms for subroutines such as pretokenization and BPE with a goal of streaming input to output as the input arrives online. However, the search for efficient online tokenization remains open.

In this work, we formalize and resolve online tokenization. We introduce the notion of delay for online tokenization which characterizes the extent to which the algorithm output lags behind the (possibly computationally intractable) optimum. We give a near-linear time algorithm with optimal delay (up to constant factors). We complement this algorithm with a conditional lower bound: unless $\mathbf{P} = \mathbf{NP}$, there is no polynomial time algorithm that beats the delay of our algorithm by more than a constant factor.

We implement a practical variant for tokenizers from OpenAI's tiktoken and HuggingFace's tokenizers. We obtain a $7.9\times$-$34.8\times$ latency advantage (measured as time to first token) over tiktoken (as well as the recent gigatoken tokenizer) and our implementation maintains approximately constant memory usage independent of input size. Our implementation also achieves a $2.4\times$-$7.0\times$ improvement in throughput over tiktoken and $15.8\times$-$38.1\times$ over tokenizers.

1 INTRODUCTION

In many natural language processing (NLP) applications, including powerful and popular large language models (LLMs), raw text undergoes tokenization before being provided as model input. Tokenization defines a vocabulary containing as many common words and sequences as possible, which help compress raw text into tokens, balancing expressivity and computational efficiency.

Byte-Pair Encoding (BPE) is a popular method for tokenization proposed by Gage (1994) that has been widely adopted in LLMs (Sennrich et al., 2016). Indeed, BPE Tokenization is both the subject of intense recent theoretical study (Berglund & van der Merwe, 2023; Jiang & Gong, 2026; Mamouras et al., 2026) and the basis for popular practical tokenizers, including OpenAI's `tiktoken` and HuggingFace's `tokenizers` (OpenAI, 2026; HuggingFace, 2026).

Concretely, tokenization consists of a three step process: Special Token, Pretokenization, and BPE. As observed in Jiang & Gong (2026), existing implementations of tokenization are inherently offline and sequential: we take in the full input, apply segmentation to the full input, then apply pretokenization on each segment, and then apply BPE to each pretoken. While completed segments and pretokens can be passed on to the next stage, one should in principle be able to apply pretokenization to a segment as it forms and similarly apply BPE to a pretoken as it forms. Following this observation, Jiang & Gong (2026) raise the question of online tokenization: can we process each stage as the input arrives in an online setting?

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.

While offline tokenization is now well understood both theoretically and empirically, the question of efficient online tokenization remains wide open. Online algorithms are desirable for many reasons. First, we can drastically reduce the the memory footprint of our algorithm by avoiding holding the entire input or output in memory, a shortcoming of all existing end-to-end tokenization algorithms. Second, streaming can reduce latency of tokenization, drastically reducing the time to first token. Indeed, streaming and online algorithms are a fundamental and well-studied paradigm both in theory and in practice, with efficient algorithms being developed in many applications including counting (Morris, 1978), selection and sorting (Munro & Paterson, 1980), heavy hitters (Misra & Gries, 1982), frequency moments (Alon et al., 1996), and cardinality estimation (Flajolet & Martin, 1985; Durand & Flajolet, 2003; Flajolet et al., 2007).

Motivated by latency and memory constraints, online algorithms have been developed for individual steps of tokenization: Li et al. (2026) studies the problem of online pretokenization and Berglund & van der Merwe (2023); Jiang & Gong (2026); Mamouras et al. (2026) study online BPE algorithms. However, none of these works provide an online algorithm for the full tokenization pipeline. Indeed, Jiang & Gong (2026) raise the design of end-to-end online tokenization as a crucial direction for future research. Our work is thus guided by the following question:

Is there an efficient algorithm for online tokenization?

1.1 Formalizing Online Tokenization

We begin with formalizing tokenization which consists of the following three stage process:

1. Special Token (Seg): Given a list of special tokens S^* and a string x, split the string by occurrences of special tokens (see Definition B.1) into segments. For example, if $S^* = \{\texttt{"<|endoftext|>"}\}$ and the input string is $x_1 = \texttt{"file1<|endoftext|>file2"}$, we have
$$\mathrm{Seg}(x_1; S^*) = \texttt{"file1"} \parallel \texttt{"<|endoftext|>"} \parallel \texttt{"file2"}.$$
We use $\parallel$ to denote segment boundaries.
2. Pretokenization (Split): Given a regular expression (regex) R and a string x, split each segment of x into pretokens by repeatedly finding the longest match of R in x (see Definition B.2). For example, if $R = a^* \mid b^*$ and $x_2 = \texttt{"aaabbba"} \parallel \texttt{"aaaaabbbbaaa"}$,
$$\mathrm{Split}(x_2; R) = \texttt{"aaa"} \parallel \texttt{"bbb"} \parallel \texttt{"a"} \parallel \texttt{"aaaaa"} \parallel \texttt{"bbbb"} \parallel \texttt{"aaa"}.$$
As we see in our example, matches cannot extend across segment boundaries.
3. BPE (Merge): Given a merge list M and a list of pretokens x, apply BPE to each pretoken of x, by repeatedly applying merges in order of priority left to right (see Definition B.4). For example, if $x_3 = \texttt{"aaabbbaaa"}$ and $M = \{(a, b) \mapsto X, (a, a) \mapsto A, (b, b) \mapsto B\}$,
$$\mathrm{Merge}(x_3; M) = \texttt{"AXBAa"}.$$

Tokenization, given special tokens S^*, regex R, and merge list M, is defined
$$\mathrm{Tok}(x; S^*, R, M) = \mathrm{Merge}(\mathrm{Split}(\mathrm{Seg}(x; S^*); R); M).$$
Note that each token τ in the output corresponds to a unique string x denoted $\mathrm{raw}(\tau)$ where $\mathrm{Merge}(x) = \tau$. For example, using the same R and M as above,
$$\mathrm{Tok}(x_3; \emptyset, R, M) = \mathrm{Merge}(\texttt{"aaa"} \parallel \texttt{"bbb"} \parallel \texttt{"aaa"}) = \texttt{"AaBbAa"}.$$
That is, merges cannot merge across pretoken boundaries, and the merged pretokens are concatenated. In the offline setting, efficient algorithms are now well known. For example, Seg and Split have classical linear time algorithms (Reps, 1998). The naive algorithm for Merge runs in $O(n\mathfrak{m})$ time, which is already efficient when $\mathfrak{m} = |M|$ is small. When $\mathfrak{m}$ is large, there are well known $\tilde{O}(n)$ time heap-based algorithms (see e.g. the `tiktoken` implementation (OpenAI, 2026)).

We now define *online* Seg, Split, Merge, and Tok. First, for two strings w, w' in an alphabet Σ, if w is a prefix of w', we show it by $w \preceq w'$. Consider a function $f : \Sigma^* \to (\Sigma')^*$ where $\Sigma \subseteq \Sigma'$ are alphabets (e.g. characters and tokens). In our context, f could be any of the Seg, Split,

Merge, or Tok functions. We say that $z \in (\Sigma')^*$ is a *stable prefix* with respect to x if $z \preceq f(xy)$ for any $y \in \Sigma^*$. It is easy to show that stable prefixes are totally ordered (see Claim B.8). The *maximum stable prefix* is then denoted by $f_{\preceq}(x)$. We define online tokenization (analogously each of the three sub-routines) as $\mathrm{Tok}_{\preceq}$. In particular, given that input x is all we have observed so far, online tokenization outputs $\mathrm{Tok}_{\preceq}(x)$, the longest possible prefix of tokens that can be safely emitted, regardless of what characters appear in the future. This is exactly the definition that allows us to stream input characters and output tokens.

Surprisingly, our first result establishes that online tokenization, unlike its offline counterpart, is **NP**-Hard. Given that it is unlikely that even a polynomial (let alone linear time) algorithm for online tokenization exists, a natural relaxation is to design efficient approximation algorithms. We introduce a natural approximation of online tokenization which we call *delay* (see Definition B.9). An online tokenization with delay k produces on input $x = x_1 \dots x_n$, output z satisfying:

$$\mathrm{Tok}_{\preceq}(x_1 \dots x_{n-k}) \preceq z \preceq \mathrm{Tok}_{\preceq}(x).$$

In particular, an algorithm with small delay produces a stable prefix that might not be the maximum stable prefix, but would have been the maximum stable prefix not too long ago. A trivial algorithm achieves delay n: output the empty string ϵ. So we focus on answering the following question:

What is the minimum delay k with efficient algorithms for online tokenization?

1.2 Our Contributions

We give optimal algorithms for online tokenization, tightly characterizing when efficient online tokenization is possible, resolving the main open question of Jiang & Gong (2026). First, our lower bound rules out the existence of efficient algorithms with small delay. For any list of special tokens S^*, let $\ell_\infty(S^*) = \max_{t \in S^*} |t|$ be the longest special token and $\ell_1(S^*) = \sum_{t \in S^*} |t|$ be the sum of the lengths of the special tokens. For any merge list M, let $\mathfrak{m} = |M|$ and $\mathfrak{t} = \max\{|x| \text{ s.t. } |\mathrm{Merge}(x)| = 1\}$ denote the length of the longest token.

Note $\ell_\infty(S^*)$, $\ell_1(S^*)$, $|R|$ are constants independent of x, so we omit them from bounds for clarity. While $\mathfrak{m}$ (and therefore $\mathfrak{t}$) is also a constant (since the BPE dictionary is independent of the input string), since $\mathfrak{m}$ can be quite large in practice (around 100,000 for CL100K in `tiktoken`), we will account for $\mathfrak{m}, \mathfrak{t}$ in space and run-time, keeping consistent with prior work (Jiang & Gong, 2026).

Theorem 1.1 (Hardness of Online Tokenization)**.** *Unless* $\mathbf{P} = \mathbf{NP}$*, there is no algorithm for exact (zero-delay) online tokenization with* $\mathrm{poly}(\ell_1(S^*), |R|, \mathfrak{m}, \mathfrak{t})$ *preprocessing and total time* $\mathrm{poly}(n, \ell_1(S^*), |R|, \mathfrak{m}, \mathfrak{t})$*, even when* $S^* = \emptyset$*. Furthermore, the lower bounds hold for algorithms with* $o(\mathfrak{m}\mathfrak{t})$ *delay.*

A natural attempt at an online tokenization algorithm is to compose online algorithms for special token, pretokenization, and BPE. We show that this idea of composition does in fact yield an efficient online tokenization algorithm with non-trivial delay.

The performance of this algorithm, which we call composition, is parameterized by the following quantity which depends on the pretokenization regular expression. Given a regular language L, we define the lookback of the regular language, denoted $\mathfrak{b}(L)$, to be the smallest integer (or ∞ if no such number exists) such that every string $x \in L$ of length $|x| > \mathfrak{b}$ has a strict prefix $y \prec x$ such that $y \in L$, and $|y| \geqslant |x| - \mathfrak{b}$ (see Definition B.3), i.e. y is in the language and y is shorter than x by at most $\mathfrak{b}$. For example, if $R = a^*$, we have $\mathfrak{b}(R) = 1$. On the other hand, if $R = a^*b$, $\mathfrak{b}(R) = \infty$.

Theorem 1.2 (Online Tokenization via Composition)**.** *Let* $S^* \subset \Sigma^*$ *be a set of special tokens,* R *a total regular expression over* Σ*, and* M *a merge list. Then, there is an* $O(n \log^2 \mathfrak{t})$ *time and* $O(\mathfrak{b}(R) + \mathfrak{m}\mathfrak{t})$ *space algorithm for online tokenization with* $\mathfrak{b}(R) + (\mathfrak{m}+1)\mathfrak{t} + \ell_\infty(S^*)$ *delay.*

Remark 1.3. *As part of our algorithm for Theorem 1.2, we design a simple, novel linear time algorithm for online pretokenization (see Lemma B.17). In contrast, the previous algorithm of Li et al. (2026) incurs preprocessing time and space* $2^{\Theta(\mathfrak{b}(R))}$*. In particular, we achieve online pretokenization without delay in linear time for all regular expressions under the longest-match convention, whereas the previous algorithm was only efficient when* $\mathfrak{b}(R) = O(\log n)$ *is small.*

Remark 1.4. *While we have efficient algorithms for online BPE (Jiang & Gong, 2026) and pretokenization via Lemma B.17, Theorem 1.1 still establishes that online tokenization (with no delay) is*

NP-*hard. Our algorithm for online pretokenization has* 0 *delay. Known algorithms for online BPE (Jiang & Gong, 2026) have delay* $O(\mathfrak{mt})$ *(see proof of Lemma B.18) which suffices for our purposes. We leave as an interesting open question whether zero delay is achievable for online BPE.*

While our first algorithm does not achieve optimal delay, we show that its space is optimal (up to word size). In particular, $\Theta(\mathfrak{b}(R))$ bits is necessary for online tokenization.

Theorem 1.5 (Space Lower Bound)**.** *For every integer* f*, there exists* R *where* $\mathfrak{b}(R) = 2f$ *and online tokenization requires* $\Omega(\mathfrak{b}(R))$ *bits in space.*

While our algorithm requires $O(\mathfrak{b}(R) + \mathfrak{mt})$ space, we note that $\mathfrak{mt}$ is a constant independent of the input x, while $\mathfrak{b}(R)$ can be arbitrarily large (see e.g. $R = a^*b$ above). We note that Theorem B.24 shows that $\Omega(\mathfrak{m})$ bits of space are necessary for online BPE, and therefore online tokenization, essentially showing that our space usage is optimal (up to a $\mathfrak{t}$ factor and word size).

Finally, we discuss delay-optimal algorithms for online tokenization. Note that Theorem 1.2 incurs delay $O(\mathfrak{b}(R)+\mathfrak{mt})$[1] and thus incurs additional $O(\mathfrak{b}(R))$ delay over the lower bound of Theorem 1.1. We show that this is not an artifact of our analysis: composition alone will always incur delay $\Omega(\mathfrak{b}(R))$ on some instances.

Theorem 1.6 (Composition Delay Bound)**.** *For every integer* $f \geqslant 2$ *there exists alphabet* Σ*, a total regular expression* R *over* Σ^**, merge list* M *and some input* x *where* $\mathfrak{b}(R) = f$ *and*

$$\mathrm{Merge}_{\preceq}(\mathrm{Split}_{\preceq}(\mathrm{Seg}_{\preceq}(x))) \prec \mathrm{Tok}_{\preceq}(x[:-(f-2)]).$$

Thus, if we want to obtain a delay-optimal algorithm, it cannot be from composition alone. Using these insights, our main result establishes an algorithm that achieves near linear time, optimal space, and optimal delay.

Theorem 1.7 (Online Tokenization with Constant Delay)**.** *Let* $S^* \subset \Sigma^*$ *be a set of special tokens,* R *a total regular expression over* Σ*, and* M *a merge list. There is an online tokenization algorithm with preprocessing time* $O(\mathfrak{mt})$*, total time* $O(n(\log^2 \mathfrak{t} + \log \mathfrak{m} + \log \min(n, \mathfrak{b})))$ *and space* $O(\min(n, \mathfrak{b}) + \mathfrak{mt})$ *with* $O(\mathfrak{mt})$ *delay.*

In particular, we characterize (up to constant factors) for what delay efficient online tokenization is possible. When delay is $O(\mathfrak{mt})$, there is a near linear time algorithm. On the other hand, when delay is $o(\mathfrak{mt})$, there is no polynomial time algorithm (unless $\mathbf{P} = \mathbf{NP}$).

2 Experiments and Empirical Results

We implement a practical variant and obtain an efficient tokenizer that streams an input file of characters to an output file of tokens. We evaluate our tokenizer against `tiktoken` and `tokenizers` as a drop-in replacement on code (Github), English, and Chinese datasets. See Section D for experimental setup, dataset composition, as well as implementation details and optimizations. Our code is available in the supplementary material. Our experiments reveal that our implementation exhibits a significant advantage over all evaluated baselines with respect to latency and memory consumption. We also describe some simple optimizations that lead to significant throughput advantages over `tiktoken` and `tokenizers`.

2.1 Latency Benchmark

We measure latency as time to first token (TTFT). Since `tiktoken` and `gigatoken` do not natively support TTFT measurement, we make minimal changes to their code base to allow TTFT measurement, which is measured as the time from when the input is provided to the tokenizer to the time that the first token is formed. Table 1 shows that our tokenizer achieves 7.9×-34.8× TTFT advantage over the best of `tiktoken` and `gigatoken` for every encoding and dataset, exhibiting the significant latency advantage of online tokenization.

[1] Note that the size of S^* is a constant independent of all other parameters. We focus on R and $\mathfrak{m}$ parameters here as S^* is typically not a bottleneck in practice, and we set $S^* = \emptyset$ in the construction of Theorem 1.1

Table 1: Time to first token (TTFT) speedup of our tokenizer over the better of `tiktoken` and `gigatoken` on each corpus.

Encoding	Github	English	Chinese
`r50k`	21.7×	17.8×	14.3×
`p50k`	19.2×	20.3×	13.8×
`cl100k`	34.8×	11.1×	7.9×
`o200k`	17.8×	11.8×	12.5×

2.2 Memory Benchmark

We measure peak RSS (Resident Set Size) as input size increases. Our implementation exhibits approximately constant peak RSS over the range of input sizes, while existing tokenizers retain the full input or output in memory. See Figure 2 for Github, Figure 3 for English, and Figure 4 for Chinese. Our tokenizer reports lower memory usage than `gigatoken` across all encodings, datasets, and input sizes. Compared to `tiktoken`, our tokenizer reports higher memory usage on small inputs due to initialization overhead but lower memory usage on large inputs. The crossover point is between 2 and 32 MiB-sized inputs depending on the encoding and dataset.

2.3 Throughput Benchmark

Finally, Table 2 exhibits the throughput comparison of our tokenizer against `tiktoken` and `tokenizers` baselines. Depending on the encoding and dataset, our tokenizer exhibits a throughput advantage of 2.4×-7.0× over `tiktoken` and 15.8×-38.1× over `tokenizers`.

Our main optimization replaces the default regular expression of each encoding with a simplified regular expression that allows for faster matching, coupled with a simple, efficient cleanup step that ensures equivalence with the reference regular expression. The optimizations of our streaming algorithms (e.g. simplified regular expressions) are *provably* verified in Section D. Due to their simplicity, the optimizations are portable across different encodings, achieving consistent speedups for all evaluated tokenizers that fall within the framework considered in this work. See Section D for implementation details and practical optimizations.

Table 2: Throughput speedup of our tokenizer over `tiktoken` (top) and `tokenizers` (bottom).

Encoding	Github	English	Chinese
`r50k`	3.9×	2.7×	2.5×
`p50k`	4.2×	2.7×	2.4×
`cl100k`	4.4×	2.9×	2.8×
`o200k`	7.0×	4.8×	3.7×
`GPT2`	38.1×	36.0×	32.2×
`Mistral`	32.8×	27.9×	23.1×
`Llama4`	31.3×	24.2×	19.8×
`Qwen-3`	19.2×	19.7×	15.8×

Recent Work: `gigatoken` Recently, the `gigatoken` tokenizer presents a parallelized tokenizer that achieves significant throughput speedups over previous tokenizers (Rød, 2026). While `gigatoken` achieves high throughput, our implementation, due to its practical streaming design, achieves lower measured TTFT and approximately constant peak RSS over the tested input sizes. We remark that some optimizations of `gigatoken` are compatible with online tokenization, and observe that a combination of the two can achieve improved throughput for some encodings and datasets, especially on the Chinese dataset (see Section E.3.1 for details).[2]

[2]Some `gigatoken` optimizations, such as more aggressive caching of observed pretokens can introduce additional memory usage that grows with the number and size of distinct pretokens encountered.

3 TECHNICAL OVERVIEW

We now describe the main ideas behind our algorithms.

Efficient Online Pretokenization We begin with Theorem 1.2. Our algorithm is a simple composition of online segmentation, pretokenization, and BPE. Thus, in order for composition to be efficient, we need efficient implementations of each online component. Given the efficient online BPE algorithm (Jiang & Gong, 2026), it suffices for us to design a linear time online pretokenization algorithm with zero delay.

Lemma 3.1 (Online Pretokenization (Informal Lemma B.17))**.** *There is an algorithm that computes online pretokenization in $O(n)$ time and $O(\mathfrak{b}(R))$ space.*

Given a string x, let $F(x)$ denote the longest prefix of x that matches R. For simplicity, suppose R can be recognized by a DFA $\mathcal{A}$ of size $\mathrm{poly}(|R|)$ (in Lemma B.17 we describe how to handle NFAs). Let $\mathcal{Q}$ denote the set of states and $\mathcal{P}$ the accepting states of $\mathcal{A}$. For a string $w \in \Sigma^+$, let $\delta(q, w)$ denote the state computed if we begin at state q and read w, and let $\mathrm{last}(q, w)$ be the length of the longest prefix $w' \preceq w$ such that $\delta(q, w') \in \mathcal{P}$; i.e. reading w' from q ends up at an accepting state. A state $q \in \mathcal{A}$ is dead if there is no $y \in \Sigma^+$ such that $\delta(q, y) \in \mathcal{P}$.[3]

Consider the following simple algorithm: initialize the variable $\mathbf{state}$ to start and on input x_i update $\mathbf{state} \leftarrow \delta(\mathbf{state}, x_i) = \delta(\mathrm{start}, x_{\leqslant i})$. Whenever $\mathbf{state} \in \mathcal{P}$, we know that the current prefix $x_{\leqslant i} \preceq F(x)$. That is, $x_{\leqslant i}$ is a prefix of the longest prefix of x that matches R, so we can emit $x_{\leqslant i}$ and set $\mathrm{last} \leftarrow i = \mathrm{last}(\mathrm{start}, x_{\leqslant i})$. Whenever $\mathbf{state} \in \mathrm{dead}$, we know that the last accepting prefix (denoted $x_{\leqslant \mathrm{last}}$) is maximal and therefore $F(x) = x_{\leqslant \mathrm{last}}$. Since we have already emitted $x_{\leqslant \mathrm{last}}$, we can emit $\|$, and restart the algorithm on input $x_{\mathrm{last}+1}$.

We now analyze the performance of this simple algorithm. On each input x_i, our algorithm needs to maintain (1) $\mathbf{state} = \delta(\mathrm{start}, x_{\leqslant i})$ to check whether $\mathbf{state} \in \mathcal{P}$ is accepting or $\mathbf{state} \in \mathrm{dead}$ is dead and (2) $\mathrm{last} = \mathrm{last}(\mathrm{start}, x_{\leqslant i})$ to track the longest accepting prefix so far. By restarting the algorithm at $x_{\mathrm{last}+1}$, naively scanning the input again requires $O(i - \mathrm{last})$ time. Since every accepting path through the DFA reaches an accepting state once every $\mathfrak{b} := \mathfrak{b}(R)$ steps we have $i \leqslant \mathrm{last} + \mathfrak{b}$ (Lemma B.20). Thus, our algorithm backtracks by at most $\mathfrak{b}$ steps on every output, yielding a running time of $O(n\mathfrak{b})$. If we could instead recompute $\delta(\mathrm{start}, x_{\mathrm{last}+1} \dots x_i)$ and $\mathrm{last}(\mathrm{start}, x_{\mathrm{last}+1} \dots x_i)$ efficiently, then we would have an efficient algorithm for online pretokenization.

Li et al. (2026) address this by precomputing δ, last for all possible strings of length $\mathfrak{b}$, resulting in preprocessing time and space that is exponential in $\mathfrak{b}$. In this work, we instead give a simple data structure cQueue that supports exactly the required operations.

Lemma 3.2. *There is a data structure* cQueue *supporting the following operations on a string w:*

1. *(Push) Append a character c to w i.e. update $w \leftarrow wc$.*
2. *(Pop) Pop a prefix of w, i.e. return $w_1 \dots w_p$ and update $w \leftarrow w_{p+1} \dots w_\ell$ for some $p > 0$.*
3. *(Query) Given a state q, compute $\delta(q, w)$ and $\mathrm{last}(q, w)$.*

Furthermore, the data structure supports the operations in amortized $O(1)$ time.

Now, to implement the online pretokenization algorithm efficiently, it suffices to observe that the string $x_{\mathrm{last}+1} \dots x_i$ which we want to query only undergoes queue updates. In particular, we push x_i on each input, and whenever $\mathbf{state} \in \mathrm{dead}$, we pop up to x_{last}. Thus, by supporting the required operations in amortized $O(1)$ time, we obtain a total time $O(n)$ algorithm for online pretokenization.

Composition Delay Bound We now argue that composition already achieves non-trivial delay $O(\mathfrak{b} + \mathfrak{m}\mathfrak{t})$. Composition always outputs a stable prefix (and this in fact holds for composition of any two functions f, g see Claim B.19). If $x' \preceq g(xy)$ for any continuation y, and $z \preceq f(x'y')$ for any continuation y', then $z \preceq f(g(xy))$ for any continuation y, since $g(xy) = x'y'$ for some y'.

[3] An accepting state can be dead, for example, if every transition leads to a dead state.

To bound the delay, it suffices to bound $|x| - |\mathrm{raw}(z)|$ where z is the output of our algorithm (see Claim B.21). Let $z_1 = \mathrm{Seg}_{\preceq}(x)$. In Lemma B.16, we show $|\mathrm{raw}(z_1)| \geqslant |x| - \ell_\infty(S^*)$ where $\ell_\infty(S^*) = \max_{\tau \in S^*} |\tau|$ is the length of the longest special token. Now, let $z_2 = \mathrm{Split}_{\preceq}(z_1)$. As discussed in the pretokenization algorithm, there are two cases when we can safely emit a stable prefix of Split: either $\mathbf{state} \in \mathcal{P}$ and we can emit the accepted prefix, or $\mathbf{state} \in \mathrm{dead}$ and we can emit $\parallel$ and rewind to the last accept. Since the DFA must reach a $\mathcal{P}$ state every $\mathfrak{b} = \mathfrak{b}(R)$ steps, $\mathrm{Split}_{\preceq}$ outputs all but the last $\mathfrak{b}(R)$ characters of z_1 (see Lemma B.20), and $|\mathrm{raw}(z_2)| \geqslant |\mathrm{raw}(z_1)| - \mathfrak{b}$. Finally, let $z = \mathrm{Merge}_{\preceq}(z_2)$ be the output of our algorithm.

We use the key property of BPE which states that for all inputs x,

$$|\mathrm{raw}(\mathrm{Merge}_{\preceq}(x))| \geqslant |x| - (\mathfrak{m}+1)\mathfrak{t}.^{4} \tag{1}$$

Combining all three bounds we can upper bound the delay by

$$|x| - |\mathrm{raw}(z)| \leqslant |x| - |\mathrm{raw}(z_2)| + (\mathfrak{m}+1)\mathfrak{t} \leqslant |x| - |\mathrm{raw}(z_1)| + \mathfrak{b} + (\mathfrak{m}+1)\mathfrak{t} \leqslant \ell_\infty(S^*) + \mathfrak{b} + (\mathfrak{m}+1)\mathfrak{t}.$$

Limitations of Composition While composition produces a stable prefix with non-trivial delay, it falls short of an optimal delay algorithm for online tokenization. While the additive $\ell_\infty(S^*)$ term is an input-independent constant (and usually not a large one in practice), the additive $\mathfrak{b}(R)$ term can be arbitrarily large. Theorem 1.6 shows that this delay is inherent to composition.

Consider the following simple example of a tokenizer: $R = a^n b \mid a \mid b$ and $M = [bb \mapsto B]$; and an input $x = a^n$. On the one hand, $\mathfrak{b}(R) = n$ and indeed $\mathrm{Split}_{\preceq}(x) = a$, since two possible continuations produce $\mathrm{Split}(x) = a \parallel \ldots \parallel a$ and $\mathrm{Split}(xb) = a^n b$ with a longest common prefix of a. On the other hand, since no pretoken contains bb, then regardless of the pretokenization, no merge ever occurs. For our specific example, we in fact have $\mathrm{Tok}_{\preceq}(y) = y$ for any input y. Concretely, we argue the delay is greater than $n-2$ as the algorithm outputs a and

$$\mathrm{Tok}_{\preceq}(x[:-(n-2)]) = \mathrm{Tok}_{\preceq}(x_{\leqslant 2}) = \mathrm{Tok}_{\preceq}(aa) = aa \not\preceq a.$$

From the above example, we observe that composition will always produce an output P where $\mathrm{raw}(P) \preceq x_{\leqslant \mathrm{last}}$, where $x_{\leqslant \mathrm{last}}$ is the longest accepting prefix in the pretokenization. However, a zero-delay online tokenization algorithm might produce output P' that is strictly longer than $\leqslant_{\mathrm{last}}$, i.e. $x_{\leqslant \mathrm{last}} \prec P'$. This is because different continuations may produce different pretokenization outputs, but they nevertheless can produce the same tokenization at the end.

Optimal Delay Algorithm We are now ready to describe our algorithm with optimal delay. Suppose our algorithm proceeds inductively, beginning with an empty output which is a stable prefix of all possible outputs. Consider some stable prefix P of the online tokenization output, i.e. $P \preceq \mathrm{Tok}_{\preceq}(x)$ which we have computed so far. Since $\mathrm{Tok}_{\preceq}(x) \preceq \mathrm{Tok}(x)$, any stable prefix $P' \preceq \mathrm{Tok}_{\preceq}(x)$ must be a prefix of $\mathrm{Tok}(x)$. Let θ be the token following P in $\mathrm{Tok}(x)$, i.e. $P\theta \preceq \mathrm{Tok}(x)$. Then, P is not a maximum stable prefix if and only if $P\theta \preceq \mathrm{Tok}_{\preceq}(x)$: θ is the only candidate token to extend P to a longer stable prefix. It thus suffices to decide whether $P\theta \preceq \mathrm{Tok}_{\preceq}(x)$, equivalently, whether for all $y \in \Sigma^+$ we have $P\theta \preceq \mathrm{Tok}(xy)$.

Recall that $\mathrm{last} = \mathrm{last}(\mathrm{start}, x)$ is the index of the last accepting state in the pretokenization DFA. Denote $\mathrm{Split}(x) = z_1 \parallel \ldots \parallel z_p$ so that $z_1 = x_1 \ldots x_{\mathrm{last}}$[5] and

$$\mathrm{Tok}(x) = \mathrm{Merge}(z_1) \circ \mathrm{Merge}(z_2) \circ \cdots \circ \mathrm{Merge}(z_p).$$

Let out be the index of the last output character so that $\mathrm{raw}(P) = x_1 \ldots x_{\mathrm{out}}$. Let $\mathrm{last}(0) = 0$ and $1 \leqslant q \leqslant p$, let $\mathrm{last}(q)$ be the index such that $z_q = x_{\mathrm{last}(q-1)+1} \ldots x_{\mathrm{last}(q)}$. Note that $\mathrm{last} = \mathrm{last}(1)$.

We now describe our optimal $O(\mathfrak{m}\mathfrak{t})$ delay tokenization algorithm. Let x be the current input of length i and P be the output so far (stable by assumption). In the appendix, we will describe how to use offline tokenization to reduce $\mathbf{state} \in \mathrm{dead}$ to the $\mathbf{state} \notin \mathrm{dead}$ case. Thus, without loss of generality, assume $\mathbf{state} = \delta(\mathrm{start}, x) \notin \mathrm{dead}$. First, if $i - \mathrm{out} \leqslant 3\mathfrak{m}\mathfrak{t}$, then we already have that P has delay $O(\mathfrak{m}\mathfrak{t})$. Thus, assume $i - \mathrm{out} > 3\mathfrak{m}\mathfrak{t}$.

[4] While known online BPE algorithms do not provably compute $\mathrm{Merge}_{\preceq}$, the algorithms nevertheless guarantee that the above bound holds (Lemma B.18)

[5] Note that a different continuation xy may have different outcome $\mathrm{Split}(xy) = z_1 \parallel z_2' \parallel \ldots \parallel z_{p'}'$.

Now, since $i - \text{out} > 3\mathfrak{mt}$, we have

$$|x_{\text{out}+1} \dots x_i y| \geqslant |x_{\text{out}+1} \dots x_i| \geqslant i - \text{out} > 3\mathfrak{mt}.$$

From Equation (1), we can conclude that $\tau := \phi(x_{\text{out}+1} \dots x_i y)$ is independent of y, where $\phi(w)$ denotes the first token in $\text{Merge}(w)$ for any w. In fact, $\tau = \phi(x_{\text{out}+1} \dots x_i y) \preceq \text{Merge}_{\preceq}(x_{\text{out}+1} \dots x_i)$ can be computed with online BPE applied to a suffix of the current input. Of course, if $\tau = \theta$, then we have $P\theta \preceq \text{Tok}_{\preceq}(x)$ so that we can safely emit θ and proceed inductively. On the other hand, if $\tau \neq \theta$, the fact that $\texttt{state} \notin \text{dead}$ already tells us that $P = \text{Tok}_{\preceq}(x)$ is a maximum stable prefix, since there is a continuation y such that

$$\text{Tok}(xy) = \text{Merge}(xy) = P \circ \text{Merge}(x_{\text{out}+1} \dots x_i y) \succeq P\tau.$$

To implement the above algorithm efficiently, we need a few additional data structures. First, we need to compute θ, the next token in $\text{Tok}(x)$, denoted by $\phi(x_{\text{out}+1} \dots x_{\text{last}(q)})$ where $x_{\text{last}(q-1)+1} \dots x_{\text{last}(q)} = z_q$ is the pretoken in $\text{Split}(x)$ with $\text{last}(q-1) \leqslant \text{out} < \text{last}(q)$. Second, we need to compute $\tau := \phi(x_{\text{out}+1} \dots x_i)$, the next token in the event that a future continuation extends the current pretoken match. To support these operations efficiently, we implement two useful data structures. Note that both data structures support pushing string characters and $\|$ to denote the beginning of a new pretoken (pushing $\|$ does not contribute to the length of the maintained string).

The first data structure incBPE allows us to compute θ efficiently.

Lemma 3.3 (Incremental BPE). *Given a BPE merge list M with longest token length $\mathfrak{t}$, there is a data structure* incBPE *that maintains implicitly an append-only history string h, an index $0 \leqslant I \leqslant |h|$, and explicitly string $w = h_{>I}$ under the following operations:*

1. *(Push) append c to $h \leftarrow hc$. Note that we can push $\|$ to denote a pretoken split.*
2. *(Pop) if $I < |h|$, increment I by 1.*
3. *(First Token) if $\text{Merge}(h_{\leqslant I}) \prec \text{Merge}(h)$, compute $\phi(w)$. Otherwise return $\perp$.*

The data structure has $O(\mathfrak{mt})$ preprocessing, space $O(\mathfrak{mt}+|w|)$, worst-case update time $O(\log^2 \mathfrak{t} + \log \mathfrak{m})$ and query time $O(\log \mathfrak{t} + \log |w|)$.

The second data structure bpeQ allows us to compute τ efficiently.

Lemma 3.4 (BPE Queue). *Given a BPE merge list M, there is a data structure* bpeQ *supporting the following operations on a string w:*

1. *(Push) Append a character c to w i.e. update $w \leftarrow wc$.*
2. *(Pop) Pop w_1, i.e. return w_1 and update $w \leftarrow w_2 \dots w_\ell$.*
3. *(First Token) If $|w| \geqslant 3\mathfrak{mt}$, then compute $\phi(w)$. Otherwise, return $\perp$.*

Furthermore, the data structure has $O(\mathfrak{mt})$ preprocessing, $O(\mathfrak{mt}+|w|)$ space, and supports updates and queries in amortized time $O(\log^2 \mathfrak{t})$.

Furthermore, Lemma C.4 gives a data structure cutQ that allows us to efficiently compute $\text{last}(q)$ satisfying $\text{last}(q-1) \leqslant \text{out} < \text{last}(q)$. We note that $\text{last}(q)$ is monotonically increasing: since we are always looking for the longest match, additional characters can only extend a pretoken.

We now sketch our final algorithm on input x_i (see Theorem 1.7 for the full algorithm and Figure 1 for an illustration of the data structures in our algorithm). As before, we will assume $\delta(\text{start}, x) \notin \text{dead}$. Initialize data structures incBPE and bpeQ. Denote data structure queries by incBPE.ϕ and bpeQ.ϕ.

Sketch of **`OptimalDelayTokenization`**(x_i)

1. Push x_i onto bpeQ. Ensure incBPE maintains $x_{\text{out}+1} \dots x_{\text{last}(q)}$.
2. While incBPE.ϕ = bpeQ.$\phi \neq \perp$ (i.e. first token queries agree), output incBPE.ϕ and pop incBPE.ϕ from incBPE and bpeQ.

3. If incBPE $= \epsilon$ and out $< i$: add $q \leftarrow q+1$, push $\| \, x_{\text{last}(q-1)+1} \dots x_{\text{last}(q)}$ to incBPE, and return to Step 2.
   (When incBPE is empty, we have processed the current pretoken z_q, so increment q and proceed to the next pretoken.)

Our algorithm maintains the following invariants (see Figure 1):

1. if out $< i$, q satisfies $\text{last}(q-1) \leqslant \text{out} < \text{last}(q)$.
2. incBPE maintains $x_{\text{out}+1} \dots x_{\text{last}(q)}$.
3. bpeQ maintains $x_{\text{out}+1} \dots x_i$.

Observe that whenever $i - \text{out} > 3\mathfrak{mt}$, incBPE is guaranteed to produce the correct output. We only need **push** operations in Step 1, since $\text{last}(q)$ is monotonically increasing. We will show in the proof of Theorem 1.7 that every query to incBPE is valid. Following the above discussion, the displayed algorithm obtains a delay bound of $O(\mathfrak{mt})$. The running time follows from the near linear running time of both data structures.

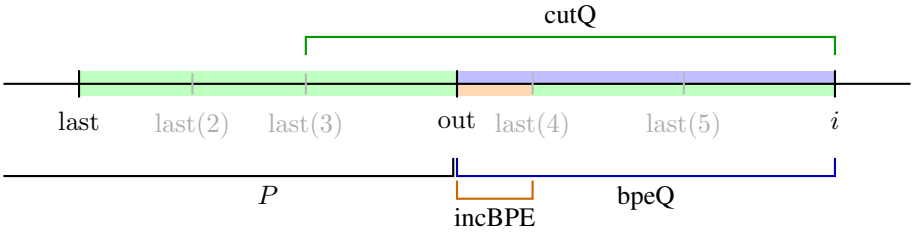


Figure 1: Illustration of $O(\mathfrak{mt})$ delay algorithm. Characters up to out are output in P. $\text{last}, \text{last}(2), \dots, \text{last}(5), i$ are endpoints of pretokens if the current input $x_{\leqslant i}$ terminates. Square brackets describe the substring of the input maintained by each data structure. Here, we have $q = 4$.

Hardness of Exact Online Tokenization Our algorithm does not achieve exact online tokenization (zero delay), and we give a lower bound ruling out any polynomial algorithm unless $\mathbf{P} = \mathbf{NP}$. We will reduce 3-SAT to online tokenization. Consider 3-SAT on d variables. First, we construct an alphabet Σ, regular expression R, and merge list M depending only on d. We will ensure that the alphabet size, the size of a DFA recognizing the regular expression, and the length of the merge list all have size $\text{poly}(d)$. Then, we give a reduction from any d-variable 3-SAT instance ψ to input strings z_ψ of length $\text{poly}(d)$ such that there is a specific alphabet character $\alpha \in \Sigma$ that appears in $\text{Tok}_\preceq(x)$ if and only if ψ is unsatisfiable (see Theorem 1.1 for details and proof with delay).

The high level idea of our approach is as follows. For each of the $O(d^3)$ possible width 3 clauses $c \in \mathcal{C}$, define a symbol ℓ_c. Define $\Sigma_\mathcal{C} = \{\ell_c\}_{c \in \mathcal{C}}$. Define the alphabet

$$\Sigma = \left\{a_1^0, a_1^1, b_1, \dots, a_d^0, a_d^1, b_d\right\} \cup \{\alpha, \beta, \gamma, \delta\} \cup \Sigma_\mathcal{C}.$$

For every assignment $x \in \{0,1\}^d$, we encode x as $E(x) = a_1^{x_1} b_1 \dots a_d^{x_d} b_d$. For a clause c, let $\mathcal{E}_c = \{E(x) \text{ s.t. } c(x) = 0\}$ denote the encodings of all assignments that falsify c and build the regular expression $R_c = \ell_c \Sigma_\mathcal{C}^* \alpha \beta \mathcal{E}_c \gamma$.

To define the tokenization, fix $S^* = \emptyset$ and $R = R_\mathcal{C} \mid \alpha\beta\mathcal{E}\gamma\delta \mid \Sigma$ where $R_\mathcal{C}$ is the alternation of all R_c, $\mathcal{E}$ is the alternation of all encodings, and Σ is the alternation of all characters added to ensure R is total. The merge rules will be chosen so that within any pretoken, α is merged if and only if the suffix of the pretoken beginning with α has even length. Observe that this occurs if and only if α occurs in the pretoken $\alpha\beta\mathcal{E}\gamma\delta$ (since $\Sigma, R_\mathcal{C}$ both have odd length suffixes beginning with α). Then, given a 3-SAT instance $\psi = c_1 \wedge \dots \wedge c_k$, construct the input $z = \ell_{c_1} \dots \ell_{c_k} \alpha$.

If ψ is satisfiable (say by x^*), we will argue that the continuation $z\beta E(x^*)\gamma\delta$ corresponding to the satisfying assignment x^* has no match with any $R_\mathcal{C}$, and thus matches $\alpha\beta\mathcal{E}\gamma\delta$, in which case α is merged and does not appear in $\text{Tok}_\preceq(z)$. On the other hand, when ψ is unsatisfiable, we argue that any match involving α matches $R_\mathcal{C}$ corresponding to some unsatisfied clause, so that α is never merged and always appears in $\text{Tok}_\preceq(z)$.

AI USE STATEMENT

In this work, we used generative AI tools for providing critical ingredients for proving mathematical claims, formulating mathematical claims, assisting in the writing of proofs, designing or providing feedback on research methodology or experiments, implementing methods, and cleaning and reformatting datasets. We have not used generative AI tools for help developing theoretical models or conceptual frameworks, proposing or refining hypotheses, supporting qualitative and thematic data analysis, or interpreting results. Generating synthetic data sets and assisting with translation are not applicable to this work.

Additionally, we used generative AI tools for creating or modifying scientific figures or images, suggesting experimental parameters, creating or editing software code, creation of artifacts, summarizing or analyzing existing literature, brainstorming, sourcing/searching for information, editing a research paper to improve readability, and identifying relevant literature. We have reviewed all AI-assisted work. On a theoretical side, generative AI was particularly useful in designing the reduction behind Theorem 1.1. On the empirical side, essentially all of our code was implemented by AI. The code was reviewed and manually verified. For our DFA algorithm, the main empirical contribution of this work, the code was written by AI but we manually prove the correctness of the key optimization (DFA simplification, see Section D). We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The authors have made significant efforts to ensure that the work is reproducible. The provided supplementary material contains essentially all of the code used by the authors and supports out of the box end-to-end reproducibility of the evaluated benchmarks. In particular, the README of the provided supplementary material details (1) how to run the implemented tokenizer, (2) how to install additional tooling (e.g. downloading data) required to run all necessary benchmarking, and (3) how to run the benchmarks implemented in the paper. Furthermore, during the experimentation phase, the authors tested the tokenizer on both Apple M1 chip and the described hardware in the reported experiments, with similar performance observed on both types of hardware available to the authors. All datasets used in the experiments are described in the paper, and furthermore our code provides a simple script to prepare identical data. All theoretical claims are proven in the paper.

REFERENCES

Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975. doi: 10.1145/360825.360855.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson/Addison-Wesley, 2nd edition, 2006.

Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 20–29, 1996.

Michael A Bender, Richard Cole, Erik D Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *European symposium on algorithms*, pp. 152–164. Springer, 2002.

Martin Berglund and Brink van der Merwe. Formalizing bpe tokenization. *arXiv preprint arXiv:2309.08715*, 2023.

BurntSushi. memchr 2.8.2. https://docs.rs/memchr/latest/memchr/, 2026. Docs.rs documentation. Accessed 26 June 2026.

Davide Della Giustina, Nicola Prezza, and Rossano Venturini. A new linear-time algorithm for centroid decomposition. In *International Symposium on String Processing and Information Retrieval*, pp. 274–282. Springer, 2019.

Paul Dietz and Daniel Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pp. 365–372, 1987.

Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA)*, pp. 605–617, 2003.

Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Proceedings of the 2007 International Conference on Analysis of Algorithms (AofA)*, volume AH of *DMTCS Proceedings*, pp. 127–146, 2007.

Philip Gage. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38, 1994.

HuggingFace. Tokenizers: Fast state-of-the-art tokenizers optimized for research and production. https://github.com/huggingface/tokenizers, 2026.

Shenghu Jiang and Ruihao Gong. Incremental BPE tokenization. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2605.30813.

Angela W Li, Yudi Yang, and Konstantinos Mamouras. Static analysis for efficient streaming tokenization. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 1880–1896, 2026.

Konstantinos Mamouras, Angela W Li, and Yudi Yang. An efficient algorithm for streaming bpe tokenization. *Proceedings of the ACM on Programming Languages*, 10(PLDI):2085–2108, 2026.

J. Misra and David Gries. Finding repeated elements. *Science of Computer Programming*, 2(2): 143–152, 1982.

Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

J. Ian Munro and Mike S. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12(3):315–323, 1980.

OpenAI. tiktoken. https://github.com/openai/tiktoken, 2026. Accessed: 2026-06-15.

OpenAI. tiktoken: src/lib.rs. https://github.com/openai/tiktoken/blob/main/src/lib.rs, 2026. Accessed: 2026-07-07.

Oracle. Quantifiers. https://docs.oracle.com/javase/tutorial/essential/regex/quant.html, 2024. Accessed: 2026-06-29.

Regular-Expressions.info. Possessive quantifiers. https://www.regular-expressions.info/possessive.html, 2025. Accessed: 2026-06-29.

Thomas Reps. "maximal-munch" tokenization in linear time. *ACM Transactions on Programming Languages and Systems*, 20(2):259–273, 1998. doi: 10.1145/276393.276394.

Marcel Rød. Gigatoken: Simd and cache hierarchies for 1000x faster byte-pair encoding tokenization on modern cpus, 2026. URL https://github.com/marcelroed/gigatoken.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: long papers)*, pp. 1715–1725, 2016.

Together Computer. RedPajama-Data-1T dataset. Hugging Face dataset repository, April 2023. URL https://huggingface.co/datasets/togethercomputer/RedPajama-Data-1T.

Wikimedia Foundation. Wikipedia dataset. Hugging Face dataset repository, November 2023. URL https://huggingface.co/datasets/wikimedia/wikipedia.

A PRELIMINARIES

Let Σ denote an alphabet, Σ^k the set of length k strings over Σ, and $\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$ the set of (any length) strings over Σ. Let $[n] = \{1, \ldots, n\}$. Let ϵ denote the empty string and $\circ$ concatenation. Let $|x|$ denote the length of a string. Let $x[i:j]$ or $x_i \ldots x_j$ denote the substring of x starting at the i-th index and ending at the j-th index (inclusive and indexed at 1). We let $x_{\leqslant i} = x_1 \ldots x_i$ and $x_{\geqslant i} = x_i \ldots x_n$ where $n = |x|$. $x_{<i}$ and $x_{>i}$ are defined analogously. Let $x[:-k]$ denote the prefix of x obtained by deleting the last k characters and $x[-k:]$ the suffix containing the last k characters. We say $x \preceq y$ if x is a prefix of y. We use $y \succeq x$ to mean that y contains x as a prefix, not that y is a suffix of x. We use $a \xleftarrow{+} b$ to denote $a \leftarrow a + b$.

Regular Expressions The regular languages over an alphabet Σ consist of the smallest collection $\mathcal{R} = \mathcal{R}(\Sigma)$ of languages $L \subseteq \Sigma^*$ that satisfy the following rules:

1. $\emptyset \in \mathcal{R}$, $\{\epsilon\} \in \mathcal{R}$, and $\{\sigma\} \in \mathcal{R}$ for all $\sigma \in \Sigma$.
2. (Concatenation) If $L_1, L_2 \in \mathcal{R}$, then $L_1 \circ L_2 = \{x_1 \circ x_2 \mid x_1 \in L_1, x_2 \in L_2\} \in \mathcal{R}$.
3. (Alternation) If $L_1, L_2 \in \mathcal{R}$, then $L_1 \mid L_2 = L_1 \cup L_2 \in \mathcal{R}$.
4. (Star) If $L \in \mathcal{R}$, then $L^* \in \mathcal{R}$.

When convenient, we use $L_1 L_2$ to denote $L_1 \circ L_2$. A regular expression R is *total* if every non-empty $x \in \Sigma^*$ has a non-empty prefix matched by R (possibly including x itself). Note that this implies $\Sigma \subseteq L(R)$. Throughout this work, we consider only total regular expressions.

Regular languages are closed under complement, union and finite intersection. For our theoretical results, a match of regular expression R in x is the longest prefix of x that matches R, or equivalently the longest prefix of x in $L(R)$. Since we assume R is total, this is a non-empty prefix.

For a total regular expression R and non-empty input x, define the longest match of R in x, denoted $F(x; R)$ as $x_{\leqslant J(x;R)}$ where $J(x; R) = \max\{j > 0 \text{ s.t. } x_{\leqslant j} \in L(R)\}$. When the context is clear, we simply write $F(x), J(x)$. We note that for any regular expression R, there is a non-deterministic finite automata (NFA) that recognizes exactly $L(R)$ of size $O(|R|)$ (see e.g. Aho et al. (2006)). In Section D, we will separately discuss examples of regular expressions used in practice, which use regular expressions with priority rules and lookahead.

Given a DFA or an NFA $\mathcal{A}$, let $\mathcal{Q}$ be its states. Let $\mathcal{P} \subseteq \mathcal{Q}$ be the accepting states, and $\mathrm{dead} \subseteq \mathcal{Q}$ be the set of states that have no *non-trivial* path to a state in $\mathcal{P}$. Note that a dead state can be a $\mathcal{P}$ state, since we do not count the trivial transition on the empty string ϵ. For our purposes of simulating a regular expression with a DFA, if a state is dead and in $\mathcal{P}$, it is a maximal match. Let start denote the start state. For a state q and word w, $\delta(q, w)$ denotes the state computed if we start on q and read w. $\mathrm{last}(q, w)$ denotes the maximum i where $\delta(q, w_{\leqslant i}) \in \mathcal{P}$, or $\perp$ if no such i exists. For an NFA, we let Q be a set of states and $\delta(Q, w)$ be the set of states reachable if starting at any $q \in Q$ and reading w and $\mathrm{last}(Q, w)$ the maximum i where $\delta(Q, w_{\leqslant i}) \cap \mathcal{P} \neq \emptyset$, or $\perp$ if no such i exists.

For a merge list M, we use merge list and BPE encoding interchangeably. For a given token τ, let $\mathrm{raw}(\tau)$ denote the expansion of τ, i.e. the raw string x such that $\mathrm{Merge}(x) = \tau$. For a sequence of tokens $\tau_1 \ldots \tau_k$, let $\mathrm{raw}(\tau_1 \ldots \tau_k)$ denote the concatenation of $\mathrm{raw}(\tau_1), \ldots, \mathrm{raw}(\tau_k)$. Let the length of a token or a sequence of tokens be defined $|\tau| = |\mathrm{raw}(\tau)|$.

B ONLINE TOKENIZATION

We begin by defining tokenization in the standard offline batch setting.

Definition B.1 (Special Tokens)**.** *Let Σ be a finite alphabet and $S^* \subset \Sigma^*$ a finite set of special tokens. For each string $x \in S^*$, $s^*(x)$ is a single special character corresponding to x. Given a string $x \in \Sigma^*$, the segmenting of x by S^* is given by*

$$\mathrm{Seg}(x; S^*) = \begin{cases} \epsilon & \text{if } x = \epsilon \\ \parallel s^*(x_{\leqslant j}) \parallel \mathrm{Seg}(x_{>j}; S^*) & \text{if } j = \max\{i \text{ s.t. } x_{\leqslant i} \in S^*\} \geqslant 1 \\ x_1 \circ \mathrm{Seg}(x_{>1}; S^*) & \text{o/w} \end{cases}.$$

The output consists of $\|$ *symbols with strings between. Each string is called a special token if it is* $s^*(x)$ *for* $x \in S^*$*, and a segment otherwise. Let* $\ell_1(S^*) = \sum_{x \in S^*} |x|$ *and* $\ell_\infty(S^*) = \max_{x \in S^*} |x|$*.*

Definition B.2 (Pretokenization)**.** *Let* Σ *be a finite alphabet and* R *a total regular expression over* Σ*. Then, the pretokenization of* x *with respect to* R *is*

$$\mathrm{Split}(x; R) = \begin{cases} \epsilon & x = \epsilon \\ F(x; R) \parallel \mathrm{Split}(x_{>|F(x;R)|}; R) & o/w \end{cases}.$$

Given a list of segments and special tokens $y = x_1 \parallel \ldots \parallel x_m$*, define* $\mathrm{Split}(y; R)$ *to be* Split *applied to each segment with the results concatenated. Note that* Split *is not applied to special tokens and trailing* $\|$ *is removed. The output of* Split *on a single segment consists of* $\|$ *symbols with strings between; we call these strings pretokens. Thus, on a list of segments and special tokens,* Split *produces a list of pretokens and special tokens.*

We formally define bounded lookback, a quantity which will characterize both the delay and space usage of our algorithms.

Definition B.3 (Lookback)**.** *A regular language* L *has* $\mathfrak{b}$*-lookback if for any* $x \in L$ *with* $|x| > \mathfrak{b}$*,*

$$\max\{|y| \text{ s.t. } y \prec x, y \in L\} \geqslant |x| - \mathfrak{b}$$

with the convention $\max \emptyset = -\infty$*. That is, every word in the regular language of length greater than* $\mathfrak{b}$ *has a strict prefix in the language that is shorter by at most* $\mathfrak{b}$*. We let* $\mathfrak{b}(L)$ *denote the minimum* $\mathfrak{b}$ *such that* L *has* $\mathfrak{b}$*-lookback.* L *has unbounded lookback if* $\mathfrak{b}(L) > c$ *for any finite* c*.*

Lookback ensures that if we do not find an accepting prefix within $\mathfrak{b}$ characters, the current match (if any) cannot extend by reading more characters of the input.

Definition B.4 (Byte-Pair Encoding)**.** *Let* $0 < n_0 < n$ *be positive integers. Let* $\Sigma = [n_0]$ *be a finite alphabet. Let* M *be a trained Byte-Pair Encoding (BPE) list of merges with corresponding dictionary* $D = n_0 + [M]$ *(i.e. a list of ordered pairs* $(m_1, m_2) \mapsto d$ *for* $d \in D$ *and* $0 < m_1, m_2 < d$*). Let* $\Sigma_D = \Sigma \cup D = [n]$ *denote the full BPE dictionary. We assume the given merge list* M *is canonical:* $\mathrm{Merge}(\mathrm{raw}(\tau); M) = \tau$ *for every dictionary token* $\tau \in \Sigma_D$*, as in Jiang & Gong (2026, Section 3.3).*

Let $m = (m_1, m_2) \mapsto d$ *be a merge. Given a string* $x \in [d-1]^*$*, the BPE of* x *with respect to* m *is*

$$\mathrm{Merge}(x; m) = \begin{cases} x & |x| \leqslant 1 \\ d \circ \mathrm{Merge}([x_3, \ldots]; m) & |x| \geqslant 2 \text{ and } (x_1, x_2) = (m_1, m_2) \\ x_1 \circ \mathrm{Merge}([x_2, \ldots]; m) & \text{otherwise} \end{cases}.$$

For a string $x \in \Sigma^*$*, the BPE of* x *with respect to* M *is*

$$\mathrm{Merge}(x; M) = \begin{cases} x & M = \emptyset \\ \mathrm{Merge}(\mathrm{Merge}(x; \mu_1); M \setminus \mu_1) & M = (\mu_1, \ldots) \end{cases}.$$

Given a list of pretokens and special tokens $y = x_1 \parallel \ldots \parallel x_p$*, we define* $\mathrm{Merge}(y; M)$ *to be* Merge *applied to each pretoken with the results concatenated. Note that on a special token,* Merge *outputs the special token identically. We call each element in the output list (i.e. number) a token.*

We note that a BPE encoding is determined by its list of merges M alone. For succinctness, we use M alone to denote a BPE. We define the longest token.

Definition B.5 (Longest Token)**.** *Define* $\mathfrak{t}(M) = \max\{|x| \text{ s.t. } |\mathrm{Merge}(x; M)| = 1\}$ *to be the length of the longest token. When the context is clear, we write* $\mathfrak{t} = \mathfrak{t}(M)$*.*

Throughout, we use the $\|$ symbol to denote hard boundaries in the input string through which no merges can occur. The tokenization problem is then a composition of the above three subroutines.

Definition B.6 (Tokenization)**.** *We define the Tokenization problem as follows. Let* Σ *be a finite alphabet. Let* $S^* \subset \Sigma^*$ *be a set of special tokens,* R *a total regular expression over* Σ*, and* M *a BPE encoding. Given a string* $x \in \Sigma^*$*, the tokenization of* x *is*

$$\mathrm{Tok}(x; S^*, R, M) = \mathrm{Merge}(\mathrm{Split}(\mathrm{Seg}(x; S^*); R); M).$$

In running times, we use R to denote $|R|$ and $\mathfrak{m}$ to denote $|M|$. We now define tokenization in the online setting. We begin with the notion of a stable output prefix.

Definition B.7 (Stable)**.** *Let $S^* \subset \Sigma^*$ be a set of special tokens, R a regular expression over Σ, and M a BPE encoding. A list of tokens z is* stable *with respect to a string $x \in \Sigma^*$ if z is a prefix of $\mathrm{Tok}(x \circ y; S^*, R, M)$ for every string $y \in \Sigma^*$. That is, Tok outputs z as a prefix for every continuation of x. Let $\mathcal{S}(x; S^*, R, M)$ denote the set of stable prefixes with respect to x. When the context is clear, we use $\mathcal{S}(x)$ to denote the set of stable prefixes.*

We show that any two stable prefixes are comparable, and therefore the maximal/maximum stable prefix is unique. For two lists z_1, z_2, we define the partial order $z_1 \preceq z_2$ if z_1 is a prefix of z_2.

Claim B.8. *If z_1, z_2 are stable with respect to x, then $z_1 \preceq z_2$ or $z_2 \preceq z_1$. In particular, $\mathcal{S}(x; S^*, R, M)$ has a unique greatest element z^* under $\preceq$, and every stable prefix is a prefix of z^*.*

Proof. We begin with the following fact: if $z \preceq L$ and $u \preceq L$, then $z \preceq u$ or $u \preceq z$. Assume without loss of generality $|z| \leq |u|$. Since both are prefixes of L and $|z| \leq |u| \leq |L|$, we have $z \preceq u$.

Define $\mathrm{Tok}(x) := \mathrm{Tok}(x; S^*, R, M)$. Then, since $z_1, z_2 \in \mathcal{S}(x)$, we have $z_1, z_2 \preceq \mathrm{Tok}(x)$ so that either $z_1 \preceq z_2$ or $z_2 \preceq z_1$. Thus, $\preceq$ is in fact a total order, so that there exists a unique greatest element z^*. □

Thus, we may define the following Online Tokenization problem. A string is incrementally updated if it is updated only by appending new characters. An algorithm outputs y incrementally if it outputs by appending to the previous output.

Definition B.9 (Online Tokenization)**.** *Let $S^* \subset \Sigma^*$ be a set of special tokens, R a total regular expression over Σ, and M a BPE encoding. Online tokenization, given an incrementally updated string x, incrementally outputs $\mathrm{Tok}_\preceq(x) = \max_\preceq \mathcal{S}(x; S^*, R, M)$. An algorithm for online tokenization with delay k incrementally outputs z such that*

$$\mathrm{Tok}_\preceq(x[:-k]) \preceq z \preceq \mathrm{Tok}_\preceq(x).$$

Exact online tokenization is the problem of online tokenization with delay 0.

We argue that given an incrementally updated string x, $\mathrm{Tok}(x)$ also updates incrementally, and thus (at least in principle) an algorithm could maintain the output efficiently.

Claim B.10. *If $x \preceq y$, then $\mathcal{S}(x) \subseteq \mathcal{S}(y)$. Consequently, $(\max_\preceq \mathcal{S}(x)) \preceq (\max_\preceq \mathcal{S}(y))$.*

Proof. Let $z \in \mathcal{S}(x)$ be a stable prefix. Then, $z \preceq \mathrm{Tok}(xx')$ for any $x' \in \Sigma^*$. Write $y = xa$ for some $a \in \Sigma^*$. Then, $z \preceq \mathrm{Tok}(xay')$ for any $y' \in \Sigma^*$ since $ay' \in \Sigma^*$ is a continuation of x. Hence, $z \in \mathcal{S}(y)$, as desired. □

B.1 Exact Online Tokenization

We begin with a conditional lower bound showing that it is likely hard to design an efficient algorithm for online tokenization without delay.

Theorem 1.1 (Hardness of Online Tokenization)**.** *Unless $\mathbf{P} = \mathbf{NP}$, there is no algorithm for exact (zero-delay) online tokenization with $\mathrm{poly}(\ell_1(S^*), |R|, \mathfrak{m}, \mathfrak{t})$ preprocessing and total time $\mathrm{poly}(n, \ell_1(S^*), |R|, \mathfrak{m}, \mathfrak{t})$, even when $S^* = \emptyset$. Furthermore, the lower bounds hold for algorithms with $o(\mathfrak{m}\mathfrak{t})$ delay.*

Proof of Theorem 1.1. We begin with the lower bound against algorithms for exact online tokenization with zero delay. Suppose we have some algorithm running in time $\mathrm{poly}(n, \ell_1(S^*), R, \mathfrak{m})$. We will construct a reduction from 3-SAT on d variables. We construct the following tokenization instance. There are $O(d^3)$ possible clauses of width 3 on d variables. Let $\mathcal{C}$ be the set of all such clauses. Define $\Sigma_\mathcal{C} = \{\ell_c\}_{c \in \mathcal{C}}$. The alphabet consists of

$$\Sigma = \left\{a_1^0, a_1^1, b_1, \ldots, a_d^0, a_d^1, b_d\right\} \cup \{\alpha, \beta, \gamma, \delta\} \cup \Sigma_\mathcal{C}.$$

Let $S^* = \emptyset$. For every bit-string $x \in \{0,1\}^d$, define

$$E(x) = a_1^{x_1} b_1 \dots a_d^{x_d} b_d.$$

For each clause c, let $\mathcal{E}_c = \{E(x) \text{ s.t. } c(x) = 0\}$ be the set of assignments on d variables that falsify c. Observe that $\mathcal{E}_c$ can be recognized by a DFA of size $O(d)$: there is 1 assignment to the three variables in c that falsifies c, and the remaining variables are unconstrained. Similarly, $\mathcal{E} = \left\{E(x) \text{ s.t. } x \in \{0,1\}^d\right\}$ is the set of all assignments and can be recognized by a DFA of size $O(d)$. Consider the regular expression

$$R_c = \ell_c \Sigma_{\mathcal{C}}^* \alpha\beta\mathcal{E}_c\gamma$$

where we abuse notation and use $\mathcal{E}_c$ to denote the regular expression that recognizes $\mathcal{E}_c$. Let $R_{\mathcal{C}}$ recognize the alternation of all R_c for $c \in \mathcal{C}$. Then, the full regular expression is

$$R = R_{\mathcal{C}} \mid \alpha\beta\mathcal{E}\gamma\delta \mid \Sigma.$$

Note that Σ is the alternation of all single characters, and ensures that R is total. Observe also that all rules that match strings of length > 1 have distinct starting characters, so that any longest match automatically chooses the match of highest priority. In particular, longest match already prioritizes matches of length > 1 over single character matches. Thus, R can be recognized by a DFA of size $O(d^4)$.

The merge rules are described below in order of priority:

$$\begin{aligned}
&(\gamma, \delta) \mapsto \Gamma \\
&(b_d, \gamma) \mapsto B_d \\
&(a_d^0, b_d) \mapsto A_d^0 && (a_d^1, b_d) \mapsto A_d^1 \\
&(b_{d-1}, a_d^0) \mapsto B_{d-1}^0 && (b_{d-1}, a_d^1) \mapsto B_{d-1}^1 \\
&(a_{d-1}^0, b_{d-1}) \mapsto A_{d-1}^0 && (a_{d-1}^1, b_{d-1}) \mapsto A_{d-1}^1 \\
&\vdots \\
&(b_1, a_2^0) \mapsto B_1^0 && (b_1, a_2^1) \mapsto B_1^1 \\
&(a_1^0, b_1) \mapsto A_1^0 && (a_1^1, b_1) \mapsto A_1^1 \\
&(\beta, a_1^0) \mapsto \Delta^0 && (\beta, a_1^1) \mapsto \Delta^1 \\
&(\alpha, \beta) \mapsto \Phi.
\end{aligned}$$

Note that $\mathfrak{m} = 4d + 3$. Thus, in the preprocessing phase, the assumed algorithm for online pretokenization given S^*, R, M requires $\text{poly}(d)$ time.

Consider now a 3-CNF formula $\psi = c_1 \wedge \dots \wedge c_k$ on d variables and $k = \text{poly}(d)$ clauses. Construct the input

$$x_\psi = \ell_{c_1} \ell_{c_2} \dots \ell_{c_k} \alpha.$$

Note that $|x_\psi| = \text{poly}(d)$. We claim that $\text{Tok}_{\preceq}(x_\psi)$ decides whether ψ is satisfiable.

Claim B.11. *For any 3-CNF formula ψ, we have the following:*

1. *(Completeness) If $\psi \in$ SAT, then $\text{Tok}_{\preceq}(x_\psi) = \ell_{c_1} \dots \ell_{c_k}$.*
2. *(Soundness) If $\psi \notin$ SAT, then $\text{Tok}_{\preceq}(x_\psi) = \ell_{c_1} \dots \ell_{c_k} \alpha$.*

Thus, if an algorithm computed online tokenization (without delay) in $\text{poly}(n, \ell_1(S^*), R, \mathfrak{m})$ time, we would have an algorithm that determines satisfiability of $\text{poly}(d)$ size formulas in $\text{poly}(d)$ time. In particular, since 3-SAT is **NP**-complete, we would have $\mathbf{P} = \mathbf{NP}$, a contradiction. It remains to prove Claim B.11.

Proof of Claim B.11. First, we argue $\text{Tok}_{\preceq}(\ell_{c_1} \ell_{c_2} \dots \ell_{c_k}) = \ell_{c_1} \ell_{c_2} \dots \ell_{c_k}$. This follows simply as ℓ_c are not involved in any merge rules, and thus are output regardless of the choice of continuation. Next, note that $\text{Tok}(x_\psi) = x_\psi$ since every pretoken that contains more than one character ends with

either γ or δ, which does not occur in x_ψ. Thus, Split splits x_ψ into single characters, and no merges occur.

Suppose $\psi \in \text{SAT}$ is satisfiable. Let $z \in \{0,1\}^d$ be a satisfying assignment so $\psi(z) = 1$. Consider the continuation $\beta E(z)\gamma\delta$. Then, we claim that no substring of $x_\psi\beta E(z)\gamma\delta$ matches R_c for any $c \in \mathcal{C}$. Suppose for contradiction some substring of $x_\psi\beta E(z)\gamma\delta$ matches R_c for some c. Since there is only one α, β, γ in the input, the match must take the form $\ell_{c_i} \dots \ell_{c_k}\alpha\beta E(z)\gamma$ for some $i \in [k]$. We thus have $E(z) \in \mathcal{E}_{c_i}$ for some clause $c_i \in \psi$, a contradiction since z is a satisfying assignment.

Since $R_{\mathcal{C}}$ does not match any substring of $x_\psi\beta E(z)\gamma\delta$, we have

$$\text{Tok}(x_\psi\beta E(z)\gamma\delta) = \text{Merge}(\ell_{c_1} \parallel \dots \parallel \ell_{c_k} \parallel \alpha\beta E(z)\gamma\delta) = \ell_{c_1}\ell_{c_2}\dots\ell_{c_k}\Phi A_1^{z_1}\dots A_d^{z_d}\Gamma.$$

Thus, $\text{Tok}_{\preceq}(x_\psi) \preceq \ell_{c_1}\ell_{c_2}\dots\ell_{c_k}$, establishing equality.

On the other hand, suppose $\psi \notin \text{SAT}$ is unsatisfiable. Our goal is to show that for any continuation $w \in \Sigma^*$, $x_\psi \preceq \text{Tok}(x_\psi w)$. Observe that α appears in a pretoken only when followed by a β, a string in $\mathcal{E}$ and a γ. Thus, it suffices to only consider continuations of the form $\beta E(z)\gamma w$ where $z \in \{0,1\}^d$ and $w \in \Sigma^*$. Otherwise, no rule R_c matches a pretoken of length > 1 containing α, so α is emitted as its own token. Since ψ is not satisfiable, we have $\psi(z) = 0$, or $c_i(z) = 0$ for some $i \in [k]$. Take the smallest such i with $E(z) \in \mathcal{E}_{c_i}$ and since R_{c_i} gets matched first, we have

$$\begin{aligned}\text{Tok}(x_\psi\beta E(z)\gamma w) &= \text{Merge}(\ell_{c_1} \parallel \dots \parallel \ell_{c_{i-1}} \parallel \ell_{c_i}\dots\ell_{c_k}\alpha\beta E(z)\gamma) \circ \text{Tok}(w)\\ &= \ell_{c_1}\dots\ell_{c_{i-1}} \circ \text{Merge}(\ell_{c_i}\dots\ell_{c_k}\alpha\beta E(z)\gamma) \circ \text{Tok}(w)\\ &= \ell_{c_1}\dots\ell_{c_k}\alpha\Delta^{z_1}B_1^{z_2}\dots B_{d-1}^{z_d}B_d \circ \text{Tok}(w).\end{aligned}$$

Thus, we have $\text{Tok}_{\preceq}(x_\psi) = \ell_{c_1}\dots\ell_{c_k}\alpha$, as desired. □

Extension to $o(\mathfrak{mt})$ Delay. Now, we describe how to modify the lower bound against algorithms with small delay. We add a character ζ to the alphabet so that

$$\Sigma = \left\{a_1^0, a_1^1, b_1, \dots, a_d^0, a_d^1, b_d\right\} \cup \{\alpha, \beta, \gamma, \delta, \zeta\} \cup \Sigma_{\mathcal{C}}.$$

The special token set $S^* = \emptyset$ is identical. Note that in the previous construction, $\mathfrak{t} = \mathfrak{t}(M) = 2$. The merge rules now begin with

$$(\zeta, \zeta) \mapsto \zeta^{(1)}, (\zeta^{(1)}, \zeta^{(1)}) \mapsto \zeta^{(2)}, \dots, (\zeta^{(p-1)}, \zeta^{(p-1)}) \mapsto \zeta^{(p)}.$$

Then, we have for every $1 \leqslant i \leqslant d$, the merge rule $(b_i, \zeta^{(p)}) \mapsto \beta_i$. The remaining merge rules are as essentially identical, replacing b_i with β_i.

$$\begin{aligned}&(\gamma, \delta) \mapsto \Gamma\\ &(\beta_d, \gamma) \mapsto B_d\\ &(a_d^0, \beta_d) \mapsto A_d^0 \qquad (a_d^1, \beta_d) \mapsto A_d^1\\ &(\beta_{d-1}, a_d^0) \mapsto B_{d-1}^0 \qquad (\beta_{d-1}, a_d^1) \mapsto B_{d-1}^1\\ &(a_{d-1}^0, \beta_{d-1}) \mapsto A_{d-1}^0 \qquad (a_{d-1}^1, \beta_{d-1}) \mapsto A_{d-1}^1\\ &\vdots\\ &(\beta_1, a_2^0) \mapsto B_1^0 \qquad (\beta_1, a_2^1) \mapsto B_1^1\\ &(a_1^0, \beta_1) \mapsto A_1^0 \qquad (a_1^1, \beta_1) \mapsto A_1^1\\ &(\beta, a_1^0) \mapsto \Delta^0 \qquad (\beta, a_1^1) \mapsto \Delta^1\\ &(\alpha, \beta) \mapsto \Phi.\end{aligned}$$

Observe that now as long as $p \leqslant d$, we have $\mathfrak{m} = p + d + 4d + 3 = p + 5d + 3$. For example, if we choose $p = \lceil \log d \rceil$, then $\mathfrak{t} \leqslant 2^p + 2 \leqslant 2d + 2$. As argued above, we have $|\Sigma| = O(d^3)$, $\mathfrak{m} = O(d)$, and $\mathfrak{t} = O(d)$. We reduce instead from 3-SAT on $d/2$ variables (assuming without loss of generality that d is even). Consider the fixed string

$$F = a_1^0 b_1 \zeta^{2^p} \dots a_{d/2}^0 b_{d/2} \zeta^{2^p}.$$

For every assignment $z \in \{0,1\}^{d/2}$ on $d/2$ variables, define

$$E(z) = a_{d/2+1}^{z_1} b_{d/2+1} \zeta^{2^p} \dots a_d^{z_{d/2}} b_d \zeta^{2^p}.$$

For every possible clause c of width 3 on $d/2$ variables, we let $\mathcal{E}_c = \{E(z) \text{ s.t. } c(z) = 0\}$ denote the assignments on $d/2$ variables that falsify c, and we write $R_c = \ell_c \Sigma_{\mathcal{C}}^* \alpha\beta F \mathcal{E}_c \gamma$. By our choice of $p = \lceil \log d \rceil$, R_c can be recognized by a DFA of size $O(d^2)$ since we may replace the at most d states for recognizing b_i for any $1 \leqslant i \leqslant d$ with a path of length at most $O(d)$ that recognizes $b_i \zeta^{2^p}$. We let $R_{\mathcal{C}}$ be the alternation of all R_c for $c \in \mathcal{C}$. Then, if we let $\mathcal{E} = \left\{E(z) \text{ s.t. } z \in \{0,1\}^{d/2}\right\}$, we define the full regular expression analogously as $R = R_{\mathcal{C}} \mid \alpha\beta F \mathcal{E} \gamma\delta \mid \Sigma$. Following similar arguments as before, and noting that once again all pretoken matching strings of length > 1 have distinct starting characters, R can be recognized by a DFA of size $O(d^5)$, so that the total preprocessing time of the assumed online tokenization algorithm is $\mathrm{poly}(d)$.

For a 3-CNF formula $\psi = c_1 \wedge \dots \wedge c_k$ on $d/2$ variables and $k = \mathrm{poly}(d)$ clauses, we construct the input

$$x_\psi = \ell_{c_1} \ell_{c_2} \dots \ell_{c_k} \alpha\beta F.$$

Note that $|x_\psi| = \mathrm{poly}(d)$. We claim that $\mathrm{Tok}_{\preceq}(x_\psi)$ decides whether ψ is satisfiable, even with delay.

Claim B.12. *Let $f = |F| = \frac{d}{2}(2^p + 2)$. For any 3-CNF formula ψ, we have the following:*

1. *(Completeness) If $\psi \in \mathrm{SAT}$, then for all $y \preceq x_\psi$ with $|y| \geqslant |x_\psi| - f$, $\mathrm{Tok}_{\preceq}(y) = \ell_{c_1} \ell_{c_2} \dots \ell_{c_k}$.*
2. *(Soundness) If $\psi \notin \mathrm{SAT}$, then for all $y \preceq x_\psi$ with $|y| \geqslant |x_\psi| - f$, $\mathrm{Tok}_{\preceq}(y) = \ell_{c_1} \ell_{c_2} \dots \ell_{c_k} \alpha$.*

Proof of Claim B.12. As before, we have $\mathrm{Tok}_{\preceq}(\ell_{c_1} \ell_{c_2} \dots \ell_{c_k}) = \ell_{c_1} \ell_{c_2} \dots \ell_{c_k}$ since no ℓ_c participates in any merge rule and for any $y \preceq x_\psi$, we have $\mathrm{Tok}(y) = y$, since x_ψ contains no γ, δ.

Suppose $\psi \in \mathrm{SAT}$ is satisfiable, so that there exists an assignment $z \in \{0,1\}^{d/2}$ with $\psi(z) = 1$. Fix any $y \preceq x_\psi$ of length at least $|x_\psi| - f$. Then, $y = \ell_{c_1} \ell_{c_2} \dots \ell_{c_k} \alpha\beta F_{\leqslant i}$ for some $0 \leqslant i \leqslant f$. Consider the extension $F_{>i} E(z) \gamma\delta$. Following similar arguments to Claim B.11, we have

$$\mathrm{Tok}(y F_{>i} E(z) \gamma\delta) = \mathrm{Tok}(x_\psi E(z) \gamma\delta) = \ell_{c_1} \ell_{c_2} \dots \ell_{c_k} \Phi A_1^0 \dots A_{d/2}^0 A_{d/2+1}^{z_1} \dots A_d^{z_{d/2}} \Gamma.$$

Thus, $\mathrm{Tok}_{\preceq}(y) \preceq \ell_{c_1} \ell_{c_2} \dots \ell_{c_k}$, establishing equality.

Suppose $\psi \notin \mathrm{SAT}$. Again, fix $y = \ell_{c_1} \ell_{c_2} \dots \ell_{c_k} \alpha\beta F_{\leqslant i}$ for some $0 \leqslant i \leqslant f$. Since every pretoken of length > 1 involving α follows α with β, F, a string in $\mathcal{E}$, and γ, it suffices to consider extensions of the form $F_{>i} E(z) \gamma w$ where $z \in \{0,1\}^{d/2}$ and $w \in \Sigma^*$. Otherwise, the pretoken does not match $\alpha\beta F_{\leqslant i}$, so that $\alpha\beta F_{\leqslant i}$ is not merged and output as single characters. Since ψ is unsatisfiable, there exists a clause c_j in ψ such that $E(z) \in \mathcal{E}_{c_j}$. Following similar arguments to Claim B.11, we have

$$\begin{aligned}\mathrm{Tok}(y F_{>i} E(z) \gamma w) &= \mathrm{Tok}(x_\psi E(z) \gamma w) \\ &= \ell_{c_1} \ell_{c_2} \dots \ell_{c_k} \alpha \Delta^0 B_1^0 \dots B_{d/2-1}^0 B_{d/2}^{z_1} \dots B_{d-1}^{z_{d/2}} B_d \circ \mathrm{Tok}(w).\end{aligned}$$

Thus, $\mathrm{Tok}_{\preceq}(y) = \ell_{c_1} \ell_{c_2} \dots \ell_{c_k} \alpha$. □

Suppose for contradiction that we have an algorithm with delay $o(\mathfrak{mt})$. Let z be the algorithm's output on x_ψ, and let k be its delay. Hence

$$\mathrm{Tok}_{\preceq}(x_\psi[:-k]) \preceq z \preceq \mathrm{Tok}_{\preceq}(x_\psi).$$

Since $\mathfrak{m} = \Theta(d)$, $\mathfrak{t} = \Theta(d)$, and $f = \Theta(d^2)$, we have $k \leqslant f$ for sufficiently large d. By Claim B.12, $\mathrm{Tok}_{\preceq}(x_\psi[:-k]) = \mathrm{Tok}_{\preceq}(x_\psi)$ determines whether ψ is satisfiable. Thus z also determines whether ψ is satisfiable. Since the input x_ψ has length $\mathrm{poly}(d)$, we decide whether ψ is satisfiable in $\mathrm{poly}(d)$ time, contradicting the assumption $\mathbf{P} \neq \mathbf{NP}$. □

B.2 Online Tokenization via Composition

Given Theorem 1.1, our goal is to design fast algorithms with minimal delay. In particular, we would like to design algorithms running in almost linear time (e.g. $n^{1+o(1)}$). We begin with a simple algorithm that composes online segmentation, pretokenization, and BPE merging.

We first define online segmentation, pretokenization, and BPE merging. Note that Claim B.8 and Claim B.10 do not use anything specific about the tokenization process, and can be shown for any function that takes an (incremental) list as input and outputs an (incremental) list. Thus, again in principle, we can hope for efficient online segmentation, pretokenization, and BPE merging. Since pretokenization takes a list of lists as an input and outputs a list of strings, we say a list of strings is updated incrementally if it is updated by either appending characters to the current last string, or starting a new string. Similarly, an algorithm outputs a list of strings incrementally by either appending characters to the current last string, or starting a new string. We let $\mathcal{S}_{\mathrm{Seg}}(x; S^*)$, $\mathcal{S}_{\mathrm{Split}}(L; R)$, and $\mathcal{S}_{\mathrm{Merge}}(L; M)$ denote the set of stable prefixes with respect to S^*, R, M respectively, where x is a string input to segment with S^*, and L is a list of strings input to pretokenization or BPE.

Definition B.13 (Online Segmentation)**.** *Let $S^* \subset \Sigma^*$ be a set of special tokens. Online Segmentation, given an incrementally updated string x, incrementally outputs* $\mathrm{Seg}_{\preceq}(x; S^*) = \max_{\preceq} \mathcal{S}_{\mathrm{Seg}}(x; S^*)$.

Definition B.14 (Online Pretokenization)**.** *Let R be a total regular expression over Σ. Online Pretokenization, given an incrementally updated list of strings L, incrementally outputs* $\mathrm{Split}_{\preceq}(L; R) = \max_{\preceq} \mathcal{S}_{\mathrm{Split}}(L; R)$.

Definition B.15 (Online Byte-Pair Encoding)**.** *Let M be a BPE encoding. Online BPE, given an incrementally updated list of strings L, incrementally outputs* $\mathrm{Merge}_{\preceq}(L; M) = \max_{\preceq} \mathcal{S}_{\mathrm{Merge}}(L; M)$.

As before, we omit S^*, R, M when convenient. We observe that for a list $L = a_1 \parallel \ldots \parallel a_s$, $\mathrm{Split}_{\preceq}(L) = \mathrm{Split}(a_1) \parallel \ldots \parallel \mathrm{Split}(a_{s-1}) \parallel \mathrm{Split}_{\preceq}(a_s)$. Indeed, any continuation of L has the first $s-1$ segments identical to L and the s-th segment is a continuation of a_s. Similarly, $\mathrm{Merge}_{\preceq}(L) = \mathrm{Merge}(a_1) \circ \cdots \circ \mathrm{Merge}(a_{s-1}) \circ \mathrm{Merge}_{\preceq}(a_s)$.

We give a linear time algorithm for online tokenization.

Theorem 1.2 (Online Tokenization via Composition)**.** *Let $S^* \subset \Sigma^*$ be a set of special tokens, R a total regular expression over Σ, and M a merge list. Then, there is an $O(n \log^2 \mathfrak{t})$ time and $O(\mathfrak{b}(R) + \mathfrak{m}\mathfrak{t})$ space algorithm for online tokenization with $\mathfrak{b}(R) + (\mathfrak{m}+1)\mathfrak{t} + \ell_\infty(S^*)$ delay.*

Accounting only for n on large inputs, our algorithm runs in $O(n)$ time and $O(\mathfrak{b})$ space with $\mathfrak{b}+O(1)$ delay.

Proof of Theorem 1.2. As subroutines, we require algorithms `Segment`, `Pretokenize`, `BPEMerge` with the following guarantees.

Lemma B.16 (Online Segmentation)**.** *There is an algorithm `Segment` that computes online segmentation in $O(n)$ time and $O(1)$ space. Furthermore, for any $x \in \Sigma^*$, $|\mathrm{raw}(\mathrm{Seg}_{\preceq}(x))| \geqslant \max(0, |x| - \ell_\infty(S^*))$.*

Lemma B.17 (Online Pretokenization (Formal Lemma 3.1))**.** *There is an algorithm `Pretokenize` that computes online pretokenization in $O(n)$ time and $O(\mathfrak{b}(R))$ space. Furthermore, for any list of segments L, $\mathrm{Split}_{\preceq}(L)$ outputs all but the last $\mathfrak{b}(R)$ characters of L.*

Lemma B.18 (Online BPE (Jiang & Gong, 2026))**.** *There is an algorithm `BPEMerge` that in $O(n \log^2 \mathfrak{t})$ total time and $O(\mathfrak{m}\mathfrak{t})$ space incrementally outputs y satisfying*

$$\mathrm{Merge}(x[:-C]; M) \preceq y \preceq \mathrm{Merge}_{\preceq}(x; M)$$

for some $C = C(x, M) \leqslant (\mathfrak{m}+1)\mathfrak{t}$.

First, we argue that the above suffices to prove Theorem 1.2. Our algorithm simply outputs

$$\texttt{BPEMerge}(\texttt{Pretokenize}(\texttt{Segment}(x; S^*); R); M) \preceq \text{Merge}_{\preceq}(\text{Split}_{\preceq}(\text{Seg}_{\preceq}(x))).$$

Note that each of the three algorithms are online (i.e. handles incremental input and incremental output) so that the algorithm is well defined. The running time and space bounds follow trivially, so it suffices to prove correctness. Let x be the current input and $\text{Tok}_{\preceq}(x)$ the desired output. First, we prove that our algorithm is sound: it outputs a stable prefix.

Claim B.19. *For any S^*, R, M and input string x,*

$$\text{Merge}_{\preceq}(\text{Split}_{\preceq}(\text{Seg}_{\preceq}(x; S^*); R); M) \preceq \text{Tok}_{\preceq}(x; S^*, R, M).$$

Proof. Throughout, we omit S^*, R, M for clarity. Let $x_1 = \text{Seg}_{\preceq}(x)$, $x_2 = \text{Split}_{\preceq}(x_1)$, and $x_3 = \text{Merge}_{\preceq}(x_2)$ denote the outputs of our algorithm. Fix any continuation $y \in \Sigma^*$. Then, by correctness of `Segment`, $x_1 = \text{Seg}_{\preceq}(x) \preceq \text{Seg}(xy)$ so that $\text{Seg}(xy) = x_1 w$ for some w. By correctness of `Pretokenize`, $x_2 \preceq \text{Split}(x_1 w) = \text{Split}(\text{Seg}(xy))$. Then, $\text{Split}(\text{Seg}(xy)) = x_2 v$ for some v. By correctness of `BPEMerge`, we have $x_3 \preceq \text{Merge}(x_2 v) = \text{Merge}(\text{Split}(\text{Seg}(xy))) = \text{Tok}(xy)$. Thus, x_3 is a stable prefix. □

Next, we prove completeness: our algorithm outputs the maximum stable prefix with delay $\mathfrak{b}(R) + O(\mathfrak{mt})$. We require the following lemma, which states that any accepted word must reach an accepting state every $\mathfrak{b}(R)$ characters.

Lemma B.20. *Let $\mathcal{A}$ be a DFA recognizing $L(R)$. For $u \in \Sigma^+$, let $\mathcal{A}(u)$ denote whether $\mathcal{A}$ is in an accepting state after reading u. Let $\text{last}(u) := \max\{|w| \text{ s.t. } w \preceq u, \mathcal{A}(w) = 1\}$ denote the last accepting state when reading u (possibly including u). If $\text{last}(u) < |u| - \mathfrak{b}(R)$, then $\mathcal{A}(ux) = 0$ for all $x \in \Sigma^*$.*

Proof. Suppose for contradiction $\mathcal{A}(ux) = 1$ for some continuation $x \in \Sigma^*$. Let $\mathfrak{b} := \mathfrak{b}(R)$. Since R is total, we have $\text{last}(u) \geqslant 1$ so that $|u| > \mathfrak{b}$. Set $y_0 = ux$. By Definition B.3, for any $|y_i| > \mathfrak{b}$, there exists some $y_{i+1} \prec y_i$ such that $|y_{i+1}| \geqslant |y_i| - \mathfrak{b}$ and $\mathcal{A}(y_{i+1}) = 1$. Let i^* be the largest i such that $u \preceq y_i$ so that $y_{i^*+1} \prec u$. Then, $\text{last}(u) \geqslant |y_{i^*+1}| \geqslant |y_{i^*}| - \mathfrak{b} \geqslant |u| - \mathfrak{b}$, a contradiction. □

Let $\mathfrak{b} := \mathfrak{b}(R)$. Let $\text{Seg}_{\preceq}(x) = a_1 \parallel \ldots \parallel a_{s-1} \parallel a'$ where $x = a_1 a_2 \ldots a_{s-1} a_s$ and a_s (possibly empty) contains no special token. We can assume this without loss of generality, as if x ends on a special token, then $a_s = \epsilon$ is empty. Note that we can thus assume a_{s-1} is a special token.

Since $\text{Seg}_{\preceq}(x)$ only holds back characters of a_s if a suffix of a_s is a prefix of a special token in S^*, we conclude that $a_s[:-\ell_\infty(S^*)] \preceq a' \preceq a_s$ since $a_s[-\ell_\infty(S^*):] \notin S^*$ and therefore is not a prefix of any special token $t \in S^*$. Then, we write

$$\text{Split}_{\preceq}(\text{Seg}_{\preceq}(x)) = \text{Split}(a_1) \parallel \ldots \parallel \text{Split}(a_{s-1}) \parallel \text{Split}_{\preceq}(a') = P \parallel \text{Split}_{\preceq}(a')$$

where $P = \parallel_{i=1}^{s-1} \text{Split}(a_i)$. If $a' = \epsilon$ is empty, then $\text{Seg}_{\preceq}(x) = a_1 \parallel \ldots \parallel a_{s-1} \parallel$ so that

$$\text{Merge}(\text{Split}(a_1 \parallel \ldots \parallel a_{s-1} \parallel)) \preceq \text{Tok}_{\preceq}(x).$$

In other words, since a_{s-1} is a special token, we have

$$\text{Tok}_{\preceq}(a_1 \ldots a_{s-1}) \preceq \text{Tok}(a_1 \ldots a_{s-1}) = \text{Merge}(\text{Split}(a_1 \parallel \ldots \parallel a_{s-1} \parallel)) \preceq \text{Tok}_{\preceq}(x)$$

so that the algorithm has delay $\ell_\infty(S^*)$, since $a' = \varepsilon$ implies $|a_s| \leqslant |a'| + \ell_\infty(S^*) \leqslant \ell_\infty(S^*)$.

Thus, assume $a' \neq \epsilon$ is not empty. Let $\text{Split}_{\preceq}(a') = b_1 \parallel \ldots \parallel b_{t-1} \parallel b'$ where $a' = b_1 \ldots b_t$ and $b' \preceq b_t$. Lemma B.17 guarantees that $|b'| \geqslant |b_t| - \mathfrak{b}(R)$.

Then, our algorithm outputs

$$\begin{aligned}\texttt{BPEMerge}(P \parallel \text{Split}_{\preceq}(a')) &= \text{Merge}(P) \circ \text{Merge}(b_1) \circ \cdots \circ \text{Merge}(b_{t-1}) \circ \texttt{BPEMerge}(b') \\ &= Q \circ \texttt{BPEMerge}(b')\end{aligned}$$

where $Q = \text{Merge}(P) \circ \left(\bigcirc_{i=1}^{t-1} \text{Merge}(b_i)\right)$. Recall that we proved above

$$|a_1 \ldots a_{s-1} b_1 \ldots b_{t-1} b'| \geqslant |a_1 \ldots a_{s-1} a'| - \mathfrak{b}(R) \geqslant |x| - (\ell_\infty(S^*) + \mathfrak{b}(R)).$$

We have the following claim.

Claim B.21. *Suppose $y \preceq x$ and $z \preceq \mathrm{Tok}_{\preceq}(x)$, and $|z| \geqslant |y|$. Then, $\mathrm{Tok}_{\preceq}(y) \preceq z$.*

Proof of Claim B.21. Note $\mathrm{Tok}_{\preceq}(y) \preceq \mathrm{Tok}_{\preceq}(x)$, so $\mathrm{Tok}_{\preceq}(y), z$ are prefixes of $\mathrm{Tok}_{\preceq}(x)$. Thus, if $|z| \geqslant |y|$, we have the desired claim. □

Our algorithm outputs $z = Q \circ \texttt{BPEMerge}(b') \preceq \mathrm{Tok}_{\preceq}(x)$. From Lemma B.18 we have

$$\begin{aligned} |z| &\geqslant |a_1 \dots a_{s-1} b_1 \dots b_{t-1}| + |b'| - (\mathfrak{m}+1)\mathfrak{t} \\ &\geqslant |x| - \ell_\infty(S^*) - \mathfrak{b} - (\mathfrak{m}+1)\mathfrak{t}. \end{aligned}$$

Thus, Claim B.21 implies the desired delay bound. It remains to prove the required lemmas.

Lemma B.16 (Online Segmentation)**.** *There is an algorithm* `Segment` *that computes online segmentation in $O(n)$ time and $O(1)$ space. Furthermore, for any $x \in \Sigma^*$, $|\mathrm{raw}(\mathrm{Seg}_{\preceq}(x))| \geqslant \max(0, |x| - \ell_\infty(S^*))$.*

Proof of Lemma B.16. Consider a trie $\mathcal{T}$ that recognizes strings in S^*. This trie has size $O(\ell_1(S^*))$. Regard the trie as a deterministic automaton with an additional rejecting sink. Missing child transitions lead to the sink, which loops on every character. Accepting nodes correspond to special tokens. The function $\mathrm{last}(\mathrm{start}, w)$ returns the length of the longest accepting prefix of w, or $\perp$ if none exists.

We note some properties of $\mathrm{Seg}_{\preceq}(x)$. The input x consists of the characters in the committed output $\mathrm{Seg}_{\preceq}(x)$ and (possibly empty) buffer u such that $x = \mathrm{raw}(\mathrm{Seg}_{\preceq}(x)) \circ u$. We note that u is a (non-maximal) trie prefix. Otherwise, if it is not a trie prefix, it is not (and cannot be extended to a special token), so that we can commit at least one character of u to output. If it is a maximal trie prefix, it is a special token (and cannot be extended to a longer one), so we can commit $s^*(u)$ as a special token. Thus, we have

$$|\mathrm{raw}(\mathrm{Seg}_{\preceq}(x))| \geqslant |x| - |u| \geqslant |x| - \ell_\infty(S^*).$$

For some partition $x = y_1 \dots y_s u$ where we may assume without loss of generality that (possibly empty) y_s is not a special token, we have

$$\mathrm{Seg}_{\preceq}(x) = \begin{cases} y_1 \parallel \dots \parallel y_s \parallel & \text{if } u \text{ contains a special token as a prefix} \\ y_1 \parallel \dots \parallel y_s & \text{o/w} \end{cases}. \tag{2}$$

Note that each y_j is either a special token (corresponding to a trie match) or a segment of alphabet characters. Only the latter is provided as input to pretokenization and BPE in later stages.

In the first case, since u contains a special token as a prefix, any continuation of x contains a special token after y_s, so the prefix contains $\parallel$ after y_s. In the latter case, we do not know if u will extend to a special token. In particular, we cannot yet commit any character of u, since if u is extended to a match (e.g. $\tilde{u} = uu' \in S^*$) then $\mathrm{Seg}(xu')$ ends with $\dots y_s \parallel s^*(\tilde{u})$.

We now give a simple algorithm. Intuitively, buffer will maintain the uncommitted suffix of the input. Initialize $\mathbf{state} \leftarrow \mathrm{start}$ to the root of $\mathcal{T}$, $\mathrm{last} \leftarrow \perp$, and $\mathrm{buffer} \leftarrow \epsilon$. Recall that δ denotes the transition function and last the last acceptance. In the following, reset means $\mathbf{state} \leftarrow \mathrm{start}, \mathrm{last} \leftarrow \perp, \mathrm{buffer} \leftarrow \epsilon$. Our algorithm never outputs two consecutive $\parallel$; this can be checked by examining the last output. On input x_i, do the following:

`SimpleMatch`(x_i)

1. Update $\mathbf{state} \leftarrow \delta(\mathbf{state}, x_i)$ and push x_i to buffer.
2. While buffer is not a trie prefix:
   (a) If $\mathrm{last} = \perp$, output buffer_1 and update $\mathrm{buffer} \leftarrow \mathrm{buffer}_{>1}$.
   (Prefix of special token unmatched. Flush first character.)
   (b) If $\mathrm{last} \neq \perp$, output and pop $\parallel s^*(\mathrm{buffer}_1 \dots \mathrm{buffer}_{\mathrm{last}}) \parallel$.
   (Backtrack to the last special token. Restore state.)
   (c) Update $\mathbf{state} \leftarrow \delta(\mathrm{start}, \mathrm{buffer})$ and $\mathrm{last} \leftarrow \mathrm{last}(\mathrm{start}, \mathrm{buffer})$.

3. If $\mathbf{state} \in \mathcal{P}$ and buffer is a maximal trie prefix, output $\| s^*(\text{buffer}) \|$. Reset.
   (Found maximal special token.)
4. Else if $\mathbf{state} \in \mathcal{P}$ set last $\leftarrow |\text{buffer}|$. If last $\neq \perp$, output $\|$.
   (Found special token, not maximal.)

Observe that if we make the assumption that S^* is prefix-free (as is often the case in practice), the algorithm and ensuing analysis can be simplified by ignoring Step 4. The space of the algorithm is $O(|\text{buffer}|) = O(\ell_\infty(S^*))$, since buffer contains exactly the uncommitted input u. The time of the $\mathbf{state} \in \mathcal{P}$ cases is $O(|\text{buffer}|)$, $O(|\text{buffer}|)$ in the maximal case and $O(1)$ otherwise. We bound the time of Step 2. We observe that each iteration of the while loop takes $O(|\text{buffer}|)$ time (e.g. to recompute $\mathbf{state}$, last). Furthermore, there are at most $O(|\text{buffer}|)$ iterations since in each iteration, the size of buffer decreases by at least 1. Thus, the overall time per update is $O(|\text{buffer}|^2) = O(\ell_\infty(S^*)^2) = O(1)$. With a slightly tighter analysis, we can achieve $O(n|\text{buffer}|)$ total time, by observing that every step that requires $O(|\text{buffer}|)$ time produces at least one character of output, so there are at most $O(n)$ such steps.

We now prove correctness, inducting on x_i with the following hypothesis. Let u be the uncommitted input after processing x_{i-1}.

1. buffer contains the uncommitted input.
2. $\mathbf{state} = \delta(\text{start}, \text{buffer})$ and $\text{last} = \text{last}(\text{start}, \text{buffer})$.
3. The output is $\text{Seg}_{\preceq}(x_{\leqslant i})$ (satisfying Equation (2)).

The base case is trivial. On input x_i, either x_i extends u as a prefix of a special token, or it does not. In the former case, if ux_i is a maximal special token, our algorithm has $\mathbf{state} \in \mathcal{P}$ and buffer is maximal, so we output $\| s^*(ux_i) \|$ and reset, satisfying the inductive hypothesis. Otherwise, ux_i is not a maximal special token. If it is a special token, we output $\|$ since $\mathbf{state} \in \mathcal{P}$, push x_i to buffer and update $\mathbf{state}$, last, again satisfying the inductive hypothesis. Otherwise, our algorithm pushes x_i to buffer and updates $\mathbf{state}$, again satisfying the inductive hypothesis.

If x_i does not extend u, our algorithm enters the case where buffer is not a trie prefix. Whenever buffer is not a trie prefix, we pop some prefix of it (either one character or a prefix of buffer that is a special token). Eventually, buffer (which is possibly empty) must be a trie prefix. If this prefix is maximal, we output it, leaving buffer empty and satisfying the first condition. If not, we retain buffer, again satisfying the first condition, as this is the uncommitted input. It is easily verified that whenever buffer is updated, $\mathbf{state}$ and last are recomputed appropriately, satisfying the second condition. In each iteration, we output any prefix of buffer that is a special token (that cannot be extended), and also outputting $\|$ if the buffer contains a trie prefix after exiting the while loop, maintaining the correctness of the output as desired. □

Lemma B.17 (Online Pretokenization (Formal Lemma 3.1))**.** *There is an algorithm* `Pretokenize` *that computes online pretokenization in $O(n)$ time and $O(\mathfrak{b}(R))$ space. Furthermore, for any list of segments L, $\text{Split}_{\preceq}(L)$ outputs all but the last $\mathfrak{b}(R)$ characters of L.*

We note that in the offline setting, there is a linear time pretokenization algorithm (called maximal munch tokenization) of Reps (1998). In the worst case, their algorithm requires linear space. We design an algorithm parameterized by $\mathfrak{b} := \mathfrak{b}(R)$. Consider a DFA $\mathcal{A}$ that recognizes exactly $L(R)$ with starting state start.

Observe that a simple modification of Lemma B.16 gives an algorithm for online pretokenization that runs in $O(n\mathfrak{b})$ time and $O(\mathfrak{b})$ space: Simulate $\mathcal{A}$. On each accepting state, emit the current match; otherwise buffer the current input in memory. On each dead state, backtrack to the last accepting state, emit a $\|$, and simulate $\mathcal{A}$ on the buffer in memory from the starting state. Lemma B.20 guarantees that a dead state must be encountered within $O(\mathfrak{b})$ characters of the last accept, so that we backtrack at most $O(\mathfrak{b})$ characters, resulting in $O(n\mathfrak{b})$ time. While in practice $\mathfrak{b}$ is not large,

in the worst case this can lead to a quadratic time algorithm (e.g. when $\mathfrak{b}$ is sufficiently large). To guarantee linear time in general, we require the following useful data structure:

Lemma 3.2. *There is a data structure* cQueue *supporting the following operations on a string w:*

1. *(Push) Append a character c to w i.e. update $w \leftarrow wc$.*
2. *(Pop) Pop a prefix of w, i.e. return $w_1 \dots w_p$ and update $w \leftarrow w_{p+1} \dots w_\ell$ for some $p > 0$.*
3. *(Query) Given a state q, compute $\delta(q, w)$ and $\mathrm{last}(q, w)$.*

Furthermore, the data structure supports the operations in amortized $O(1)$ time.

The data structure is initialized with two empty stacks: the push stack and the pop stack.

$\mathbf{push}(c)$ for character $c \in \Sigma$:

1. Compute $\delta(q, c)$ and $\mathrm{last}(q, c)$ for all $q \in \mathcal{Q}$.
2. Let $w_{\mathbf{push}}$ be the top element of the push stack (ϵ if empty). Compute and push

$$\delta(q, w_{\mathbf{push}}c) \leftarrow \delta(\delta(q, w_{\mathbf{push}}), c),$$
$$\mathrm{last}(q, w_{\mathbf{push}}c) \leftarrow \begin{cases} \mathrm{last}(q, w_{\mathbf{push}}) & \text{if } \mathrm{last}(\delta(q, w_{\mathbf{push}}), c) = \perp \\ |w_{\mathbf{push}}| + \mathrm{last}(\delta(q, w_{\mathbf{push}}), c) & \text{o/w} \end{cases}.$$

$\mathbf{pop}(p)$ for integer $p \geqslant 1$:

1. If the pop stack has $k \geqslant p$ elements, pop p elements from the pop stack.
2. If the pop stack has $k < p$ elements:
   (a) Pop k elements from the pop stack. Let the push stack contain $w_1 \dots w_\ell$ (with w_ℓ on top).
   (b) If $\ell + k \leqslant p$, empty both stacks.
   (c) If $\ell + k > p$, reverse the elements of the push stack onto the pop stack, so that the pop stack contains $w_\ell \dots w_1$ (with w_1 on top). For each $i \in [\ell]$ in decreasing order, compute:

$$\delta(q, w_i \dots w_\ell) \leftarrow \delta(\delta(q, w_i), w_{i+1} \dots w_\ell),$$
$$\mathrm{last}(q, w_i \dots w_\ell) \leftarrow \max \begin{cases} \mathrm{last}(q, w_i) \\ 1 + \mathrm{last}(\delta(q, w_i), w_{i+1} \dots w_\ell) \end{cases}.$$

   (For comparison purposes, if $\mathrm{last}(q, w) = \perp$, then $\perp = -\infty$.)
   Then, pop $p - k$ elements from the pop stack.

$\mathbf{query}(q)$ for state $q \in \mathcal{Q}$:

1. Let the pop stack contain $w_j \dots w_1$ (with w_1 on top) and the push stack contain $w_{j+1} \dots w_\ell$ (with w_ℓ on top).
2. Compute:

$$\delta(q, w_1 \dots w_\ell) \leftarrow \delta(\delta(q, w_1 \dots w_j), w_{j+1} \dots w_\ell),$$
$$\mathrm{last}(q, w_1 \dots w_\ell) \leftarrow \max \begin{cases} \mathrm{last}(q, w_1 \dots w_j) \\ j + \mathrm{last}(\delta(q, w_1 \dots w_j), w_{j+1} \dots w_\ell) \end{cases}.$$

We prove that the data structure satisfies the desired guarantees.

Proof of Lemma 3.2. We maintain the following invariant: every element maintains δ and last for the string obtained by concatenating all characters beneath it and itself. This consists of prefixes of the string on the push stack and suffixes of the string on the pop stack. Formally,

Claim B.22 (Below Consistency). *Let the pop stack contain $w_j \dots w_1$ (with w_1 on top) and the push stack contain $w_{j+1} \dots w_\ell$ (with w_ℓ on top). Our algorithm maintains for every $q \in \mathcal{Q}$:*

1. *(Pop) $\delta(q, w_i \dots w_j)$ and $\text{last}(q, w_i \dots w_j)$ for all $1 \leqslant i \leqslant j$,*
2. *(Push) $\delta(q, w_{j+1} \dots w_i)$ and $\text{last}(q, w_{j+1} \dots w_i)$ for all $j+1 \leqslant i \leqslant \ell$.*

Proof of Claim B.22. Note that for any word $w_1 \dots w_j w_{j+1} \dots w_\ell$, we have by the definition of a DFA for all $j \in [\ell]$,

$$\delta(q, w_1 \dots w_\ell) = \delta(\delta(q, w_1 \dots w_j), w_{j+1} \dots w_\ell). \tag{3}$$

Similarly, for all $j \in [\ell]$,

$$\begin{aligned} \text{last}(q, w_1 \dots w_\ell) &= \max \begin{cases} \text{last}(q, w_1 \dots w_j) \\ j + \text{last}(\delta(q, w_1 \dots w_j), w_{j+1} \dots w_\ell) \end{cases} \\ &= \begin{cases} \text{last}(q, w_1 \dots w_j) & \text{if } \text{last}(\delta(q, w_{\leqslant j}), w_{j+1} \dots w_\ell) = \bot \\ j + \text{last}(\delta(q, w_{\leqslant j}), w_{j+1} \dots w_\ell) & \text{o/w} \end{cases}. \end{aligned} \tag{4}$$

The above equality follows as either the DFA reaches an accepting state in the second half of the computation when scanning $w_{j+1} \dots w_\ell$ (in which case the second case triggers) or it does not, in which case it suffices to find an accepting state in the first half when scanning $w_1 \dots w_j$ (which is handled by the first case).

We now argue that the invariant is maintained on the push and pop stacks. This follows directly from Equation (3) and Equation (4) and the update rule of **push** and **pop**. □

Given Claim B.22, we observe that Equation (3) and Equation (4) ensure that **query** returns the correct values. We now argue that our data structure has the desired performance. Consider a single element: it is pushed onto the push stack once, moved from the push to pop stack (at most) once, and popped from the pop stack (at most) once. It then suffices to argue that each operation requires $O(1)$ time. For **push**, the computation of the transitions δ and last can be computed in $O(1)$ time given $\delta(q, w_{\textbf{push}})$ and $\text{last}(q, w_{\textbf{push}})$. When moving from the push stack to the pop stack, we compute $\delta(q, w_i)$ and $\text{last}(q, w_i)$ in $O(1)$ time and then query $\delta(\cdot, w_{i+1} \dots w_\ell)$ and $\text{last}(\cdot, w_{i+1} \dots w_\ell)$ in $O(1)$ time. Popping an element of the pop stack requires $O(1)$ time, completing the proof. □

Using the given data structure, run the following algorithm. We begin by initializing an empty cQueue (Lemma 3.2) and **state** $\leftarrow$ start. Then, for each character x_i in the input, do the following.

`FastPretokenization`(x_i)

1. Push cQueue.**push**(x_i) and update **state** $\leftarrow \delta(\textbf{state}, x_i)$.
2. If **state** $\in \mathcal{P}$, output cQueue.**pop**$(|\text{cQueue}|)$.
   (Found a match, so emit the current buffer and reset cQueue.)
3. While **state** $\in$ dead:
   (The unresolved suffix cannot continue the current match, so $\|$ at the last $\mathcal{P}$.)
   (a) If cQueue $= \epsilon$, output $\|$, set **state** $\leftarrow$ start and exit.
   (b) Query $\delta(\text{start}), \text{last}(\text{start}) \leftarrow$ cQueue.**query**(start).
   (c) Output $\|$ and cQueue.**pop**(last(start)).
   (d) Update **state** $\leftarrow \delta(\text{start})$.

We prove that our algorithm satisfies the desired properties.

Proof of Lemma B.17. We begin by analyzing the correctness of our algorithm. We proceed by induction, and argue that for all $i \geqslant 0$, after witnessing $x_{\leqslant i}$, the algorithm maintains the following:

A The output is $\mathrm{Split}_{\preceq}(x_{\leqslant i})$. The output $\mathrm{Split}_{\preceq}(x_{\leqslant i})$ consists of a string consisting of pretokens and separators $\|$. We let P_i denote the output before the last separator in $\mathrm{Split}_{\preceq}(x_{\leqslant i})$. Thus, we observe that the output has one of the following forms

- $P_i \parallel x_{b(i)+1} \dots x_{\ell(i)}$ for $\ell(i) = b(i) + \mathrm{last}(\mathrm{start}, x_{b(i)+1} \dots x_i)$, for some $b(i)$ indicating the current "live" pretoken,
- $P_i \parallel$, in which case we write $b(i) = \ell(i) = i$.

B cQueue maintains $x_{\ell(i)+1} \dots x_i$.

C **state** maintains $\delta(\mathrm{start}, x_{b(i)+1} \dots x_i)$.

We call the above induction the outer induction, which inducts over each input x_i provided incrementally to the algorithm. The base case is trivial. If $i = 0$, we have $b(0) = \ell(0) = 0$ so the initial output is ϵ, cQueue is empty, and $\mathbf{state} \leftarrow \mathrm{start} = \delta(\mathrm{start}, \epsilon)$.

We now proceed to the inductive case. Consider input x_i. To prove the outer induction above, we will use an inner induction over the while loop within `FastPretokenization`. To prove the output of the i-th iteration of the outer induction has Properties A, B, C, we assume the output of the $(i-1)$-th iteration starting with $P_{i-1} \parallel$ satisfies Properties A, B, C after processing x_{i-1}. Note that $P_{i-1} \preceq P_i$ since we can only incrementally extend the output of $\mathrm{Split}_{\preceq}(x_{\leqslant i})$. We now claim that after the j-th iteration of the while loop, the following hold:

1. The current output is $P_{i-1} \parallel Q_j \preceq \mathrm{Split}_{\preceq}(x_{\leqslant i})$ where
$$Q_j = x_{b(i-1)+1} \dots x_{r(0)} \parallel x_{r(0)+1} \dots x_{r(1)} \parallel \dots \parallel x_{r(j-1)+1} \dots x_{r(j)}$$
for $r(j) = r(j-1) + \mathrm{last}(\mathrm{start}, x_{r(j-1)+1} \dots x_i)$ with the convention $r(-1) = b(i-1)$.
2. cQueue maintains $x_{r(j)+1} \dots x_i$. Furthermore, **state** maintains $\delta(\mathrm{start}, x_{r(j-1)+1} \dots x_i)$.
3. If $\mathbf{state} \notin \mathrm{dead}$, then the output is $\mathrm{Split}_{\preceq}(x_{\leqslant i})$.
4. $\mathbf{state} \notin \mathcal{P}$ if and only if $r(j) < i$.

First, we claim that if the inner induction claim holds and the while loop terminates, then the outer induction holds. Towards Item A, we observe that the while loop only terminates when $\mathbf{state} \notin \mathrm{dead}$ and thus Item 3 ensures that whenever $\mathbf{state} \notin \mathrm{dead}$, the output is $\mathrm{Split}_{\preceq}(x_{\leqslant i})$ as desired. Towards Item B and Item C, we set $b(i) = r(j-1)$ and $\ell(i) = r(j)$ after the termination of the while loop so that Item 2 ensures the desired properties. Thus, it suffices to prove (1) the inner induction holds and (2) that the algorithm does indeed terminate after each input.

Base Case. We begin with the base case. By convention, we say that the 0-th iteration of the while loop ends after Step 2, i.e. immediately before the first iteration of the while loop. When input x_i arrives, the output is $\mathrm{Split}_{\preceq}(x_{<i}) \preceq \mathrm{Split}_{\preceq}(x_{\leqslant i})$ by Claim B.10.

If $\mathbf{state} \notin \mathcal{P}$, then the algorithm does not output or modify cQueue in Step 2. Furthermore, since
$$\mathbf{state} = \delta(\delta(\mathrm{start}, x_{b(i-1)+1} \dots x_{i-1}), x_i) = \delta(\mathrm{start}, x_{b(i-1)+1} \dots x_i) \notin \mathcal{P},$$
we have $r(0) \leqslant \ell(i-1)$, satisfying the first, second, and fourth conditions.

If $\mathbf{state} \in \mathcal{P}$, then we have
$$\mathbf{state} = \delta(\mathrm{start}, x_{b(i-1)+1} \dots x_i) \in \mathcal{P}.$$
In particular, since for any continuation y,
$$\mathrm{Split}(x_{\leqslant i} y) \succeq \mathrm{Split}_{\preceq}(x_{\leqslant i}) \succeq \mathrm{Split}_{\preceq}(x_{<i}) \succeq P_{i-1} \parallel$$

we have $\text{Split}(x_{\leqslant i}y) = P_{i-1} \parallel \text{Split}(x_{b(i-1)+1} \dots x_i y)$ so that

$$x_{b(i-1)+1} \dots x_i = F(x_{b(i-1)+1} \dots x_i) \preceq F(x_{b(i-1)+1} \dots x_i y) \preceq \text{Split}(x_{b(i-1)+1} \dots x_i y)$$

where $F(x)$ represents the first match of x (see Definition B.2). Thus, the first, second, and fourth conditions are satisfied with $r(0) = i$.

It remains to check the third condition. Suppose $\mathbf{state} \notin \text{dead}$, we claim the output is $\text{Split}_{\preceq}(x_{\leqslant i})$. Since $\mathbf{state} \notin \text{dead}$, there is some continuation $|y| > 0$ where

$$\delta(\text{start}, x_{b(i-1)+1} \dots x_i y) = \delta(\delta(\text{start}, x_{b(i-1)+1} \dots x_i), y) = \delta(\mathbf{state}, y) \in \mathcal{P}.$$

If $\mathbf{state} \notin \mathcal{P}$, then above we had $r(0) = \ell(i-1) < i$,

$$\text{Split}(x_{\leqslant i}) = P_{i-1} \parallel x_{b(i-1)+1} \dots x_{\ell(i-1)} \parallel \text{Split}(x_{\ell(i-1)+1} \dots x_i)$$
$$\text{Split}(x_{\leqslant i}y) = P_{i-1} \parallel x_{b(i-1)+1} \dots x_i y$$

so that $\text{Split}_{\preceq}(x_{\leqslant i}) \preceq P_{i-1} \parallel x_{b(i-1)+1} \dots x_{\ell(i-1)} = P_{i-1} \parallel x_{b(i-1)+1} \dots x_{r(0)}$.

If $\mathbf{state} \in \mathcal{P}$, after Step 2 the output is $\text{Split}(x_{\leqslant i})$ since we output

$$P_{i-1} \parallel x_{b(i-1)+1} \dots x_i = P_{i-1} \parallel F(x_{b(i-1)+1} \dots x_i) = \text{Split}(x_{\leqslant i}).$$

We claim this is $\text{Split}_{\preceq}(x_{\leqslant i})$ as

$$\text{Split}(x_{\leqslant i}) = P_{i-1} \parallel x_{b(i-1)+1} \dots x_i$$
$$\text{Split}(x_{\leqslant i}y) = P_{i-1} \parallel x_{b(i-1)+1} \dots x_i y$$

have a maximum common prefix of $P_{i-1} \parallel x_{b(i-1)+1} \dots x_i$. Thus, in both cases we have that the output is $\text{Split}_{\preceq}(x_{\leqslant i})$.

Inductive Case. Now, let $j \geqslant 1$. We check that all conditions are maintained through each iteration of the while loop. From the second condition, we have cQueue maintains $x_{r(j-1)+1} \dots x_i$ so that we compute

$$\delta(\text{start}) = \delta(\text{start}, x_{r(j-1)+1} \dots x_i) \quad \text{and} \quad \text{last}(\text{start}) = \text{last}(\text{start}, x_{r(j-1)+1} \dots x_i).$$

Let δ_0 denote $\mathbf{state}$ at the end of the previous iteration. If $\delta_0 \in \mathcal{P}$, the fourth condition implies that $r(j-1) = i$ and the second implies cQueue maintains ϵ. Furthermore, since $\delta_0 \in \text{dead}$ (otherwise this iteration would never have run), then for every continuation y

$$\text{Split}(x_{\leqslant i}y) = \text{Split}(x_{\leqslant i}) \parallel \text{Split}(y)$$

so that $\text{Split}_{\preceq}(x_{\leqslant i}) = P_{i-1} \parallel Q_{j-1} \parallel$. Our algorithm outputs exactly $\parallel$ in the j-th iteration. Furthermore, $\mathbf{state} \leftarrow \delta(\text{start}) = \text{start} \notin \text{dead}$, so the while loop terminates, completing the inductive step.

Thus, in the following assume $\delta_0 \notin \mathcal{P}$, so that $r(j-1) < i$ and $r(j) > r(j-1)$. By induction, we output $P_{i-1} \parallel Q_j$. We argue $P_{i-1} \parallel Q_j \preceq \text{Split}_{\preceq}(x_{\leqslant i})$. Since $\delta_0 \in \text{dead}$, we have for any continuation y,

$$\text{Split}(x_{\leqslant i}y) \succeq P_{i-1} \parallel Q_{j-1} \parallel \text{Split}(x_{r(j-1)+1} \dots x_i y).$$

Since $x_{r(j-1)+1} \dots x_{r(j)} = F(x_{r(j-1)+1} \dots x_i)$, we have

$$P_{i-1} \parallel Q_j = P_{i-1} \parallel Q_{j-1} \parallel F(x_{r(j-1)+1} \dots x_i) \preceq \text{Split}(x_{\leqslant i}y).$$

The algorithm pops and outputs $x_{r(j-1)+1} \dots x_{r(j)}$, satisfying the first and second conditions. If $\delta(\text{start}) \notin \mathcal{P}$, we have $r(j) < i$, satisfying the fourth condition.

We verify the third condition. Suppose $\mathbf{state} \notin \text{dead}$, so there is some continuation $|y| > 0$ where

$$\delta(\text{start}, x_{r(j-1)+1} \dots x_i y) = \delta(\delta(\text{start}, x_{r(j-1)+1} \dots x_i), y) = \delta(\mathbf{state}, y) \in \mathcal{P}.$$

Then, we have

$$\text{Split}(x_{\leqslant i}) = P_{i-1} \parallel Q_j \parallel \text{Split}(x_{r(j)+1} \dots x_i)$$
$$\text{Split}(x_{\leqslant i}y) = P_{i-1} \parallel Q_{j-1} \parallel x_{r(j-1)+1} \dots x_i y$$

so that the maximum common prefix is at most $P_{i-1} \parallel Q_{j-1} \parallel x_{r(j-1)+1} \dots x_{r(j)} = P_{i-1} \parallel Q_j$, as desired.

Termination Condition. We now argue that the while loop terminates. Above we have argued that whenever $\mathbf{state} \in \mathcal{P}$, the while loop terminates in the next iteration. On the other hand, whenever $\mathbf{state} \notin \mathcal{P}$, we have $r(j) < i$, and the first condition ensures that $r(j) > r(j-1)$ since $\mathcal{A}$ is total so that $\mathrm{last}(\mathrm{start}, x_{r(j-1)+1} \dots x_i) > 0$, as every non-empty string has a non-empty prefix that matches the regular expression.

Performance. Finally, we bound the amortized running time of our algorithm. From Lemma 3.2, it suffices to argue that the running time of our algorithm is dominated by operations of cQueue (up to constant factors). To do so, we can simply charge the check of $\delta(\mathrm{start}) \in \mathrm{dead}$ and outputting $\|$ in each iteration of the while loop to query from cQueue. To bound the space, we use Lemma B.20. Suppose $|\mathrm{cQueue}| > \mathfrak{b}$ maintains more than $\mathfrak{b}$ elements e.g. $x_{i-\mathfrak{b}} \dots x_i$. Recall $\ell(i) = b(i) + \mathrm{last}(\mathrm{start}, x_{b(i)+1} \dots x_i)$ is the last output index, corresponding to the last index where $\mathbf{state} \in \mathcal{P}$ on the current attempted match. Then, we have $\ell(i) < i - \mathfrak{b}$ since we pop from cQueue and output whenever an accepting state is encountered. In particular, Lemma B.20 ensures that $\delta(\mathrm{start}, x_{b(i)+1} \dots x_i) \in \mathrm{dead}$, so that we enter the while loop and output $\|$. Since $\mathcal{A}$ is total, on the new match we have $\ell' \geqslant \ell(i) + 1$, so that the algorithm outputs $\|$ and pops (at least) $x_{\ell(i)+1}$ from cQueue. Thus, whenever $|\mathrm{cQueue}| > \mathfrak{b}$, the algorithm pops at least one element from cQueue in each iteration, ensuring that the size of the string maintained by cQueue does not grow. Thus, cQueue requires space at most $O(\mathfrak{b})$.

Extension to NFA So far, we have assumed that $\mathcal{A}$ is a DFA. However, given a regex R, a DFA might have size $\exp(|R|)$, while an NFA is promised to have size $O(|R|)$. We thus describe how to extend our algorithm to handle NFAs. It suffices to describe how to modify the data structure given by Lemma 3.2 to handle queries on NFAs. Note that we only ever query the data structure with the start state. In addition, in the algorithm of Lemma B.17, we modify $\mathbf{state} \subseteq \mathcal{Q}$ to be a subset of states and in the first step, modify the update rule $\mathbf{state} \leftarrow \cup_{q \in \mathbf{state}} \delta(q, x_i)$. We say a subset $\mathbf{state}$ is dead if $\mathbf{state} \subseteq \mathrm{dead}$.

We modify Lemma 3.2 as follows. For each state q, we maintain $\delta(q, w) \subseteq \mathcal{Q}$ which is now a subset of states. For any $w = uv$, we have the identities (analogous to Equation (3) and Equation (4)):

$$\delta(q, w) = \bigcup_{a \in \delta(q,u)} \delta(a, v)$$

$$\mathrm{last}(q, w) = \max \left\{ \mathrm{last}(q, u), |u| + \max_{a \in \delta(q,u)} \mathrm{last}(a, v) \right\}.$$

Above, max of an empty set or $\perp$ is $-\infty$. Given $\delta(\cdot, u), \delta(\cdot, v)$ and $\mathrm{last}(\cdot, u), \mathrm{last}(\cdot, v)$ note that $\delta(\cdot, w), \mathrm{last}(\cdot, w)$ can be computed in $\mathrm{poly}(|\mathcal{Q}|) = O(1)$ time. Thus, the amortized time of each operation of the cQueue data structure is still $O(1)$, as desired.

The analyses of correctness and performance follow identically. Note that all of the modifications described incur a $\mathrm{poly}(|R|) = O(1)$ blowup in runtime. This concludes the proof of Lemma B.17. □

Finally, for online BPE we use the algorithm of Jiang & Gong (2026).

Lemma B.18 (Online BPE (Jiang & Gong, 2026))**.** *There is an algorithm* `BPEMerge` *that in* $O(n \log^2 \mathfrak{t})$ *total time and* $O(\mathfrak{m}\mathfrak{t})$ *space incrementally outputs* y *satisfying*

$$\mathrm{Merge}(x[:-C]; M) \preceq y \preceq \mathrm{Merge}_{\preceq}(x; M)$$

for some $C = C(x, M) \leqslant (\mathfrak{m} + 1)\mathfrak{t}$.

Proof of Lemma B.18. We take the algorithm of Jiang & Gong (2026) (see their eager output algorithm and accompanying analysis). Their algorithm has time $O(n \log^2 \mathfrak{t})$ and space $O(\mathfrak{m}\mathfrak{t})$. While they do not explicitly analyze the delay of their algorithm (indeed it does not compute zero-delay online BPE, see Claim B.25), we show that their algorithm has delay at most $(\mathfrak{m} + 1)\mathfrak{t}$.

Let $d(x) \leqslant \mathfrak{t}$ denote the depth of the Aho & Corasick (1975) automaton state after reading x. The Jiang & Gong (2026) algorithm outputs the longest common prefix of $\mathrm{Merge}(x[:-j])$ for $0 \leqslant j \leqslant$

$d(x)$. Every input in this common-prefix computation extends $x[:-\mathfrak{t}]$ so that $\mathrm{Merge}_{\preceq}(x[:-\mathfrak{t}])$ is a prefix of their output. In particular, $\mathrm{Merge}_{\preceq}$ on input x tokenizes all but a $\mathfrak{mt}$ length suffix of x.

Lemma B.23 (Theorem 2 of Berglund & van der Merwe (2023))**.** *For any merge list M and input $x \in \Sigma^*$, we have*

$$\mathrm{Merge}(x[:-C]) \preceq \mathrm{Merge}_{\preceq}(x)$$

where $C := C(x, M) \leqslant \mathfrak{mt}$ and $\mathfrak{t} = \mathfrak{t}(M)$.

The bound on C is tight up to constant factors (see Theorem B.24 for details). We remark that while $C(x, M)$ may vary on input x, it is always upper bounded by $\mathfrak{mt}$. By Lemma B.23, the output emits all but at most $(\mathfrak{m}+1)\mathfrak{t}$ characters. Alternatively, we can also use the algorithm of Mamouras et al. (2026), which obtains similar guarantees. □

This concludes the proof of Theorem 1.2. □

B.3 Space Lower Bounds and Limitations of Composition

We give a lower bound showing that the space usage of Theorem 1.2 is optimal.

Theorem 1.5 (Space Lower Bound)**.** *For every integer f, there exists R where $\mathfrak{b}(R) = 2f$ and online tokenization requires $\Omega(\mathfrak{b}(R))$ bits in space.*

Proof. Fix the alphabet $\Sigma = \{a, b, c\}$. Fix a function f. Let $S^* = \emptyset$,

$$R = (aa \mid bb)^f c \mid a \mid b \mid c$$

and $M = [aa \mapsto A, bb \mapsto B]$.

First, we claim that $\mathfrak{b}(R) = 2f$. Consider any string of the form $(aa \mid bb)^f c \in L(R)$. Then, the only proper prefixes that are in $L(R)$ are a, b which are length $2f$ shorter than the original string. Thus, we have $\mathfrak{b}(R) \geqslant 2f$. To prove the upper bound, we observe that all singletons are in the language, and the longest word in the language is length $2f + 1$.

We now show that online tokenization requires $\Omega(\mathfrak{b}(R))$ space. Consider the following inputs, where x is a string $(aa \mid bb)^f$. Upon seeing this, there are (at least) two possibilities. The input ends, in which case, Split partitions x into single characters, so that $\mathrm{Tok}(x) = x$. On the other hand, if the next character is a c so that the entire string xc becomes one pretoken, we apply the merge rules to the consecutive aa and bb pairs, so the output is $\mathrm{Tok}(xc) \in (A \mid B)^f c$. In particular, since the two outputs have no common prefix we have $\mathrm{Tok}_{\preceq}(x) = \epsilon$. When the next character is a c and the input ends, the algorithm must output a string in $(A \mid B)^f c$. Note that the output is purely a function of the internal state of the algorithm, since the final character c is constant across all 2^f possible inputs. Since there are 2^f such strings, the algorithm requires $\Omega(f)$ space, as desired. □

Using similar ideas, we show that our space dependence on $\mathfrak{m}$ and the bound in Lemma B.23 are both optimal.

Theorem B.24 (BPE Space Bound)**.** *For every pair of functions f, g with $f = \Omega(\log g)$, there exists M where the size of the merge list $\mathfrak{m} = \Theta(f)$ and $\mathfrak{t} = \Theta(g)$ and online BPE merging (and therefore online tokenization) requires $\Omega(\mathfrak{m})$ bits of space. Furthermore, there exists some $x \in \Sigma^*$ where*

$$\mathrm{Merge}_{\preceq}(x) \prec \mathrm{Merge}(x[:-\Theta(\mathfrak{mt})]).$$

Proof of Theorem B.24. We give an instance of BPE, which can easily be extended to a tokenization instance by setting $S^* = \emptyset$ and $R = \Sigma^*$. Let $t = f$. Let $\Sigma = \{a, b, c_0, c_1, \ldots, c_t\}$. We build the list of merges as follows, in order of priority. For $s = \lceil \log g \rceil$ we define

$$aa \mapsto A_1, A_1A_1 \mapsto A_2, \ldots, A_{s-1}A_{s-1} \mapsto A$$
$$bb \mapsto B_1, B_1B_1 \mapsto B_2, \ldots, B_{s-1}B_{s-1} \mapsto B$$

so that with $L = 2^s$ we have $\text{Merge}(a^L) = A$ and $\text{Merge}(b^L) = B$. Next, define the following rules, where the priority order is given first by row and then left-to-right in each row:

$$\begin{aligned} & & Ac_t &\mapsto R_t \\ c_{t-1}A &\mapsto L_{t-1} & Ac_{t-1} &\mapsto R_{t-1} \\ &\dots & &\dots \\ c_1A &\mapsto L_1 & Ac_1 &\mapsto R_1 \\ c_0A &\mapsto L_0. \end{aligned}$$

Analogously, we define $Bc_i \mapsto R'_i$ and $c_iB \mapsto L'_i$, placing each immediately after its corresponding A-rule in the above priority order.

We now describe the inputs. For any $v \in \{a, b\}^t$, we can build the input

$$x = c_0 v_1^L c_1 \dots c_{t-1} v_t^L.$$

Where $v = (v_1, \dots, v_t)$. For $i \in [t]$, we define $X_i = L_{i-1}$ if $v_i = a$ and L'_{i-1} otherwise and $Y_i = R_i$ if $v_i = a$ and R'_i otherwise. Then, consider the ϵ (empty) continuation and continuation $y = c_t$. We have

$$\begin{aligned} \text{Merge}(x) &= X_1 \dots X_t \\ \text{Merge}(xy) &= c_0 Y_1 \dots Y_t. \end{aligned}$$

Thus, $\text{Merge}_{\preceq}(x) = \epsilon \prec c_0 = \text{Merge}(x_{\leqslant 1}) = \text{Merge}(x[: -\Theta(fg)])$. Note $|x| = t(2^s + 1) = \Theta(fg)$. As in Theorem 1.5, after viewing x, the output is $\text{Merge}_{\preceq}(x) = \epsilon$, but the algorithm must produce 2^t distinct outputs after receiving the continuation y. Then, the number of bits of space is at least $\Omega(t) = \Omega(f)$. □

Next, we can extend the example of Theorem B.24 to show that the Online BPE (eager output) algorithm of Jiang & Gong (2026) in fact incurs delay $\Omega(\mathfrak{t})$. Since in our application, delay $O(\mathfrak{mt})$ suffices (and is optimal), we leave as an open question to develop an online BPE algorithm with zero delay.

Claim B.25. *For every function g, there exists M where $\mathfrak{t} = \Theta(g)$ and the online BPE algorithm of Jiang & Gong (2026) incurs delay $\Omega(\mathfrak{t})$.*

We begin by recapping the Aho & Corasick (1975) automaton, a crucial ingredient in describing the Jiang & Gong (2026) algorithm. For any node z in the trie, define $\text{fail}(z)$ to be the longest proper suffix of z that is a trie prefix. Note $\text{fail}(z)$ could be the root corresponding to ϵ. We use strings to denote their corresponding nodes in the trie. Aho and Corasick (Aho & Corasick, 1975) give an efficient linear time algorithm for computing $\text{fail}(z)$ for all $z \in V(\mathcal{T})$ and the Aho-Corasick automaton, denoted $\mathcal{A}(\mathcal{T})$, which has states $V(\mathcal{T})$ and transitions:

$$\delta(z, c) = \begin{cases} zc & zc \in V(\mathcal{T}) \\ \delta(\text{fail}(z), c) & \text{o/w .} \end{cases}$$

That is, transition to the child if it exists, otherwise find the longest suffix of zc (possibly empty) that is a prefix of the trie. The *depth* of a state is the depth of the node in the trie.

Proof of Claim B.25. Let $\Sigma = \{a\}$ and $M = \left\{aa \mapsto A^1, A^1A^1 \mapsto A^2, \dots, A^kA^k \mapsto A^{k+1}\right\}$ for $k = \lceil \log g \rceil$ so that $2g \leqslant \mathfrak{t} = 2^{k+1} \leqslant 4g$. Consider the input $x = a^{2\mathfrak{t}-1}$ so that $\text{Merge}_{\preceq}(x) = A^{k+1}$. Indeed, we have $\text{Merge}_{\preceq}(x_{\leqslant \mathfrak{t}}) = A^{k+1}$.

Note that the depth of the Aho-Corasick automaton on the trie consisting of the strings corresponding to all tokens $\left\{a, A^1, \dots, A^{k+1}\right\}$ after scanning x is $\mathfrak{t}$. Thus, the Jiang & Gong (2026) algorithm outputs the longest common prefix of $\text{Merge}(x), \dots, \text{Merge}(x[: -\mathfrak{t}])$. In particular,

$$\begin{aligned} \text{Merge}(x) &= A^{k+1}A^k \dots A^1 a \\ \text{Merge}(x[: -\mathfrak{t}]) &= A^k \dots A^1 a. \end{aligned}$$

so the longest common prefix is ϵ, the empty output. This fails the requirement for the algorithm to have delay $\mathfrak{t} - 1$, thus establishing that the delay is $\Omega(\mathfrak{t})$. □

So far, we have given an algorithm that is essentially optimal in both time and space. However, the worst-case delay of the algorithm is still $\Omega(\mathfrak{b})$. On the other hand, Theorem 1.1 does not rule out efficient algorithms with delay $O(\mathfrak{m}\mathfrak{t})$, which can be much smaller than $\mathfrak{b}$. Below, we show that this is an artifact of composition: naively composing online algorithms for `Segment`, Split, and Merge cannot achieve online tokenization with delay less than lookback.

Theorem 1.6 (Composition Delay Bound)**.** *For every integer $f \geqslant 2$ there exists alphabet Σ, a total regular expression R over Σ^*, merge list M and some input x where $\mathfrak{b}(R) = f$ and*

$$\mathrm{Merge}_{\preceq}(\mathrm{Split}_{\preceq}(\mathrm{Seg}_{\preceq}(x))) \prec \mathrm{Tok}_{\preceq}(x[:-(f-2)]).$$

Proof of Theorem 1.6. Let $\Sigma = \{a, b\}$. For any function f, we provide an example where composition of segmenting, pretokenization, and merging alone achieve delay at least $f - 1 = \mathfrak{b}(R) - 1$. Let $S^* = \emptyset$, $R = a^f b \mid a \mid b$ and $M = [bb \mapsto B]$. Observe that the lookback is f since the only strict prefix of $a^f b$ that is matched by the regular expression R is a, so the lookback is at least f. To upper bound the lookback, note singletons are accepted and the longest pretoken has length $f + 1$.

Consider the input a^f. Note $\mathrm{Tok}_{\preceq}(a^f) = a^f$ since no pretoken contains bb as a substring and no merge ever occurs. However, $\mathrm{Split}_{\preceq}(a^f) \preceq a$ since it must be a common prefix of

$$\mathrm{Split}(a^f) = a \parallel a \parallel \ldots \parallel a$$
$$\mathrm{Split}(a^f b) = a^f b \parallel .$$

Thus, $\mathrm{Merge}_{\preceq}(\mathrm{Split}_{\preceq}(a^f)) \preceq \mathrm{Merge}_{\preceq}(a) = a \prec aa = \mathrm{Tok}_{\preceq}(aa)$. In particular, the delay is at least $f - 1$, as desired. □

While the above example is not hard for tokenization (in fact, $\mathrm{Tok}_{\preceq}(x) = \mathrm{Tok}(x) = x$ for any x), it shows that any algorithm achieving $o(\mathfrak{b})$ delay against arbitrary pretokenization splitting and BPE merging cannot naively compose algorithms for segmenting, pretokenization, and BPE merging.

C Improved Delay Beyond Composition

We show that a more careful algorithm can achieve delay $O(\mathfrak{m}\mathfrak{t})$ independent of $\mathfrak{b}$. Theorem 1.1 establishes that this delay is optimal for polynomial time algorithms. We begin with some notation and useful properties. Recall that for any string x, $\phi(x)$ is the first token in $\mathrm{Merge}(x)$.

Claim C.1 (Theorem 2 of Berglund & van der Merwe (2023))**.** *Let M be a merge list with $\mathfrak{m}$ pairs and $\mathfrak{t}$ the length of the longest token. For any two input strings x, y, if $x_{\leqslant(\mathfrak{m}+1)\mathfrak{t}} = y_{\leqslant(\mathfrak{m}+1)\mathfrak{t}}$, then $\phi(x) = \phi(y)$.*

Throughout this section, we will view BPE as the following process on an input string x. Every pair of adjacent characters initially has a cut between them, that is, every character begins as a single token. Pretokenization fixes certain cuts in place (i.e. the cuts that represent cuts between pretokens denoted by $\parallel$). For example, the list of pretokens

`"aaa"` $\parallel$ `"bbb"` $\parallel$ `"aaaa"`

can be viewed as list of characters with cuts

$$a \mid a \mid a \parallel b \mid b \mid b \parallel a \mid a \mid a \mid a$$

where $\parallel$ represents fixed cuts given by the pretokenization phase and $\mid$ represents unfixed cuts (or simply cuts) that can be removed by the BPE merge process. BPE merge on each pretoken then removes cuts, i.e. every merge removes a cut between two adjacent tokens. Following the above example, the merge rule $(a, a) \mapsto A$ removes cuts so that the output is

$$aa \mid a \parallel b \mid b \mid b \parallel aa \mid aa$$

corresponding to the output string `"AabbbAA"`.

An ordered pair of tokens (τ, μ) is consistent, denoted $\tau \to \mu$, if

$$\mathrm{Merge}(\mathrm{raw}(\tau)\mathrm{raw}(\mu)) = \tau\mu.$$

We show that consistency extends to sequences of tokens.

Claim C.2. *Let $\tau_1, \dots, \tau_k$ be a sequence of tokens. Then*

$$\mathrm{Merge}(\mathrm{raw}(\tau_1)\dots\mathrm{raw}(\tau_k)) = \tau_1\dots\tau_k \iff \tau_i \to \tau_{i+1}\ \forall\, 1 \leqslant i < k.$$

We say $\tau_1, \dots, \tau_k$ is a consistent sequence of tokens if they satisfy the left hand side. Consequently, for any token μ and consistent sequence of tokens $\tau_1, \dots, \tau_k$, we have $\mu, \tau_1, \dots, \tau_k$ is consistent iff $\mu \to \tau_1$.

Following the above convention of viewing BPE as a process of removing cuts between characters, a sequence of tokens $\tau_1 \dots \tau_k$ can be viewed as the raw string $\mathrm{raw}(\tau_1)\dots\mathrm{raw}(\tau_k)$ with the cuts surviving the BPE merge process exactly denoting the cuts in the final output.

$$\mathrm{raw}(\tau_1)\ |_1\ \mathrm{raw}(\tau_2)\ |_2 \cdots |_{k-1}\ \mathrm{raw}(\tau_k)$$

where $|_i$ denotes the surviving cut between token τ_1 and τ_2.

Proof of Claim C.2. Suppose $\tau_1, \dots, \tau_k$ is consistent. Consider the interval $\mathrm{raw}(\tau_i)\mathrm{raw}(\tau_{i+1})$ for any $1 \leqslant i < k$ with endpoint cuts $|_{i-1}, |_{i+1}$. Observe that for two strings containing this interval, if after applying merge $m \in M$, neither endpoint cut is merged, then the cuts internal to the interval are identical after applying merge m. Then, consider the strings $\mathrm{raw}(\tau_1)\dots\mathrm{raw}(\tau_k)$ and $\mathrm{raw}(\tau_i)\mathrm{raw}(\tau_{i+1})$. Since $\tau_1, \dots, \tau_k$ is consistent, the endpoint cuts $|_{i-1}, |_{i+1}$ are never merged. The same statement holds trivially over the string $\mathrm{raw}(\tau_i)\mathrm{raw}(\tau_{i+1})$. Thus, since the cuts internal to the interval are identical, we have that $\tau_i \to \tau_{i+1}$.

Conversely, suppose every pair $\tau_i \to \tau_{i+1}$ is consistent. If every cut between τ_i, τ_{i+1} is not merged, then we have $\mathrm{Merge}(\mathrm{raw}(\tau_1)\dots\mathrm{raw}(\tau_k)) = \tau_1\dots\tau_k$ by the above argument. Thus, assume some cut between τ_i, τ_{i+1} is merged. Let m be the highest priority rule that removes a boundary cut, and among the boundary cuts removed by m, let $|_i$ be the one of minimum index. We claim that at the instant $|_i$ is removed, neither endpoint cut $|_{i-1}$ nor $|_{i+1}$ has been removed. Indeed, by the choice of m, no rule of higher priority than m removes a boundary cut, and every rule of lower priority is applied only after m; so it suffices to rule out that m itself removes an endpoint cut before $|_i$. For $|_{i-1}$, this holds by the minimality of i. For $|_{i+1}$, note that minimality does not apply, since $i+1 > i$; instead, if m also removes $|_{i+1}$, then it does so only after removing $|_i$, as $|_i$ lies to the left of $|_{i+1}$. Consequently, the cuts internal to the interval $\mathrm{raw}(\tau_i)\mathrm{raw}(\tau_{i+1})$ are identical to those in the Merge process on $\mathrm{raw}(\tau_i)\mathrm{raw}(\tau_{i+1})$ alone, so $|_i$ is removed there as well. This contradicts $\tau_i \to \tau_{i+1}$. □

We now present our algorithm.

Theorem 1.7 (Online Tokenization with Constant Delay)**.** *Let $S^* \subset \Sigma^*$ be a set of special tokens, R a total regular expression over Σ, and M a merge list. There is an online tokenization algorithm with preprocessing time $O(\mathfrak{mt})$, total time $O(n(\log^2 \mathfrak{t} + \log \mathfrak{m} + \log \min(n, \mathfrak{b})))$ and space $O(\min(n, \mathfrak{b}) + \mathfrak{mt})$ with $O(\mathfrak{mt})$ delay.*

Proof of Theorem 1.7. We present an algorithm that computes $(\mathrm{Merge} \circ \mathrm{Split})_{\preceq}$ with delay $O(\mathfrak{mt})$. Following similar arguments to Theorem 1.2, we can obtain an online tokenization algorithm with delay $O(\mathfrak{mt}) + \ell_\infty(S^*) = O(\mathfrak{mt})$, since $\mathrm{Seg}_{\preceq}$ outputs all but the last $\ell_\infty(S^*) = O(1)$ characters of the input. We begin with the following useful data structures.

BPE Queue On any string w, let $\phi(w; M) = \mathrm{Merge}(w; M)_1$ be the first token in $\mathrm{Merge}(w)$. As before, we use $\phi(w)$ when the merge list M is clear.

Lemma 3.4 (BPE Queue)**.** *Given a BPE merge list M, there is a data structure* bpeQ *supporting the following operations on a string w:*

1. *(Push) Append a character c to w i.e. update $w \leftarrow wc$.*
2. *(Pop) Pop w_1, i.e. return w_1 and update $w \leftarrow w_2 \dots w_\ell$.*
3. *(First Token) If $|w| \geqslant 3\mathfrak{mt}$, then compute $\phi(w)$. Otherwise, return $\perp$.*

Furthermore, the data structure has $O(\mathfrak{mt})$ preprocessing, $O(\mathfrak{mt} + |w|)$ space, and supports updates and queries in amortized time $O(\log^2 \mathfrak{t})$.

Note that queries are only required to be correct when $|w|$ is sufficiently large ($\gg \mathfrak{mt}$).

Proof of Lemma 3.4. We require a modification of the incremental algorithm of Jiang & Gong (2026).

Lemma C.3. *Given BPE merge list M, there is an algorithm Φ with $O(\mathfrak{mt})$ preprocessing and space that given any string x, computes $\phi(x_{\geqslant i})$ for all i in $O(|x| \log^2 \mathfrak{t})$ time.*

Note that Jiang & Gong (2026) give an algorithm that computes in $O(n \log^2 \mathfrak{t})$ time the *last* token of every *prefix* of x. In contrast, we compute the *first* token of every *suffix* of x. Since the data structure essentially follows from Jiang & Gong (2026), we defer the proof to Section F.

Finally, we can use a standard trick to convert the batch algorithm into an algorithm with efficient amortized update and query time. We maintain two queues U and V such that the maintained string w satisfies $w = UV$. Initially, both $U, V = \epsilon$. We maintain the invariant that:

$$\text{If } |w| \geqslant 3\mathfrak{mt}, \text{ then } 2\mathfrak{mt} \leqslant |U| \leqslant 3\mathfrak{mt} \text{ and } \phi(U_{\geqslant i}) \text{ is computed for all } i \leqslant |U| + 1.$$

On a push, we push the character to V. On a pop, we pop the character from U if it is not empty. Otherwise, V is not empty and we remove a character from V. After each operation, if $|U| < 2\mathfrak{mt}$ and $|w| \geqslant 3\mathfrak{mt}$, then we have $|V| > \mathfrak{mt}$, in which case we let $U \leftarrow w_{\leqslant 3\mathfrak{mt}}$ and apply the algorithm of Lemma C.3 to U.

We check that the invariant is always maintained. Consider the first time $|w| \geqslant 3\mathfrak{mt}$, at this time we have $|V| = 3\mathfrak{mt}$ and U is empty. Our algorithm detects this, and moves w to U, applying the batch algorithm to U, maintaining the invariant. Subsequently, anytime $|U| < 2\mathfrak{mt}$ and $|w| \geqslant 3\mathfrak{mt}$, we again transfer $w_{\leqslant 3\mathfrak{mt}}$ to U and apply the batch algorithm. Note that $|U| \leqslant 3\mathfrak{mt}$ always and $\phi(U_{\geqslant i})$ is always computed, since whenever we add elements to U, we recompute $\phi(U_{\geqslant i})$ on the whole object U, while popping elements from U ensures that $\phi(U_{\geqslant i})$ is still computed on the remaining elements. Each application of the batch algorithm requires $O(\mathfrak{mt} \log^2 \mathfrak{t})$ time and ends with $|U| = 3\mathfrak{mt}$. Since we do not apply the algorithm again until $|U| < 2\mathfrak{mt}$, or after $\Omega(\mathfrak{mt})$ operations, the amortized time is $O(\log^2 \mathfrak{t})$.

Finally, correctness follows since whenever $|w| \geqslant 3\mathfrak{mt}$, the invariant guarantees $|U| \geqslant 2\mathfrak{mt}$. Claim C.1 then ensures that $\phi(w) = \phi(U)$. □

Incremental BPE We also extend the incremental BPE algorithm of Jiang & Gong (2026).

Lemma 3.3 (Incremental BPE)**.** *Given a BPE merge list M with longest token length $\mathfrak{t}$, there is a data structure* incBPE *that maintains implicitly an append-only history string h, an index $0 \leqslant I \leqslant |h|$, and explicitly string $w = h_{>I}$ under the following operations:*

1. *(Push) append c to $h \leftarrow hc$. Note that we can push $\|$ to denote a pretoken split.*
2. *(Pop) if $I < |h|$, increment I by 1.*
3. *(First Token) if $\mathrm{Merge}(h_{\leqslant I}) \prec \mathrm{Merge}(h)$, compute $\phi(w)$. Otherwise return $\perp$.*

The data structure has $O(\mathfrak{mt})$ preprocessing, space $O(\mathfrak{mt} + |w|)$, worst-case update time $O(\log^2 \mathfrak{t} + \log \mathfrak{m})$ and query time $O(\log \mathfrak{t} + \log |w|)$.

Popping is allowed only when $I < |h|$. First token queries are only required to return $\phi(w)$ when $\mathrm{Merge}(h_{\leqslant I}) \prec \mathrm{Merge}(h)$, i.e. when I is a token boundary in the tokenization of the full implicit hidden string h.

Proof of Lemma 3.3. Jiang & Gong (2026) give an algorithm with $O(\mathfrak{mt})$ preprocessing and space (see Section 5 and 6 of (Jiang & Gong, 2026) and Section F for details) $O(\log^2 \mathfrak{t})$ worst-case update time supporting push operations and last token queries after any number of push operations. For example, given a string h, they can compute the last token of every prefix of h, i.e. the last token of $\mathrm{Tok}(h_{\leqslant i})$ for all i, denoted $\lambda(h_{\leqslant i})$.

We describe how to augment their data structure with first token ϕ queries, as pops can trivially be maintained with a separate counter for I, initialized at 0. Note that Jiang & Gong (2026) provide $\lambda(h_{\leqslant i})$ queries for all i. Let $J = |h|$ denote the total number of characters pushed onto h, including those that have been popped. Consider the tree $\mathcal{T}$ whose nodes are indices $0 \leqslant i \leqslant J$ and node i has parent $p(i) = i - |\lambda(h_{\leqslant i})|$. Intuitively, node i points to its parent, the last character of the token immediately preceding $\lambda(h_{\leqslant i})$. Thus, if we wanted to compute the BPE output of $h_{\leqslant i}$, it suffices to follow the path from i to the root of the tree $\mathcal{T}$.

Let I be the number of characters popped so $I \leqslant J$. Note that I, J naturally correspond to indices in the input string. We abuse notation and let I, J also denote the nodes in the tree $\mathcal{T}$ corresponding to the indices I, J. Our data structure maintains $w = h_{>I}$, and induced forest of $\mathcal{T}$ on the nodes corresponding to indices $I \leqslant i \leqslant J$, which we denote $\mathcal{T}_{\geqslant I}$. Note that for any push, we only need to query $\lambda(h_{\leqslant i})$ from the Jiang & Gong (2026) data structure, and add this node to $\mathcal{T}_{\geqslant I}$ (either to its parent if it is in the tree, or as the root of a new tree otherwise). We maintain $\mathcal{T}_{\geqslant I}$ using standard data structures (for example, an Euler Tour and efficient order queries using Dietz & Sleator (1987); Bender et al. (2002)) so that one can efficiently insert and delete nodes, and support queries of the form is a a descendant of b, and if so, what is the next vertex on that path? This allows us to support first token queries, as we can query whether J is a descendant of I. If so, report $\phi(w)$ as the child of I on the path form I to J (note that this could be J itself), otherwise return $\perp$. Finding the child of I can be performed with binary search in $O(\log |w|)$ time. Overall, this requires space $O(|w| + \mathfrak{mt})$, since we also use the data structure of Jiang & Gong (2026).

□

Next, we give an efficient data structure for computing pretokens. Recall from the proof of Lemma B.17, for an incrementally updated string $x_{\leqslant i}$: $\mathrm{Split}_{\preceq}(x_{\leqslant i}) = Q \parallel x_{b(i)+1} \dots x_{\ell(i)}$ where $\ell(i) = b(i) + \mathrm{last}(\mathrm{start}, x_{b(i)+1} \dots x_i)$ or $\mathrm{Split}_{\preceq}(x_{\leqslant i}) = Q \parallel$ in which case $b(i) = \ell(i) = i$. Here, $b(i)$ indicates the start of the current "live" pretoken in $x_{\leqslant i}$.

Lemma C.4. *Given a total regular expression R, there is a data structure* cutQ *that maintains implicitly an append-only string x (of length i), and index $0 \leqslant \mathrm{out} \leqslant |x|$. Denote*

1. $\mathrm{Split}(x) = z_1 \parallel \dots \parallel z_\ell$ *where* $z_j = x_{r(j-1)+1} \dots x_{r(j)}$ *for* $1 \leqslant j \leqslant \ell$.
2. $\mathrm{Split}_{\preceq}(x) = z_1 \parallel \dots \parallel z_{\mathrm{base}}$ *where* $z_{\mathrm{base}} = x_{b(i)+1} \dots x_{\ell(i)}$.

Then, cutQ *supports the following operations:*

1. *(Push) append c to $x_{\leqslant i} \leftarrow x_{<i}c$.*
2. *(Pop) increment* out *by 1.*
3. *(Cut) if* $\mathrm{out} = i$*, return i, otherwise return $r(j)$ where $r(j-1) \leqslant \mathrm{out} < r(j)$.*
4. *(Base) return $b(i)$.*

Furthermore, the data structure requires $O(1)$ preprocessing, amortized update and query time $O(1)$ and $O(1 + i - \mathrm{out})$ space.

Proof of Lemma C.4. We will maintain a linked list $\mathcal{L} \subseteq \{(r(j-1), r(j))\}_{1 \leqslant j \leqslant \ell}$ that intuitively contains pretokens in $\mathrm{Split}(x_{\leqslant i})$ where $r(j) > \mathrm{out}$. We will also maintain a data structure $\mathcal{A}$ that contains for some subset of $1 \leqslant j \leqslant \ell$, the tuple $r(j-1)$ and $\mathbf{state}(j) := \delta(\mathrm{start}, x_{r(j-1)+1} \dots x_i)$. For the analysis, denote $\mathbf{state}_0(j) = \delta(\mathrm{start}, x_{r(j-1)+1} \dots x_{i-1})$. Let $\mathcal{Q}$ be the state space of the DFA recognizing $L(R)$. We will ensure that $\mathcal{A}$ contains at most $|\mathcal{Q}|$ tuples by ensuring that for every $q \in \mathcal{Q}$, we keep only $(s^*, q) \in \mathcal{A}$ where $s^* = \min\{s \mid (s, q) \in \mathcal{A}\}$. This operation, which can be computed in $O(1)$ time, is denoted $\mathrm{trim}(\mathcal{A})$. At initialization, both $\mathcal{L}, \mathcal{A} \leftarrow \emptyset$. Upon receiving x_i, run the following algorithm.

```
cutQ.push(x_i)
```

1. For $(s, q) \in \mathcal{A}$, update $q \leftarrow \delta(q, x_i)$. $\mathrm{trim}(\mathcal{A})$.
2. Let $X = \{s \mid (s, q) \in \mathcal{A}, q \in \mathcal{P}\}$.
3. If $X \neq \emptyset$, let $s^* = \min X$. Remove all (s, q) from $\mathcal{A}$ with $s > s^*$.
   (a) If $s^* \leqslant \mathrm{out}$, set $\mathcal{L} \leftarrow \{(s^*, i)\}$.
   (b) If $s^* > \mathrm{out}$, remove all $(s, e) \in \mathcal{L}$ where $s \geqslant s^*$ and insert (s^*, i) to $\mathcal{L}$.
4. Otherwise, if $X = \emptyset$, add $(i - 1, i)$ to $\mathcal{L}$ and $(i - 1, \delta(\mathrm{start}, x_i))$ to $\mathcal{A}$.
   (Totality ensures $\delta(\mathrm{start}, x_i) \in \mathcal{P}$.)
5. Remove (s, q) from $\mathcal{A}$ if $q \in \mathrm{dead}$.

$\mathrm{cutQ}.\mathbf{pop}()$

1. Increment $\mathrm{out} \leftarrow \mathrm{out} + 1$.
2. Remove any $(s, e) \in \mathcal{L}$ where $e \leqslant \mathrm{out}$.

We claim that the following hold after each update (push or pop):

1. If $\mathrm{out} = i$, $\mathcal{L} = \emptyset$. Otherwise, $\mathcal{L} = \{(r(j_0 - 1), r(j_0)), \ldots, (r(\ell - 1), r(\ell))\}$ where $r(j_0 - 1) \leqslant \mathrm{out} < r(j_0)$.
2. $\mathcal{A}$ contains for every $q \notin \mathrm{dead}$ such that $s_q = \min\{r(j - 1) \mid \mathbf{state}(j) = q\}$ exists, a tuple (s_q, q), and no other tuples.

Before proving the invariants, we see that this efficiently supports the desired queries. Cut queries can be supported in $O(1)$ time by examining the first element of $\mathcal{L}$ (or returning i if $\mathcal{L}$ is empty). Base queries can be supported by computing $\mu \leftarrow \min\{s \text{ s.t. } (s, q) \in \mathcal{A}\}$ in $O(|\mathcal{A}|) = O(1)$ time. If the set is empty, we set $\mu \leftarrow i$. We claim that $\mu = b(i)$, since

$$\begin{aligned}\mu &= \min\{s \text{ s.t. } (s, q) \in \mathcal{A}\} \\ &= \min\left\{r(j - 1) \text{ s.t. } \delta(\mathrm{start}, x_{r(j-1)+1} \ldots x_i) \notin \mathrm{dead}\right\} = b(i).\end{aligned}$$

When the set is empty, we have $b(i) = \mu = i$, as desired. It remains to prove that the above invariants hold, which we do so by induction. In the base case, before any operations, the invariants are trivially satisfied.

Consider a push operation of x_i. There are two cases: either x_i does not extend any existing pretoken, or it does.

In the former case, we have

$$\mathrm{Split}(x_{\leqslant i}) = z_1 \parallel \ldots \parallel z_\ell \parallel x_i.$$

Observe that $X = \emptyset$ since every $\delta(\mathrm{start}, x_{r(j-1)+1} \ldots x_i) \notin \mathcal{P}$, so we add $(i - 1, i)$ to $\mathcal{L}$ and $(i - 1, \delta(\mathrm{start}, x_i))$ to $\mathcal{A}$. This satisfies the first property.

Towards the second property, consider some $q \notin \mathrm{dead}$ and the minimum $r(j - 1)$ such that $\mathbf{state}(j) = q$. We claim $(r(j - 1), \mathbf{state}(j)) \in \mathcal{A}$. Suppose $\mathbf{state}(j) \notin \mathcal{P}$, which covers all but the new singleton pretoken. Note that for any j, j', if $\mathbf{state}(j) \neq \mathbf{state}(j')$, then $\mathbf{state}_0(j) \neq \mathbf{state}_0(j')$. Thus, at the start of the **push**, the same j must have been the minimum j with state $\mathbf{state}_0(j)$. Thus, we have $(r(j - 1), \mathbf{state}_0(j)) \in \mathcal{A}$ by induction. If $(r(j - 1), \mathbf{state}(j)) \notin \mathcal{A}$, it must have been deleted by $\mathrm{trim}(\mathcal{A})$ since $X = \emptyset$ and Step 3 does not run. This cannot happen, since $r(j - 1)$ is the minimum over all j with $\mathbf{state}(j) = q$. The unique $\mathbf{state}(j) \in \mathcal{P}$ is the new singleton pretoken, which we insert into $\mathcal{A}$.

Conversely, consider a tuple $(s, q) \in \mathcal{A}$. Any tuple in $\mathcal{A}$ is $(r(j - 1), \mathbf{state}(j))$ for some j (since all pretokens from before the **push** survive). Suppose for contradiction that there is some j' with $r(j') < r(j)$ and $\mathbf{state}(j) = \mathbf{state}(j')$. Take the minimum such j'. By the inductive hypothesis,

$(r(j'-1), \mathbf{state}_0(j'))$ was in $\mathcal{A}$ before the push, so that $(r(j'-1), \mathbf{state}(j')) \in \mathcal{A}$ is in $\mathcal{A}$, implying that $(r(j-1), \mathbf{state}(j))$ is removed by $\mathrm{trim}(\mathcal{A})$. This concludes the proof of the second property.

In the latter case, for some $k \geqslant \mathrm{base}$, we have

$$\mathrm{Split}(x_{\leqslant i}) = z_1 \parallel \ldots \parallel z_k \ldots z_\ell x_i.$$

By the choice of k, it is the earliest old pretoken extended by x_i. Since $\mathbf{state}(k) \in \mathcal{P}$, its previous state $\mathbf{state}_0(k)$ is not dead as reading x_i provides a nonempty accepting continuation. We claim that $(r(k-1), \mathbf{state}_0(k))$ is present in $\mathcal{A}$ before the push. Otherwise, the second invariant gives an earlier $j < k$ with $\mathbf{state}_0(j) = \mathbf{state}_0(k)$. Then $\mathbf{state}(j) = \mathbf{state}(k) \in \mathcal{P}$, so x_i would extend the earlier pretoken z_j, contradicting the choice of k. Consequently, $r(k-1) \in X$, and no earlier old pretoken start belongs to X. Thus $s^* = r(k-1)$.

Towards the first property, we see that if $r(k-1) \leqslant \mathrm{out}$, then $\mathcal{L} = \{(r(k-1), i)\}$ satisfies the invariant. Similarly, if $r(k-1) > \mathrm{out}$, we have $\mathcal{L} = \{\ldots, (r(k-1), i)\}$, as desired.

Towards the second property, again consider some $q \notin \mathrm{dead}$ and the minimum j such that $\mathbf{state}(j) = q$. We claim $(r(j-1), \mathbf{state}(j)) \in \mathcal{A}$. If not, following similar arguments as the previous case, it must have been removed either by $\mathrm{trim}(\mathcal{A})$ or Step 3. We have already argued that $\mathrm{trim}(\mathcal{A})$ will not remove the tuple, so we may focus on Step 3. Here, we only remove tuples (s, q) where $s > s^* = r(k-1)$. Since all pretokens have $r(j-1) \leqslant r(k-1)$, this cannot happen either.

Conversely, consider $(s, q) \in \mathcal{A}$. If $(s, q) = (r(j-1), \mathbf{state}(j))$ corresponds to a pretoken in $\mathrm{Split}(x_{\leqslant i})$, a similar argument to above proves that $r(j-1)$ is the minimum among all tuples with state $\mathbf{state}(j)$. However, it may now be the case that (s, q) corresponds to some pretoken z_j with $j > k$. In this case, we have $r(j-1) > r(k-1)$, so that Step 3 removes (s, q), as desired.

Finally, consider a **pop** operation. Note that out that does not affect the second invariant at all, so this remains satisfied. We remove all $(s, e) \in \mathcal{L}$ with $e \leqslant \mathrm{out}$, so that the first interval in $\mathcal{L}$ has $e > \mathrm{out}$, as desired.

Performance It remains to bound the performance. Preprocessing is constant, since we just need to build the DFA and initialize $\mathcal{L}, \mathcal{A}$. All queries require $O(1)$ time, since we ensure $\mathcal{A}$ has at most size $|\mathcal{Q}| = O(1)$. Each **push** and **pop** operation requires $O(1)$ time for all operations that do not involve $\mathcal{L}$, again because $|\mathcal{A}| = O(1)$. To bound the amortized update time, note that each iteration adds at most one interval, and each interval is removed at most once. Finally, we note that the space is $O(\mathcal{L}) = O(1 + i - \mathrm{out})$. □

We now use Lemma 3.4, Lemma 3.3, and Lemma C.4 to prove Theorem 1.7, i.e. give an online tokenization algorithm with constant delay. We begin by describing the data structures our algorithm will use. We initialize $\mathrm{cut}, \mathrm{out} \leftarrow 0$. We ensure that out maintains the last index in the output and cut to maintain the end of the pretoken containing out in $\mathrm{Tok}(x)$, i.e. the current pretoken being output if the input were to immediately terminate.

We maintain a raw string buffer buffer that supports **push** and **pop**. We initialize empty bpeQ and incBPE data structures. We initialize cutQ with $\mathrm{out} \leftarrow 0$ (the same out as the counter). Whenever out is incremented, we accordingly pop from cutQ. When a data structure is used as an input, this denotes reading the string maintained by the structures. When **pop** is supplied with an integer argument p, we **pop** p items. When **push** is supplied with a string, we push the string character by character. For each character x_i in the input, do the following:

`OptimalDelayTokenization`(x_i)

1. Push x_i to bpeQ, cutQ, and buffer.
2. If $\mathrm{cutQ.cut} > \mathrm{cut}$:

   (The current pretoken containing out extends.)

   Push $\mathrm{incBPE}.\mathbf{push}(\mathrm{buffer}.\mathbf{pop}(\mathrm{cutQ.cut} - \mathrm{cut}))$.

   Update $\mathrm{cut} \leftarrow \mathrm{cutQ.cut}$.

3. While $\mathrm{cutQ.base} > \mathrm{out}$:

   (Cannot extend the current pretoken containing out. Process dead pretokens, and find the next live pretoken. Compare to Step 3 of **`FastPretokenization`**.)

   (a) Push $\mathrm{incBPE}.\mathbf{push}(\ \|\ )$.
   (b) While $\mathrm{incBPE} \neq \epsilon$:
   Output $\phi \leftarrow \mathrm{incBPE}.\phi$.
   Update $\mathrm{bpeQ}.\mathbf{pop}(|\phi|)$, $\mathrm{incBPE}.\mathbf{pop}(|\phi|)$, and $\mathrm{out} \xleftarrow{+} |\phi|$.
   (Compute BPE on the current dead pretoken with incBPE.)
   (c) Push $\mathrm{incBPE}.\mathbf{push}(\mathrm{buffer}.\mathbf{pop}(\mathrm{cutQ.cut} - \mathrm{cut}))$.
   (d) Update $\mathrm{cut} \leftarrow \mathrm{cutQ.cut}$.
   (Move the next pretoken to incBPE.)

4. While $\mathrm{incBPE}.\phi = \mathrm{bpeQ}.\phi \neq \perp$:

   (a) Output $\phi \leftarrow \mathrm{incBPE}.\phi$.
   (b) Update $\mathrm{bpeQ}.\mathbf{pop}(|\phi|)$, $\mathrm{incBPE}.\mathbf{pop}(|\phi|)$, and $\mathrm{out} \xleftarrow{+} |\phi|$.

   (When incBPE and bpeQ agree, $\mathrm{incBPE}.\phi$ extends the stable prefix.)

5. If $\mathrm{incBPE} = \epsilon$ and $\mathrm{out} < i$:

   (Load the next pretoken into incBPE.)

   (a) Push $\mathrm{incBPE}.\mathbf{push}(\ \|\ \mathrm{buffer}.\mathbf{pop}(\mathrm{cutQ.cut} - \mathrm{cut}))$.
   (b) Update $\mathrm{cut} \leftarrow \mathrm{cutQ.cut}$ and go to Step 4.

When $\mathrm{out} < i$, write $\mathrm{Split}(x_{\leqslant i}) = Q \parallel z_{-p} \parallel \ldots \parallel z_0 \parallel z_1 \parallel \ldots \parallel z_\ell$ where $x_{b(i)+1} \ldots x_i = z_{-p} \ldots z_\ell$ with $\ell(i) = |Q| + |z_{-p}|$ and $|z_j| = \mathrm{last}(\mathrm{start}, z_j \ldots z_\ell)$. Above, $z_{-p}, \ldots, z_\ell$ are the pretokens in $x_{\leqslant i}$ if the input were to immediately terminate. We use $z_j \ldots z_\ell$ to denote the concatenation of pretokens $z_j, z_{j+1}, \ldots, z_\ell$, so that z_j is indeed the longest accepting prefix in $z_j \ldots z_\ell$ for each $-p \leqslant j \leqslant \ell$. Define $r(-p-1) = b(i)$ and for each $-p \leqslant j \leqslant \ell$, $r(j) = r(j-1) + |z_j|$ so that $z_j = x_{r(j-1)+1} \ldots x_{r(j)}$. Index the pretokens so that $r(-1) \leqslant \mathrm{out} < r(0)$; thus z_0 contains the first un-emitted character. When $\mathrm{out} = i$, no such z_0 is needed.

We now make the following claim at after processing each input x_i:

1. buffer maintains $x_{\mathrm{cut}+1} \ldots x_i$.
2. bpeQ maintains $x_{\mathrm{out}+1} \ldots x_i$.
3. incBPE explicitly maintains $x_{\mathrm{out}+1} \ldots x_{\mathrm{cut}}$. Consequently, $\mathrm{out} \leqslant \mathrm{cut}$.
4. The output y of the algorithm satisfies $\mathrm{raw}(y) = x_{\leqslant \mathrm{out}}$.
5. If $\mathrm{out} < i$, then $r(-1) \leqslant \mathrm{out} < r(0)$ and $\mathrm{cut} = r(0)$. If $\mathrm{out} = i$, then $\mathrm{cut} = i$.
6. $\mathrm{out} \geqslant b(i)$.

Trivially, the conditions are satisfied at initialization. Observe that $\mathrm{cut}, \mathrm{out}$ are monotonically non-decreasing. To prove the first three properties, note that (1) x_i is always pushed onto $\mathrm{buffer}, \mathrm{bpeQ}$, (2) we always push and pop $\mathrm{cutQ.cut} - \mathrm{cut}$ elements from buffer to incBPE and increment cut by the same amount, and (3) we always increment out by $|\phi|$, which is popped from bpeQ and incBPE. Similarly, the fourth property follows as every output is accompanied by a corresponding increment to out.

Next, we argue the fifth property. If $\mathrm{out} = i$, the third property and $\mathrm{cut} \leqslant i$ give $\mathrm{cut} = i$. Otherwise, $r(-1) \leqslant \mathrm{out} < r(0)$ holds by definition. We argue $\mathrm{cut} = r(0)$. After Step 2 and Step 3, $\mathrm{cut} = r(0)$ by correctness of Lemma C.4, since we query $\mathrm{cutQ.cut}$ after any pushes and pops (i.e. increments

to out). In Step 4, we increment out, but as long as incBPE $\neq \epsilon$, we still have out $<$ cut $= r(0)$ by the third property, in which case $r(0)$ does not change and cut does not need to updated. On the other hand, whenever incBPE $= \epsilon$, in Step 5 we recompute cut $\leftarrow$ cutQ.cut $= r(0)$ by the correctness of Lemma C.4.

Finally, we argue out $\geqslant$ cutQ.base $= b(i)$. In Step 3, we already have cutQ.base $= b(i)$. Thus, as long as the while loop terminates (which we will show later), we must have out $\geqslant b(i)$.

We are now ready to prove correctness, and proceed with the inductive claim that the output y satisfies

$$(\mathrm{Merge} \circ \mathrm{Split})_{\preceq}(x_{\leqslant i-3\mathfrak{mt}}) \preceq y \preceq (\mathrm{Merge} \circ \mathrm{Split})_{\preceq}(x_{\leqslant i}).$$

We argue inductively over individual token emissions within each input iteration. Thus, before considering the next emission, we may assume that the output already produced is a stable prefix.

We first justify first-token queries to incBPE. Consider a query with out $<$ cut, and let $w = x_{\mathrm{out}+1} \dots x_{\mathrm{cut}}$ be the string maintained by incBPE. Let a be the position such that incBPE implicit hidden history is $\dots \parallel x_{a+1} \dots x_{\mathrm{cut}}$, possibly followed by a trailing separator at cut. We denote $u = x_{a+1} \dots x_{\mathrm{out}}$.

Observe that both a and out are token boundaries in the tokenization of the current input: a is either a pretoken boundary or a boundary of the already-emitted stable output, and out is the end of that output. The block uw lies within a single pretoken of the current $\mathrm{Tok}(x)$ tokenization: it is loaded within one pretoken, and later longest-match extensions can only preserve or join existing pretokens. Thus a and out are surviving token boundaries within that pretoken, so Claim C.2 gives $\mathrm{Merge}(uw) = \mathrm{Merge}(u) \circ \mathrm{Merge}(w)$. Since the implicit hidden history of incBPE is $\dots \parallel uw$, we have $\mathrm{Merge}(h_{\leqslant \mathrm{out}}) \prec \mathrm{Merge}(h)$ and Lemma 3.3 applies. In particular, since the first token query condition is satisfied,

$$\text{incBPE}.\phi = \phi(x_{\mathrm{out}+1} \dots x_{\mathrm{cut}}). \tag{5}$$

Beginning with soundness, it is clear that if we do not output any tokens after reading x_i, then the output is a stable prefix by induction as

$$y \preceq (\mathrm{Merge} \circ \mathrm{Split})_{\preceq}(x_{<i}) \preceq (\mathrm{Merge} \circ \mathrm{Split})_{\preceq}(x_{\leqslant i}).$$

Suppose a token is output in Step 3 (i.e. cutQ.base $>$ out). By the correctness of Lemma C.4, we have cutQ.base $= b(i)$ so that $b(i-1) \leqslant \mathrm{out} < b(i)$. Following the proof of Lemma B.17, write

$$\begin{aligned}
\mathrm{Split}_{\preceq}(x_{<i}) &= P_{i-1} \parallel x_{b(i-1)+1} \dots x_{\ell(i-1)} \\
\mathrm{Split}_{\preceq}(x_{\leqslant i}) &= P_{i-1} \parallel x_{b(i-1)+1} \dots x_{r'(0)} \parallel x_{r'(0)+1} \dots x_{r'(1)} \parallel \dots \parallel x_{b(i)+1} \dots x_{\ell(i)}
\end{aligned}$$

where for some $t \geqslant 1$, we have $r'(-1) = b(i-1)$, $r'(t) = \ell(i)$, and $r(i) = r(i-1) + \mathrm{last}(\mathrm{start}, x_{r(i-1)+1} \dots x_i)$ for $0 \leqslant i \leqslant t$. Following our above notation, we have $z_{-p} = x_{r'(t-1)+1} \dots x_{r'(t)}$. Then, observe

$$\begin{aligned}
\mathrm{Merge}_{\preceq}(\mathrm{Split}_{\preceq}(x_{\leqslant i})) &= \mathrm{Merge}(P_{i-1} \parallel x_{b(i-1)+1} \dots x_{r'(0)} \parallel x_{r'(0)+1} \dots x_{r'(1)} \parallel \dots \parallel) \\
&\quad \circ \mathrm{Merge}_{\preceq}(x_{r'(t-1)+1} \dots x_{r'(t)}) \\
&\preceq (\mathrm{Merge} \circ \mathrm{Split})_{\preceq}(x_{\leqslant i}).
\end{aligned}$$

Consider an iteration of Step 3. Since out $< b(i)$, the current pretoken ends at cut $\leqslant b(i)$, so this pretoken is output by $\mathrm{Split}_{\preceq}$ and is unchanged under every continuation. By (5), the query returns $\phi(x_{\mathrm{out}+1} \dots x_{\mathrm{cut}})$, the next token of the current pretoken. Since the already-emitted output is stable, this is also the next token under every continuation. Thus each emission preserves soundness, and the argument applies again after updating out.

Upon exiting Step 3, we have out $= b(i)$, and the emitted prefix is the following stable prefix:

$$\mathrm{Merge}(P_{i-1} \parallel x_{b(i-1)+1} \dots x_{r'(0)} \parallel x_{r'(0)+1} \dots x_{r'(1)} \parallel \dots \parallel) \preceq (\mathrm{Merge} \circ \mathrm{Split})_{\preceq}(x_{\leqslant i}).$$

Now suppose a token ϕ is emitted in Step 4. Write $z_0 = z_0' z_0''$, where $|z_0'| = \mathrm{out} - r(-1)$. By (5) $\phi = \text{incBPE}.\phi = \phi(z_0'')$. Also, bpeQ maintains $z_0'' z_1 \dots z_\ell$ and $\phi = \text{bpeQ}.\phi \neq \perp$ so $|\text{bpeQ}| \geqslant 3\mathfrak{mt}$.

Fix any continuation z. If no pretoken extends, or the first pretoken that extends occurs strictly after z_0, then z_0 is unchanged. The next token is therefore $\phi(z_0'') = \phi$.

Otherwise, let z_k, with $k \leqslant 0$, be the first pretoken that extends. Because its longer match was unavailable before reading z, the new match ends strictly after position i. Consequently, this pretoken has raw string $z_k \dots z_\ell z'$ for some nonempty prefix $z' \preceq z$. The remaining suffix of z may form further pretokens.

By the inductive stability of the emitted output, the boundary at out survives within this extended pretoken. In particular, we have

$$\mathrm{Merge}(z_k \dots z_\ell z') = \mathrm{Merge}(z_k \dots z_0') \circ \mathrm{Merge}(z_0'' \dots z_\ell z').$$

By Claim C.1,

$$\phi(z_0'' z_1 \dots z_\ell z') = \phi(z_0'' z_1 \dots z_\ell) = \mathrm{bpeQ}.\phi = \phi.$$

Thus ϕ is the next token for every continuation and emitting it preserves soundness.

Finally, we prove completeness, i.e. $(\mathrm{Merge} \circ \mathrm{Split})_{\preceq}(x_{\leqslant i-3\mathfrak{mt}}) \preceq y$. As in the proof of soundness, we note that after exiting Step 3, we have $\mathrm{out} \geqslant b(i)$. We may assume without loss of generality that $\mathrm{out} < i - 3\mathfrak{mt}$, as otherwise Claim B.21 already bounds the delay by $3\mathfrak{mt}$.

Thus $|\mathrm{bpeQ}| = i - \mathrm{out} > 3\mathfrak{mt}$. At termination, Step 5 ensures that incBPE is nonempty, since $\mathrm{out} < i$. Define

$$\theta = \mathrm{incBPE}.\phi = \phi(x_{\mathrm{out}+1} \dots x_{\mathrm{cut}}), \qquad \tau = \mathrm{bpeQ}.\phi = \phi(x_{\mathrm{out}+1} \dots x_i).$$

The first equality is justified by (5). Both queries are non-bottom, so termination of Step 4 implies $\theta \neq \tau$.

Since $b(i) \leqslant \mathrm{out} < i$, there is a live pretoken $x_{b(i)+1} \dots x_{\ell(i)}$ with

$$q = \delta(\mathrm{start}, x_{b(i)+1} \dots x_i) \notin \mathrm{dead}.$$

Choose a nonempty continuation z such that $\delta(q, z) \in \mathcal{P}$. Note that all pretokens preceding $b(i)$ are dead, while $x_{b(i)+1} \dots x_i z$ is accepted as a single pretoken under the continuation z.

For the empty continuation, the next token after the already-emitted output is θ. For continuation z, the boundary at out survives by soundness. The next token is

$$\phi(x_{\mathrm{out}+1} \dots x_i z) = \phi(x_{\mathrm{out}+1} \dots x_i) = \tau,$$

where the first equality follows from Claim C.1. Hence, if P denotes the already emitted output,

$$P \circ \theta \preceq \mathrm{Merge}(\mathrm{Split}(x_{\leqslant i})), \qquad P \circ \tau \preceq \mathrm{Merge}(\mathrm{Split}(x_{\leqslant i} z)).$$

Because $\theta \neq \tau$, these two tokenizations have longest common prefix exactly P. Together with soundness, this proves

$$P = (\mathrm{Merge} \circ \mathrm{Split})_{\preceq}(x_{\leqslant i}).$$

In particular,

$$(\mathrm{Merge} \circ \mathrm{Split})_{\preceq}(x_{\leqslant i-3\mathfrak{mt}}) \preceq P$$

as required.

Termination Condition. It remains to prove that the algorithm terminates on each iteration. Step 3 terminates since out increases in every iteration of the while loop. In particular, since $\mathrm{out} < b(i)$, we have $\mathrm{out} < r(0)$ so that incBPE is not empty. Step 4 terminates as each iteration pops at least 1 character from both incBPE and bpeQ, so the loop must terminate when $|\mathrm{bpeQ}| < 3\mathfrak{mt}$. Step 5 must terminate whenever Step 4 terminates with a mismatch, in which case $\mathrm{incBPE} \neq \epsilon$. Otherwise, we increment cut until eventually $\mathrm{out} = i$, and the algorithm terminates.

Performance Finally, we bound the space and runtime of our algorithm. To bound the space, we observe that similar to Lemma B.17, we can bound the space of cutQ by $O(\min(n, \mathfrak{b}) + \mathfrak{mt})$. This follows as

$$\mathrm{out} \geqslant \ell(i) - 3\mathfrak{mt} \geqslant i - \mathfrak{b} - 3\mathfrak{mt}.$$

The first inequality follows from the fact that if $\mathrm{out} < \ell(i) - 3\mathfrak{mt}$, then bpeQ and incBPE share a prefix of length at least $3\mathfrak{mt}$, so that Step 4 will output a token. The second inequality follows from

the definition of $\mathfrak{b}$. Similarly, the space of bpeQ and incBPE is $O(\min(n, \mathfrak{b}) + \mathfrak{mt})$. The space of buffer is $O(i - \text{cut}) = O(\min(n, \mathfrak{b}))$ as $\text{cut} \geqslant \ell(i)$.

To bound the time, our algorithm does $O(1)$ work per input in addition to various calls to the data structures. Combining Lemma 3.3, Lemma 3.4, and Lemma C.4 gives an algorithm with $O(\mathfrak{mt})$ preprocessing and $O(n(\log^2 \mathfrak{t} + \log \mathfrak{m} + \log \min(n, \mathfrak{b})))$ total time. This concludes the proof of Theorem 1.7. □

D Experimental Setup

We compare our end-to-end tokenizer on a selection of encodings from `tiktoken` (OpenAI, 2026) and `tokenizers` (HuggingFace, 2026). Our code can be found in the supplementary material.

We test our tokenizer on similar datasets as Jiang & Gong (2026).[6] In particular, we use the following three datasets:

1. English and Chinese - We utilized the first Parquet shard from the standard training split of the Wikipedia dataset (20231101 dump) (Wikimedia Foundation, 2023). The filtered dataset is obtained by a strided sampling strategy. For English, we keep documents 0, 5, 10, and 16 in every block of 21 documents. For Chinese, we keep documents 0 and 6 in every block of 13 documents.
2. Github - We extracted data from the RedPajama GitHub subset (Together Computer, 2023), specifically using the file with the filename `"filtered_08cdfa755e6d4d89b673d5bd1acee5f6.sampled.jsonl"`. The filtered dataset contains every 17th document.

All experiments were conducted on an Intel Xeon Platinum 8462Y+, comprising 32 cores. All computations were performed exclusively on the CPU and executed on a single core. The system was otherwise idle during benchmarking, with no other user processes competing for CPU resources.

D.1 Applications: R50K, CL100K, and O200K.

We study the full R50K, CL100K, O200K reference patterns below. The three patterns: R50K (used by both R50K and P50K encodings), CL100K, and O200K are given below:

```
R50K = r"'(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+";

CL100K = concat!(
    r"'(?i:[sdmt]|ll|ve|re)|[^\r\n\p{L}\p{N}]?+\p{L}++|\p{N}{1,3}+|",
    r" ?[^\s\p{L}\p{N}]++[\r\n]*+|\s++$|\s*[\r\n]|\s+(?!\S)|\s"
);

O200K = concat!(
    r"[^\r\n\p{L}\p{N}]?[\p{Lu}\p{Lt}\p{Lm}\p{Lo}\p{M}]*[\p{Ll}\p{Lm}\p{Lo}\p{M}]",
    r"+(?i:'s|'t|'re|'ve|'m|'ll|'d)?",
    r"|[^\r\n\p{L}\p{N}]?[\p{Lu}\p{Lt}\p{Lm}\p{Lo}\p{M}]+[\p{Ll}\p{Lm}\p{Lo}\p{M}]",
    r"*(?i:'s|'t|'re|'ve|'m|'ll|'d)?",
    r"|\p{N}{1,3}",
    r"| ?[^\s\p{L}\p{N}]+[\r\n/]*",
    r"|\s*[\r\n]+",
    r"|\s+(?!\S)",
    r"|\s+"
);
```

Each regex is an alternation of several cases. We explain some of the illustrative cases below (focusing on R50K and CL100K, the interested reader is encouraged to investigate in detail with a tool such as `https://regex101.com`).

1. `r"'(?i:[sdmt]|ll|ve|re)"` matches common apostrophed suffixes such as `"'ll"`.
2. `r" ?\p{L}+"` matches consecutive letters with optional leading whitespace, e.g. `" hello"`.

[6] We note that we do not explicitly compare our tokenizer against Jiang & Gong (2026), since their code supports only BPE, whereas our goal is to benchmark the full end-to-end tokenization pipeline.

3. `r" ?\p{N}+"` matches consecutive digits with optional leading whitespace, e.g. `" 123"`. We note that CL100K caps the length of a run of digits at 3.
4. `r" ?[^\s\p{L}\p{N}]+"` matches consecutive symbols with optional leading whitespace e.g. `" !?!?"`.
5. `r"\s+(?!\S)"` matches consecutive white space such that the *next* character is a space or end of input.
6. `r"\s+$"` matches a consecutive whitespace until the end of the file. Throughout, we use $ as a special character to denote EOF.
7. `r"\s*[\r\n]"` matches consecutive whitespace ending in a newline.
8. `r"\s+"` and `r"\s"` matches consecutive whitespace and single whitespace respectively.

In these reference patterns, matching is computed in order of priority, then length. In particular, given a regular expression $R = R_1 \mid R_2 \mid \cdots \mid R_i$ the *first* match of R in x, denoted $F'(x; R)$ is defined as $x_{\leqslant J'(x;R)}$ where

$$J'(x; R) = \max \{j > 0 \mid x_{\leqslant j} \in L(R_i) \text{ and no non-empty prefix of } x \text{ is in } L(R_k) \text{ for } k < i\}.$$

That is, instead of just taking the longest prefix that matches any rule, we take the highest priority rule that has any match with a prefix of x, and the longest match under that rule. We observe that given the left-to-right priority of the alternation rules in the regex pattern, a string is accepted by a rule if (1) it matches the pattern of that rule and (2) it has no prefix that matches any higher priority rule. Note that (online) pretokenization (Definition B.2) is defined identically, replacing F with F_{ref}.

Beyond left-to-right priorities, practical patterns also implement the lookahead operations. In particular, CL100K and O200K contain the rule `r"\s+(?!\S)"` which matches runs of whitespace that are followed by whitespace or end of input (but not runs of whitespace followed by a non-whitespace character). In this case, a DFA may need to scan ahead by one character to accept a prefix. For example, if $x =$ `"\t\t\ta"`, the DFA must scan `"\t\t\t"` before accepting `"\t\t"`. We note that even with the priority rules and lookahead operation the CL100K and O200K examples can still be represented with simple DFAs (see Section F). In our examples, the DFA needs to scan ahead by at most one character.

With both priority and lookahead, let $F_{\text{ref}}(x; R)$ be the prefix of x matched by R incorporating all priority and lookahead rules. Define $J_{\text{ref}}(x; R) = |F_{\text{ref}}(x; R)|$.

In view of priority and lookahead, we give a generalized definition of lookback (Definition B.3).

Definition D.1. *A regular expression R has $\mathfrak{b}_{\text{ref}}$-generalized-lookback if for any $x \in \Sigma^*$ and $a \in \Sigma$, $|J_{\text{ref}}(xa) - J_{\text{ref}}(x)| \leqslant \mathfrak{b}_{\text{ref}}$.*

First, one can see that when R has no priority rules (i.e. just one rule) and no lookahead, $|J_{\text{ref}}(xa) - J_{\text{ref}}(x)| = J(xa) - J(x) \geqslant 0$ is only non-zero when $xa \in L$, in which case $J(x)$ is exactly the longest match among strict prefixes, so that $\mathfrak{b}_{\text{ref}}(R) = \mathfrak{b}(R)$.

For the reference patterns considered here, we adapt the simple backtracking algorithm of Lemma B.16 to compute pretokenization on a pattern with priority rules and one character lookahead. When $\mathfrak{b}_{\text{ref}}(R)$ is finite, a similar proof can show that the structure of the DFAs ensure that each restart reprocesses only $O(\mathfrak{b}_{\text{ref}}(R))$ characters. Charging each restart to progress through the input gives $O(n\mathfrak{b}_{\text{ref}}(R))$ total time. We use this approach in practice because it is simple and avoids the overhead of the more general data structure in Lemma B.17. We further justify this choice by showing that in most practical cases, $\mathfrak{b}_{\text{ref}}(R)$ is small.

Claim D.2. *The* R50K *pattern satisfies* $\mathfrak{b}_{\text{ref}}(\text{R50K}) \leqslant 2$.

Proof of Claim D.2. Appending a character can extend an existing run of letters, digits, or symbols by at 1. For runs of letters and numbers, appending a character cannot lead to a match with a higher priority rule. On the other hand, an initial apostrophe could be extended to a contraction e.g. `"'s"`, which extends the match by at most 1. Finally, note that a character can extend the match to a contraction, for example, $x =$ `"'v"` and $a =$ `"e"`, which can extend the first match by at most 2.

For a run of whitespace, appending a non-whitespace character can shorten the first match by one character because of the lookahead rule, or extend the match to a higher priority rule by at most one

character. Appending a whitespace character on the other hand, can extend the match by at most one character. This exhausts all the cases. □

Claim D.3. *The* CL100K *and* O200K *patterns have unbounded* $\mathfrak{b}_{\text{ref}}$.

Proof of Claim D.3. It suffices to produce a single accepted string where any proper prefix that is also accepted is much shorter.

Consider CL100K. For any $m \geqslant 1$, consider $x =$ `"\n(\t)^{m}"` and a to be any letter so that xa matches the pattern Item 7 and $J_{\text{ref}}(xa) = 1$. On the other hand, x matches the pattern Item 6 and $J_{\text{ref}}(x) = m + 1$. This establishes a gap of size m.

For O200K, we consider a similar string $x =$ `"\n(\t)^{m}"` and $a =$ `"\n"`. Note that xa is accepted by the (7) (i.e. the fifth rule of O200K, but any length > 1 prefix of x is not accepted: the first rule it matches, either (5) or (8), is lower priority than (7), which is matched by the prefix `"\n"`. Thus, $J_{\text{ref}}(xa) - J_{\text{ref}}(x) = m + 1$ as well. □

However, given that CL100K and O200K only have unbounded lookback on relatively rare outputs (basically files that are all whitespace), all three patterns have lookback that are typically much smaller than input size on normal text.

D.2 Optimization

We now describe some optimizations in our code to improve the practical efficiency of our algorithm.

Chunk Size We read input in fixed-size chunks to amortize input overhead, observing that chunks of 64KiB (or larger) are most efficient in practice. The read chunk size controls input batching. For each chunk, the unresolved suffix may contain a candidate pretoken that is retained until it is resolved by reading the next chunk.

Pretokenization Algorithm We remark that in practice, the simple backtracking algorithm for online pretokenization that requires $O(n\mathfrak{b})$ time is more efficient than the $O(n)$ time algorithm given by Lemma B.17. In particular, the overhead of the linear-time algorithm depends on the size of the regular expression, which is typically larger than the overhead incurred by backtracking (since in practice $\mathfrak{b}$ is often very small, see e.g. Claim D.2).

BPE Algorithm. To favor throughput, our implementation invokes the incremental (but not online) BPE algorithm of Jiang & Gong (2026), which only merges after a pretoken boundary is resolved. This is consistent with Jiang & Gong (2026), which observe that the online BPE algorithm is slower in practice. The implementation therefore differs here from the theoretical composition algorithm, which emits accepted prefixes of candidate pretokens to an online BPE merging algorithm. The practical implementation can consequently (in the worst case) retain an arbitrarily long unfinished pretoken and defer applying BPE to merge this pretoken until the pretoken is complete. Its worst-case memory and latency can therefore depend on pretoken length, even for patterns with bounded lookback. We note that for the implemented regular expressions, this scenario typically does not occur on natural language, evidenced by the memory and latency advantages of our implementation.

Linear Buffering Suppose that we have read in a chunk into a buffer, and our online tokenization algorithm has decided that some small prefix of the chunk can be stably tokenized and emitted as output. We now need to remove this prefix on the chunk and recurse the tokenizer on the remaining suffix. Naively draining the buffer (e.g. `std::String::drain`) after each token is emitted requires linear time, since we shift the entire array of characters forwards. Since tokens are typically much shorter than a chunk, this requires time that is quadratic in the chunk size. Instead, whenever we emit a token, we simply advance a counter on the buffer to mark which characters have been successfully processed. At the arrival of a new chunk, we keep the last trailing token (typically quite short) and drain the processed prefix in time linear in the chunk size.

DFA Implementation A key optimization of our code is a faster implementation of the regular expression matching required by pretokenization. Indeed, regex parsing is the dominant cost of tokenization (see OpenAI (2026)). Instead of directly implementing the pretokenization patterns (e.g. R50K, CL100K) as a DFA, we use a combination of (1) a simpler pattern that can be efficiently implemented as a DFA, and (2) an auxiliary `ApplyFixup` function that ensures our algorithm exhibits identical behavior to a DFA directly implemented on the original pattern.

We note that the choice of DFA implementation (either the lazy DFA or dense DFA in `regex_automata`) once the pattern is fixed does not lead to any significant differences in performance. Furthermore, since we work exclusively with total regular expressions, we use anchored DFAs, which ensure that the first character is matched (removing the overhead caused by searching for alternative starting characters). In Section D.3, we present theoretical and empirical evidence that our alternative implementation is in fact equivalent to a direct implementation of the DFA.

Segmenting Finally, we optimize the special token segmenting. Since special tokens typically start with the same character (e.g. <) and this character is relatively rare in natural text, we use `memchr` (BurntSushi, 2026), an optimized library for string search in Rust to efficiently find instances of the starting characters.

D.3 Data Validation: DFA Optimization

For `tiktoken`, we provide theoretical and empirical evidence that our DFA optimizations preserve the same pretoken matching and splitting as the reference patterns. We consider the R50K, CL100K, and O200K, beginning with their simplified forms.

```
R50K_SIMP = r"'(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+";

CL100K_SIMP = concat!(
    r"'(?i:[sdmt]|ll|ve|re)|[^\r\n\p{L}\p{N}]?\p{L}+|\p{N}{1,3}|",
    r" ?[^\s\p{L}\p{N}]+[\r\n]*|\s*[\r\n]|[^\S\r\n]+"
);
```

Similarly to R50K, the simplified O200K pattern is identical to the reference O200K pattern with the `r"\s+(?!\S)"` rule removed. The `ApplyFixup` function is described as follows. Every time the DFA finds a resolved match in the substring (in the language of Lemma B.17, this corresponds to outputting $\|$), we call the `ApplyFixup` function on the candidate pretoken. Note that if we reach the end of the current chunk in buffer and the DFA is not in an $\mathcal{P}$ or dead state, we simply suspend the DFA computation until the arrival of the next chunk and do not call `ApplyFixup`.

`ApplyFixup`$(x, j; R)$

1. **Input:** remaining input x and candidate boundary j.
2. **extend:** if x_j is a newline, $x_{\leqslant j}$ is all whitespace, and the remaining suffix of x through EOF is all horizontal whitespace, then extend the pretoken to the end of the buffer (equivalently return x).
   (extend is only applied for the CL100K pattern.)
3. **trim:** if $j > 1$ and x_j is whitespace, $j < |x|$ and next character x_{j+1} is not whitespace, then return $x_{<j}$.
   (for the CL100K and O200K patterns, we check if x_j is horizontal whitespace.)
4. Otherwise, return $x_{\leqslant j}$.

We now show that the simplified patterns along with `ApplyFixup` as described above is equivalent to the original pattern.

Proof of Equivalence Observe the following crucial property of Split: any algorithm that successfully computes $F_{\text{ref}}(x; R)$ immediately gives an algorithm that computes Split. The proof is essentially immediate from the definition of pretokenization (see Definition B.2). Recall that given a string x and a total expression R, $F_{\text{ref}}(x; R)$ is the first match of R in x starting at the first character. Note $F_{\text{ref}}(\epsilon; R) = \epsilon$.

Claim D.4 (Local Reduction)**.** *Let R be a total regular expression. Suppose there is a function f_0 such that $f_0(x) = F_{\text{ref}}(x; R)$ for all $x \in \Sigma^*$. Then, $f(\epsilon) = \epsilon$ and $f(x) := f_0(x) \parallel f(x_{>|f_0(x)|})$ for $|x| > 0$ computes $\text{Split}(x)$ for all $x \in \Sigma^*$.*

Thus, it suffices to argue that the first match is identical for all strings. Before doing so, we formally describe our algorithm `FastMatch`$(x; R)$. For regular expression R (e.g. R50K), let $\tilde{R}$ denote the simplified pattern.

`FastMatch`$(x; R)$

1. **Input:** string $x \in \Sigma^*$.
2. If $x = \epsilon$, return ϵ.
3. Compute $J_{\text{ref}}(x; \tilde{R})$.
4. Output `ApplyFixup`$(x, J_{\text{ref}}(x; \tilde{R}); R)$.

Proposition D.5. *For all $x \in \Sigma^*$ and pattern R in* R50K*,* CL100K*, and* O200K*, we have*

$$\texttt{FastMatch}(x; R) = F_{\text{ref}}(x; R).$$

Proof of Proposition D.5. We analyze each pattern separately, beginning with R50K.

R50K. Note that the first four rules are identical in the original and simplified patterns. None of these patterns trigger the **trim** subroutine of `ApplyFixup`, since the last character is not a whitespace. Therefore, the returned match is identical, if original pattern first matches one of these rules.

Suppose the fifth rule `r"\s+(?!\S)"` matches in the original pattern. Let w be the maximal initial run of whitespace. If w reaches EOF, both the original and simplified patterns match all of w, and **trim** does not trigger because there is no following character. Otherwise, w is followed by a non-whitespace character. For the fifth rule to match, $|w| \geqslant 2$, and the original pattern matches w without its final character. The simplified pattern matches all of w, and **trim** removes its final character, yielding the same match.

Finally, suppose the sixth rule matches in the original pattern. Since the fifth rule did not match, the initial whitespace run consists of a single character followed by a non-whitespace character. This whitespace cannot be a literal space, since one of the preceding letter, number, or symbol rules would then match. The simplified pattern matches the same single whitespace character, and `trim` does not trigger because the candidate has length one.

CL100K. Consider now CL100K. The first rule is identical in the original and simplified pattern. For the second, third, and fourth rules, we claim that removing the possessive quantifier `"+"` does not change the match. First, a general characterization of when possessive quantifiers change the output of a match.

Claim D.6. *Let A_0, B_0 be disjoint sets of literals in Σ. Consider a regular expression R_1, defined by `r"A++B"` and R_2, defined by `r"A+B"`. If $A \subseteq A_0^*$ and $B \subseteq B_0^*$, then $F_{\text{ref}}(x; R_1) = F_{\text{ref}}(x; R_2)$ for all $x \in \Sigma^*$.*

Proof of Claim D.6. We begin with a brief refresher on possessive quantifiers (see e.g. Oracle (2024); Regular-Expressions.info (2025)). Given a pattern `r"A++B"`, a possessive quantifier tries to (1) find the longest match satisfying `r"A+"`, and (2) the longest match of `r"B"` in the remaining string. If no such match exists, then the possessive quantifier reports that no match is found. In contrast, if no such match exists, the greedy quantifier given by `r"A+B"` backtracks to the next longest prefix matching `r"A+"`, and attempts to match `r"B"` from there. As an example, `r".++x"` would fail to match `"abcx"`, since (1) matches the whole string leaving no characters for (2), whereas `r".+x"` would find a match by backtracking one character.

Suppose for contradiction that $F_{\text{ref}}(x; R_1) \neq F_{\text{ref}}(x; R_2)$ for some x. Let $a_1 = F_{\text{ref}}(x; \texttt{r"A+"})$ be the longest prefix match of `r"A+"` in x. Then, there is no match for `r"B"` in $x \setminus a_1$. On the other

hand, there is a prefix $a_2 \prec a_1$ such that $x \setminus a_2$ contains a match for `r"B"`. In particular, a prefix of $x \setminus a_2$ is a suffix of a_1, a contradiction since `r"A"` and `r"B"` are over disjoint sets of literals. □

Claim D.6 applies to the possessive symbol run followed by the disjoint newline class in the fourth rule. In the second rule, the optional possessive prefix excludes letters, so giving back its character cannot enable the following letter match. The bounded digit pattern in the third rule and the remaining possessive quantifiers in these three rules occur at the ends of their rules, where no subsequent sub-pattern can require backtracking. Thus removing the possessive modifiers preserves the matches of the second, third, and fourth rules.

Consider the fifth rule (6). In this case, the simplified pattern matches (7) or the final case of all horizontal whitespace. In the former case, we trigger **extend** since the match ends with a newline, is all whitespace, and the remaining string is all whitespace, so we extend the pretoken to the whole string. In the latter case, the entire input string is all horizontal whitespace followed by EOF, so that neither **extend** or **trim** trigger, and we return the full matched string, as desired.

For the sixth rule, observe that the sixth rule occurs in the simplified string, and since we did not match the fifth rule, **extend** will not trigger since the remaining string is not all whitespace. Since x_j is not horizontal whitespace, **trim** does not trigger either.

Consider the seventh rule, i.e. a run of whitespace with no newlines (since the sixth rule did not match) such that the following character is a horizontal whitespace (but there is some non-whitespace character afterwards since the fifth rule did not match). In this case, we match the last rule of the simplified pattern (which matches all runs of horizontal whitespace). Since we matched also the trailing horizontal whitespace **trim** will remove this to produce an identical match to the original pattern.

In the final rule, we have a single whitespace character; a similar argument follows as in the last rule of R50K.

O200K. As with R50K, the simplified pattern and the original pattern are identical for the first five rules, so we may assume that the first match does not involve the first five rules. The fifth rule did not match, hence the run contains no newline. In particular, we may assume that the first match is all horizontal whitespace.

Suppose the original pattern matches the sixth rule `r"\s+(?!\S)"`. Let w be the maximal initial run of horizontal whitespace. If w reaches EOF, both patterns match all of w, **trim** does not trigger. Otherwise, w is followed by a non-whitespace character. The sixth rule can match only if $|w| \geqslant 2$, in which case it matches w without its final character. The simplified pattern matches all of w, and **trim** removes its final character, yielding the same match.

Finally, suppose the original pattern matches the seventh rule. Since the sixth rule did not match, the initial run consists of a single horizontal whitespace character followed by a non-whitespace character. The simplified pattern matches this same character, and **trim** does not trigger because the candidate has length one. This concludes the proof. □

Next, we verify the correctness of our algorithm empirically by enforcing equal output with reference tokenizers on test datasets.

Empirical Validation: Equivalence on Datasets Finally, we ensure that our algorithm's output is byte-identical to `tiktoken` and `tokenizers` on the three given datasets (English, Chinese, and Github). We also empirically verify that our implementation is byte-identical to `tiktoken` and `tokenizers` on a large number of short strings.

E Experimental Results

We now present the results of our empirical evaluations.

E.1 Time to First Token

Next, we measure the time to first token (TTFT), a key metric in streaming and online applications. For each `tiktoken` encoding and dataset, we ignore encoder construction time, and measure the time from when the input is provided to the encoder to when the first token is formed. Note that we measure the time to when the first token is formed, not when it is output, since both `tiktoken` and `gigatoken` do not output any tokens until all tokens are formed. For our tokenizer, we provide the input as a file, so that input-reading time is included. For `tiktoken` and `gigatoken`, we provide the string as input in memory, thus excluding input-reading time. We note that `tiktoken` and `gigatoken` do not natively provide support for TTFT profiling: we make minimal changes to their code to allow for this measurement, as the alternative simpler method of measuring TTFT as the total time of encoding excessively advantages our algorithm (see code for details). TTFT results are provided in Table 3.

Table 3: Time to first token (TTFT) of all comparison of all tokenizers on `tiktoken` encodings measured in milliseconds (ms). We report the median TTFT over 30 runs for each row with upper and lower quartiles in the parentheses.

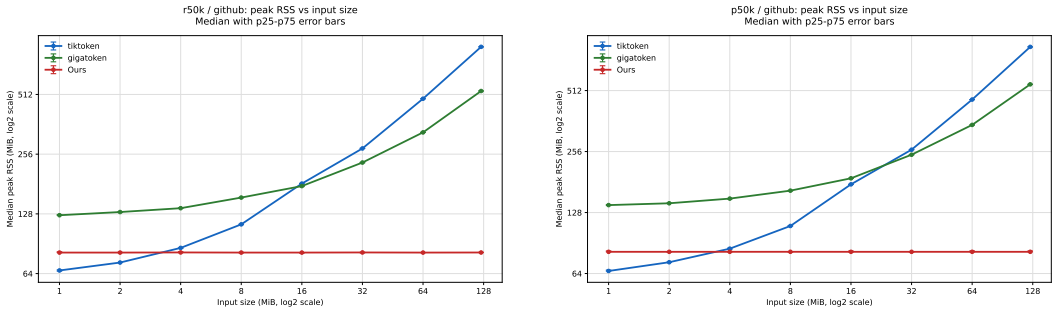
Encoding	**Dataset**	`tiktoken`	`gigatoken`	**Ours**	Speedup
`r50k`	Github	10.91 (10.89-16.77)	10.70 (10.67-10.74)	**0.49** (0.45-0.56)	21.7×
`r50k`	English	10.95 (10.94-10.99)	10.86 (10.85-10.89)	**0.61** (0.51-0.84)	17.8×
`r50k`	Chinese	11.04 (11.00-11.11)	50.24 (46.48-172.62)	**0.77** (0.72-0.85)	14.3×
`p50k`	Github	10.88 (10.86-10.91)	10.70 (10.68-10.72)	**0.56** (0.49-0.61)	19.2×
`p50k`	English	10.95 (10.93-10.99)	10.83 (10.81-10.86)	**0.53** (0.47-0.68)	20.3×
`p50k`	Chinese	11.01 (10.99-11.04)	46.63 (31.03-64.26)	**0.80** (0.72-0.87)	13.8×
`cl100k`	Github	19.93 (19.91-19.96)	18.21 (18.19-18.23)	**0.52** (0.47-0.56)	34.8×
`cl100k`	English	6.44 (6.41-6.46)	6.96 (6.92-7.00)	**0.58** (0.50-0.70)	11.1×
`cl100k`	Chinese	6.51 (6.50-6.54)	236.34 (233.10-238.22)	**0.82** (0.77-0.86)	7.9×
`o200k`	Github	10.62 (10.61-10.67)	18.78 (18.73-18.82)	**0.60** (0.56-0.66)	17.8×
`o200k`	English	10.95 (10.92-10.97)	7.03 (6.96-7.05)	**0.60** (0.51-0.72)	11.8×
`o200k`	Chinese	10.99 (10.98-11.02)	235.69 (94.11-239.96)	**0.88** (0.78-0.90)	12.5×

E.2 Memory

Next, we examine the memory required by each tokenizer. For each encoding and dataset, we measure the peak RSS (Resident Set Size) required by tokenization for input sizes of 1, 2, 4, 8, 16, 32, 64, 128 MiB. We invoke `tiktoken`, `tokenizers` and `gigatoken` through their public APIs, which produce the output as an array in memory. Recall that our algorithm streams an input file of characters to an output file of tokens. The peak RSS is plotted against input size on a log-log scale. Across the evaluated datasets of size 1–128 MiB, our implementation exhibits approximately constant peak RSS, while existing tokenizers show memory growth with input size. Note that on small inputs, our algorithm will incur large RSS relative to `tiktoken` but still smaller than `gigatoken` due to initialization costs.

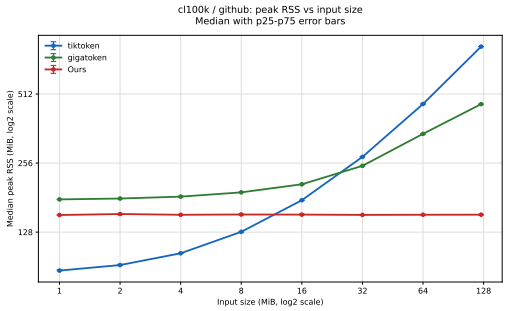
E.3 Throughput Comparison

Finally, we measure throughput. The size of each dataset is approximately 128 MiB. For every dataset and every `tiktoken` encoding, we compare the throughput between `tiktoken` and our tokenizer. For every dataset and every `tokenizers` model, we compare the throughput between `tokenizers` and our tokenizer. For both `tiktoken` and `tokenizers`, input is provided as a string in memory, while output is given as an array in memory. For our algorithm, input is provided as a file path and output is streamed to a temporary file. For the throughput benchmarking, output is streamed to a temporary file (`tmpfs`) backed by memory instead of an actual file on disk. In particular, the cost of reading and writing the input and output file is factored into the throughput of our algorithm. Note that encoder construction time is not timed. We measure performance of the encoding function itself. Table 4 shows the throughput comparison on `tiktoken` encodings and Table 5 shows the throughput comparison on `tokenizers` encodings. Our algorithm significantly

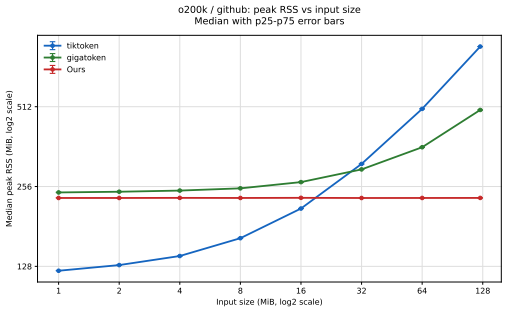


(a) `r50k` peak RSS vs. input size

(b) `p50k` peak RSS vs. input size



(c) `cl100k` peak RSS vs. input size



(d) `o200k` peak RSS vs. input size

Figure 2: Peak RSS vs. input size on Github datasets. The blue line is `tiktoken`, the green line is `gigatoken`, while the red line is our algorithm.

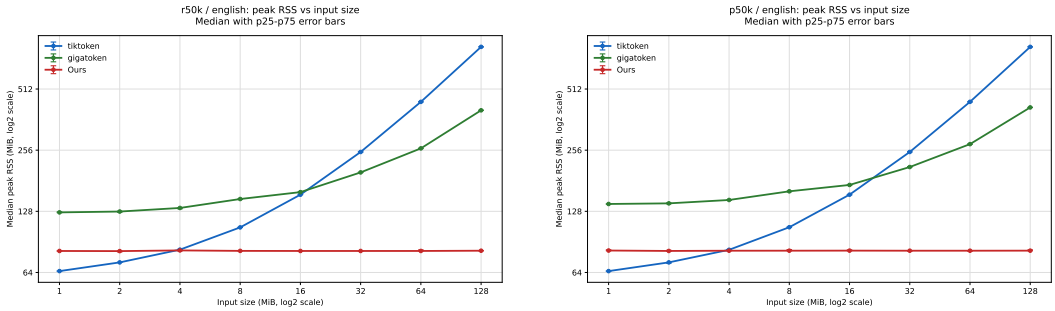
outperforms `tiktoken`, achieving throughput speedups of 2.4×-7.0× depending on the encoding and dataset. We achieve a more dramatic speedup over `tokenizers`, with speedup factors between 15.8×-38.1×.

Table 4: Throughput comparison of our tokenizer on `tiktoken` encodings measured in MiB/s. `tiktoken` is provided with a string as input and gives an array as output, both in memory. Our algorithm is provided with a file path as input and writes output to file. We report the median throughput over 10 runs for each row with upper and lower quartiles in the parentheses.

Encoding	**Dataset**	`tiktoken`	**Ours**	Speedup
`r50k`	Github	11.3 (11.3-11.3)	**43.6** (43.5-43.7)	3.9×
`r50k`	English	17.8 (17.7-17.8)	**47.7** (47.4-48.3)	2.7×
`r50k`	Chinese	15.0 (14.9-15.0)	**38.0** (38.0-38.1)	2.5×
`p50k`	Github	10.8 (10.8-10.8)	**45.0** (44.7-45.2)	4.2×
`p50k`	English	18.0 (17.9-18.0)	**49.2** (48.8-49.4)	2.7×
`p50k`	Chinese	15.0 (14.9-15.0)	**36.6** (36.5-36.7)	2.4×
`cl100k`	Github	11.1 (11.1-11.1)	**48.4** (48.1-48.6)	4.4×
`cl100k`	English	15.5 (15.4-15.6)	**45.3** (44.3-46.4)	2.9×
`cl100k`	Chinese	13.6 (13.6-13.6)	**37.6** (37.5-37.6)	2.8×
`o200k`	Github	7.8 (7.7-7.8)	**54.8** (54.7-55.0)	7.0×
`o200k`	English	8.7 (8.7-8.7)	**41.5** (41.3-41.7)	4.8×
`o200k`	Chinese	9.9 (9.9-9.9)	**36.5** (36.2-36.9)	3.7×

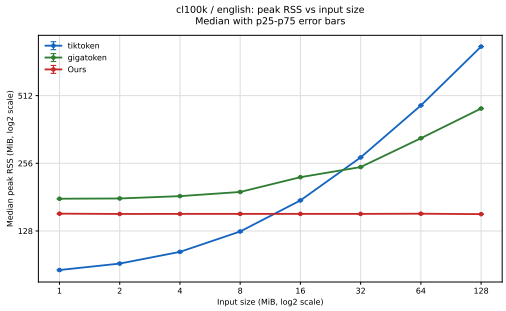
E.3.1 Comparison with `gigatoken`

Recently, the release of the `gigatoken` tokenizer presents a parallelized tokenizer that achieves significant throughput speedups over previous tokenizers (Rød, 2026). At a high level, `gigatoken` uses the following ideas to dramatically speed up tokenization:

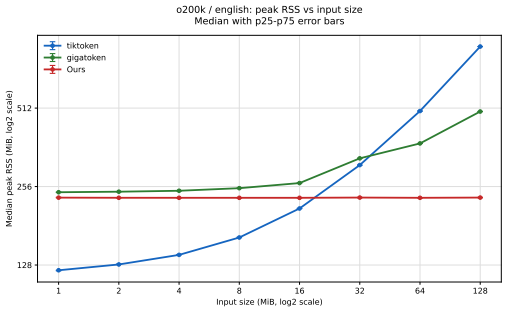


(a) `r50k` peak RSS vs. input size

(b) `p50k` peak RSS vs. input size



(c) `cl100k` peak RSS vs. input size



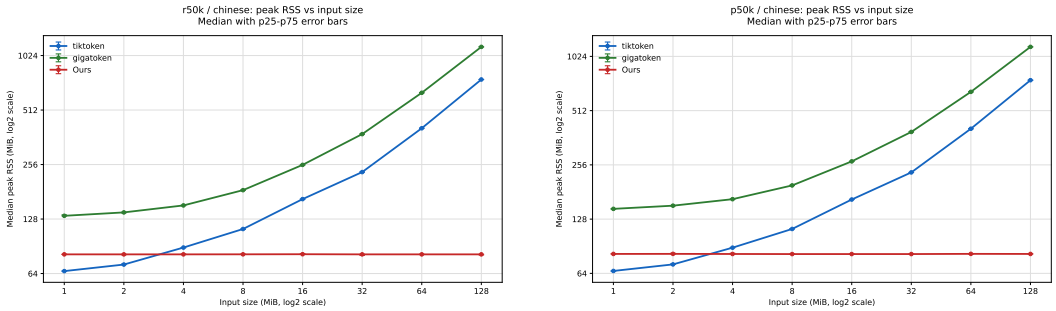
(d) `o200k` peak RSS vs. input size

Figure 3: Peak RSS vs. input size on English datasets. The blue line is `tiktoken`, the green line is `gigatoken`, while the red line is our algorithm.

1. Parallelism - Rød (2026) attempts to process a batch of files or a single file in parallel. In order to break up a single file, one can attempt certain heuristics to split up a large file based on known priors for pretokenization. For example, examining the `tiktoken` patterns, if we find a space between two words (a fairly common occurence), we can safely split a file at this space.
2. Optimized Pretokenization - Using SIMD and many other tricks, Rød (2026) drastically reduces the workload of pretokenization, which is typically deferred to a regex engine.
3. Caching - In order to avoid repeatedly merging the same pretoken, Rød (2026) maintains a cached mapping from pretokens to tokens. While some form of caching is already implemented in `tiktoken`, Rød (2026) uses a heavily optimized cache to improve performance.

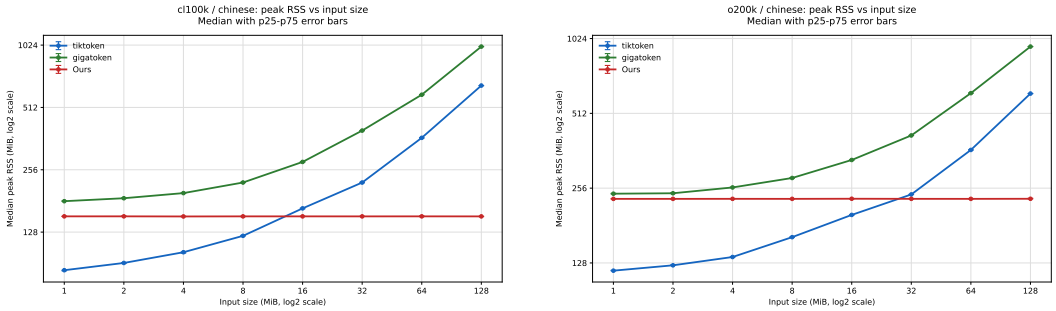
To make a fair comparison with `gigatoken` in the streaming and online settings which are inherently sequential, we compare with the serial implementation of `gigatoken` (the package already provides such an option by setting `parallel=`**`False`**). Furthermore, we note that given the above tricks to split a single file or a batch of files, one can easily obtain a parallel algorithm by applying an optimized serial tokenizer to each part of the resulting split. Regarding the optimized pretokenization and caching, we observe that many optimizations remain compatible with an online framework, particularly composition: one can replace our DFA-based pretokenization algorithm with the SIMD-optimized pretokenization, and one can incorporate an optimized pretoken caching policy into any online BPE algorithm. We note that our original algorithm, identically to `tiktoken`, maintains a cache of all pretokens that map directly to tokens, and thus maintains a cache of approximately size $O(\mathfrak{mt})$. The cache of `gigatoken` instead caches all observed pretokens, thus resulting in a cache that grows with input size in the worst case. In particular, after incorporating the more aggressive caching policy which caches all pretokens, online tokenization will no longer maintain a constant memory footprint independent of input size.

In Table 6, we compare the throughput of `tiktoken`, `gigatoken`, and two implementations of our algorithm with optimized pretokenization and caching from Rød (2026). Both our algorithm and `gigatoken` are provided with a file path as input and supports two output types: an array



(a) `r50k` peak RSS vs. input size

(b) `p50k` peak RSS vs. input size



(c) `cl100k` peak RSS vs. input size

(d) `o200k` peak RSS vs. input size

Figure 4: Peak RSS vs. input size on Chinese datasets. The blue line is `tiktoken`, the green line is `gigatoken`, while the red line is our algorithm.

in memory, and a written output file (for the benchmark, we use a temporary file backed by RAM to avoid blocking on disk). Since `gigatoken` does not support streaming, we write the array to file after output. On array output, we attain similar performance to `gigatoken`, producing lower throughput on English and code datasets, while producing higher throughput on Chinese. The more aggressive caching policy can use increasing memory as additional distinct pretokens are encountered. Our DFA-based implementation voids this growing pretoken cache, but its input buffer can still grow with an unfinished pretoken (as discussed earlier).

F Omitted Proofs

We prove Lemma C.3.

Proof of Lemma C.3. We use the incremental BPE algorithm of Jiang & Gong (2026). Their algorithm computes the last token of every prefix, whereas we require the first token of every suffix. The algorithm and proof are almost identical, with some minor modifications to handle left-to-right tie-breaking in the computation of first token ϕ rather than last token λ.

We recap the algorithm of Jiang & Gong (2026). In preprocessing, they compute the following data structures:

1. A successor forest $\mathcal{F}$ where nodes are tokens τ and edges $(\tau, \mathrm{suc}(\tau))$ where τ is produced by the merge rule $(\mathrm{pre}(\tau), \mathrm{suc}(\tau)) \mapsto \tau$. We call $\mathrm{pre}(\tau), \mathrm{suc}(\tau)$ the predecessor and successor of τ. We replace this with a predecessor forest $\mathcal{F}$ where the nodes are tokens and edges $(\tau, \mathrm{pre}(\tau))$. This requires $O(\mathfrak{m})$ time.

2. For each token τ, compute the suffix-successor tree of τ, which is the induced subgraph of $\mathcal{F}$ containing tokens μ where $\mathrm{raw}(\mu)$ is a suffix of $\mathrm{raw}(\tau)$. On each of these trees, compute a centroid decomposition (using e.g. Della Giustina et al. (2019)). Instead, we replace this with a prefix-predecessor tree, i.e. the induced subgraph of the predecessor

Table 5: Throughput comparison of our tokenizer on `tokenizers` models measured in MiB/s. `tokenizers` is provided with a string as input and gives an array as output, both in memory. Our algorithm is provided with a file path as input and writes output to file. We report the median throughput over 10 runs for each row with upper and lower quartiles in the parentheses.

Model	**Dataset**	`tokenizers`	**Ours**	Speedup
`GPT2`	Github	1.2 (1.2-1.2)	**45.7** (45.5-45.8)	38.1×
`GPT2`	English	1.4 (1.4-1.4)	**50.4** (50.0-51.3)	36.0×
`GPT2`	Chinese	1.2 (1.2-1.2)	**38.6** (38.4-38.6)	32.2×
`RoBERTa`	Github	1.1 (1.1-1.2)	**40.7** (37.5-41.0)	37.0×
`RoBERTa`	English	1.3 (1.3-1.3)	**46.2** (45.7-46.4)	35.5×
`RoBERTa`	Chinese	1.2 (1.2-1.2)	**35.0** (34.7-35.2)	29.2×
`StarCoder`	Github	1.4 (1.4-1.4)	**51.3** (50.8-51.5)	36.6×
`StarCoder`	English	1.3 (1.3-1.4)	**35.8** (35.7-35.8)	27.5×
`StarCoder`	Chinese	1.5 (1.4-1.5)	**32.1** (32.0-32.2)	21.4×
`Mistral`	Github	1.7 (1.6-1.7)	**55.8** (55.4-56.3)	32.8×
`Mistral`	English	1.7 (1.7-1.7)	**47.4** (47.1-47.6)	27.9×
`Mistral`	Chinese	1.7 (1.7-1.7)	**39.3** (39.3-39.4)	23.1×
`Llama4`	Github	1.8 (1.8-1.8)	**56.3** (56.2-56.7)	31.3×
`Llama4`	English	1.7 (1.7-1.7)	**41.1** (40.9-41.4)	24.2×
`Llama4`	Chinese	1.7 (1.7-1.7)	**33.7** (33.5-33.9)	19.8×
`Qwen-3`	Github	1.6 (1.6-1.6)	**30.8** (30.6-31.0)	19.2×
`Qwen-3`	English	1.5 (1.5-1.5)	**29.6** (29.5-30.1)	19.7×
`Qwen-3`	Chinese	1.7 (1.7-1.7)	**26.8** (26.6-26.9)	15.8×
`gpt-oss`	Github	1.9 (1.9-1.9)	**56.6** (56.4-57.1)	29.8×
`gpt-oss`	English	1.8 (1.8-1.8)	**44.4** (44.2-44.7)	24.7×
`gpt-oss`	Chinese	1.9 (1.9-1.9)	**36.8** (36.6-37.0)	19.4×

Table 6: Throughput comparison of optimized tokenizer on `tiktoken` encodings (MiB/s). `gigatoken` and our algorithm are provided with a file path as input. The array (A) column reports throughput when output is collected as array in memory, the stream (S) column reports when output is written to a temporary file. We report the median of 10 runs (quartiles omitted for space).

Encoding	**Dataset**	`gigatoken` (A)	`gigatoken` (S)	**Ours** (A)	**Ours** (S)
`r50k`	Github	178.3	26.9	**185.5**	117.5
`r50k`	English	**331.1**	44.6	298.7	181.9
`r50k`	Chinese	39.9	13.1	**45.3**	37.8
`p50k`	Github	188.7	32.2	**191.5**	126.8
`p50k`	English	**329.2**	44.6	297.6	180.0
`p50k`	Chinese	39.8	13.0	**45.9**	37.8
`cl100k`	Github	**172.2**	40.4	159.7	119.7
`cl100k`	English	**278.3**	44.2	248.3	158.4
`cl100k`	Chinese	27.8	13.5	**43.7**	36.5
`o200k`	Github	**253.4**	43.3	236.6	155.9
`o200k`	English	**240.9**	43.6	216.0	140.8
`o200k`	Chinese	24.0	15.0	**41.2**	36.4

forest $\mathcal{F}$ consisting of tokens μ where $\mathrm{raw}(\mu) \preceq \mathrm{raw}(\tau)$. There are $\mathfrak{m}$ tokens, each with at most $\mathfrak{t}$ prefixes, so this requires $O(\mathfrak{mt})$ time.

3. Jiang & Gong (2026) use an Aho–Corasick automaton to report the longest suffix of $x_{\leqslant i}$ that is a token. Instead, we scan right-to-left using an automaton on reversed token strings to report the longest prefix of $x_{\geqslant i}$ that is a token. This requires $O(\mathfrak{mt})$ preprocessing.

We then process the input string x right-to-left. Assume that $\phi(x_{\geqslant j})$ is computed for $j > i$. We have an analogue of Jiang & Gong (2026) Prefix Last Token condition (see Definition 4.1), which

we call the Suffix First Token condition. That is, τ satisfies the Suffix First Token condition if it is an atomic token (i.e. $|\mathrm{raw}(\tau)| = 1$) or satisfies the following:

1. (Reachability) $\phi(x_{\geqslant i+|\mathrm{pre}(\tau)|})$ is in the subtree of the predecessor forest rooted at $\mathrm{suc}(\tau)$. This ensures that $\mathrm{suc}(\tau)$ is formed in the suffix $x_{\geqslant i+|\mathrm{pre}(\tau)|}$.

2. (Priority Dominance) If $\phi(x_{\geqslant i+|\mathrm{pre}(\tau)|}) \neq \mathrm{suc}(\tau)$, let u be the unique child of $\mathrm{suc}(\tau)$ that lies on the path to $\phi(x_{\geqslant i+|\mathrm{pre}(\tau)|})$. Then, the rule producing τ has priority at least as high as the rule producing u. This ensures that τ is produced before $\mathrm{suc}(\tau)$ is consumed. Note that here we require priority to be at least as high, instead of strictly higher as in Jiang & Gong (2026), due to left-to-right tie-breaking.

Let ρ be the longest token such that $\mathrm{raw}(\rho) \preceq x_{\geqslant i}$. Then $\mathrm{raw}(\phi(x_{\geqslant i})) \preceq \mathrm{raw}(\rho)$. Analogously to Jiang & Gong (2026), one can show that on the prefix-predecessor tree of ρ, a vertex is on the path from x_i to $\phi(x_{\geqslant i})$ iff it satisfies the Suffix First Token condition, which can be checked in $O(1)$ time. The idea then is to start from the root x_i and follow the path of children satisfying the Suffix First Token (SFT) condition. Since we have constructed a centroid decomposition, we begin from the centroid. We first test the centroid's SFT condition. If it is valid, we search for a child satisfying the condition. We identify the valid child by binary search, with children ordered by the left endpoints of their disjoint valid intervals, as in Jiang & Gong (2026), with predecessors and successors interchanged. Here children refer to the original prefix-predecessor tree. If the centroid is valid but has no valid child, return it. Otherwise, either the centroid has a valid child, in which case we remove the centroid and continue at the centroid of the remaining component containing the valid child, or the centroid is invalid, in which case we remove it and continue at the centroid of the remaining component containing its original-tree parent, as in Jiang & Gong (2026). This proceeds for at most $O(\log \mathfrak{t})$ steps, and searching for the right child requires $O(\log \mathfrak{t})$ time, leading to $O(\log^2 \mathfrak{t})$ time. □

We provide the reference and simplified DFA illustrations for R50K, CL100K, and O200K.

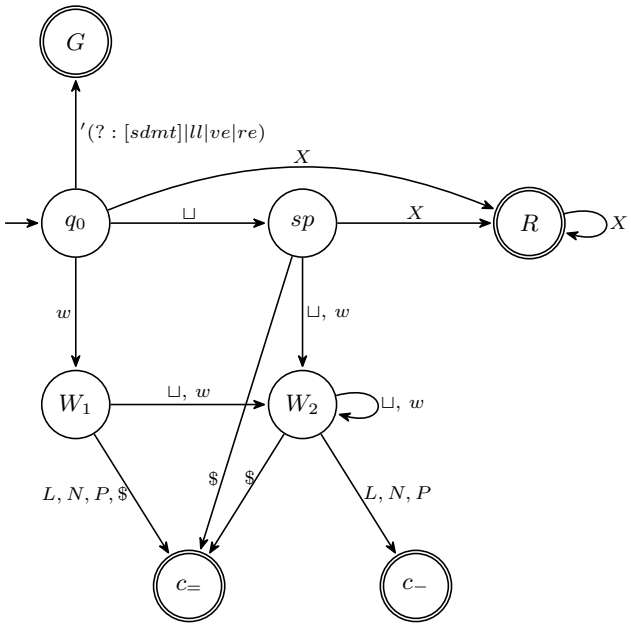


Figure 5: R50K reference pattern as a delayed-accept DFA. X denotes non-whitespace characters as in Figure 6; ␣ = space, w = other whitespace, $ = end of input. The non-whitespace part (q_0, sp, R, G) is an ordinary DFA (boundary unconditional at run/contraction end); the whitespace part $(W_1, W_2, c_=, c_-)$ is a delayed DFA. $c_=$ includes the last ws char; c_- trims it (imposed by the rule $\backslash s^+(?!\backslash S)$). Peeling fires only from W_2, i.e. when the run of whitespace is at least 2 characters long.

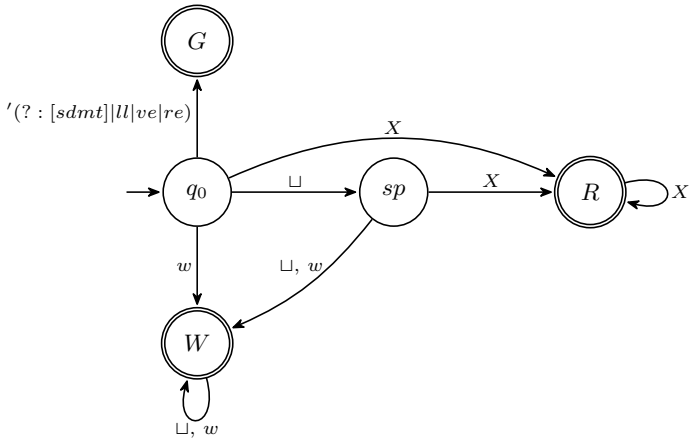


Figure 6: R50K simplified pattern. X denotes non-whitespace characters such as letters, digits, and symbols. In each case, the run continues on the *same* class X; ␣ = literal space, w = other whitespace $(\backslash s \setminus ␣)$. W accepts the maximal $\backslash s^+$ run *unconditionally* (no lookahead).

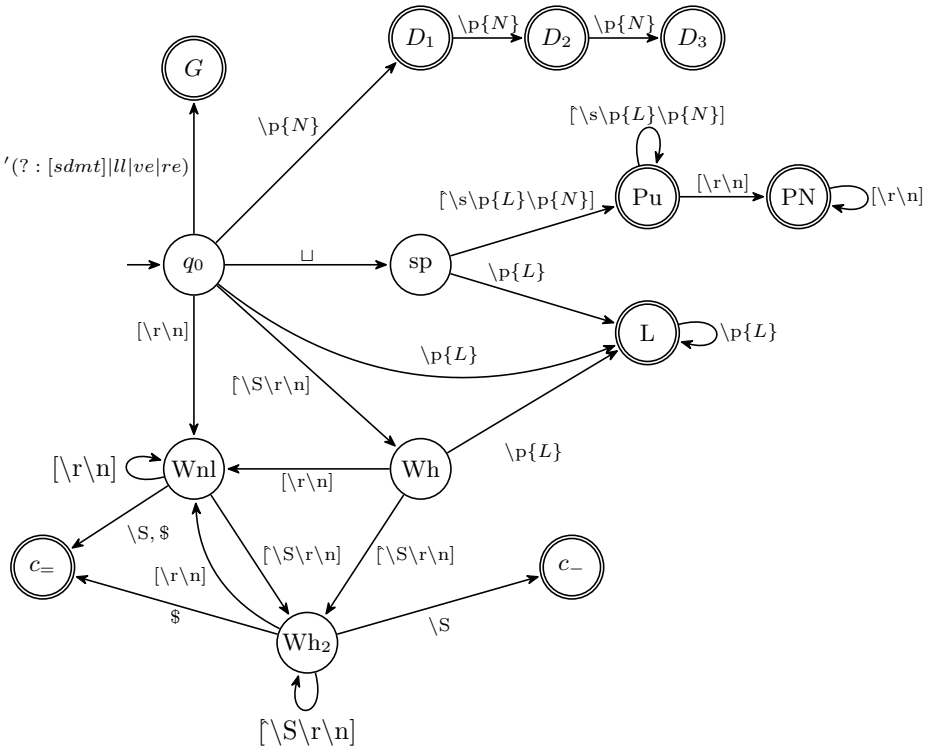


Figure 7: CL100K reference pattern. G denotes the DFA to compute the first rule. Next, L denotes the DFA to compute runs of letters with leading horizontal whitespace or punctuation (e.g. $\lceil\backslash r\backslash n\backslash p\{L\}\backslash p\{N\}\rceil$). D_i denotes the DFA to compute runs of $\leqslant 3$ digits. Pu and PN denote the DFA to compute runs of symbols followed by newlines. Priority: contraction, word, digit, punct, ws. Wh and Wh_2 denote the first and subsequent horizontal whitespace respectively. Wnl denotes vertical whitespace/newline. From Wnl we always keep the last whitespace. From Wh_2, we trim the last character when a non-whitespace is encountered.

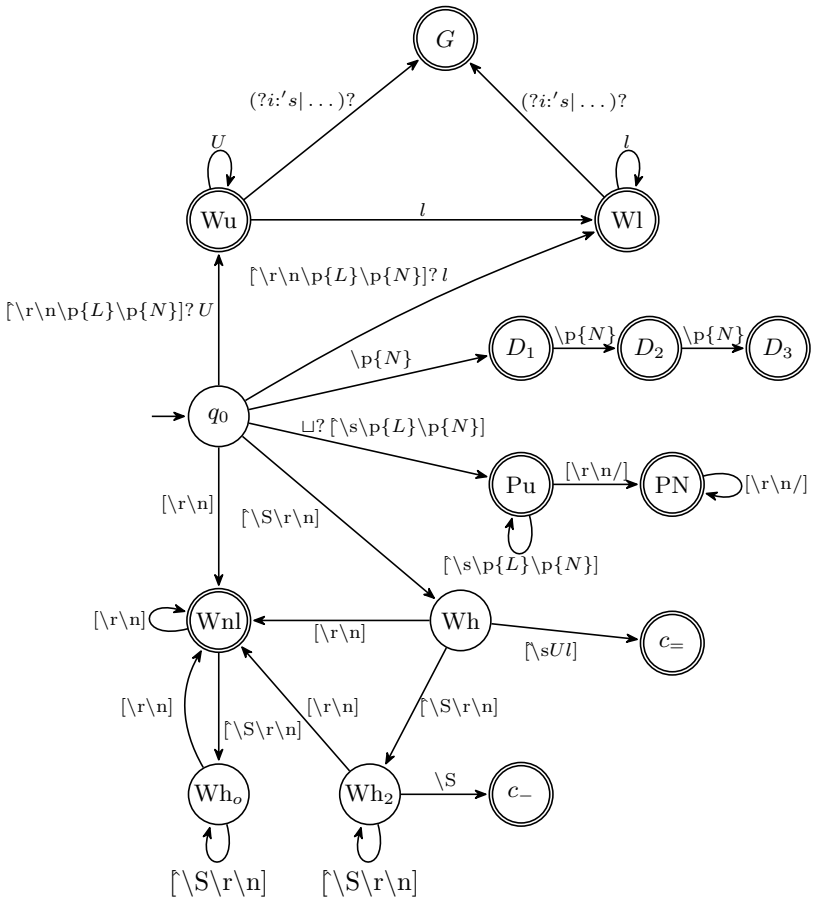


Figure 8: O200K reference pattern. Words are case split, with upper case characters $U = [\p{Lu, Lt, Lm, Lo, M}]$ first, followed by lower case characters $l = [\p{Ll, Lm, Lo, M}]$ with an optional contraction suffix G. Digits have run length $\leqslant 3$. Symbols can have a leading space, followed by an arbitrary number of newlines. Horizontal whitespace followed by letters reaches the first two cases, while followed by any other non-whitespace character immediately outputs the single space. Remaining runs of whitespace either must end with a newline or leave a trailing space before the next whitespace character.

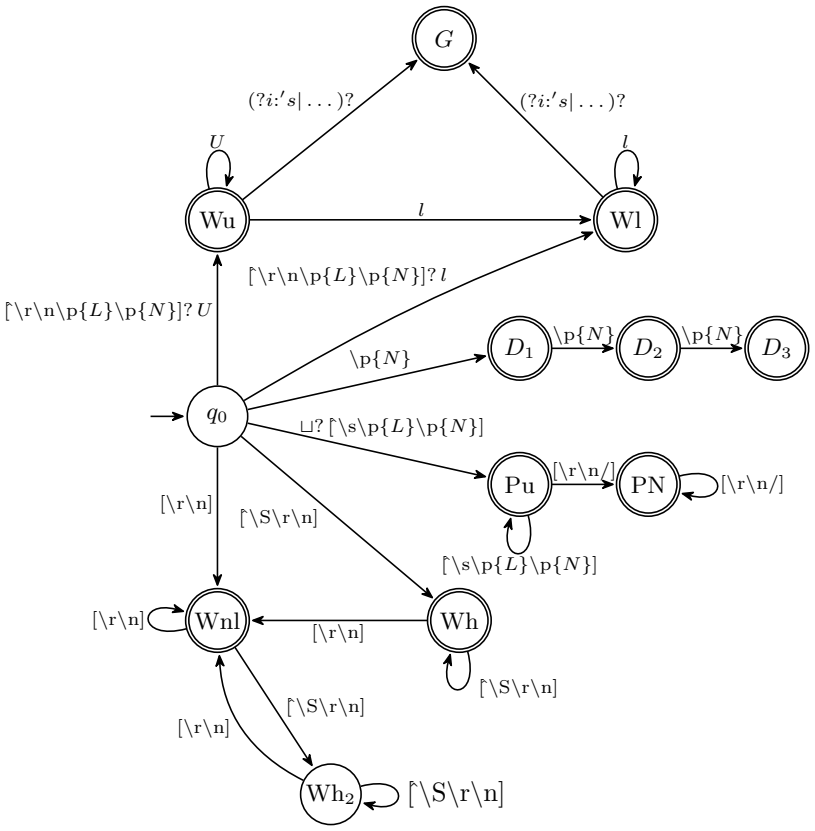


Figure 9: O200K simplified pattern. Most of the DFA (except whitespace handling) is identical to Figure 8. For whitespace, any run of whitespace that contains vertical whitespace/newline must end on vertical whitespace/newline. Otherwise, a run that contains only horizontal whitespace is matched as a single run.