Advanced Algorithms

Lecture Notes

Luca Simonetti

Università degli Studi di Udine · A.A. 2025–2026

Contents

CHAPTER 1

String Matching

1.1 Exact Pattern Matching

1.1.1 Definition

The exact pattern matching problem is defined as follows:

Problem 1.1.1 (Exact pattern matching)**.** Given an alphabet Σ, and two strings $T, P \in \Sigma^*$, for which $|T| = n$ and $|P| = m$, find all occurrences of P in T, i.e. compute $\{\, i \mid T[i,\ i+m-1] = P \,\}$.

Notation

Throughout these notes, Σ denotes a finite alphabet of constant size ($|\Sigma| \in O(1)$). For a string S of length $|S|$:

- $S[i]$ denotes the i-th character of S (1-based indexing).
- $S[i, j]$ denotes the substring $S[i]\,S[i+1]\cdots S[j]$ (empty string if $i > j$).
- $S[,j] = S[1, j]$ is the prefix of length j; $S[i,] = S[i, |S|]$ is the suffix starting at i.

A *proper* prefix (resp. suffix) of S is any prefix (resp. suffix) strictly shorter than S itself.

Example 1.1.2. Let $\Sigma = \{a, b\}$, $T = \texttt{ababbab}$ and $P = \texttt{abb}$. The only occurrence of P in T is at position 3.

1.1.2 Naive approach

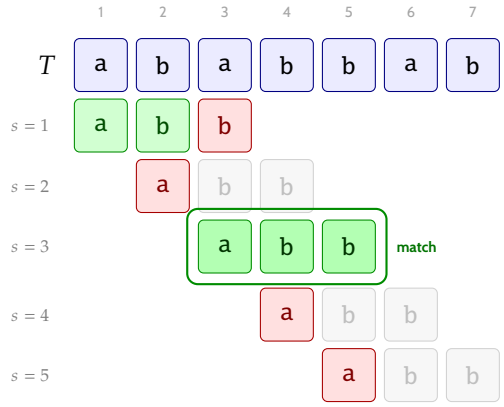


Figure 1.1: Naive exact pattern matching of P = abb in T = ababbab (1-indexed). Each row corresponds to one shift s; the pattern P is aligned starting at $T[s]$. **Green** = character matched, **red** = first mismatch (algorithm skips to next shift), gray = not compared.

What the Figure 1.1 illustrates is one basic naive approach. In this approach we nest two loops: the outer loop iterates over all possible "first positions" s in T where the pattern P could potentially match, and the inner loop checks character by character if P matches T starting from position s. For example the first row of the figure corresponds to starting the search from index $s = 1$ of T. Given s we then loop one by one each character of P, moving along T as well, and check if all the characters match. In this case we find a mismatch at the third character of P (which is b, whereas the corresponding character in T is a). When this happens, we break this inner loop and move to the next shift $s = 2$. If the length of the pattern was m and the length of the text was n, the outer loop could have up to $O(n)$ iterations, and each inner iteration could take up to $O(m)$ time, leading to a worst-case time complexity of $O(nm)$ for this naive approach.

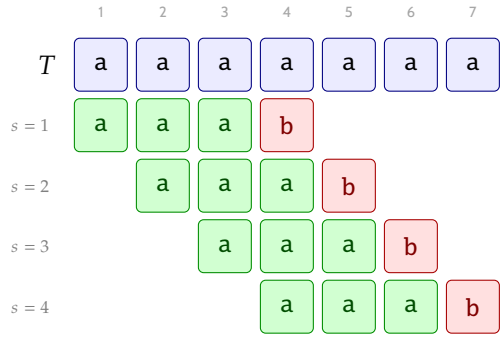


Figure 1.2: Worst-case input for the naive algorithm: T = aaaaaaa ($n = 7$) and P = aaab ($m = 4$). Every shift s performs $m - 1$ successful comparisons before the inevitable mismatch on $P[m]$ = b, giving $(n - m + 1)(m - 1) = \Theta(nm)$ total comparisons.

In figure 1.2 we see an example of a worst-case input for the naive algorithm. In this case the text T consists of n a characters, and the pattern P consists of $m - 1$ a characters followed by a b. There are $n - m + 1$ possible shifts s where

the pattern could potentially match, and for each of these shifts the algorithm will successfully match the first $m-1$ characters of P. In the example of the figure we have $n = 7$ and $m = 4$, so there are $7 - 4 + 1 = 4$ shifts to check, and for each shift we make m checks, of which $m - 1 = 3$ are successful matches before we encounter the mismatch at the last character of P. This results in a total of $(n - m + 1)(m - 1)$ comparisons, which is $\Theta(nm)$ in the worst case.

Algorithm 1: Naive exact pattern matching

Input: Text $T[1 \ldots n]$, Pattern $P[1 \ldots m]$
Output: All positions s such that $T[s, s + m - 1] = P$

```
1 for s ← 1 to n − m + 1 do
2     j ← 1
3     while j ≤ m and T[s + j − 1] = P[j] do
4         j ← j + 1
5     if j > m then
6         output s
```

Correctness

To pin down what "the inner loop does" precisely, let's give a name to the obvious quantity: for any position i in T, write

$$\text{common}(i) \;=\; \max\{\, k \geq 0 \mid T[i, i + k - 1] = P[1, k] \,\}$$

for the length of the longest common prefix of $T[i,]$ and P — basically, how many characters of P would match if you aligned it at position i.

Formal details — Correctness of the naive algorithm

Lemma 1.1.3. *At shift s, the inner loop exits with $j = \text{common}(s) + 1$.*

Proof. Track the invariant: entering each iteration, we have $T[s, s + j - 2] = P[1, j - 1]$ (the first $j - 1$ characters already matched). The loop keeps going as long as the next character also matches; it stops the moment it doesn't, or when j runs past m. Since $\text{common}(s)$ is exactly how many consecutive characters agree, the loop stops right at $j = \text{common}(s) + 1$. □

Theorem 1.1.4. *The naive algorithm finds every occurrence of P in T, and nothing more.*

Proof. The outer loop visits every shift $s \in \{1, \ldots, n - m + 1\}$, so no candidate starting position is ever skipped. By Lemma 1.1.3, after the inner loop we have $j = \text{common}(s) + 1$; the algorithm outputs s exactly when $j > m$, i.e. $\text{common}(s) \geq m$, i.e. $T[s, s + m - 1] = P$ — which is precisely the definition of an occurrence. □

Lower bound

Formal details — Lower bound for exact pattern matching

Theorem 1.1.5. *Any algorithm for exact pattern matching must make $\Omega(n+m)$ character comparisons in the worst case.*

Proof. Two independent arguments, one for each term.

- **You have to read all of** P ($\Omega(m)$). Suppose the algorithm skips position $P[q]$. An adversary flips just that character, producing a different pattern P'. The algorithm sees the same comparisons on both inputs and therefore gives the same output — which can't be correct for both P and P'.
- **You have to read all of** T ($\Omega(n)$). Suppose position $T[i]$ is never read. The adversary flips that character, potentially creating or destroying a match right around position i. Since the algorithm never looked at $T[i]$, it has no way to tell the difference.

The two bounds add up: $\Omega(m) + \Omega(n) = \Omega(n + m)$. □

The naive algorithm achieves $O(nm)$ in the worst case, which is far from this lower bound. KMP will close the gap, achieving $\Theta(n + m)$.

This $\Omega(n + m)$ lower bound applies to any *algorithm, not just naive.*

Properties

We evaluate string matching algorithms along four axes:

Online. The algorithm processes T strictly left-to-right, one character at a time, and can emit results before reading future characters of T. Useful when T arrives as a stream.

Real-time. A stronger requirement: each new character of T is fully processed in $O(1)$ *worst-case* time before the next one is read.

Blind (no backtracking). The algorithm never re-reads a position of T: each character of T is accessed at most once. A blind algorithm is necessarily online; the converse is false (the naive algorithm is online but not blind).

Succinct. The algorithm uses only $O(1)$ extra space beyond storing P and T (no large auxiliary data structures).

Algorithm	Online	Real-time	Blind	Succinct
Naive	✓	×	×	✓
KMP	✓	×	✓	×
KMP (sp')	✓	✓	✓	×

The naive algorithm is **online** (it reads T left-to-right) but not **blind**: at each new shift it backtracks and re-compares characters of T that were already seen. It is also not **real-time**: a single character of T can trigger up to $O(m)$ comparisons.

The naive algorithm backtracks in T: at each new shift s it re-reads characters already compared at shift $s-1$.

This naive approach basically starts over from scratch at each shift, without leveraging precious information from previous comparisons.

1.2 Knuth-Morris-Pratt (KMP) algorithm

Intermezzo — A brief history of KMP

The algorithm is named after Donald Knuth, James Morris, and Vaughan Pratt, who discovered it independently around 1970. Morris came up with the idea while writing a text editor and needed a fast way to search for patterns in a stream of characters. Knuth and Pratt arrived at the same algorithm through purely theoretical analysis — a rare case of practice and theory converging on the same solution simultaneously. The three joined forces, but the paper was not published until 1977 — seven years after the initial discovery.

1.2.1 Intuition

What if we could somehow "remember" the information from a previous comparison, and use it to skip some of the checks at the next shift? Let's introduce the intuition behind the Knuth-Morris-Pratt (KMP) algorithm, with an example.

Example 1.2.1. Let P = `abcdeabdef` and suppose we are running the naive algorithm at shift $s = 3$ over some text T whose characters at positions 3–9 are `abcdeab`. We match $P[1 \ldots 7]$ successfully, then hit a mismatch at $P[8]$ = `d` vs $T[10]$ = `x`.

As shown in Figure 1.3, at this point the naive algorithm would discard everything and try shift $s = 4$. But notice that the matched portion `abcdeab` starts *and* ends with α = `ab`: the prefix $P[1 \ldots 2]$ and the suffix $P[6 \ldots 7]$ of the matched region are identical. This means that when we slide P forward, the α that was at the end of the match is now perfectly aligned with the α at the start of P — we can jump directly to shift $s = 8$ and resume comparing from $P[3]$, skipping five shifts entirely.

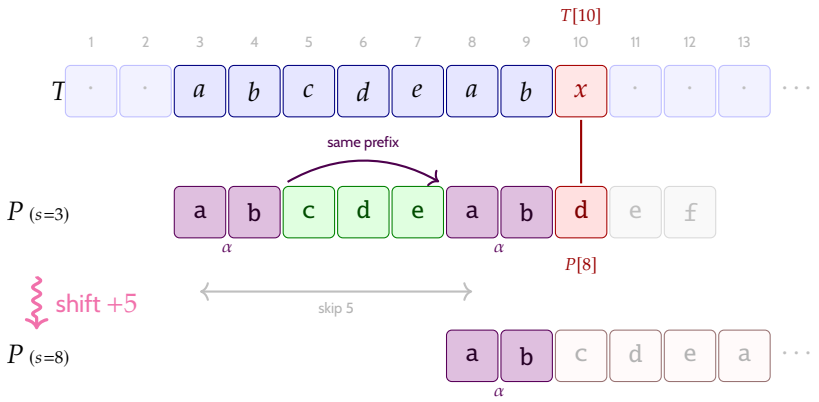


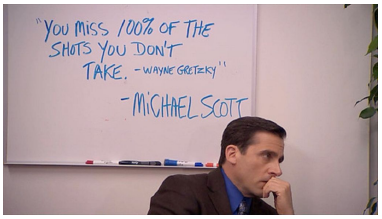
Figure 1.3: KMP shift intuition with P = `abcdeabdef`. At shift $s = 3$, the algorithm matches $P[1 \ldots 7]$ = `abcdeab` then hits a mismatch at $P[8]$. The matched portion ends with α = `ab`, which also appears at the very start of P — so instead of shifting by 1, we jump directly to $s = 8$, re-aligning α and resuming from $P[3]$.

Basically what the example shows is that if we take the longest prefix of P that is also a suffix of the **matched** portion of P, we can shift the pattern so that this prefix is now aligned with the suffix — and we can resume the

*When we say suffix here we mean suffix of the **matched** portion of P, not of P itself. In the example, α is a suffix of $P[1 \ldots 7]$ but not of the whole P.*

search from the character right after this prefix, since we know it must match (otherwise we would have had a mismatch earlier). In other words we have saved some work (namely, five shift indexes) by leveraging the information from the previous comparisons, instead of starting over from scratch, from the very next shift.

But is this correct? Or do we miss some potential matches by jumping directly to $s = 8$?



"You miss 100% of the shots you don't take." — Michael Scott

Well, it turns out that this is indeed correct, and we won't miss any matches by doing this. The KMP algorithm is based on this very idea, and it guarantees that we will find all occurrences of P in T without missing any, while also achieving a much better time complexity than the naive approach.

1.2.2 Preprocessing: computing sp values

To exploit this idea, we need to precompute for every prefix $P[1 \dots i]$ its *longest proper prefix that is also a suffix*. We write this length as $\mathrm{sp}_i(P)$.

A proper *prefix/suffix excludes the string itself;* $\mathrm{sp}_i(P)$ *is thus an integer.*

Definition 1.2.2 (sp values)**.** Let P be a pattern of length m. For each $i \in \{1, \dots, m\}$, define

$$\mathrm{sp}_i(P) \;=\; \max\big\{\, k < i \mid P[1 \dots k] = P[i-k+1 \dots i] \,\big\},$$

i.e. the length of the longest proper prefix of $P[1 \dots i]$ that is also a suffix of $P[1 \dots i]$.

The key observation is that $\mathrm{sp}_i(P)$ can be computed recursively from $\mathrm{sp}_{i-1}(P)$, giving an efficient $O(m)$ preprocessing algorithm. There are two cases:

1. **Extension is possible.** Let $k = \mathrm{sp}_{i-1}(P)$. If $P[k+1] = P[i]$, then the longest prefix-suffix of $P[1 \dots i]$ is simply one character longer than that of $P[1 \dots i-1]$, so $\mathrm{sp}_i(P) = k + 1$.

   Example: with $P =$ `abcdeabdef` at $i = 7$, we have $\mathrm{sp}_6(P) = 1$ (the prefix-suffix of `abcdea` is `a`, length 1). We check $P[2] =$ `b` against $P[7] =$ `b`: they match, so $\mathrm{sp}_7(P) = 2$ (see Figure 1.4).

2. **Extension is not possible.** If $P[k+1] \neq P[i]$, we cannot extend the current prefix-suffix. We then ask: what is the longest prefix-suffix of the prefix-suffix itself? In other words, we fall back to $k \leftarrow \mathrm{sp}_k(P)$ and try to extend that instead. We repeat this process until either a match is found or we reach $k = 0$, in which case $\mathrm{sp}_i(P) = 0$.

   Example: still with $P =$ `abcdeabdef` at $i = 8$, we have $\mathrm{sp}_7(P) = 2$ (the prefix-suffix of `abcdeab` is `ab`). We check $P[3] =$ `c` against $P[8] =$ `d`: no

match. We fall back to $k \leftarrow \mathrm{sp}_2(P) = 0$, so we check $P[1] = \mathtt{a}$ against $P[8] = \mathtt{d}$, still no match. We have reached $k = 0$, so $\mathrm{sp}_8(P) = 0$.

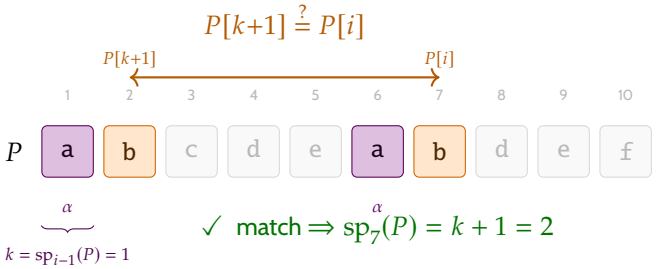


Figure 1.4: Computing $\mathrm{sp}_7(P)$ for $P = \mathtt{abcdeabdef}$. We know $k = \mathrm{sp}_6(P) = 1$: the suffix of $P[1 \ldots 6]$ matching a prefix of P has length 1 (namely $\alpha = \mathtt{a}$, violet). We compare $P[k+1] = P[2] = \mathtt{b}$ against $P[i] = P[7] = \mathtt{b}$ (orange): they match, so $\mathrm{sp}_7(P) = 2$.

The resulting algorithm is the following:

Algorithm 2: Computing sp values

Input: Pattern $P[1 \ldots m]$
Output: Array $sp[1 \ldots m]$ where $sp[i] = \mathrm{sp}_i(P)$

```
1 sp[1] ← 0
2 k ← 0                    // length of current candidate prefix-suffix
3 for i ← 2 to m do
4     while k > 0 and P[k + 1] ≠ P[i] do
5         k ← sp[k]                                   // fall back
6     if P[k + 1] = P[i] then
7         k ← k + 1
8     sp[i] ← k
9 return sp
```

Remark 1.2.3 (Why fall back to $sp[k]$?). At every point in the loop, k has a precise meaning: it is the length of the longest prefix of P that currently matches a suffix of $P[1 \ldots i-1]$. In other words,

$$P[1 \ldots k] \;=\; P[i-k \ldots i-1],$$

the k characters immediately before position i are exactly the first k characters of the pattern. When $P[k+1] = P[i]$ we can extend this match by one: $sp[i] = k+1$. When $P[k+1] \neq P[i]$ the match of length k cannot be extended, and we need a shorter candidate $j < k$ that still satisfies the same property: $P[1 \ldots j] = P[i-j \ldots i-1]$.

Key insight. $P[1 \ldots k]$ is not an arbitrary string — it is a *prefix* of P. Any valid candidate j must be a prefix of P that also appears as a suffix of $P[1 \ldots i-1]$. Two simple observations handle this:

- *Prefix of a prefix is a prefix*: if $P[1 \ldots j]$ is a prefix of $P[1 \ldots k]$, it is a prefix of P.
- *Suffix of a suffix is a suffix*: since $P[1 \ldots k]$ is a suffix of $P[1 \ldots i-1]$ (by the invariant), any suffix of $P[1 \ldots k]$ of length j is also a suffix of

$P[1 \dots i{-}1]$.

Therefore j is a valid candidate *if and only if* $P[1 \dots j]$ is a *border* of $P[1 \dots k]$ (a string that is both a prefix and a suffix of it). The longest such border is $sp[k]$, already computed during preprocessing. Any value strictly between $sp[k]$ and k is, by definition of $sp[k]$, not a border of $P[1 \dots k]$ and therefore not a valid candidate — no check is needed. We jump directly to $k \leftarrow sp[k]$ and repeat. Figure 1.5 shows this chain for $i = 6$, $k = 3$.

Note 1.2.4 (A fallback that succeeds). Let $P = \texttt{ababcababa}$ and consider $i = 10$. We have $k = 4$ since $P[1 \dots 4] = P[6 \dots 9] = \texttt{abab}$. Check $P[5] = \texttt{c}$ vs. $P[10] = \texttt{a}$: mismatch. Could $k' = 3$ be a valid candidate? It would need $P[1 \dots 3]$ to be a border of $\texttt{abab}$, i.e. $\texttt{aba} = \texttt{bab}$: no. The longest border of $\texttt{abab}$ is $\texttt{ab}$ (length 2), so $sp[4] = 2$. We fall back to $k = 2$: $P[1 \dots 2] = P[8 \dots 9] = \texttt{ab}$. Check $P[3] = \texttt{a}$ vs. $P[10] = \texttt{a}$: **match**! Hence $sp[10] = 3$; skipping $k' = 3$ was the only correct move.

Formal details — Why no candidate between $sp[k]$ and k exists

Suppose for contradiction that $sp[k] < k' < k$ and that k' satisfies the same property as k, i.e. $P[1 \dots k'] = P[i - k' \dots i - 1]$. Since $k' < k$, the string $P[i - k' \dots i - 1]$ is a suffix of $P[i - k \dots i - 1] = P[1 \dots k]$. Combined with $P[1 \dots k']$ being a prefix of P, this makes $P[1 \dots k']$ a border of $P[1 \dots k]$ of length $k' > sp[k]$, contradicting $sp[k]$ being the *longest* proper border of $P[1 \dots k]$. □

Formal details — Correctness of the sp preprocessing algorithm

We prove that at the end of iteration i, $sp[i]$ equals the length of the longest proper border of $P[1 \dots i]$.

Precondition	$P[1 \dots m]$ is given; $sp[1] = 0$ and $k = 0$ are set before the loop.
Loop guard	i ranges from 2 to m.
Invariant	At the *start* of iteration i: $k = sp[i{-}1]$, i.e. k is the length of the longest proper border of $P[1 \dots i{-}1]$ (and in particular $P[1 \dots k] = P[i{-}k \dots i{-}1]$).
Postcondition	$sp[i]$ is correctly set for all $1 \le i \le m$.
Base case	Before the first iteration ($i = 2$), $k = 0 = sp[1]$ since $P[1 \dots 1]$ has no proper border. ✓
Preservation	Assume the invariant holds at the start of iteration i, i.e. $k = sp[i{-}1]$. The inner **while** loop traverses the border chain $k \to sp[k] \to sp[sp[k]] \to \cdots$, stopping as soon as $P[k{+}1] = P[i]$ or $k = 0$. By the border characterisation (a candidate j is valid iff $P[1 \dots j]$ is a border of $P[1 \dots k]$), this finds the longest extendable border. The subsequent **if** either increments k by one or leaves $k = 0$. In both cases $sp[i] \leftarrow k$ is correct, and $k = sp[i]$ is the invariant for iteration $i{+}1$. ✓
Postcondition	When the loop exits ($i = m + 1$), all $sp[i]$ have been correctly computed. ✓

Step-by-step: Computing sp values

Let's trace the algorithm on $P = \texttt{ababaca}$.

1. $i = 1$: By definition $sp[1] = 0$. $k \leftarrow 0$.
2. $i = 2$: $k = 0$, $P[1] = \mathsf{a} \neq P[2] = \mathsf{b}$. $sp[2] = 0$.
3. $i = 3$: $k = 0$, $P[1] = \mathsf{a} = P[3] = \mathsf{a}$. $k \leftarrow k + 1 = 1$. $sp[3] = 1$.
4. $i = 4$: $k = 1$, $P[2] = \mathsf{b} = P[4] = \mathsf{b}$. $k \leftarrow k + 1 = 2$. $sp[4] = 2$.
5. $i = 5$: $k = 2$, $P[3] = \mathsf{a} = P[5] = \mathsf{a}$. $k \leftarrow k + 1 = 3$. $sp[5] = 3$.
6. $i = 6$: $k = 3$. The invariant gives $P[1 \ldots 3] = P[3 \ldots 5] = \mathsf{aba}$. We check $P[4] = \mathsf{b}$ against $P[6] = \mathsf{c}$: mismatch, so we fall back. The fallback chain is shown in Figure 1.5:
   - $k \leftarrow sp[3] = 1$: the longest border of aba is a (length 1). Check $P[2] = \mathsf{b}$ against $P[6] = \mathsf{c}$: mismatch.
   - $k \leftarrow sp[1] = 0$: the longest border of a is empty. Check $P[1] = \mathsf{a}$ against $P[6] = \mathsf{c}$: mismatch.
   - $k = 0$: no candidate left. $sp[6] = 0$.

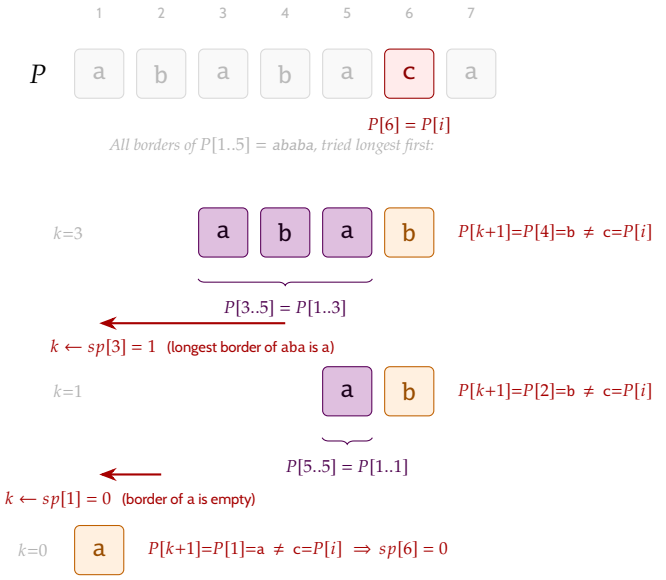


Figure 1.5: Fallback chain at step $i = 6$ for $P = \mathsf{ababaca}$. The three rows show *all* borders of $P[1..5] = \mathsf{ababa}$ (lengths 3, 1, 0), tried longest first. For each candidate k, the violet cells show the matched suffix $P[i-k..i-1] = P[1..k]$; the orange cell is the character $P[k+1]$ we would need at position i to extend it. All three fail, so $sp[6] = 0$.

7. $i = 7$: $k = 0$, $P[1] = \mathsf{a} = P[7] = \mathsf{a}$. $k \leftarrow k + 1 = 1$. $sp[7] = 1$.

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$sp[i]$	0	0	1	2	3	0	1

Remark 1.2.5. The "fall back to the sp value of the current sp value" is the recursive heart of KMP: the same shift logic used during the search phase is also used during preprocessing.

Exercise 1.2.6. Design an algorithm that, given a pattern P of length m, computes $\mathrm{sp}_i(P)$ for every $i \in \{1, \ldots, m\}$ in $O(m)$ time.

Hints:

1. Try to express $\mathrm{sp}_i(P)$ in terms of $\mathrm{sp}_{i-1}(P)$. What is the relationship between the longest prefix-suffix of $P[1 \dots i]$ and that of $P[1 \dots i-1]$?
2. There are two cases depending on whether $P[\mathrm{sp}_{i-1}(P)+1]$ equals $P[i]$ or not. What can you do in each case?
3. For the harder case, think recursively: if you cannot extend a prefix-suffix of length k, which shorter candidate should you try next?

Exercise 1.2.7 (Period of a string). The *period* of a string S of length n is the smallest integer $p \geq 1$ such that $S[i] = S[i+p]$ for all $1 \leq i \leq n-p$. Equivalently, p is the length of the shortest string Π such that S is a prefix of $\Pi^\infty = \Pi\Pi\Pi\cdots$

Design an $O(n)$-time algorithm that, given S, computes its period.

Hints:

1. Compute the sp values of S.
2. What is the relationship between $sp_n(S)$ and the period of S? In particular, is $n - sp_n(S)$ always a period?
3. Is it always the *shortest* period? What additional condition on $sp_n(S)$ guarantees minimality?

Intermezzo — Amortised analysis — the credit method

Some algorithms contain operations that look expensive in isolation but are cheap *on average* over any sequence of calls. *Amortised analysis* makes this precise by distributing the cost of occasional expensive operations across the many cheap ones that made them possible.

A clean way to formalise this is the **credit method**.

Definition 1.2.8 (Credit method)**.** Assign each operation an *amortised cost* $\hat{c}$ (which we choose freely). When an operation with real cost c runs:

- if $\hat{c} > c$, the surplus $\hat{c} - c$ is deposited as *credits* into a bank;
- if $\hat{c} < c$, the deficit $c - \hat{c}$ is withdrawn from the bank.

Invariant: the bank balance must never go negative.

Consequence: since the bank starts at 0 and never goes below 0,

$$\sum_i c_i \;\leq\; \sum_i \hat{c}_i,$$

i.e. the total real cost is bounded by the total amortised cost.

Example: stack with PUSH and MULTI-POP. Consider a stack supporting two operations:

- PUSH(x): push element x onto the stack; real cost 1.
- MULTI-POP(k): pop the top $\min(k, |S|)$ elements; real cost = number of elements actually popped.

A single MULTI-POP can cost $\Theta(n)$, so a naive worst-case analysis gives $O(n^2)$ for n operations. Amortised analysis does much better.

Credit assignment. Attach 1 credit to each element at the moment it is pushed:

- PUSH: amortised cost $\hat{c} = 2$ (pay 1 for the real push, deposit 1 credit *on the element*).
- MULTI-POP(k): amortised cost $\hat{c} = 0$ (use the 1 credit stored on each popped element to pay for its removal).

The bank balance equals the number of elements on the stack, which is always ≥ 0, so the invariant holds.

Example 1.2.9 (Credit trace for a stack)**.** The table below traces the sequence PUSH(a), PUSH(b), PUSH(c), MULTI-POP(2), PUSH(d), PUSH(e), MULTI-POP(3).

Operation	Stack after	Real cost	Amortised cost	Bank
PUSH(a)	$[a]$	1	2	1
PUSH(b)	$[a, b]$	1	2	2
PUSH(c)	$[a, b, c]$	1	2	3
MULTI-POP(2)	$[a]$	2	0	1
PUSH(d)	$[a, d]$	1	2	2
PUSH(e)	$[a, d, e]$	1	2	3
MULTI-POP(3)	$[\,]$	3	0	0
Total		**10**	**10**	

The bank never goes negative, confirming the invariant. The total real cost 10 is indeed $\leq$ the total amortised cost 10.

Lemma 1.2.10. *Any sequence of n PUSH and MULTI-POP operations on an initially empty stack costs $O(n)$ in total.*

Proof. There are at most n pushes, each with amortised cost 2, and all MULTI-POPS have amortised cost 0. The total amortised cost is therefore at most $2n$. By the credit-method bound, the total real cost is at most $2n = O(n)$. □

1.2.3 Application to KMP preprocessing

Formal details — Amortised $O(m)$ analysis of sp preprocessing

The sp-computation algorithm runs in $O(m)$ total time despite the nested loops. We apply the credit method with k as the potential: credits measure how much k can still decrease.

- k increases by at most 1 per outer iteration (there are $m - 1$ iterations, so k grows at most $m - 1$ times in total).
- Each execution of the inner **while** body sets $k \leftarrow sp[k] < k$, strictly decreasing k.
- $k \geq 0$ always holds.

Assign amortised cost 2 to each outer step that increments k (real cost 1 plus deposit 1 credit), and amortised cost 0 to each inner iteration that decrements k (use the saved credit). The bank balance equals $k \geq 0$, so the invariant holds, and the total number of inner iterations is at most the total number of increments, which is $O(m)$. Together with the $O(m)$ overhead of the outer loop, the total preprocessing time is $O(m)$.

Example 1.2.11. Consider the pattern $P = \texttt{aaaaa}$ of length 5. The sp values are $sp[1] = 0$, $sp[2] = 1$, $sp[3] = 2$, $sp[4] = 3$, $sp[5] = 4$. The variable k starts at 0 and increases to 1, then to 2, then to 3, then to 4, as we process each character of P. There are no decreases in this case, so the inner loop runs only once per outer iteration, for a total of $O(m)$ time.

1.2.4 The search phase

Once the sp array is computed, we scan T left-to-right, maintaining j: the number of characters of P currently matched.

Algorithm 3: KMP search

Input: Text $T[1 \ldots n]$, pattern $P[1 \ldots m]$, precomputed $sp[1 \ldots m]$
Output: All starting positions s where P occurs in T

```
1 j ← 0                                  // characters of P matched so far
2 for i ← 1 to n do
3     while j > 0 and P[j + 1] ≠ T[i] do
4         j ← sp[j]                      // fall back along the border chain
5     if P[j + 1] = T[i] then
6         j ← j + 1
7     if j = m then
8         output i − m + 1               // occurrence starting here
9         j ← sp[j]                      // continue searching
```

Figure 1.6 traces the search on a concrete example, showing how sp values let the algorithm carry previously matched characters across fallbacks instead of starting from scratch.

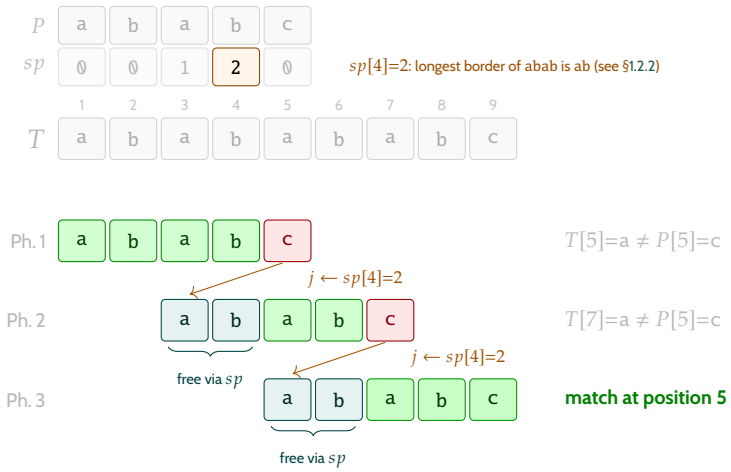


Figure 1.6: KMP search for P=ababc (m=5) in T=abababababc (n=9). The sp array for P is shown at the top (cf. the preprocessing in §1.2.2). Green: characters newly matched in this phase. Teal: characters carried for free from the previous phase via $j \leftarrow sp[j]$. Red: mismatch. After each fallback, the algorithm reuses the last $sp[j]$ matched characters without any comparison, then continues from where it left off in T.

The same amortised argument applies: j increases by at most 1 per outer step and each inner fallback strictly decreases $j \geq 0$, so the inner loop runs

$O(n)$ times in total.

Theorem 1.2.12. *The KMP algorithm finds all occurrences of P in T in $O(n+m)$ time.*

1.3 Z-Algorithm

1.3.1 Definition

The Z-algorithm is a second linear-time preprocessing approach. Where $\mathrm{sp}_i(P)$ looks *backward* (longest prefix that is also a suffix of $P[1 \dots i]$), the Z-value Z_i looks *forward* from position i.

Definition 1.3.1 (Z values)**.** Let A be a string of length n. For $i \geq 2$, define

$$Z_i(A) = \begin{cases} 0 & \text{if } A[i] \neq A[1], \\ \max\{\, \ell > 0 \mid A[1,\ell] = A[i,\, i+\ell-1] \,\} & \text{otherwise.} \end{cases}$$

In words, $Z_i(A)$ is the length of the longest prefix of A that matches A starting at position i, as shown in Figure 1.7.

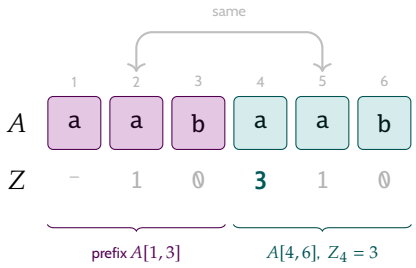


Figure 1.7: Z values for $A = \texttt{aabaab}$ ($n = 6$). The prefix $A[1,3] = \texttt{aab}$ (violet) reappears at position 4 (teal), giving $Z_4 = 3$. Also $Z_2 = Z_5 = 1$ since $A[1] = A[2] = A[5] = \texttt{a}$, and $Z_3 = Z_6 = 0$ since $A[3] = A[6] = \texttt{b} \neq A[1]$. Z_1 is left undefined.

Connection to pattern matching. Choose a separator $\$ \notin \Sigma$ and form $A = P\$T$ (of length $m + 1 + n$). For any index i in the T-part of A (i.e. $i > m + 1$):

$$Z_i(P\$T) = m \quad \Longleftrightarrow \quad P \text{ occurs in } T \text{ starting at position } i - m - 1.$$

The separator prevents Z_i from exceeding m, so equality with m detects all occurrences.

Formal details — Proof of the connection

Write $A = P\$T$ and fix $i > m + 1$ (i.e. i is in the T-part of A).

The separator bounds $Z_i \leq m$. If $Z_i > m$ then $A[i \dots i+m] = A[1 \dots m+1]$, so in particular $A[i+m] = A[m+1] = \$$. But $i + m > 2m + 1 > m + 1$, so $A[i+m]$ lies in the T-part of A, hence $A[i+m] \in \Sigma$. Since $\$ \notin \Sigma$ we have a contradiction: $Z_i \leq m$.

($\Longrightarrow$) Suppose $Z_i = m$. Then $A[i \dots i+m-1] = A[1 \dots m] = P$. Since $A[i+j-1] = T[i-m-1+j-1]$ for $j = 1, \dots, m$, this gives $T[i-m-1 \dots i-1] = P$, i.e. P occurs at position $i - m - 1$ in T.

(⇐) Suppose P occurs at position $j = i - m - 1$ in T, i.e. $T[j \dots j + m - 1] = P$. Then $A[i \dots i + m - 1] = T[j \dots j + m - 1] = P = A[1 \dots m]$, so $Z_i \geq m$. Combined with $Z_i \leq m$ we get $Z_i = m$. □

Example 1.3.2. Let $P = \texttt{aba}$ ($m = 3$) and $T = \texttt{ababac}$ ($n = 6$). Form $A = \texttt{aba\$ababac}$ and compute Z values for positions $i > 4$:

i	1	2	3	4	5	6	7	8	9	10
$A[i]$	a	b	a	$	a	b	a	b	a	c
Z_i	–	0	1	0	**3**	0	**3**	0	1	0

The two positions with $Z_i = 3 = m$ are $i = 5$ and $i = 7$, corresponding to occurrences of P in T starting at positions $5 - 3 - 1 = 1$ and $7 - 3 - 1 = 3$. Indeed $T[1 \dots 3] = \texttt{aba}$ and $T[3 \dots 5] = \texttt{aba}$. ✓

1.3.2 Computing Z values: the Z-box algorithm

A naive computation of all Z values costs $O(n^2)$: for each i, compare character by character until a mismatch.

Intuition: reuse past work. Suppose we have processed positions $2, \dots, i-1$ and at some earlier position l we found $Z_l = z$, i.e. $A[l \dots l+z-1] = A[1 \dots z]$. Call the interval

$$[l,\ r], \quad r = l + z - 1$$

a *Z-box*: a contiguous block that we *know* is an exact copy of the prefix $A[1 \dots z]$.

Now suppose we want Z_i and i falls *inside* the box ($l \leq i \leq r$). Let $k = i-l+1$ be the *mirror* position of i in the prefix. Since the box says $A[l \dots r] = A[1 \dots r-l+1]$, we immediately know

$$A[i] = A[k], \quad A[i+1] = A[k+1], \quad \dots, \quad A[r] = A[r-l+1].$$

Whatever is already recorded in Z_k — the length of the match starting at k in the prefix — is available at i *for free*, without any character comparisons. The only open question is whether the match at i extends beyond r, which requires at most one fresh comparison per step.

The Z-box invariant. The algorithm maintains the Z-box $[l, r]$ with the **largest right endpoint** seen so far:

$$r = l + Z_l(A) - 1 \quad \text{(maximised over all processed positions).}$$

Figure 1.8 shows this setup schematically.

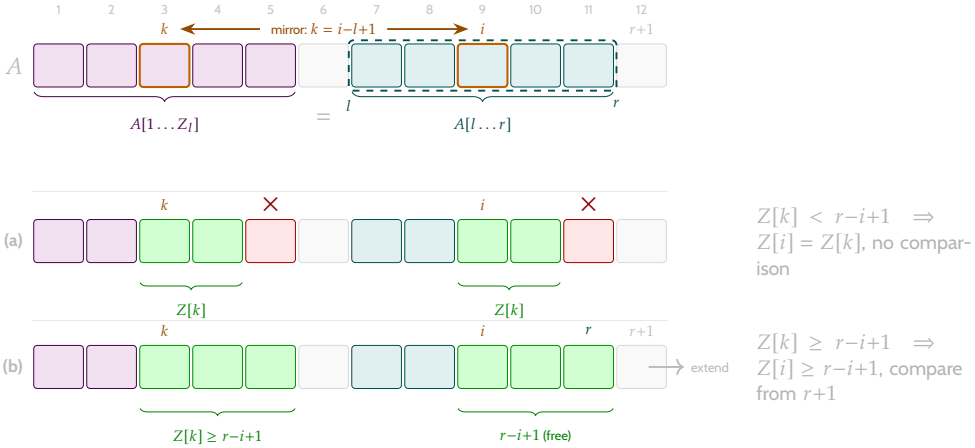


Figure 1.8: Z-box schematic (positions l=7, r=11, k=3, i=9). Violet: the prefix $A[1 \ldots Z_l]$ that established the box. Teal: the Z-box $A[l \ldots r]$, a copy of that prefix. Orange border: mirror pair (k, i) with $k = i-l+1$. Green: the $Z[k]$ characters that match at both k and i (free, by the mirror property). Red ×: the mismatch that terminated $Z[k]$, reflected at i. Case (a): the mismatch lies inside $[l, r]$, so $Z[i] = Z[k]$ at no cost. Case (b): the match reaches r, so we must compare from $r+1$. See Figure 1.9 for a concrete step-by-step trace.

At each step i (from 2 to n) exactly one of three cases applies (illustrated in Figure 1.8):

1. $i > r$ **(outside the box).** No cached information is available. Compare $A[1], A[2], \ldots$ against $A[i], A[i+1], \ldots$ directly until a mismatch; set Z_i to the number of matches. If $Z_i > 0$, update $l \leftarrow i,\ r \leftarrow i + Z_i - 1$.
2. $i \le r$ **(inside the box) — let** $k = i-l+1$**.**
   (a) $Z_k < r - i + 1$ **(mirror strictly inside).** The Z-match at k ends before the box boundary, and the same mismatch is reflected at i: set $Z_i = Z_k$, no comparisons needed, no update to $[l, r]$.
   (b) $Z_k \ge r - i + 1$ **(mirror reaches or exceeds the boundary).** We get $Z_i \ge r-i+1$ for free; continue comparing from $A[r+1]$ until a mismatch, then update $l \leftarrow i,\ r \leftarrow i + Z_i - 1$.

Figure 1.9 traces the *complete* execution on $A = \texttt{aabaaab}$, showing every step $i = 2, \ldots, 7$.

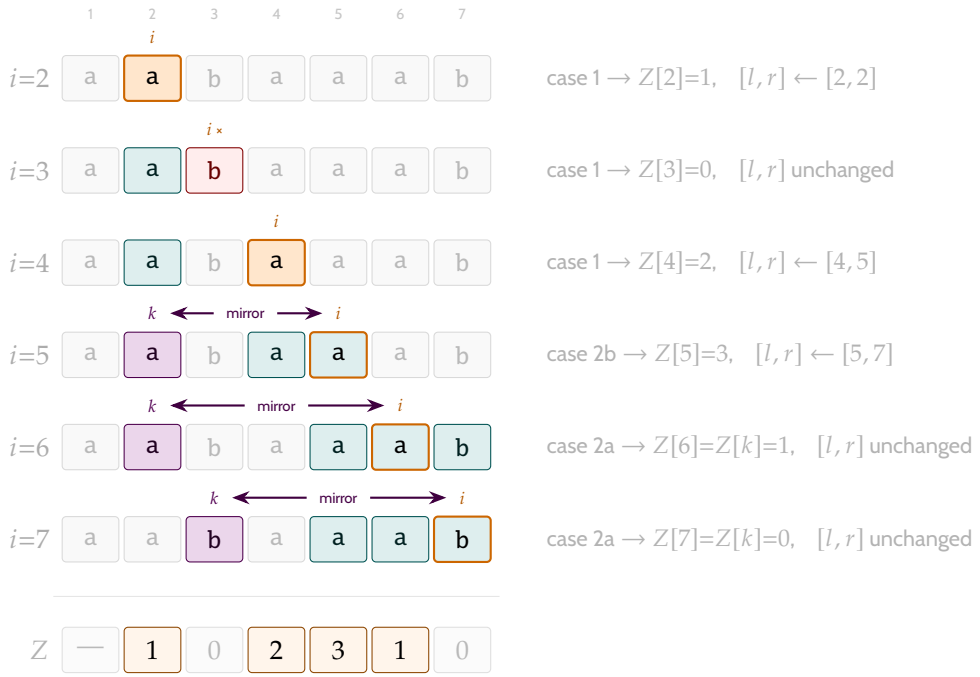


Figure 1.9: Complete Z-box trace on A = aabaaab, steps $i = 2, \ldots, 7$. Teal: current Z-box $[l, r]$. Orange: current position i (teal fill when i is inside the box). Violet: mirror $k = i - l + 1$. Red: immediate mismatch (Z=0). Bottom row: final Z array (non-zero entries highlighted).

Algorithm 4: Z-Algorithm

Input: String $A[1 \ldots n]$
Output: Array $Z[2 \ldots n]$ with $Z[i] = Z_i(A)$

```
l ← 0
r ← 0
for i ← 2 to n do
    if i > r then
        Z[i] ← 0
        while i + Z[i] ≤ n and A[1 + Z[i]] = A[i + Z[i]] do
            Z[i] ← Z[i] + 1
    else
        k ← i − l + 1
        if Z[k] < r − i + 1 then
            Z[i] ← Z[k]          // fully inside - no comparisons
        else
            Z[i] ← r − i + 1
            while i + Z[i] ≤ n and A[1 + Z[i]] = A[i + Z[i]] do
                Z[i] ← Z[i] + 1
    if Z[i] > 0 and i + Z[i] − 1 > r then
        l ← i
        r ← i + Z[i] − 1
return Z
```

Formal details — Correctness of the Z-box algorithm

We prove that the algorithm sets $Z[i] = Z_i(A)$ for every $i \geq 2$.

Precondition $A[1 \ldots n]$ is given; $l = r = 0$ before the loop.

Loop guard i ranges from 2 to n.

Invariant At the *start* of iteration i: (i) $Z[j]$ is correctly set for all $2 \leq j < i$, and (ii) $[l, r]$ is the Z-box with the largest right endpoint seen so far, i.e. $A[l \ldots r] = A[1 \ldots r{-}l{+}1]$ (with $l = r = 0$ meaning no box yet).

Postcondition $Z[i] = Z_i(A)$ for all $i \in \{2, \ldots, n\}$.

Base case Before $i = 2$: Z is empty and $l = r = 0$, so the invariant holds vacuously. ✓

Preservation Assume the invariant holds at the start of iteration i. Three cases arise depending on whether i falls inside the current Z-box $[l, r]$, and if so, what the "mirror position" tells us. Throughout, $k = i - l + 1$ denotes i's mirror position inside the Z-box (i.e. k is the position in the prefix that corresponds to i).

Case 1: $i > r$ (outside the Z-box).
We have no information about $A[i]$ from previous work, so we compare from scratch: $A[1]$ vs $A[i]$, then $A[2]$ vs $A[i{+}1]$, and so on until a mismatch. This directly computes $Z[i]$ by definition. If $Z[i] > 0$, we have found a new Z-box $[l, r] = [i,\ i{+}Z[i]{-}1]$ with a larger r than before. ✓

Case 2a: $i \leq r$ and $Z[k] < r - i + 1$ (mirror match fits entirely inside the Z-box).
Since i is inside the Z-box, we know that $A[l \ldots r] = A[1 \ldots r{-}l{+}1]$ (from the invariant). This means the substring starting at i is a "copy" of the substring starting at position k in the prefix — at least for the characters up to the box boundary r.
Now, $Z[k]$ tells us how far the match at k extended in the prefix. Since $Z[k] < r - i + 1$, that match ended *before* the box boundary, so the mismatch character at $k + Z[k]$ is also inside the box. By the copy property, the same mismatch occurs at $i + Z[k]$, so we can conclude $Z[i] = Z[k]$ without any new comparisons. The box is not updated (no extension beyond r). ✓

Case 2b: $i \leq r$ and $Z[k] \geq r - i + 1$ (mirror match reaches or exceeds the box boundary).
The mirror tells us that at least $r - i + 1$ characters match (the portion inside the box), but beyond position r we have no information — the box does not cover that region. So we initialise $Z[i] = r - i + 1$ (the guaranteed part) and then extend by comparing $A[r{+}1]$ vs $A[Z[i]{+}1]$, etc., exactly as in Case 1. The box is then updated to $[i,\ i{+}Z[i]{-}1]$. ✓

Postcondition When the loop exits, all $Z[i]$ have been correctly computed by the case analysis above. ✓

Amortised $O(n)$ complexity. The variable r is non-decreasing and bounded by n, so it increments at most n times. Every comparison in the **while** loops either advances r (paid by the budget) or ends the step without advancing r (at most one per outer step i). Hence the total number of comparisons is $O(n)$.

1.3.3 From Z values to sp values

The sp values of P can be read off directly from its Z-array.

Observation 1.3.3. $Z_j(P) = k$ means $P[j \dots j+k-1] = P[1 \dots k]$, so $P[1 \dots k]$ is a prefix-suffix of $P[1 \dots i]$ for $i = j + k - 1$. We say j *maps to* i when $i = j + Z_j(P) - 1$.

Let us see what this means on a concrete string. Take $P = \texttt{aabaaab}$. Its Z-array is $Z = [7, 1, 0, 2, 1, 1, 0]$ (1-indexed, with $Z_1 = |P|$ by convention). Consider $j = 4$: $Z_4 = 2$, which tells us that $P[4 \dots 5] = P[1 \dots 2] = \texttt{aa}$. This means aa is a prefix of P that also appears ending at position $i = j + Z_j - 1 = 4 + 2 - 1 = 5$. In other words, aa is a border (prefix-suffix) of $P[1 \dots 5] = \texttt{aabaa}$, so $sp_5 \geq 2$. Is it the *longest* border? That depends on whether any other j also maps to $i = 5$ but gives a larger Z_j value — which is exactly what the proposition below formalises.

Proposition 1.3.4. *For each* $i \in \{2, \dots, m\}$,

$$\mathrm{sp}_i(P) \;=\; Z_{j^*}(P), \qquad \textit{where} \quad j^* = \min\bigl\{\, j \geq 2 \mid j + Z_j(P) - 1 = i \,\bigr\}.$$

Proof. Any j mapping to i gives a prefix-suffix of $P[1 \dots i]$ of length $Z_j(P) = i - j + 1$. The minimum such j maximises $i - j + 1$, yielding the longest prefix-suffix, which is $\mathrm{sp}_i(P)$ by definition. □

Exercise 1.3.5. Using the Z-array of P, design an $O(m)$ algorithm that computes $\mathrm{sp}_i(P)$ for every $i \in \{1, \dots, m\}$.

1.4 Strong borders and real-time KMP

1.4.1 The real-time bottleneck

KMP with plain sp values is *blind* (it never re-reads T) but not *real-time*: a single new character of T can trigger a fallback chain costing $O(m)$ in the worst case. The fix is to collapse the entire chain into a single precomputed lookup. We achieve this in two logical steps: first by skipping strictly useless borders, and then by precomputing transitions for every possible character.

1.4.2 Strong border (sp'): Skipping useless fallbacks

Definition 1.4.1 (Strong border)**.** For $i \in \{1, \dots, m-1\}$, define

$$\mathrm{sp}_i(P)' \;=\; \max\bigl\{\, k \leq \mathrm{sp}_i(P) \mid k = 0 \text{ or } P[k+1] \neq P[i+1] \,\bigr\}.$$

This is the length of the longest proper prefix-suffix of $P[1 \dots i]$ whose next character differs from $P[i+1]$.

In plain words, $\mathrm{sp}_i(P)'$ is the length of the longest border of $P[1 \dots i]$ that would not cause an immediate mismatch at $P[i+1]$.

Any mismatch on $P[i+1]$ with a text character x would also fail at the next border $k = \mathrm{sp}_i(P)$ if $P[k+1] = P[i+1]$ — we say such borders are *useless* (see

Figure 1.10). What is a useless border? Example 1.10 shows P = aabaa at $i = 4$: the border α = a of length $k = 1$ is useless since $P[k+1] = P[2]$ = a equals $P[i+1] = P[5]$ = a. If a mismatch occurs at $P[5]$ with some text character $x \neq$ a, then after falling back to the border α, we would immediately fail again at $P[k+1]$ — the border is useless. sp' allows us to skip an entier chain of useless borders in a single step, jumping directly to the longest safe border that could potentially match the next text character.

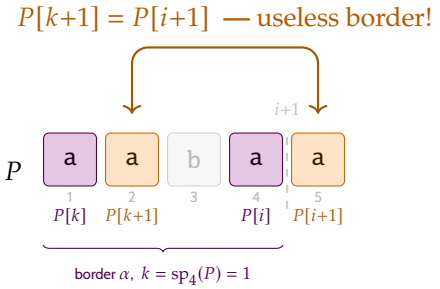


Figure 1.10: Strong border for P = aabaa at $i = 4$. The border α = a (violet, $k = \mathrm{sp}_4(P) = 1$) satisfies $P[1] = P[4]$, but its "next character" $P[k+1] = P[2]$ = a (orange) equals $P[i+1] = P[5]$ = a. Any mismatch on $P[5]$ with text character $x \neq$ a would *also* fail immediately at $P[k+1]$ after the fallback — the border is useless. Hence $\mathrm{sp}_4(P)' = 0$: skip straight to the start.

$\mathrm{sp}_i(P)'$ skips the entire chain of useless borders in one step, but it only encodes a single fallback state — the longest "safe" border for the specific character $P[i+1]$. For a completely arbitrary text character x, a fallback chain might still occur. To achieve strict $O(1)$ time per character, a full character-indexed table is needed.

1.4.3 Character-indexed borders and the DFA: True Real-Time

For a complete real-time solution we precompute, for each position i and character $x \in \Sigma$, the longest border of $P[1 \ldots i]$ whose next character is exactly x:

Definition 1.4.2 ($sp_{i,x}$ values)**.** For $i \in \{0, 1, \ldots, m\}$ and $x \in \Sigma$,

$$\mathrm{sp}_{i,x}(P) \;=\; \max\big\{\, k \leq \mathrm{sp}_i(P) \mid P[k+1] = x \,\big\},$$

with default 0 when no such k exists.

This generalises $\mathrm{sp}_i(P)'$: where the strong border encodes a single fallback state for the character $P[i+1]$, the table $\mathrm{sp}_{i,x}$ encodes the optimal fallback for *every* $x \in \Sigma$ simultaneously.

Example 1.4.3 (sp' vs. $\mathrm{sp}_{i,x}$ for P = aabaab)**.** Consider P = aabaab ($m = 6$) over $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. The sp and sp' values are:

i	1	2	3	4	5	6
$P[i]$	a	a	b	a	a	b
$\mathrm{sp}_i(P)$	0	1	0	1	2	3
$\mathrm{sp}_i(P)'$	0	1	0	0	1	–

At $i = 5$ the border aa ($\mathrm{sp}_5(P) = 2$) is useless because $P[3] = \mathsf{b} = P[6]$; the shorter border a ($k = 1$) satisfies $P[2] = \mathsf{a} \neq P[6]$, so $\mathrm{sp}_5(P)' = 1$.

At $i = 5$, the border chain of $P[1..5] =$ aabaa is $k \in \{2, 1, 0\}$. Both $k = 2$ ($P[3]$=b=$P[6]$) and $k = 1$ ($P[2]$=a) must be inspected; $\mathrm{sp}_5(P)' = 1$ skips past the useless $k = 2$. Also: $\mathrm{sp}_{5,\mathsf{a}} = 1$ (border of length 1 satisfies $P[2] = \mathsf{a} = x$); $\mathrm{sp}_{5,\mathsf{c}} = 0$ (no border has $P[k{+}1] =$ c).

Now suppose we have matched $P[1..5]$ (state $j = 5$) and the next text character is x. With $\mathrm{sp}_5(P)' = 1$, KMP falls back to state 1 for *every* mismatch $x \neq P[6] = \mathsf{b}$, and may still need further comparisons depending on x:

- $x = \mathsf{a}$: $5 \to 1$, then $P[2] = \mathsf{a} = x \Rightarrow$ state 2. **Two comparisons.**
- $x = \mathsf{c}$: $5 \to 1$, $P[2] \neq \mathsf{c}$; $1 \to 0$, $P[1] \neq \mathsf{c}$; stay at 0. **Three comparisons.**

By contrast, $\mathrm{sp}_{5,x}$ resolves every case in one lookup:

x	a	b	c
$\mathrm{sp}_{5,x}$	1	– (match $P[6]$)	0
DFA $\delta(5, x)$	2	6 (full match!)	0

$\delta(5, \mathsf{a}) = 2$ replaces two comparisons with one table read; $\delta(5, \mathsf{c}) = 0$ replaces three.

The table $\{\mathrm{sp}_{i,x}(P)\}_{i,x}$ is precisely the *transition function* of the deterministic finite automaton (DFA) for pattern P: state i encodes "i characters of P matched so far", and on reading x the automaton transitions to $\mathrm{sp}_{i,x}(P) + \mathbf{1}[P[i+1] = x]$. With this table, each character of T is processed in strict $O(1)$ worst-case time, making KMP truly **real-time**.

The DFA has $m + 1$ states and $|\Sigma|$ transitions per state, so the table takes $O(m|\Sigma|)$ space and time to build.

1.5 A Comprehensive Trace: KMP and Z-Algorithm

To solidify these concepts, let's trace KMP and then contrast it directly with the Z-Algorithm.

1.5.1 KMP Step-by-Step

Consider the pattern $P =$ ababa and the text $T =$ ababcababa. Here we trace the standard KMP behavior (using plain sp values to illustrate the fallback mechanism that DFA optimizes).

For $i = 4$, $P[1..4] =$ abab. Proper prefix-suffixes are ab (len 2) and a (len 1). Max is 2.

1. Preprocessing P. We compute the sp array for P.

i	1	2	3	4	5
$P[i]$	a	b	a	b	a
$\mathrm{sp}_i(P)$	0	0	1	2	3

2. Searching T. We maintain a pointer i into T and a pointer j representing the length of the current match in P. The complete trace is shown in Figure 1.11.

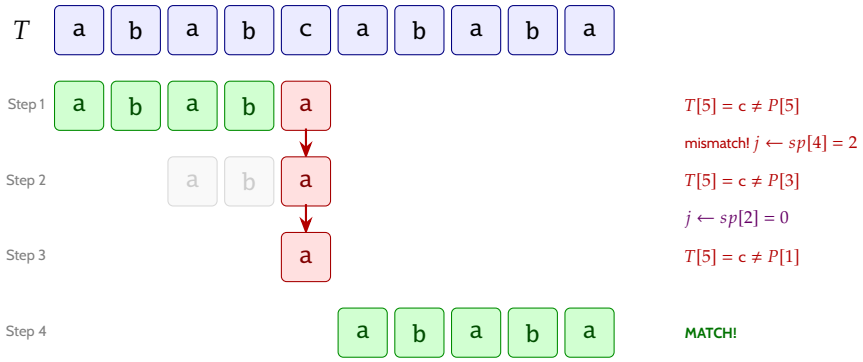


Figure 1.11: KMP trace for P = ababa in T = ababcababa. Step 1 finds a mismatch at $T[5]$. KMP falls back twice ($j = 4 \longrightarrow 2 \longrightarrow 0$) without re-reading T. Finally, Step 4 finds a complete match starting at $T[6]$.

1.5.2 Online vs. Real-Time: Bridging the Gap

The KMP trace above highlights a crucial property: KMP processes T one character at a time without ever stepping backwards in T. This makes KMP an **online** algorithm. However, as seen in Steps 1-3, processing the single character $T[5]$ = c required multiple internal operations (j falling back multiple times). If we had used the full DFA ($\text{sp}_{i,x}$) discussed previously, those steps would collapse into $O(1)$ worst-case time, upgrading the algorithm from merely online to strictly **real-time**.

By contrast, the Z-algorithm is *neither* real-time nor strictly online in the same way. The computation of the Z-box $[l, r]$ depends on reading characters ahead in the string. Let's trace it to see this structural difference in action.

1.5.3 Z-Algorithm Trace

Let's compute the Z-values for the string S = aababca. Notice how the algorithm looks ahead to establish the rightmost boundary r.

1. $i = 2$: $S[2] = \text{a} = S[1]$. Check $S[3] = \text{b} \neq S[2]$. $Z_2 = 1$. Box $[l, r] = [2, 2]$.
2. $i = 3$: $i > r = 2$. $S[3] = \text{b} \neq S[1]$. $Z_3 = 0$. Box stays $[2, 2]$.
3. $i = 4$: $i > r = 2$. $S[4] = \text{a} = S[1]$. Check $S[5] = \text{b} = S[2]$, $S[6] = \text{c} \neq S[3]$. $Z_4 = 2$. New box $[l, r] = [4, 5]$.
4. $i = 5$: $i \leq r = 5$. Mirror $k = i - l + 1 = 5 - 4 + 1 = 2$. $Z_k = Z_2 = 1$. Remaining box width $r - i + 1 = 5 - 5 + 1 = 1$. Since $Z_k \not< r - i + 1$ (Case 2b), we try to extend from $r + 1 = 6$. $S[6] = \text{c} \neq S[3]$. No extension. $Z_5 = 1 + 0 = 1$. Box stays $[4, 5]$.
5. $i = 6$: $i > r = 5$. $S[6] = \text{c} \neq S[1]$. $Z_6 = 0$.
6. $i = 7$: $i > r = 5$. $S[7] = \text{a} = S[1]$. No more chars. $Z_7 = 1$.

i	1	2	3	4	5	6	7
$S[i]$	a	a	b	a	b	c	a
Z_i	–	1	0	2	1	0	1

1.6 Boyer-Moore Algorithm

So far we have seen algorithms that scan the text strictly left to right, never revisiting a character once processed. The Boyer-Moore algorithm takes a different approach: it compares the pattern against the text *from right to left* while still sliding the pattern from left to right. Mismatches on the rightmost characters of the pattern are the cheapest failures possible—they let the algorithm skip large portions of the text.

Scanning right-to-left is the key insight: a mismatch at the last *character of the current alignment gives the most information about how far to jump. Why learn Boyer-Moore if KMP already achieves $O(n+m)$? In practice, Boyer-Moore is often* faster *than KMP on natural language and large alphabets: its right-to-left scan frequently skips large chunks of the text without even looking at them. KMP guarantees a tight worst case; Boyer-Moore wins on average.*

1.6.1 Intuition

The algorithm combines two independent heuristics. At every alignment it takes the *maximum* shift allowed by either rule:

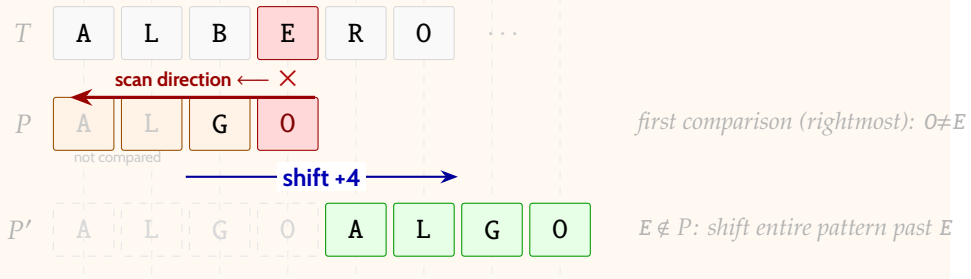
- **Bad Character Rule (BCR).** When character x in the text causes a mismatch at position i in the pattern, shift P right so that the rightmost occurrence of x in $P[1 \ldots i{-}1]$ aligns with x in the text. If x does not appear in P at all, skip past x entirely.
- **Good Suffix Rule (GSR).** When the suffix $S = P[i \ldots m]$ has already been matched and position $i{-}1$ causes a mismatch, shift P right so that another copy of S in P—preceded by a character *different* from $P[i{-}1]$—aligns with the matched copy of S in the text.

Both rules are correct in the sense that neither one ever skips a genuine occurrence of P. At each step Boyer-Moore applies whichever rule yields the larger shift.

1.6.2 Bad Character Rule

Definition 1.6.1 (Bad Character table)**.** For each position $1 \le i \le m$ and character $x \in \Sigma$, let $\mathrm{BC}(i, x)$ be the largest $j < i$ such that $P[j] = x$, or 0 if no such j exists. When a mismatch occurs at $P[i]$ against text character x, shift P right by $i - \mathrm{BC}(i, x)$ positions.

Example 1.6.2 (BCR skip)**.** Search $P =$ `ALGO` in $T =$ `...ALBERO...`. Aligning P with the first four characters, we compare right-to-left: $P[4] =$ `O` vs $T[4] =$ `E` is a mismatch. Since `E` does not appear in P, BCR shifts the entire pattern past the "bad" character.



Preprocessing options. Depending on the alphabet size $|\Sigma|$ and pattern length m, different data structures can be used to implement the Bad Character Rule:

a) **Full 2-D table** ($|\Sigma| \times m$): A matrix $M[x, i]$ storing $\mathrm{BC}(i, x)$. This provides $O(1)$ query time but requires $O(|\Sigma|m)$ space, which is only feasible for small alphabets like DNA.

b) **Compressed Matrix (Array of Lists):** For each position $1 \leq i \leq m$, store a linked list of pairs (x, j) where j is the rightmost occurrence of character x in $P[1 \ldots i-1]$. This is a sparse representation of the 2-D table (see Figure 1.12). The space complexity is $O(m \cdot \min(m, |\Sigma|))$, and the query time is $O(\min(m, |\Sigma|))$.

c) **Position Lists (Character-indexed lists):** For each character x that appears in P, store a list of its occurrence indices in P in *descending* order (see Figure 1.13). To find $\mathrm{BC}(i, x)$, we search the list for x for the largest index $j < i$. Space is $O(m)$, and query time is $O(\log m)$ with binary search.

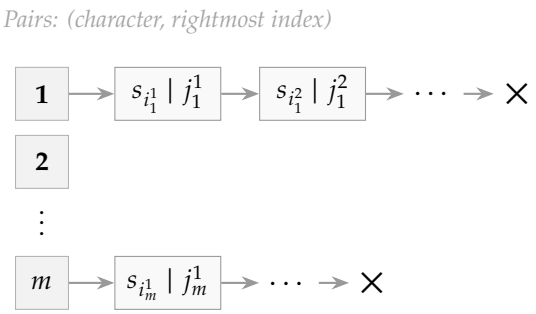


Figure 1.12: BCR as an array of $|P|$ linked lists (Sparse Matrix representation).

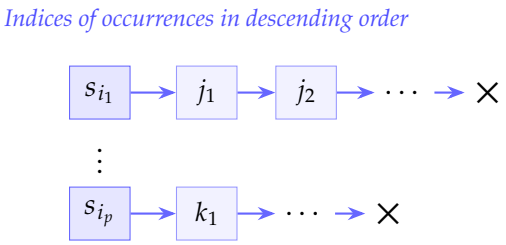


Figure 1.13: BCR as an array of character-indexed lists.

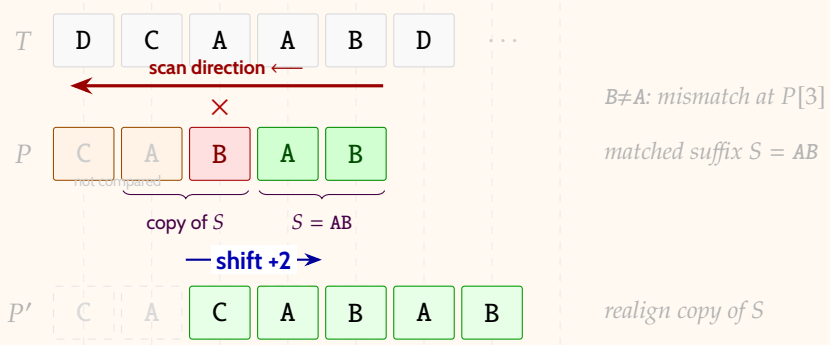
> *Remark* 1.6.3. While binary search on position lists costs $O(\log m)$, the total pattern length m is typically small enough that this overhead is negligible compared to the large shifts BCR provides.

1.6.3 Good Suffix Rule

Intuition. The Bad Character Rule ignores the characters we *already matched*. When Boyer-Moore scans right-to-left and a mismatch occurs at position i, we have successfully matched the suffix $S = P[i \ldots m]$ against the text. That

information is precious: if S appears again somewhere to the left in P, we can shift P so that second occurrence lines up with the matched region in T and continue — no re-reading of T required.

Example 1.6.4 (GSR skip). Search $P = \texttt{CABAB}$ in $T = \texttt{DCAAB}\ldots$. Aligning P at position 1, we scan right-to-left: $P[5] = \texttt{B}$ vs $T[5] = \texttt{B}$ matches, $P[4] = \texttt{A}$ vs $T[4] = \texttt{A}$ matches, $P[3] = \texttt{B}$ vs $T[3] = \texttt{A}$ is a mismatch. The matched suffix is $S = \texttt{AB}$. The same substring AB also appears at $P[2\ldots3]$, preceded by $P[1] = \texttt{C} \neq \texttt{B} = P[3]$ (safe!). GSR shifts P right by 2 so that the copy of S aligns with the matched region.



Two boundary situations arise:

- If no copy of S exists inside P (other than at the end), we fall back to the longest *prefix* of P that is a suffix of S, and align that prefix with the end of the matched region.
- If even that fails (no prefix of P matches any suffix of S), we shift the entire pattern past the matched region.

The Good Suffix Rule is the Boyer-Moore analogue of KMP's border table: both exploit internal repetition of the pattern to skip shifts that are guaranteed to fail.

After matching suffix $S = P[i\ldots m]$, we want to shift P so that another copy of S aligns with the text. We require that copy to be preceded by a character *different* from $P[i{-}1]$; otherwise the same mismatch would recur immediately.

Definition 1.6.5 ($L'(i)$). $L'(i)$ is the largest index $j < m$ such that

1. $P[i\ldots m]$ occurs in P ending at position j, and
2. the character immediately before that occurrence differs from $P[i{-}1]$: $P[j-(m-i)] \neq P[i{-}1]$.

Figure 1.14 illustrates the geometry: the suffix $S = P[i\ldots m]$ appears again ending at position $L'(i)$, and the character immediately to its left (x) differs from $y = P[i{-}1]$, guaranteeing the shift avoids an immediate re-mismatch.

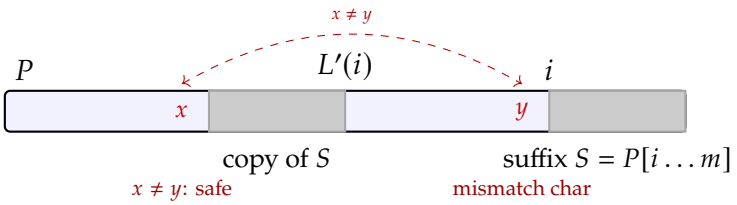


Figure 1.14: Geometry of $L'(i)$. The suffix $S = P[i \ldots m]$ occurs again ending at $L'(i)$, preceded by a character $x \neq y = P[i{-}1]$. Shifting P so that the copy of S aligns with the matched text region guarantees the mismatch character changes, so the same failure cannot repeat.

To compute $L'(i)$ efficiently we need a helper quantity that measures how much of P's own suffix repeats inside P. Intuitively, $L'(i) = j$ means that $P[j-(m-i)\ldots j] = P[i \ldots m]$, so we need to find the rightmost position j where such a match ends. That is exactly captured by the *N-values*:

Definition 1.6.6 ($N_j(P)$)**.** $N_j(P)$ is the length of the longest suffix of $P[1 \ldots j]$ that is also a suffix of P.

In plain words: stand at position j in P and look backwards — how many characters match the *end* of P? For example, let $P = \texttt{abcabc}$ ($m = 6$). The suffix of the full P is abcabc itself, but we are asking about shorter suffixes that appear earlier. At $j = 3$, $P[1 \ldots 3] = \texttt{abc}$; the last 3 characters abc match the last 3 characters of P (abc), so $N_3 = 3$. At $j = 2$, $P[1 \ldots 2] = \texttt{ab}$; neither b nor ab matches the end of P (bc, abc), so $N_2 = 0$.

Notice that N_j is essentially the Z-value idea but reading *from right to left* instead of left to right — which is exactly why the theorem below connects N_j to the Z-array of the *reversed* pattern.

Theorem 1.6.7. *The values $N_j(P)$ satisfy*

$$N_j(P) = Z_{m-j+1}(P^r),$$

where $m = |P|$ and P^r denotes P reversed. Consequently, the entire GSR table can be computed in $O(m)$ time by running the Z-algorithm on P^r.

Formal details — Proof: $N_j(P) = Z_{m-j+1}(P^r)$

Set $k = m - j + 1$ and recall the two definitions:

- $N_j(P) = \ell$ iff ℓ is the largest integer such that $P[j{-}\ell{+}1 \ldots j] = P[m{-}\ell{+}1 \ldots m]$, i.e. the last ℓ characters of $P[1 \ldots j]$ equal the last ℓ characters of P.
- $Z_k(P^r) = \ell$ iff the length-ℓ prefix of P^r matches P^r at position k: $P^r[1 \ldots \ell] = P^r[k \ldots k{+}\ell{-}1]$.

Key observation. Since $P^r[i] = P[m+1-i]$, reading P^r from left to right is the same as reading P from right to left. More precisely, for any $\ell \geq 1$ and $s = 0, 1, \ldots, \ell - 1$:

$$P^r[1+s] = P[m-s] \qquad \text{and} \qquad P^r[k+s] = P[m+1-k-s] = P[j-s],$$

where the last equality uses $k = m - j + 1$.

The two conditions are equivalent. For any $\ell \geq 0$:

$$P^r[1 \dots \ell] = P^r[k \dots k+\ell-1] \iff P[m-s] = P[j-s] \; \forall s \in \{0, \dots, \ell-1\}$$
$$\iff P[m-\ell+1 \dots m] = P[j-\ell+1 \dots j].$$

The rightmost condition is exactly the requirement that $P[1 \dots j]$ has a suffix of length ℓ that matches a suffix of P.

Conclusion. Because the two conditions agree for every ℓ, their maxima agree:

$$N_j(P) = Z_{m-j+1}(P^r). \square$$

Complexity. Running the Z-algorithm on P^r costs $O(m)$ time and produces all m values $Z_1(P^r), \dots, Z_m(P^r)$ in a single pass. Reading off $N_j = Z_{m-j+1}$ for $j = 1, \dots, m$ is then an $O(m)$ post-processing step, so the full GSR precomputation is $O(m)$ overall.

Example 1.6.8 (N_j and $L'(i)$ on a concrete pattern)**.** Let $P =$ `CABAB` ($m = 5$). The reversed pattern is $P^r =$ `BABAC`. Running the Z-algorithm on P^r:

position k	1	2	3	4	5
$P^r[k]$	B	A	B	A	C
$Z_k(P^r)$	–	0	2	0	0

Converting via $N_j = Z_{m-j+1}(P^r)$:

- $N_5 = Z_1 =$ (by convention, the full string) $= 5$
- $N_4 = Z_2 = 0$
- $N_3 = Z_3 = 2$ (suffix `AB` of $P[1 \dots 3] =$ `CAB` matches suffix `AB` of P)
- $N_2 = Z_4 = 0$
- $N_1 = Z_5 = 0$

From $N_3 = 2$ we read: the last 2 characters of $P[1 \dots 3]$ form a suffix of P, i.e. $P[2 \dots 3] = P[4 \dots 5] =$ `AB`. This directly gives $L'(4) = 3$: the rightmost occurrence of $P[4 \dots 5]$ inside P ending strictly before m ends at position 3, preceded by $P[1] =$ `C` $\neq P[3] =$ `B`. GSR shift for a mismatch at position 4: $m - L'(4) = 5 - 3 = 2$.

Figure 1.15 shows the reversal correspondence between $Z_k(P^r)$ and $N_j(P)$ visually: a match of length ℓ starting at position k in P^r corresponds exactly to a suffix-match of length ℓ ending at position j in P.

Reusing the Z-algorithm on P^r means no new code is needed for GSR preprocessing: run the same routine on the reversed pattern.

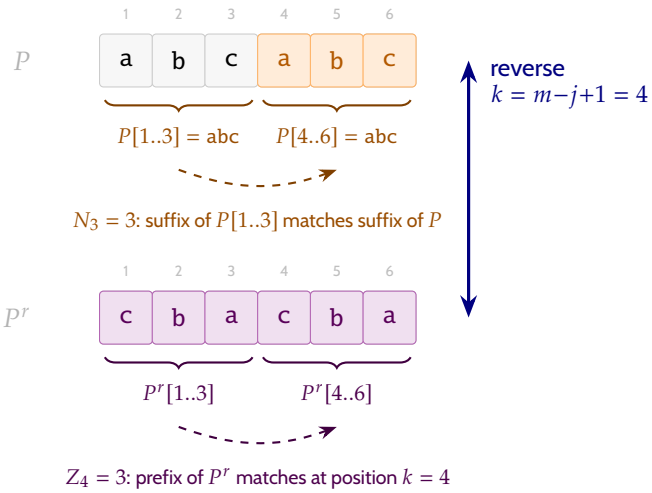


Figure 1.15: Reversal correspondence: $N_j(P) = Z_{m-j+1}(P^r)$ for P = abcabc. **Top**: $N_3 = 3$ because the last 3 characters of $P[1..3]$ (abc) match the last 3 characters of P (abc). **Bottom**: reversing P gives P^r = cbacba, and $Z_4 = 3$ because the prefix cba reappears at position $k = m - j + 1 = 4$. Reading P backwards from position j is the same as reading P^r forwards from position k.

1.6.4 Algorithm Implementation

We now have all the ingredients. Both rules are precomputed in $O(m)$ time: BCR uses the bad-character table BC(i, x) (position lists or 2-D table), and GSR uses the N_j values derived from the Z-algorithm on P^r. At each alignment, Boyer-Moore scans right-to-left, and on a mismatch at position j applies *both* rules independently, then takes the **maximum** shift — this is safe because each rule on its own guarantees not to skip any occurrence.

The full Boyer-Moore algorithm combines the precomputed Bad Character and Good Suffix tables to determine the maximum possible shift at each step.

Algorithm 5: Boyer-Moore Pattern Matching

Input: Text T of length n, Pattern P of length m
Output: Indices of all occurrences of P in T

```
BC ← PrecomputeBadCharacter(P)
GS ← PrecomputeGoodSuffix(P)
s ← 0                                          // Current shift
while s ≤ n − m do
    j ← m
    while j > 0 and P[j] = T[s + j] do
        j ← j − 1
    if j = 0 then
        Report occurrence at s + 1
        s ← s + GS[1]
    else
        shift_BC ← max(1, j − BC(j, T[s + j]))
        shift_GS ← GS[j]
        s ← s + max(shift_BC, shift_GS)
```

1.6.5 Complexity

Theorem 1.6.9 (Boyer-Moore worst case)**.** *In the worst case, Boyer-Moore runs in $O(nm)$ time.*

Proof sketch. Consider $T = a^n$ and $P = ba^{m-1}$. At every alignment the algorithm compares all m characters right-to-left (matching $m-1$ a's before mismatching on b), and both BCR and GSR produce a shift of only 1. There are $n - m + 1$ alignments, giving $(n - m + 1) \cdot m = O(nm)$ comparisons. □

Remark 1.6.10 (Average-case behaviour). For a random text over a large alphabet, BCR almost always triggers a skip at the very first (rightmost) comparison, since a random character is unlikely to appear in P. The expected shift is $O(m)$, leading to an average-case complexity of $O(n/m)$ — sub-linear in n, which is why Boyer-Moore is so effective in practice for natural-language or genomic texts.

1.6.6 Apostolico–Giancarlo Optimization

Intuition. Boyer-Moore's $O(nm)$ worst case stems from *forgetting*: after each alignment it discards all information about which text characters were compared, so the same characters can be re-examined from scratch at the next shift. This is the same problem KMP solved for single-pattern matching — and the fix is the same idea: *cache* the outcome of previous comparisons.

Apostolico and Giancarlo maintain an auxiliary array M over the text. After each alignment, whenever a suffix of P of length ℓ is matched against T ending at position k, they store $M[k] = \ell$. At a future alignment, before comparing $P[i]$ against $T[k]$, the algorithm checks whether $M[k]$ already records how much of the pattern matches at k — and if so, skips those comparisons entirely.

Definition 1.6.11 (M array)**.** $M[k]$ is the length of the longest suffix of P that was matched against T ending at position k in some previous alignment. Initially $M[k] = 0$ for all k.

When P is aligned at shift s and we are comparing $P[i]$ with $T[k]$ (where $k = s + i$), we check $w = M[k]$. Recall that $N_i(P)$ is the length of the longest suffix of $P[1 \dots i]$ that is also a suffix of P.

There are four key cases when we encounter a previously matched segment ($w > 0$):

1. **Case 1:** $w > N_i$**.** Since the text matches a longer suffix of P than what is available as a sub-occurrence ending at i, there must be a mismatch at $i - N_i$. We can immediately shift.

2. **Case 2:** $w < N_i$**.** The text matches a shorter suffix than the available sub-occurrence. This implies a mismatch at $i - w$.

3. **Case 3:** $w = N_i$**.** The match in the text exactly matches the sub-occurrence of the suffix in P. We can skip w comparisons and continue checking P from position $i - w$.

4. **Case 4:** $w \geq N_i$ **and** $N_i = i$**.** This corresponds to a prefix of P matching the text. If $i = m$, we found a full match; otherwise, we can skip and continue.

Figure 1.16 summarises the four cases geometrically.

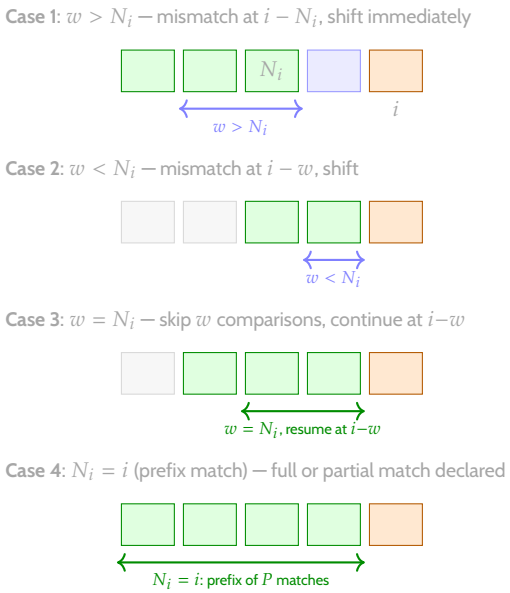
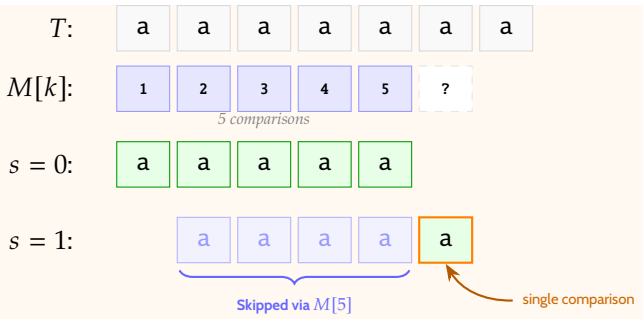


Figure 1.16: The four cases of the Apostolico–Giancarlo algorithm. Green cells are already known to match (from $M[k]$); blue cells are skipped; orange is the current comparison position i.

By using these rules, Apostolico–Giancarlo ensures that each character of the text is involved in at most a constant number of *successful* comparisons, bringing the worst-case complexity down to $O(n + m)$.

Example 1.6.12 (Apostolico–Giancarlo in action)**.** Consider the classic worst-case for Boyer-Moore: $P = \texttt{aaaaa}$ ($m = 5$) and $T = \ldots\texttt{aaaaaa}\ldots$. Precomputing $N_i(P)$: since every prefix of P is also a suffix of P, we have $N_i = i$ for all i.

1. **Shift** $s = 0$: Compare P against $T[1 \ldots 5]$ right-to-left. 5 matches found. Update array M with match lengths: $M[1] = 1, M[2] = 2, M[3] = 3, M[4] = 4, M[5] = 5$.
2. **Shift** $s = 1$: Align P starting at $T[2]$.
   - Compare $P[5]$ with $T[6]$. Match! (1 comparison).
   - Move to $P[4]$ vs $T[5]$. We see $M[5] = 5$ (previous match).
   - Check Case 4: $w = 5 \geq N_4 = 4$ and $N_4 = 4$ (prefix match).
   - **Jump**: We can declare a full match for the rest of the pattern without any further comparisons.



In this scenario, while Boyer-Moore would perform m comparisons for each shift ($O(nm)$), AG performs only one (plus the M check), guaranteeing $O(n)$ linearity even on periodic strings.

On the other hand, in favourable inputs (e.g. $T = c^n$, $P = a^{m-1}c$) the original Boyer-Moore already scans the text in as few as $\lceil n/m \rceil$ character comparisons.

Single-Pattern Algorithms at a Glance

Before moving on, here is a quick reference table comparing the single-pattern algorithms we have seen so far.

Algorithm	Worst case	Preprocessing	Key idea
Naïve	$O(nm)$	—	Try every alignment
KMP	$O(n+m)$	$O(m)$	Never re-read T; use border table
Z-Algorithm	$O(n+m)$	$O(n+m)$	Z-box reuse avoids redundant comparisons
Boyer-Moore	$O(nm)^*$	$O(m+\lvert\Sigma\rvert)$	Right-to-left scan; skip on mismatch

* With the Apostolico–Giancarlo optimisation, Boyer-Moore achieves $O(n+m)$ worst case and $O(n/m)$ best case.

Table 1.1: Comparison of single-pattern exact matching algorithms.

1.7 Generalization of the Problem: Multiple Patterns

The algorithms we have seen so far are designed to search for a single pattern P in a text T. In many applications, we need to search for multiple patterns simultaneously. For example, in DNA sequencing, we might want to find all occurrences of a set of motifs in a long genome sequence.

Definition 1.7.1 (Multiple Pattern Matching)**.** Given a set of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_z\}$ and a text T, find all occurrences of all pattern in $\mathcal{P}$ within T.

1.7.1 Naïve Approach

As usual, let's establish a baseline to compare against. The simplest idea is to run a single-pattern algorithm (e.g. KMP or the Z-algorithm) once for *each* pattern in $\mathcal{P}$ independently.

Denote $m_i = |P_i|$ and let $m = \sum_{i=1}^{z} m_i$ be the total pattern length. Each invocation costs $O(n + m_i)$, so the overall cost is

$$O(n + m_1) + \cdots + O(n + m_z) \;=\; O(z \cdot n + m)\,.$$

The culprit is the factor z multiplying n: we scan T from scratch for every pattern, wasting all the information accumulated from previous passes.

The $z \cdot n$ term dominates when many patterns are present: we are re-reading T once per pattern, ignoring any shared structure among the P_i.

Can we do better? Ideally we would like to scan T only *once*, achieving $O(n + m)$ total time regardless of z. The key insight is that instead of processing the patterns independently, we should *combine* all of them into a single shared data structure, then query it while performing a single left-to-right pass over T.

The natural structure for organising a set of strings by their shared prefixes is a tree—specifically, a **keyword tree**.

1.7.2 Keyword Tree

Definition 1.7.2 (Keyword tree, $K(\mathcal{P})$)**.** Given a set of patterns $\mathcal{P} = \{P_1, \ldots, P_z\}$ over alphabet Σ, the **keyword tree** $K(\mathcal{P})$ is a rooted tree whose edges are labelled with characters of Σ such that:

(i) edges leaving the same node carry *distinct* labels;

(ii) for each $i \in \{1, \ldots, z\}$ there is exactly one root-to-leaf path whose edge-labels, read in order, spell P_i; conversely, every root-to-leaf path spells some $P_i \in \mathcal{P}$.

Property (i) is reflected in Example 1.7.5 by the fact that every node has at most one outgoing edge per character: for instance, the root has three children labelled `p`, `t`, and `o`, all distinct, and the depth-1 node reached via `t` branches into `a` (toward `tatoo`) and `h` (toward `theater`), again with distinct labels. Property (ii) is witnessed by the four leaves, one per pattern: each root-to-leaf path spells exactly one $P_i \in \mathcal{P}$, and conversely every pattern in $\mathcal{P}$ corresponds to exactly one such path.

Remark 1.7.3 (Simplifying assumption). Throughout this section we assume no P_i is a proper prefix of another P_j. Under this assumption every pattern ends at a *leaf*, giving a **bijection** between leaves and patterns. The Aho–Corasick construction lifts this restriction.

Observation 1.7.4 (Size, height, and fan-out of $K(\mathcal{P})$).

- **Size.** The number of edges is at most $m = \sum_{i=1}^{z} m_i$: every non-root node is introduced by a distinct edge, and each root-to-leaf path contributes exactly m_i edges.
- **Height.** The longest root-to-leaf path has length $m_{\max} = \max_i m_i$, the length of the longest pattern.
- **Fan-out.** By property (i), the children of any node carry distinct labels, so the fan-out is at most $|\Sigma|$.

The bijection means: a single pointer inside $K(\mathcal{P})$ simultaneously represents all *patterns that share the prefix corresponding to the current node — the exact analogue of the single state in KMP, now for a whole set of patterns.*

We can verify all three properties in Example 1.7.5 and Figure 1.17: the tree has $6 + 5 + 7 + 5 = 23$ edges (the sum of the pattern lengths), confirming the size bound; the longest path is the one spelling `theater` (7 edges), matching $m_{\max} = 7$; and the root has fan-out 3 (edges `p`, `t`, `o`), well within $|\Sigma|$.

Example 1.7.5 (Building a keyword tree). Let $\mathcal{P} = \{$`potato`, `tatoo`, `theater`, `other`$\}$. We insert the patterns one by one from the root, sharing edges as long as prefixes agree:

- `potato` and no other pattern share a prefix → a chain of 6 edges labelled `p-o-t-a-t-o` from the root.
- `tatoo` starts with `t`; no existing edge from the root is labelled `t` → a new branch. At depth 1 the single `t`-node branches further for `a` (continuing `tatoo`) and later for `h` (from `theater`).
- `theater` shares the prefix `t` with `tatoo`; at depth 1 they diverge (`a` vs `h`).
- `other` starts with `o`; no existing root-edge is labelled `o` → a new branch.

The resulting tree has 6+5+7+5 = 23 edges (one per character of all patterns).

Figure 1.17 shows the keyword tree for $\mathcal{P} = \{$`potato`, `tatoo`, `theater`, `other`$\}$. The `t`-subtree branches at depth 2 into the `a`-path (for `tatoo`) and the `h`-path (for `theater`), reflecting the shared prefix `t`.

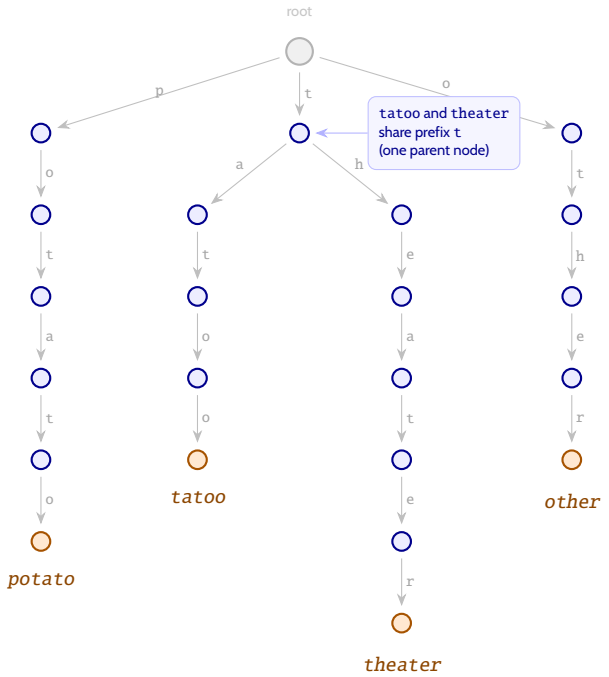


Figure 1.17: Keyword tree $K(\mathcal{P})$ for $\mathcal{P} = \{$`potato`, `tatoo`, `theater`, `other`$\}$. Grey root, blue internal nodes, orange leaves. The `tatoo` and `theater` sub-trees branch from the same `t`-node (depth 1), reflecting their shared first character.

1.7.3 Using $K(\mathcal{P})$ on T

The tree is built once in $O(m)$ time (we simply insert each pattern character by character, sharing edges whenever possible, exactly as in Example 1.7.5).

Once the tree is ready, the search works much like the naïve single-pattern algorithm: we try every starting position $s = 1, 2, \ldots, n$ in T, one after the other. For a given s we place a pointer at the root of $K(\mathcal{P})$ and start reading $T[s], T[s{+}1], T[s{+}2], \ldots$, following the corresponding edges downward. Two things can happen:

- We reach a **leaf** at some depth m_i. This means we have matched every character of some pattern P_i, so we report an occurrence of P_i starting at position s in T.
- We get **stuck**: the current node has no outgoing edge for the next character of T. No pattern in $\mathcal{P}$ begins with the piece of T we have read so far, so we give up on this starting position.

Either way, we move to starting position $s{+}1$ and go back to the root to try again.

The important difference with respect to the single-pattern naïve approach is what happens *during* each descent. Because the tree merges all patterns that share a common prefix into the same path, a single walk from the root simultaneously checks all of those patterns at once. For instance, in Example 1.7.5, reading `t` from the root is enough to know that both `tatoo` and `theater` are still candidates — we do not need separate passes for each of them. Only when we reach the branching point (depth 2, where one path continues with `a` and the other with `h`) do the two patterns diverge, and the tree automatically "picks" the right one based on the next character of T.

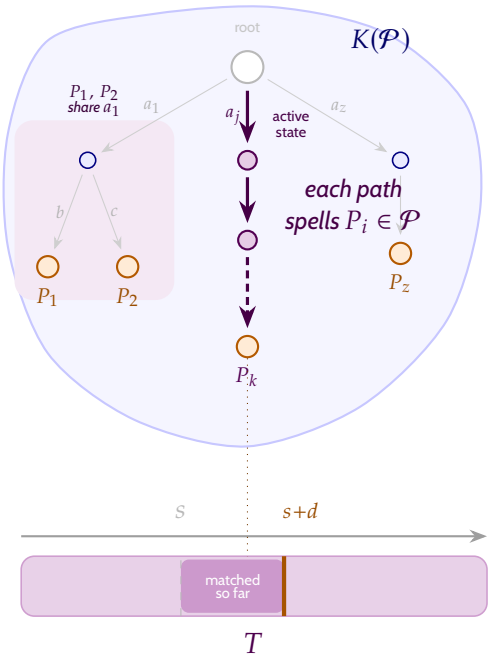


Figure 1.18: Scanning T with $K(\mathcal{P})$. At each starting position s we follow edge-labels $T[s], T[s{+}1], \ldots$ downward from the root (depth d reached so far is shaded on T); reaching a leaf reports an occurrence of the corresponding pattern. Patterns with a shared prefix — here P_1 and P_2 both start with a_1 (pale violet zone) — are tracked in parallel by a single internal node, so no extra work is done for either until they diverge. The highlighted violet path is the *active* descent for the current starting position.

This already beats the naïve approach when patterns share long prefixes, but the worst-case cost per starting position is still $O(m_{\max})$, giving $O(n \cdot m_{\max} + m)$ overall.

Why the keyword tree alone is not enough. The algorithm above is correct — it will find every occurrence of every pattern. The problem is that it wastes work. Every time a descent ends (whether in a match or a mismatch), we go back to the root and start over from position $s+1$, re-reading characters of T that we have already seen.

Consider $\mathcal{P} = \{\texttt{he}, \texttt{she}, \texttt{his}, \texttt{hers}\}$ and $T = \texttt{ushers}$. Starting at $s = 2$ we descend from the root following $\texttt{s} \to \texttt{h} \to \texttt{e}$ and reach the leaf for `she` — match found. Now we restart from the root at $s = 3$ and read $\texttt{h} \to \texttt{e}$ — we find `he`, good. But notice: the `h` and `e` we just read at positions 3 and 4 are the *same characters* we already consumed during the previous descent for `she`. We already knew that $T[3,4] = \texttt{he}$; going back to the root and re-reading them is redundant work.

There is a second issue lurking in this example. Notice that `he` is a proper prefix of `hers`: when we descend from the root following $\texttt{h} \to \texttt{e}$, we have matched `he` in full — but `hers` keeps going with $\texttt{r} \to \texttt{s}$ along the same path. Under our simplifying assumption (no pattern is a prefix of another), this situation cannot arise, because every pattern ends at a leaf and there is nothing "beyond" it. But in general it does arise, and the plain keyword tree has no way to report `he` at an internal node — it only reports matches at leaves. The Aho–Corasick automaton will handle both problems: it eliminates the redundant re-reading *and* correctly reports matches at internal nodes.

In general, the worst-case cost per starting position is $O(m_{\max})$ (we may descend all the way to the deepest leaf before failing or matching), giving $O(n \cdot m_{\max} + m)$ overall. We would like to do better: scan T once, reading each character exactly once, and still find every match.

This is exactly the problem KMP solved for a single pattern with the sp' table: instead of restarting from scratch, jump to the longest proper suffix of what we have matched so far that is itself a prefix of some pattern, and continue from there. For multiple patterns the same idea generalises: after a match or a mismatch we follow a **failure link** to the node in $K(\mathcal{P})$ that represents the longest proper suffix of the current path that is also a prefix of some pattern in $\mathcal{P}$. These failure links turn $K(\mathcal{P})$ into a finite automaton that processes T in a single left-to-right pass, never re-reading any character.

> **Definition 1.7.6** (Failure link)**.** For a node v in $K(\mathcal{P})$ representing a string α, the **failure link** of v points to the node representing the longest proper suffix of α that is a prefix of some pattern in $\mathcal{P}$ (or to the root if no such suffix exists).

Failure links in $K(\mathcal{P})$ are the direct
tion of the
encode
atched
o".

Intermezzo — Aho–Corasick and Unix `fgrep`

The resulting automaton was published by Alfred Aho and Margaret Corasick in 1975. It was originally designed for `fgrep`, the Unix command-line tool for searching a file for multiple fixed strings at once. Every time you run `grep -F` with several patterns, you are using a direct descendant of this construction — one of the most successful examples of an algorithm making

its way from a research paper into a tool that millions of developers use daily without knowing it.

To achieve a true $O(n + m)$ single pass we need to handle mismatches without restarting from the root — the same challenge KMP solved for a single pattern. The solution is to equip $K(\mathcal{P})$ with failure links, leading to the **Aho–Corasick** automaton.

Figure 1.19 previews the idea for the example $\mathcal{P} = \{\texttt{he}, \texttt{she}, \texttt{his}, \texttt{hers}\}$. Dashed arrows are failure links: when the descent in the tree gets stuck or reaches a leaf, we follow the failure link instead of restarting from the root, and continue matching from there.

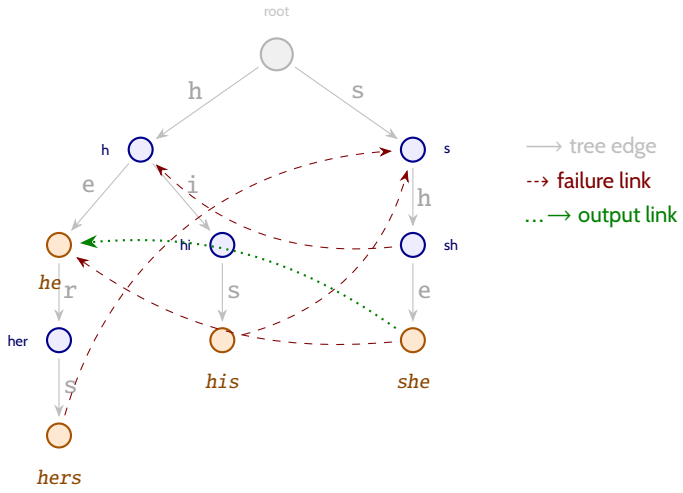


Figure 1.19: Keyword tree for $\mathcal{P} = \{\texttt{he}, \texttt{she}, \texttt{his}, \texttt{hers}\}$ with failure links (dashed) and output links (dotted). For example, after matching `she` the failure link from the `she`-node leads to the `he`-node, so the automaton knows that `he` has also been matched — without re-reading any character. The output link from `she` to `he` records this: whenever the automaton reaches the `she`-node, it must also report `he` as a match. Only non-root failure links are shown; all other nodes fall back to the root. All other output links are nil.

Output links: patterns that end at internal nodes. Failure links solve the problem of continuing the search without re-reading characters, but there is one more subtlety. Consider $\mathcal{P} = \{\texttt{acatt}, \texttt{ca}\}$ and suppose we are searching through a text that contains `acatt`. As we descend the tree, we pass through the node for `ca` on our way to the leaf for `acatt`. But `ca` is itself a pattern in $\mathcal{P}$ — and it ends at an *internal* node, not a leaf. If we only report matches at leaves, we miss it.

Failure links alone do not help here: the failure link from the `ca`-node might point to some unrelated node (or the root), and we would never know that `ca` itself was a match.

The solution is to add a second type of pointer called an **output link**. For every node v in $K(\mathcal{P})$, the output link of v points to the nearest node reachable by following failure links from v that corresponds to a complete pattern. During the search, every time we arrive at a node (whether by a tree edge or a failure link), we follow its output-link chain and report all the patterns we find.

Definition 1.7.7 (Output link)**.** For a node v in $K(\mathcal{P})$, the **output link** of v points to the nearest ancestor (via failure links) of v whose corresponding string is a complete pattern in $\mathcal{P}$. If no such node exists, the output link is nil.

In our earlier example with $\mathcal{P} = \{\texttt{he}, \texttt{she}, \texttt{his}, \texttt{hers}\}$ (Figure 1.19), the node for she has a failure link to the node for he. Since he is a complete pattern, the output link from she points to he. This is exactly how the automaton discovers that matching she also implies matching he — it follows the output link and reports both.

Together, the keyword tree, failure links, and output links form the complete **Aho–Corasick automaton**. It processes T in a single left-to-right pass in $O(n + m + k)$ time, where k is the number of pattern occurrences reported.

Output links are computed during preprocessing in a single BFS pass over the tree: for each node v, if v's failure-link target is itself a pattern, set the output link to that target; otherwise, copy the output link from the failure-link target. This takes $O(m)$ time total.

1.8 Suffix Trees

All the algorithms we have seen so far share a common pattern: we *preprocess the pattern* P and then scan the text T in a single pass. But what if the scenario is reversed? Imagine a huge, mostly static text — a genome, a legal corpus, or a search-engine index — that will be queried over and over with many different patterns. Preprocessing T *once* so that each subsequent query can be answered quickly is far more practical.

The data structure that achieves this is the **suffix tree**. Its power comes from a simple observation: every substring of T is a prefix of some suffix of T. What does this mean concretely? Take any substring — say the characters from position i to position j in T. That same piece of text also appears at the *beginning* of the suffix that starts at position i (namely $T[i,]$). In other words, $T[i, j]$ is a prefix of the suffix $T[i,]$.

Think of it as building a "phone book" for every substring of T: once built, any lookup takes time proportional to the query length, not the text length.

Example 1.8.1. Let T = banana. Consider the substring $T[3, 5]$ = nan. The suffix starting at position 3 is $T[3,]$ = nana, and indeed nan is a prefix of nana. The same holds for every substring: ba is a prefix of the suffix banana, an is a prefix of the suffix anana, and so on.

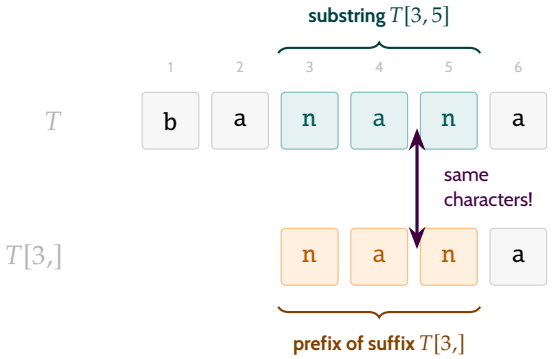


Figure 1.20: Every substring of T is a prefix of some suffix. The substring $T[3, 5]$ = nan (teal) is the first 3 characters of the suffix $T[3,]$ = nana (orange prefix). This is why indexing all suffixes in a tree automatically indexes all substrings.

So if we build a data structure that indexes *all* suffixes of T, we can find any substring by walking down from the root — because that substring is guaranteed to appear as the beginning of at least one suffix.

1.8.1 Suffixes and Substrings

The example above illustrates a general fact:

Observation 1.8.2. Let $T = t_1 t_2 \cdots t_n$. The set of substrings of T equals the set of prefixes of all suffixes of T:

$$\{\, T[i, j] \mid 1 \le i \le j \le n \,\} \;=\; \bigcup_{i=1}^{n} \{\, \text{prefixes of } T[i,] \,\}.$$

This means a trie over all suffixes of T encodes every substring. Walking from the root following the characters of a query string Q either succeeds (meaning Q is a substring of T) or fails at some point (meaning Q does not occur in T).

The word trie *comes from "re**trie**val" and was coined by Edward Fredkin in 1960. It is sometimes pronounced "tree" (as Fredkin intended) and sometimes "try" (to avoid confusion with the word "tree"). A trie is simply a tree where each edge is labelled with a character and shared prefixes share the same path from the root — exactly like the keyword tree we saw earlier, but used for a different purpose.*

1.8.2 Suffix Trie

The first idea is the most direct one: build a trie that contains every suffix of T.

If this sounds familiar, it should. In the previous chapter we built a *keyword tree* $K(\mathcal{P})$ to index a collection of patterns $\mathcal{P}$. A suffix trie is exactly the same construction, but the "patterns" are all the suffixes of T:

$$\text{STrie}(T) \;=\; K\big(\{\, T[i,] \mid 1 \le i \le n \,\}\big).$$

In other words, the suffix trie *is* the keyword tree of the set of all suffixes of T. This means everything we learned about keyword trees — shared prefixes merging into a single path, one node per distinct "pattern prefix" — applies directly here. The difference is the purpose: instead of searching for a fixed set of patterns in a text, we are indexing the text itself so that *any* query can be answered quickly.

This also explains why the suffix trie has a state for every distinct substring of T: each substring is a prefix of some suffix, and in a keyword tree every prefix of every pattern gets its own node.

Definition 1.8.3 (Suffix trie, STrie(T))**.** Let $T = t_1 t_2 \cdots t_n$ be a string over an alphabet Σ. The **suffix trie** of T, denoted STrie(T), is the (deterministic, acyclic) trie that accepts exactly the set $\sigma(T)$ of all suffixes of T (including the empty string ε).

Formally, $\text{STrie}(T) = (Q \cup \{root, \bot\},\ \Sigma,\ g',\ f,\ root)$ where:

- Q has one state for every distinct substring of T.
- $g'(s, a)$ is the transition function: from state s (representing substring w) reading character a, go to the state representing wa (if wa is a substring of T).
- f is the **suffix function**: $f(s)$ points to the state representing $w[2,]$ (i.e. w with its first character removed). This is the exact analogue of the failure function in KMP / Aho–Corasick.
- $\bot$ is an auxiliary "bottom" state with $g'(\bot, a) = root$ for every $a \in \Sigma$, and $f(root) = \bot$.

The **final states** correspond to actual suffixes of T (including $root = \varepsilon$).

Let us unpack the definition piece by piece, using Example 1.8.4 and Figure 1.21 as a running reference.

The suffix trie is a tree whose nodes represent substrings of T. The root represents the empty string ε. From every node, there is at most one outgoing edge for each character $a \in \Sigma$: following that edge takes us from the substring w to the substring wa (i.e. we append one character). This is the transition function g'. In the figure, the **violet-highlighted** edge from ca to cac shows g'(ca, c) = cac: we were at substring ca, read character c, and arrived at substring cac. If there is no edge (because wa is not a substring of T), we simply stop — just as in the keyword tree.

The suffix function f is the interesting part. From any node representing a string w, the suffix function points to the node representing w with its first character chopped off. In the figure, the **red-highlighted** dashed arrow from cac to ac shows f(cac) = ac: we drop the leading c from cac and land on ac. This is the same idea as the failure function in KMP, extended to a whole tree: when we get stuck during a search, we follow the suffix link to "fall back" to the longest suffix that is still a valid state, instead of starting from scratch.

Finally, the bottom state $\perp$ is just a bookkeeping trick: it sits below the root and has an edge to the root for every character in Σ. Its only purpose is to make sure that the suffix-link chain (following f repeatedly) always has a clean stopping point, even when we fall off the root. (The bottom state is not shown in the figure to keep it readable.)

Example 1.8.4 (Suffix trie of T = cacao)**.** The suffixes of cacao are: cacao, acao, cao, ao, o, and ε.

The suffix trie STrie(cacao) contains one state for every distinct substring: ε, c, a, o, ca, ac, ao, cac, aca, cao, caca, acao, cacao, and so on — a total of 16 states (including $root$). Figure 1.21 shows the structure.

The suffix function f is the direct analogue of KMP's failure function: it links each state to its longest proper suffix that is still a state. The bottom state $\perp$ is a convenience — it ensures that the suffix-link chain starting from any state always terminates cleanly at $\perp$, even when we fall off the root.

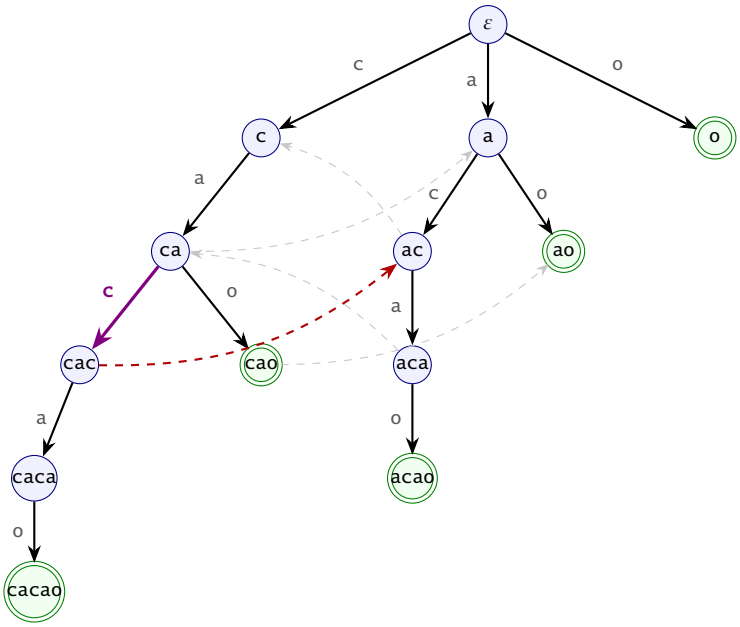


Figure 1.21: Suffix trie STrie(cacao). Solid arrows are transitions $g'(s, a)$; dashed arrows show selected suffix links $f(s)$. Double-bordered states are final (they correspond to actual suffixes). The **violet** edge highlights the transition $g'(\texttt{ca}, \texttt{c}) = \texttt{cac}$; the **red** dashed arrow highlights the suffix link $f(\texttt{cac}) = \texttt{ac}$.

The problem with suffix tries. The suffix trie is conceptually clean, but it is too large. A string of length n has $\Theta(n^2)$ distinct substrings in the worst case, so STrie(T) can have $\Theta(n^2)$ states and $\Theta(n^2)$ transitions. Both building it and storing it takes $\Theta(n^2)$ time and space — the same as explicitly listing all substrings. We need a more compact representation.

1.8.3 From Suffix Trie to Suffix Tree: Compaction

The same idea that turns a trie into a Patricia trie / radix tree*: merge every chain of single-child internal nodes into one edge with a multi-character label.*

Many states in the suffix trie have exactly one outgoing transition — they are just "pass-through" nodes on a single path. Look at the suffix trie in Figure 1.21: the node cac has only one child (caca), which in turn has only one child (cacao). These nodes do not carry any useful branching information — they just spell out the rest of a suffix one character at a time. The idea behind compaction is to collapse such chains into a single edge with a multi-character label. Instead of three nodes cac → caca → cacao connected by edges a and o, we get a single edge labelled ao going straight from cac's parent to the leaf.

Before formalising this, let us classify the nodes of the suffix trie:

Definition 1.8.5 (Branching state, leaf, implicit state)**.** A state of STrie(T) is:
- a **branching state** if it has at least two outgoing transitions;
- a **leaf** if it has zero outgoing transitions (equivalently, it corresponds to a suffix that is not a proper prefix of another suffix);
- an **implicit state** otherwise (exactly one outgoing transition — it sits in the interior of a compacted edge).

The root is always considered a branching state.

In our cacao example (Figure 1.21):
- The root (ε), a, and ca are **branching states** — each has two or more children.

- o, ao, cao, acao, and cacao are **leaves** — they have no outgoing edges.
- Everything else (c, ac, cac, aca, caca) is an **implicit state** — exactly one outgoing edge, just passing through.

The compaction step keeps only the branching states and leaves (the "interesting" nodes) and merges everything in between into multi-character edges. This gives us the suffix tree:

Definition 1.8.6 (Suffix tree, STree(T))**.** The **suffix tree** of T is the tree obtained from STrie(T) by keeping only the branching states and the leaves (the **explicit states**), and replacing every maximal chain of implicit states with a single edge.

Each edge is labelled with a *string* (not a single character). To avoid storing edge labels explicitly (which would again require $O(n^2)$ space), we represent each label as a pair (k, p) of positions in T: the label is $T[k \dots p]$.

The suffix tree STree(T) has:

- at most n leaves (one per suffix, exactly n when a unique end-marker \$ is appended);
- at most $n - 1$ internal branching nodes (by a standard tree argument);
- at most $2n - 1$ edges.

Hence STree(T) uses $O(n)$ space.

The key insight for the space bound: since every internal node branches (has at least two children), there can be at most $n - 1$ internal nodes for n leaves. This is a general fact about trees, and it is important enough to spell out:

■ **Formal details — Linear size of branching trees**

Lemma 1.8.7. *Let $\mathcal{T}$ be a rooted tree in which every internal node has at least two children, and let f be the number of leaves. Then the number of internal nodes i satisfies $i \le f - 1$, and the total number of nodes is at most $2f - 1$.*

Proof. By induction on f.

Base case ($f = 1$). The tree consists of a single node (the root), which is a leaf. There are $i = 0$ internal nodes, and $0 \le 1 - 1 = 0$. ✓

Inductive step ($f > 1$). Pick an internal node x of maximum depth. Since x is internal it has at least two children, and since x is the deepest internal node, all of its children must be leaves. Remove two of x's leaf children from the tree. Now x has lost two children:

- If x had exactly two children, it now has zero — so x itself becomes a leaf. The new tree $\mathcal{T}'$ has $f' = f - 2 + 1 = f - 1$ leaves (two removed, one gained) and $i' = i - 1$ internal nodes (we lost x as an internal node).
- If x had more than two children, it still has at least one child left, so x remains internal. The new tree has $f' = f - 2$ leaves and $i' = i$ internal nodes.

In both cases, $\mathcal{T}'$ still satisfies the branching condition (every internal node has ≥ 2 children), and $f' < f$. By the inductive hypothesis, $i' \le f' - 1$.

In the first case: $i = i' + 1 \le (f - 1) - 1 + 1 = f - 1$. ✓

In the second case: $i = i' \le (f - 2) - 1 = f - 3 < f - 1$. ✓

The total number of nodes is $i + f \le (f - 1) + f = 2f - 1$. □

Applying Lemma 1.8.7 to the suffix tree: the leaves are the suffixes (at most n), so the internal branching nodes are at most $n - 1$, giving at most $2n - 1$

nodes total. And since we store each edge label as just two integers (k, p) rather than copying the actual characters, the whole tree takes $O(n)$ space — a dramatic improvement over the $\Theta(n^2)$ suffix trie.

Compare Figure 1.23 with Figure 1.21: the 16-node trie has been compressed into just 7 nodes. For instance, the path root $\to$ c $\to$ ca in the trie (two single-character edges through the implicit node c) has become a single edge labelled ca going directly from the root to the branching node.

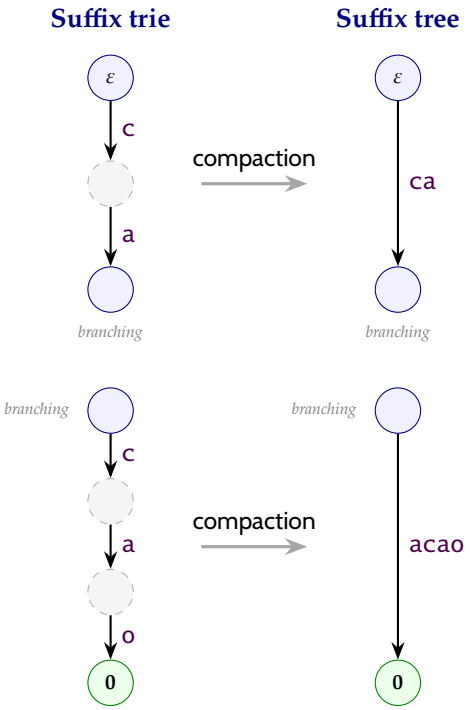


Figure 1.22: Compaction from suffix trie to suffix tree on the string cacao$. Each maximal chain of nodes with a single child (gray dashed) is replaced by one edge whose label is the concatenation of the individual characters. *Top:* the implicit node c is removed, merging two edges into the single edge ca. *Bottom:* three single-child nodes between the branching node and leaf 0 collapse into the edge acao.

By appending a special character $\$ \notin \Sigma$ to T, we guarantee that no suffix is a prefix of another. This makes every suffix end at a distinct leaf, so the tree has exactly $n + 1$ leaves (counting $\$$).

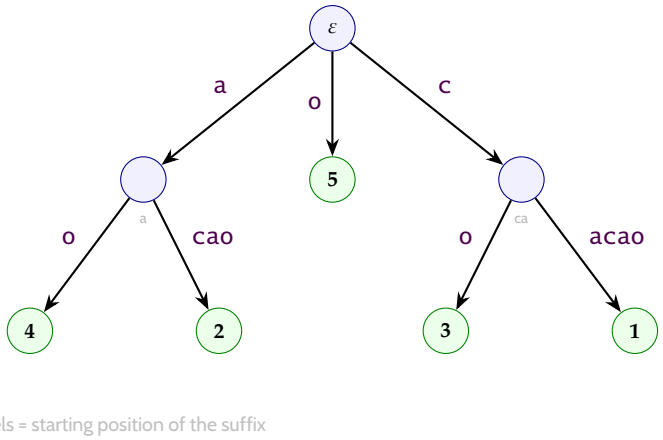


Figure 1.23: Suffix tree STree(cacao). Internal nodes correspond to branching states of the suffix trie; leaves are labelled with the starting position of the suffix they represent. Edge labels are substrings of T, stored as index pairs (k, p) pointing into T (shown here as the actual characters for readability). Compare with the suffix trie in Figure 1.21: the $\Theta(n^2)$ states have been compressed into $O(n)$ nodes.

Reference pairs. Since implicit states are not stored as nodes in the suffix tree, we need a way to talk about them. Think of it this way: in the suffix trie, the node cac existed as its own node. In the suffix tree, it has been swallowed into the edge labelled acao going from the ca-node to a leaf. But we might still need to refer to the "point" on that edge where we have read c so far (one character into the four-character edge label).

A **reference pair** (s, w) gives us exactly that: it names an explicit node s and a string w that tells us how far to walk down from s. In practice, w is stored as an index pair (k, p) so that $w = T[k \dots p]$. For our example, the implicit state cac would be represented as $(\mathtt{ca},\ (3, 3))$ — start at the explicit node ca and walk one character: $T[3] = \mathtt{c}$. When $w = \varepsilon$ (i.e. $k > p$), the reference pair points to the explicit state s itself.

A reference pair $(s, (k, p))$ is **canonical** if s is the closest explicit ancestor of the target state — that is, the entire string $T[k \dots p]$ fits within a single edge leaving s. If it does not (because w spans more than one edge), we **canonize** by walking down from s, consuming full edges one at a time, until the remaining string fits within a single edge.

1.8.4 Online Construction of the Suffix Trie

Before tackling the linear-time suffix tree construction, it is instructive to see how the suffix trie can be built *online* (left-to-right, one character at a time). This naïve construction is $O(n^2)$, but it introduces the key ideas that Ukkonen's algorithm will later exploit.

An important property of the generalised transitions in the suffix tree: each state s has at most one a-transition for each character $a \in \Sigma$, and that transition is uniquely identified by its first character t_k. *This is because no two edges out of the same node can begin with the same character (the tree would not be well-defined otherwise). As a consequence, given a state s and a character a, we can find (or determine the absence of) the corresponding edge in $O(1)$ time with a hash table, or $O(|\Sigma|)$ with an array.*

The key observation. Let $T^i = t_1 \cdots t_i$ denote the prefix of T of length i. The set of suffixes of T^i can be obtained from the suffixes of T^{i-1} by appending t_i to each one, plus the empty string:

$$\boxed{\sigma(T^i) \;=\; \sigma(T^{i-1}) \cdot t_i \;\cup\; \{\varepsilon\}} \tag{1.1}$$

"Online" means: after reading $t_1 \dots t_i$, the data structure represents STrie(T^i) *where $T^i = t_1 \cdots t_i$. We never look ahead at t_{i+1}.*

This is immediate: $T^i = T^{i-1} \cdot t_i$, so every suffix of T^i is either ε or has the form $\alpha \cdot t_i$ where α is a suffix of T^{i-1}.

In terms of the trie, to go from STrie(T^{i-1}) to STrie(T^i) we need to extend every suffix of T^{i-1} by the new character t_i. This means adding a t_i-transition from each final state of STrie(T^{i-1}) that does not already have one.

The boundary path. The final states of STrie(T^{i-1}) are exactly the states representing the suffixes of T^{i-1}:

$$T^{i-1}, \quad T^{i-1}[2,], \quad T^{i-1}[3,], \quad \ldots, \quad T^{i-1}[i-1,], \quad \varepsilon.$$

These states are connected by the suffix function f: $f(T^{i-1}) = T^{i-1}[2,]$, $f(T^{i-1}[2,]) = T^{i-1}[3,]$, and so on down to $f(\varepsilon) = \perp$. This chain of suffix links is called the **boundary path** of STrie(T^{i-1}).

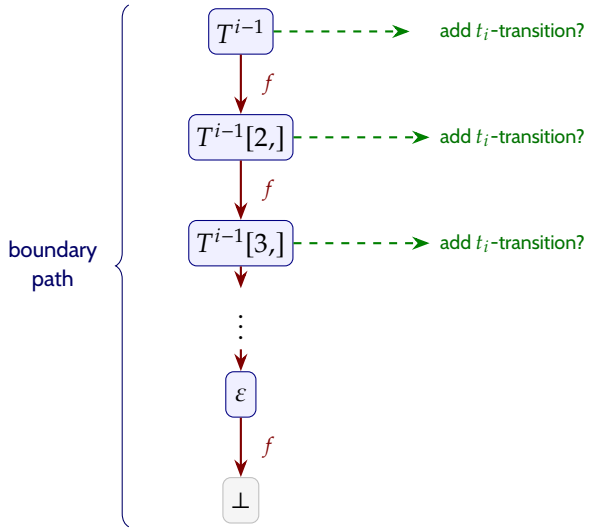


Figure 1.24: The boundary path of STrie(T^{i-1}): the chain of suffix links from the deepest state T^{i-1} down to $\perp$. To build STrie(T^i) we walk this path and add a t_i-transition from every state that does not already have one.

Active point and endpoint. When we walk down the boundary path from T^{i-1} towards $\perp$, at some point we hit a state that *already* has a t_i-transition. Once we find such a state, *all* states below it on the boundary path also have a t_i-transition (because they represent shorter suffixes, and if $\alpha \cdot t_i$ is a substring of T^{i-1}, then so is every suffix of α followed by t_i).

Definition 1.8.8 (Active point and endpoint)**.** Let $s_1 = T^{i-1}$, s_2, $\ldots$, $s_k = \varepsilon$, $s_{k+1} = \perp$ be the states on the boundary path.

- The **active point** s_j is the first state on the boundary path (from the top) that is *not* a leaf.
- The **endpoint** $s_{j'}$ is the first state on the boundary path (from the top) that already has a t_i-transition (and is not a leaf).

The boundary path is thus partitioned into three zones:

1. $s_1, \ldots, s_{j-1}$: leaves — extending an existing branch (the t_i is appended to the path leading to the leaf).
2. $s_j, \ldots, s_{j'-1}$: non-leaf states without a t_i-transition — a *new branch* is created from each.

3. $s_{j'}, \ldots, s_{k+1}$: states that already have a t_i-transition — nothing to do.

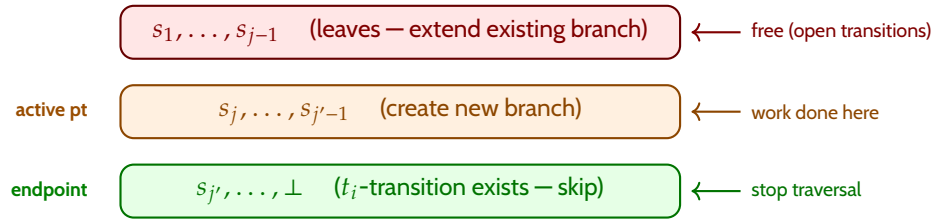


Figure 1.25: Partition of the boundary path into three zones when adding character t_i.

Algorithm 1 (suffix trie, quadratic). The online construction of STrie(T) proceeds as follows: start with STrie(ε) (just the root and $\perp$), and for each character t_i ($i = 1, \ldots, n$), walk down the boundary path from the deepest state, creating new t_i-transitions until we reach a state that already has one.

Algorithm 6: Online construction of STrie(T) (quadratic)

Input: String $T = t_1 \cdots t_n$
Output: STrie(T)

create *root* and $\perp$
$f(root) \leftarrow \perp$
for *every* $a \in \Sigma$ **do**
  $g'(\perp, a) \leftarrow root$
$r \leftarrow root$ `// r = deepest state on boundary path`
for $i \leftarrow 1$ **to** n **do**
  $top \leftarrow r$ `// save current deepest state`
  `// Walk boundary path from` top `towards` $\perp$
  while $g'(r, t_i)$ *is undefined* **do**
    create new state r' and new transition $g'(r, t_i) = r'$
    if $r \neq top$ **then**
      $f(oldr) \leftarrow r'$ `// suffix link from previous new state`
    $oldr \leftarrow r'$
    $r \leftarrow f(r)$
  $f(oldr) \leftarrow g'(r, t_i)$
  $r \leftarrow g'(top, t_i)$ `// new deepest state =` T^i

■ **Formal details — Suffix trie: $\Theta(n^2)$ time and space**

Theorem 1.8.9. STrie(T) *can be constructed on-line in time and space proportional to the size of* STrie(T)*, which is* $O(|T|^2)$ *in the worst case.*

Proof. Each iteration i traverses at most i states on the boundary path, performing $O(1)$ work per state (creating a new state, setting a suffix link, and following one suffix link). The total work is $\sum_{i=1}^{n} O(i) = O(n^2)$. The number of states is bounded by the number of distinct substrings, which is at most $\binom{n+1}{2} + 1 = O(n^2)$. □

1.8.5 Ukkonen's Linear-Time Construction

The quadratic construction above already contains the right algorithmic skeleton. To reach $O(n)$ time we need three key observations, each eliminating a source of redundant work.

Intermezzo — Three tricks for $O(n)$

1. **Open transitions (leaves extend for free).** In the suffix *tree*, a leaf edge is stored as (k, ∞) — the right endpoint "stays at infinity". When we append t_i, the label of every existing leaf edge automatically grows by one character, because ∞ now covers one more position. This means the entire first zone of the boundary path (all the leaves) is updated *implicitly*, at zero cost.
2. **The endpoint only moves to the right.** After processing t_i, the endpoint either stays in the same place or moves deeper along the boundary path for the next step. This means we never need to re-traverse the states above the current active point — the algorithm "resumes where it left off" across iterations.
3. **Suffix links + canonize.** Following a suffix link and then canonizing the reference pair takes amortised $O(1)$ time (the total canonize work over all iterations is $O(n)$).

Open transitions. This is perhaps the most elegant trick. In the suffix trie, extending a leaf means physically adding a new transition. In the suffix tree, leaf edges are labelled (k, ∞) (or in practice, $(k, |T|)$ at the end), where ∞ means "to the current end of the string". When we append t_i to T, the "current end" shifts from $i-1$ to i, so *every* leaf edge automatically extends by one character, without us touching it.

This means that once a leaf is created, it will *never* need to be revisited — it extends passively with every new character. The algorithm only needs to handle the second zone (creating new branches) and detect when to stop (the third zone).

Open transitions eliminate the need to explicitly update leaves: the first zone of the boundary path (Definition 1.8.8) costs $O(0)$ per step instead of $O(j-1)$.

Suffix links in the suffix tree. When we move from the suffix trie to the suffix tree, the suffix function f is generalised to f', now defined only for **branching states** (explicit internal nodes). For a branching state $\bar{x} \neq root$ with label $x = a\alpha$ (where $a \in \Sigma$), the suffix link points to the branching state $\bar{y}$ whose label is $y = \alpha$ (i.e. x with its first character removed): $f'(\bar{x}) = \bar{y}$. A crucial property is that f' maps branching states to branching states: if $\bar{x}$ has at least two children labelled by different characters, so does $\bar{y}$, because every string that branches after x also branches after α. One can also think of "implicit" suffix links between implicit states, but these are never materialised in the data structure — the CANONIZE procedure handles them transparently.

The fact that f' maps branching to branching is important for the algorithm: after following a suffix link, we always land on an explicit node (or the root), so the active point remains a valid reference pair.

The challenge of implicit states. There is one last difficulty that deserves attention. In the second zone of the boundary path (between the active point and the endpoint), the algorithm must create new branches from states that are *not necessarily explicit* in the suffix tree. In the suffix trie, every state was a node, so creating a transition was straightforward. In the suffix tree, the state from which we need to branch may lie *in the middle of a compacted edge* — it is an implicit state. Before we can attach a new leaf, we must first **split** the

edge, promoting the implicit state to an explicit branching node. This is exactly what the TEST-AND-SPLIT procedure does: it checks whether we need to split, and if so, creates the new node. This is a by-product of the compaction trick that keeps the tree linear: the space saving comes at the cost of occasionally "uncompressing" a piece of edge when new branches appear.

The procedures. Ukkonen's algorithm maintains a *reference pair* $(s, (k, i-1))$ for the **active point**: the deepest non-leaf state on the boundary path of STree(T^{i-1}). At each step i, three procedures cooperate:

TEST-AND-SPLIT$(s, (k, p), t)$.
Tests whether the state with reference pair $(s, (k, p))$ already has a t-transition in the suffix tree. If $(s, (k, p))$ is an explicit state, it simply checks for a t-edge from s. If it is an implicit state (mid-edge), it checks whether the next character on the edge equals t. If not, it **splits** the edge, creating a new explicit branching node, and returns it. This is the analogue of the "if there is no t_i-transition, create one" step in Algorithm 6.

CANONIZE$(s, (k, p))$.
Given a reference pair, walks down from explicit state s consuming full edges until the remaining string fits within a single edge. Returns the canonical pair $(s', (k', p))$ where s' is as deep as possible.

UPDATE$(s, (k, i))$.
The main loop. Starting from the active point $(s, (k, i-1))$, it repeatedly calls TEST-AND-SPLIT with the new character t_i. If a split occurs, a new leaf is created, a suffix link is set, and we follow the suffix link of s to move to the next state on the boundary path (then CANONIZE). The loop stops when TEST-AND-SPLIT reports that a t_i-transition already exists (the endpoint has been reached).

Algorithm 7: TEST-AND-SPLIT$(s, (k, p), t)$

```
   // Returns (done?, explicit-state)
 1 if k ≤ p then
 2     let g'(s, (k', p')) = s' be the t_k-transition from s
 3     if t = t_{k'+p-k+1} then
 4         return (true, s)
 5     else
 6         create new state r
 7         replace transition: g'(s, (k', k' + p − k)) = r and
             g'(r, (k' + p − k + 1, p')) = s'
 8         return (false, r)
 9 else
       // (s,(k,p)) is an explicit state (i.e. s itself)
10     if there is no t-transition from s then
11         return (false, s)
12     else
13         return (true, s)
```

Algorithm 8: CANONIZE$(s, (k, p))$

```
if p < k then
    return (s, (k, p))          // already canonical (empty string)
find the t_k-transition g'(s, (k', p')) = s' from s
while p' − k' ≤ p − k do
    k ← k + (p' − k') + 1
    s ← s'
    if k ≤ p then
        find the t_k-transition g'(s, (k', p')) = s' from s
return (s, (k, p))
```

Algorithm 9: UPDATE$(s, (k, i))$

```
// Returns the new active point reference pair
oldr ← root
(done, r) ← TEST-AND-SPLIT(s, (k, i − 1), t_i)
while not done do
    create new leaf r' with transition g'(r, (i, ∞)) = r'
    if oldr ≠ root then
        create suffix link f'(oldr) = r
    oldr ← r
    (s, k) ← CANONIZE(f'(s), (k, i − 1))
    (done, r) ← TEST-AND-SPLIT(s, (k, i − 1), t_i)
if oldr ≠ root then
    f'(oldr) = r                          // final suffix link
return (s, k)
```

The complete algorithm. Algorithm 10 ties everything together. It processes T one character at a time, left to right. After processing t_i, the data structure represents STree(T^i). The variable (s, k) tracks the active point across iterations.

Algorithm 10: Ukkonen's on-line construction of STree(T)

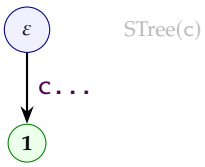
Input: String $T = t_1 t_2 \cdots t_n$ # where # is a unique end-marker
Output: STree(T)

```
create root and ⊥
for every a ∈ Σ ∪ {#} do
    create transition g'(⊥, (−j, −j)) = root for each a = t_{−j}
f'(root) ← ⊥
s ← root;   k ← 1;   i ← 0
while t_{i+1} ≠ end of input do
    i ← i + 1
    (s, k) ← UPDATE(s, (k, i))
    (s, k) ← CANONIZE(s, (k, i))
```

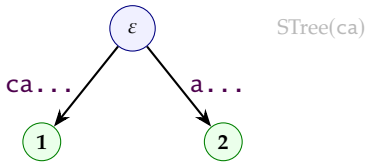
1.8.6 Worked Example

Let us trace Ukkonen's algorithm on T = `cacao$`, the same string from the paper. We denote the characters t_1 = `c`, t_2 = `a`, t_3 = `c`, t_4 = `a`, t_5 = `o`, t_6 = `$`.

Phase $i = 1$ **(**$t_1 =$ **c).** The tree is empty. UPDATE calls TEST-AND-SPLIT at the root: no c-transition exists, so it creates a new leaf with edge $(1, \infty)$ labelled `c...`. Follow the suffix link from *root* to $\perp$; $\perp$ always has a transition back to *root*, so we stop. Active point: $(root, 2)$ i.e. $(root, \varepsilon)$.



Phase $i = 2$ **(**$t_2 =$ **a).** Open transitions: leaf 1's edge $(1, \infty)$ now covers `ca...` — no explicit work. UPDATE from $(root, \varepsilon)$: root has no a-transition, so create leaf 2 with edge $(2, \infty)$. Follow suffix link to $\perp$, stop.



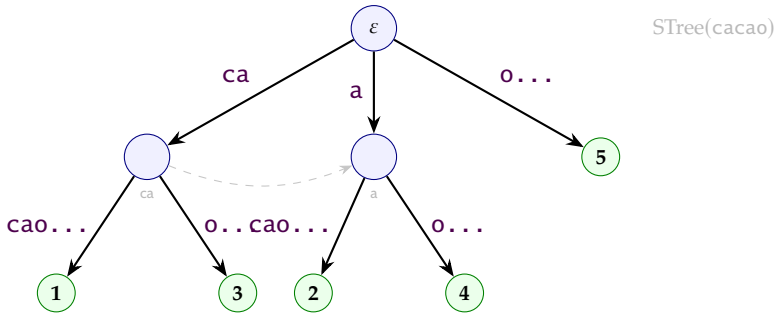
Phase $i = 3$ **(**$t_3 =$ **c).** Open transitions: leaves 1 and 2 extend to cover `cac...` and `ac...`. UPDATE from $(root, \varepsilon)$: root already has a c-transition (to leaf 1), so TEST-AND-SPLIT returns *done* = true immediately. **No new leaf is created.** Active point becomes $(root, (3, 3))$, meaning we are now "inside" the edge to leaf 1, right after the c.

The tree structure is unchanged from phase 2 (only the open transition labels have grown).

This is where the endpoint observation kicks in: the suffix c *already appears in the tree as a prefix of the existing edge, so the construction pauses. When a new character* extends *an existing path in the tree ("implicit extension"), no physical change is needed.*

Phase $i = 4$ **(**$t_4 =$ **a).** Open transitions: leaves grow to cover `caca...` and `aca...`. The active point is $(root, (3, 3))$ = implicit state after c on the root's c-edge. TEST-AND-SPLIT checks: the next character on that edge after c is a — and $t_4 =$ a, so *done* = true. No new node; active point advances to $(root, (3, 4))$ (implicit state for `ca`).

Phase $i = 5$ **(**$t_5 =$ **o).** Open transitions: leaves grow to `cacao...` and `acao...`. Active point is $(root, (3, 4))$ = implicit state for `ca`. TEST-AND-SPLIT: the next character on the edge after `ca` is c (since the edge spells `cac...`), but $t_5 =$ o $\neq$ c. So we **split**: create a new internal node v after `ca`, with edges `cao...` (leaf 1 continues) and o (new leaf 3). Then follow suffix link from $root$ to $\perp$... effectively the next state to process is for the suffix a. The active point for a is $(root, (4, 4))$ = implicit state on the a-edge. TEST-AND-SPLIT at that implicit state: next character after a on that edge is c, but we need o; so split again: create internal node w after a, with edges `cao...` (leaf 2 continues) and o (new leaf 4). Set suffix link $f'(v) = w$. Next state: $(root, \varepsilon)$, check o at root. Root has no o-edge, so create leaf 5. Set suffix link $f'(w) = root$. Follow to $\perp$, stop.



Phase $i = 6$ (t_6 = \$). The end-marker \$ does not appear anywhere in the tree, so every remaining suffix gets a new leaf. After this phase, every suffix ends at its own leaf, and the construction is complete. The final tree is STree(cacao\$).

1.8.7 Complexity Analysis

Theorem 1.8.10 (Ukkonen, 1995). *Algorithm 10 constructs the suffix tree* STree(T) *for a string $T = t_1 \cdots t_n$ on-line in $O(n)$ time (assuming a constant-size alphabet or hash-based edge lookup).*

Formal details — Proof of $O(n)$ time

We split the running time into two components and show each is $O(n)$.

Component 1: everything except CANONIZE. In each phase i, the **while** loop of UPDATE (Algorithm 9) performs one iteration for each new leaf created plus one final check that terminates the loop. Each iteration does $O(1)$ work (one call to TEST-AND-SPLIT with $O(1)$ work, one leaf creation, one suffix link update, one suffix link traversal). The total number of leaves created across all n phases is at most n (each suffix produces exactly one leaf). The number of "stop" checks is also n (one per phase). So this component is $O(n)$.

Component 2: CANONIZE. The procedure CANONIZE$(s, (k, p))$ walks down from explicit state s, consuming edges of the suffix tree one by one. Each edge consumed takes $O(1)$ time (just compare the edge length against $p - k + 1$ and follow the pointer).

To bound the total number of edges consumed by all calls to CANONIZE across the entire algorithm, we use an amortised argument. Define a potential function:

$$\Phi_i \;=\; \text{depth}(s_i) - k_i + 1$$

where (s_i, k_i) is the active point reference pair after phase i, and depth(s_i) is the string-depth (number of characters from root to s_i) of the explicit state component. Intuitively, Φ_i measures the "amount of string" in the reference pair that is accounted for by explicit nodes rather than by the remaining (k, p) part.

At each phase i:

- **The UPDATE loop** follows one suffix link per iteration. Following a suffix link from s to $f'(s)$ decreases depth(s) by at least 1 (the suffix link target is one character shallower). Meanwhile, k does not change during this step. So Φ decreases by at least 1.
- **Each CANONIZE call** consumes edges, each of which increases depth(s) while also increasing k by the same amount. So each edge consumed leaves Φ unchanged.

- **The final CANONIZE** (line 8 of Algorithm 10) moves i from $i-1$ to i, which can increase Φ by at most 1 (the reference pair extends by one character).

Since $\Phi_0 = 0$, $\Phi_n \geq 0$, and each phase increases Φ by at most 1 while each suffix-link step decreases it by at least 1, the total number of suffix-link steps (and hence the total number of CANONIZE edge-consumptions) is bounded by n.

Total: both components are $O(n)$, giving $O(n)$ overall.

Remark 1.8.11. The "on-line" property is valuable in practice: we can build the suffix tree incrementally as characters arrive. After reading $t_1 \cdots t_i$, the tree already represents STree(T^i) and can answer queries about substrings of the text seen so far.

1.8.8 Applications of Suffix Trees

Once STree(T) is built in $O(n)$ time, many string problems that would otherwise require specialised algorithms become simple tree queries:

1. **Substring search.** Given a pattern P of length m, walk down the tree following the characters of P. If we can spell out all of P, the subtree below that point contains all occurrences. Time: $O(m + \text{occ})$ where occ is the number of occurrences.
2. **Longest repeated substring.** The deepest internal node (by string-depth) corresponds to the longest substring that appears at least twice. A single DFS suffices.
3. **Longest common substring of two strings.** Build STree($T_1\$_1T_2\$_2$) and find the deepest internal node that has leaves from both T_1 and T_2 in its subtree.
4. **Number of distinct substrings.** Sum the edge-label lengths over all edges: each edge contributes as many distinct substrings as its label length.

Observation 1.8.12. The suffix tree is the "Swiss army knife" of stringology: a single $O(n)$-time preprocessing step unlocks $O(m)$-time (or better) solutions to a wide family of substring problems.

Remark 1.8.13 (Suffix automata and DAWGs). The suffix trie STrie(T), augmented with suffix links, also provides a natural characterisation of the **suffix automaton** (also known as a *Directed Acyclic Word Graph*, or **DAWG**) of T. The DAWG is the minimal DFA that accepts exactly the set of all suffixes of T; it can be obtained by merging equivalent states of STrie(T) (i.e. states that accept the same set of strings). Ukkonen's paper shows that the suffix links give an elementary criterion for identifying these equivalent states, leading to a simple linear-time construction of the DAWG as well.

1.9 Rabin–Karp Algorithm

All the algorithms we have seen so far—naive, KMP, Boyer–Moore, Aho–Corasick—are *comparison-based*: their main primitive is comparing two characters. The Rabin–Karp algorithm takes a completely different approach: it

replaces character comparisons with *arithmetic operations*. The idea is simple and elegant: interpret strings as numbers, and check whether the "number" of the pattern equals the "number" of the current window in the text.

Rabin–Karp was published by Richard Karp and Michael Rabin in 1987, although the ideas had been circulating since the late 1970s. It is historically important as one of the first randomised *algorithms for string matching.*

1.9.1 Strings as Numbers

Without loss of generality, we can assume $\Sigma = \{0, 1, \ldots, d-1\}$ where $d = |\Sigma|$. If the original alphabet is not numeric, we simply map each character to a digit: for instance, with $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ we set $\mathtt{a} \mapsto 0$, $\mathtt{b} \mapsto 1$, $\mathtt{c} \mapsto 2$, $\mathtt{d} \mapsto 3$. This mapping takes $O(n+m)$ time and is done once as a preprocessing step.

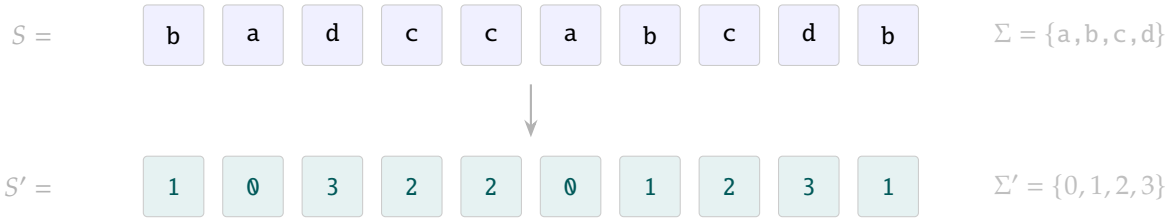


Figure 1.26: Mapping a string S over $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ to a numeric string S' over $\Sigma' = \{0, 1, 2, 3\}$. The mapping is $\mathtt{a} \mapsto 0$, $\mathtt{b} \mapsto 1$, $\mathtt{c} \mapsto 2$, $\mathtt{d} \mapsto 3$.

Once both T and P are expressed as sequences of digits in base d, we can interpret any length-m window of T as a number. Formally, define for each position $s \in \{1, \ldots, n-m+1\}$:

$$t_s \;=\; \big(T[s] \cdot d^{m-1} \;+\; T[s+1] \cdot d^{m-2} \;+\; \cdots \;+\; T[s+m-1] \cdot d^{0}\big)$$

and similarly for the pattern:

$$p \;=\; \big(P[1] \cdot d^{m-1} \;+\; P[2] \cdot d^{m-2} \;+\; \cdots \;+\; P[m] \cdot d^{0}\big).$$

Clearly, there is an occurrence of P at position s if and only if $t_s = p$. If we could compare these numbers in $O(1)$ time, one scan through T would give us an $O(n)$ algorithm.

The key obstacle is that t_s and p can be astronomically large*: a string of m characters in base d represents a number up to $d^m - 1$, which requires $\Omega(m \log d)$ bits. On a standard RAM machine, comparing two numbers of m digits takes $O(m)$ time—no better than character-by-character comparison.*

1.9.2 The Fingerprinting Idea

The Rabin–Karp approach side-steps the large-number problem by working *modulo a prime* q. Instead of comparing t_s and p directly, we compare their **fingerprints**:

$$t_s \bmod q \quad \stackrel{?}{=} \quad p \bmod q.$$

Since the fingerprints are numbers in $\{0, 1, \ldots, q-1\}$, each comparison takes $O(1)$ time (assuming q fits in a machine word).

Two things can happen:

1. $t_s \bmod q \neq p \bmod q$: then certainly $t_s \neq p$, so there is **no match** at position s. This is a *true negative*—we can safely skip this position.
2. $t_s \bmod q = p \bmod q$: this is a **hit**. Unfortunately, it does *not* guarantee $t_s = p$: two different numbers can have the same remainder modulo q. We call this a **spurious hit** (or *false positive*). When a hit occurs, we must verify character by character whether $T[s, s+m-1] = P$, which costs $O(m)$.

Think of it as a cheap "quick reject" filter: most positions fail the modular test and are discarded instantly. Only the rare hits require the expensive $O(m)$ verification.

1.9.3 Rolling Hash via Horner's Rule

Computing $p \bmod q$ from scratch is easy: we apply Horner's rule.

$$p \bmod q \;=\; \big(\big((P[1]\cdot d + P[2])\cdot d + P[3]\big)\cdot d + \cdots + P[m]\big) \bmod q.$$

This takes $O(m)$ multiplications and additions, all performed modulo q so that intermediate values never exceed $q \cdot d$. The same computation gives us $t_1 \bmod q$ (the fingerprint of the first window).

The crucial observation is that we can compute $t_{s+1} \bmod q$ from $t_s \bmod q$ in $O(1)$ time, without scanning m characters again. The relationship between consecutive windows is:

$$\begin{aligned} t_{s+1} &= \big(T[s+1]\cdot d^{m-1} + T[s+2]\cdot d^{m-2} + \cdots + T[s+m]\cdot d^{0}\big) \\ &= \big(t_s - T[s]\cdot d^{m-1}\big)\cdot d \;+\; T[s+m]. \end{aligned}$$

In words: *remove* the contribution of the leftmost character $T[s]$, *shift* everything one position to the left (multiply by d), and *add* the new rightmost character $T[s+m]$.

Taking everything modulo q:

$$t_{s+1} \bmod q \;=\; \Big[\big(t_s - T[s]\cdot h\big)\cdot d + T[s+m]\Big] \bmod q, \qquad \text{where } h = d^{m-1} \bmod q.$$

The value $h = d^{m-1} \bmod q$ is precomputed once in $O(m)$ time (or $O(\log m)$ using fast exponentiation).

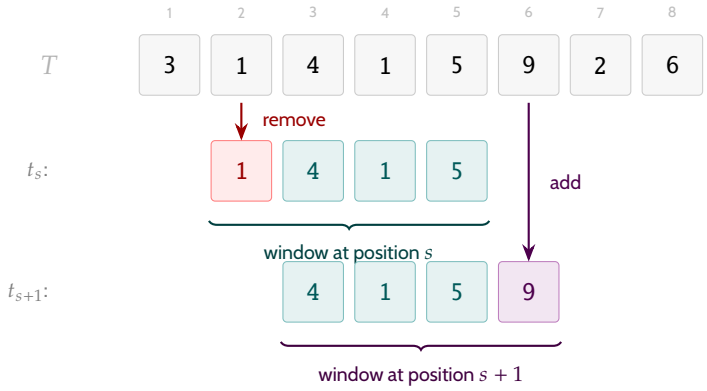


Figure 1.27: Rolling hash: computing t_{s+1} from t_s. The **red** digit $T[s]$ is *removed* from the high end, the remaining digits are shifted left (multiplied by d), and the **violet** digit $T[s+m]$ is *added* at the low end. Each update costs $O(1)$ arithmetic operations modulo q.

This "rolling" update is the heart of Rabin–Karp: after the initial $O(m)$ setup, each new window costs only $O(1)$ to fingerprint.

1.9.4 The Algorithm

Putting the pieces together, the Rabin–Karp algorithm works in three phases:

1. **Preprocessing.** Compute $h = d^{m-1} \bmod q$. Compute $p = P \bmod q$ using Horner's rule. Compute $t_1 = T[1,m] \bmod q$ using Horner's rule. This takes $O(m)$ time.
2. **Scanning.** For each position $s = 1, 2, \ldots, n-m+1$:

- If $t_s \equiv p \pmod{q}$: *hit* — verify character by character whether $T[s, s+m-1] = P$. If yes, report occurrence at position s.
- If $t_s \not\equiv p \pmod{q}$: skip (no match possible).
- Compute t_{s+1} from t_s using the rolling hash formula.

3. **Output.** All reported positions are true occurrences.

Algorithm 11: Rabin–Karp

Input: Text $T[1 \ldots n]$, Pattern $P[1 \ldots m]$, alphabet size d, prime q
Output: All positions s where P occurs in T

```
h ← d^(m−1) mod q
// Horner's rule for pattern fingerprint
p ← 0
for j ← 1 to m do
    p ← (p · d + P[j]) mod q
// Horner's rule for first window
t ← 0
for j ← 1 to m do
    t ← (t · d + T[j]) mod q
// Slide the window
for s ← 1 to n − m + 1 do
    if p = t then
        // Hit: verify character by character
        k ← 0
        while k < m and P[k + 1] = T[s + k] do
            k ← k + 1
        if k = m then
            output s
    if s < n − m + 1 then
        t ← ((t − T[s] · h) · d + T[s + m]) mod q
```

1.9.5 Correctness

The correctness relies on two observations:

1. **No false negatives.** If $t_s \bmod q \neq p \bmod q$, then certainly $t_s \neq p$, so skipping position s is safe. We never miss a true occurrence.
2. **Hits are verified.** When $t_s \bmod q = p \bmod q$, the algorithm performs a full character-by-character comparison. A position s is reported as an occurrence *only if* $T[s, s+m-1] = P$ holds exactly. This eliminates all spurious hits from the output.

■ **Formal details — Rolling hash recurrence**

Lemma 1.9.1. *The rolling hash formula correctly maintains $t \equiv t_s \pmod{q}$ throughout the algorithm.*

Proof. After the preprocessing loop, $t = t_1 \bmod q$ by Horner's rule. For the inductive step,

suppose $t = t_s \bmod q$ at the start of iteration s. The update computes:

$$t' = \big((t_s - T[s] \cdot d^{m-1}) \cdot d + T[s+m]\big) \bmod q.$$

But $t_{s+1} = (t_s - T[s] \cdot d^{m-1}) \cdot d + T[s+m]$ by the definition of the window values, so $t' = t_{s+1} \bmod q$. □

1.9.6 Complexity

The running time depends on the number of spurious hits.

Preprocessing. Computing h, p, and t_1 takes $O(m)$ time.

Scanning. In each of the $n - m + 1$ iterations, the rolling hash update and the modular comparison take $O(1)$. The expensive part is the verification upon a hit: each verification costs $O(m)$.

Let v denote the number of true occurrences and c the number of spurious hits. The total running time is:

$$O(m) + O(n-m) + (v+c) \cdot O(m) \;=\; O(n + m\,(v+c)).$$

Worst case. In the worst case, *every* position is a hit (true or spurious), giving $v + c = n - m + 1$ and a total time of $O(nm)$—no better than the naive algorithm. This happens, for instance, when $q = 2$, $P = \texttt{aa}$, and $T = \texttt{aaa...a}$: every window matches both modulo q and exactly.

Expected case. The key insight is that by choosing q to be a sufficiently large *random prime*, the probability of a spurious hit becomes very small.

> **Definition 1.9.2** (Fingerprint)**.** For a positive integer q, the **fingerprint** of P is $H_q(P) = p \bmod q$, and the fingerprint of window s is $H_q(T_s) = t_s \bmod q$. A **false match** at position s occurs when $H_q(P) = H_q(T_s)$ but $P \neq T[s, s+m-1]$.

If $P \neq T[s, s+m-1]$, then $p \neq t_s$, so q can only produce a false match if q divides $|p - t_s|$. Since $|p - t_s| < d^m$, it has at most $\pi(d^m)$ distinct prime divisors, where $\pi(u)$ denotes the number of primes up to u.

If we pick q uniformly at random among all primes up to some bound I, the probability that q divides $|p - t_s|$ at any single position s is at most $\pi(d^m)/\pi(I)$.

> **Theorem 1.9.3** (Karp–Rabin, 1987)**.** *If q is a random prime in $\{1, \dots, I\}$ with $I = nm^2$, then the probability of a false match at* any single *position is at most $O(1/m)$.*

When q is large enough (say $q \geq nm$) and chosen as a random prime, the expected number of spurious hits across all $n - m + 1$ positions is $O(n/m)$, giving an expected total time of:

$$O(n+m) + O(n/m) \cdot O(m) = O(n+m).$$

By the prime number theorem, $\pi(u) \approx u/\ln u$, so the bound works out to roughly $\frac{\pi(d^m)}{\pi(nm^2)} \approx \frac{m \ln d}{\ln(nm^2)} \cdot \frac{1}{m} = O(1/m)$ for d constant.

If the number of true occurrences v is constant, this is **expected linear time**.

1.9.7 Las Vegas vs. Monte Carlo

The Rabin–Karp algorithm as presented above always verifies hits, so it *never reports a false positive*. This makes it a **Las Vegas** randomised algorithm: the output is always correct, but the running time is a random variable (it depends on how many spurious hits occur).

Las Vegas variant.
Always verify hits. Output is *always correct*. Worst-case time $O(nm)$, expected time $O(n + m)$.

Monte Carlo variant.
Skip the verification step: when $t_s \bmod q = p \bmod q$, report position s as an occurrence without checking. Running time is $O(n + m)$ *always* (deterministic), but the output may contain false positives. The probability of any false positive is bounded by Theorem 1.9.3.

Remark 1.9.4. We can further reduce the error probability by using k independent random primes $q_1, \ldots, q_k$ and declaring a match only when *all* k fingerprints agree. The probability of a false match drops to $O\left((1/m)^k\right)$—exponentially small in k—at the cost of a factor k in running time.

1.9.8 Properties

	Online	Real-time	Blind	Succinct
Rabin–Karp (Las Vegas)	✓	×	×	✓
Rabin–Karp (Monte Carlo)	✓	✓	✓	✓

The algorithm is **online**: it processes T left-to-right. The Monte Carlo variant is even **real-time** (each character costs $O(1)$ amortised) and **blind** (no backtracking), since it never re-reads characters of T. The Las Vegas variant can backtrack during verification, so it is neither real-time nor blind.

Both variants are **succinct**: they use only $O(1)$ extra space beyond storing P and T (just the variables h, p, t, and the current position).

The Rabin–Karp approach generalises naturally to multi-pattern *matching: compute fingerprints for all patterns and store them in a hash set. Each position of T is checked against the set in $O(1)$ expected time. This gives expected $O(n + km)$ time for k patterns of length m, without building an Aho–Corasick automaton.*

1.10 Shift-And Algorithm

We turn to a different perspective on exact pattern matching—one that works *from the bottom up*, at the level of individual bits inside machine words.

The Shift-And algorithm combines two ideas:

1. **Bit vectors.** Represent states as sequences of bits (words of memory) so that many comparisons happen in a single CPU instruction.
2. **Dynamic programming.** Define a matrix that records which prefixes of P are "alive" at every position of T, then compute it column by column.

Before diving into the algorithm itself, let us briefly recall these two ingredients.

The same matrix will reappear when we tackle approximate *string matching (matching with errors).*

Intermezzo — Bit vectors and bit-level parallelism

A *bit vector* is simply a sequence of bits—zeros and ones—packed into a machine word. A modern CPU has registers of width w bits (typically $w = 64$); a single register can therefore store 64 independent boolean values.

The key advantage is *parallelism for free*: a bitwise AND (&), OR (|), or shift (<<, >>) on a w-bit register operates on all w bits in one clock cycle. If we encode a problem so that each "slot" of the answer is one bit, we effectively perform w operations at the cost of one.

Example. Let $A = 10110$ and $B = 11010$. Then:

$$\begin{aligned} A \,\&\, B &= 10010 \quad \text{(AND: both bits must be 1)} \\ A \mid B &= 11110 \quad \text{(OR: at least one bit is 1)} \\ A \ll 1 &= 01100 \quad \text{(left-shift by one position)} \end{aligned}$$

Each of these is a *single CPU instruction*, regardless of how many bits are in the word.

Intermezzo — Dynamic programming

Dynamic programming (often abbreviated DP) is a general technique for solving problems that can be decomposed into overlapping subproblems. The idea is straightforward:

1. Define a table (matrix, array, ...) whose entries represent solutions to "smaller" instances of the problem.
2. Write a *recurrence*: a formula that computes each entry from entries that have already been filled in.
3. Fill the table in an order that respects the dependencies, so that every entry we need is ready when we need it.

The payoff is that we solve each subproblem exactly once and store its answer, rather than recomputing it every time it is needed (which is what a naive recursive approach would do).

In the context of the Shift-And algorithm, the "table" is a bit matrix M where $M[i, j]$ tells us whether a certain prefix of P matches a certain suffix of T. We will fill this table column by column, left to right, using the recurrence derived below.

Important. The full $m \times n$ matrix is *never materialised in memory*. Since each column depends only on the immediately preceding column, we only need to keep *one column at a time*—a single machine word when $m \leq w$. This is a common space optimisation in dynamic programming.

1.10.1 Working assumption

> *Remark* 1.10.1. We assume $|P| \leq w$, where w is the machine word size (typically 64). Under this assumption every column of the matrix fits in a single register, and shift/AND operations cost $O(1)$.

Working with a very large text T is not a problem: the algorithm scans T once, left to right, spending constant time per character. When $|P| > w$ the bit vector is split into $\lceil m/w \rceil$ machine words and the cost per text character rises to $O(\lceil m/w \rceil)$.

1.10.2 The matrix

Define a bit matrix M of size $m \times n$ (rows indexed by the pattern, columns by the text):

Definition 1.10.2 (Shift-And matrix)**.**

$$M[i,j] \;=\; \begin{cases} 1 & \text{if } P[1\,..\,i] \text{ is a suffix of } T[1\,..\,j], \\ 0 & \text{otherwise.} \end{cases}$$

In words: $M[i,j] = 1$ exactly when the first i characters of the pattern match the last i characters of $T[1\,..\,j]$.

Observation 1.10.3. An occurrence of P ending at position j of T corresponds to $M[m,j] = 1$. Therefore, the set of all 1s in the *last row* gives us every occurrence.

As noted above, this matrix is never stored in full: the algorithm computes it one column at a time, overwriting the previous column at each step.

The 1s in the other rows carry information too: they mark partial matches that may extend as we read more of T.

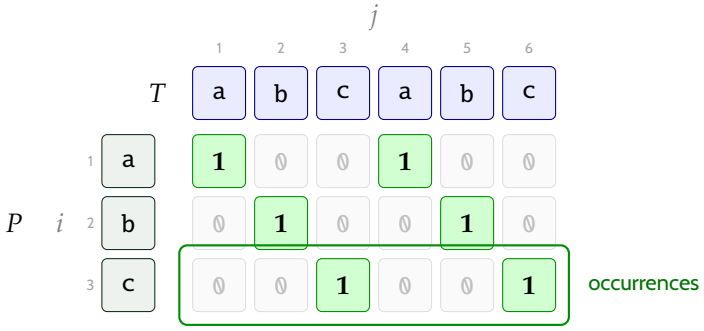


Figure 1.28: Shift-And matrix for $P = \texttt{abc}$ and $T = \texttt{abcabc}$. The 1s in the last row ($i = m = 3$) mark positions $j = 3$ and $j = 6$ where an occurrence of P ends. Note that the full matrix is never stored: the algorithm keeps only one column at a time.

1.10.3 Recurrence

How do we compute $M[i,j]$? A prefix $P[1\,..\,i]$ is a suffix of $T[1\,..\,j]$ if and only if two conditions hold simultaneously:

1. $P[1\,..\,i{-}1]$ is already a suffix of $T[1\,..\,j{-}1]$ (i.e. $M[i{-}1,\ j{-}1] = 1$), and
2. The last characters match: $P[i] = T[j]$.

This is the shift*: the shorter prefix was alive at the previous position.*

Formally:

$$M[i,j] = 1 \quad\Longleftrightarrow\quad M[i{-}1,\ j{-}1] = 1 \ \text{ AND } \ P[i] = T[j],$$

with the convention that $M[0,j] = 1$ for all j (the empty prefix is always a suffix).

This is where the name comes from: condition (a) is a **shift** (look one row up and one column left), and condition (b) is an **AND** with the alphabet mask.

1.10.4 Bit-vector reformulation

The key insight is that we can compute an entire column of M in one shot by encoding it as a bit vector.

Alphabet masks

For each character $x \in \Sigma$, pre-compute a bit vector U_x of length m:

$$U_x[i] = \begin{cases} 1 & \text{if } P[i] = x, \\ 0 & \text{otherwise.} \end{cases}$$

This records every position in the pattern where character x appears.

Example 1.10.4. For $P = \texttt{abab}$ over $\Sigma = \{a, b\}$:

$$U_a = 1010, \qquad U_b = 0101.$$

Column update

Let M_j denote the j-th column of M, viewed as a single bit vector of length m (bit 1 is the top, bit m is the bottom). The recurrence becomes:

$$M_j \;=\; \big(\text{shift}(M_{j-1}) \;\&\; U_{T[j]}\big),$$

where shift means:

- Shift the vector *down* by one position (toward higher indices),
- Insert a 1 at the top (position 1), because the empty prefix is always alive,
- The bit that falls off the bottom ($M[m]$) is discarded.

On a machine register this is:

$$M_j \;=\; \big((M_{j-1} \ll 1) \mid 1\big) \;\&\; U_{T[j]}.$$

In C-like code the "shift down" on a bit vector stored LSB = position 1 is simply a left shift: `(M<<1) | 1`.

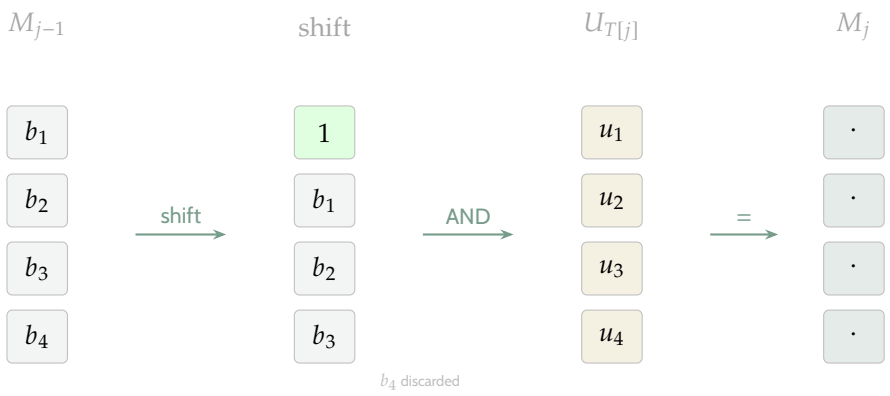


Figure 1.29: One step of the Shift-And algorithm. The previous column is shifted down (with a 1 inserted at the top), then AND-ed with the alphabet mask for the current text character.

1.10.5 Algorithm

Algorithm 12: Shift-And exact matching

Input: Text $T[1\,..\,n]$, pattern $P[1\,..\,m]$ with $m \le w$
Output: All positions j where P occurs in T

```
// Pre-compute alphabet masks
foreach x ∈ Σ do
    U_x ← 0
    for i ← 1 to m do
        if P[i] = x then
            U_x ← U_x | (1 ≪ (i−1))

// Scan text
M ← 0
for j ← 1 to n do
    M ← ((M ≪ 1) | 1) & U_{T[j]}
    if M & (1 ≪ (m−1)) ≠ 0 then
        output "occurrence ending at position j"
```

1.10.6 Complexity

- **Time.** Pre-processing the alphabet masks takes $O(|\Sigma| \cdot m)$. When $m \le w$, each iteration of the main loop costs $O(1)$ (a shift, an OR, and an AND—three CPU instructions). The loop runs n times, so the total search time is $O(n)$.
  If $m > w$, each column requires $\lceil m/w \rceil$ words, and the cost per text character rises to $O(\lceil m/w \rceil)$, giving $O(n \cdot \lceil m/w \rceil)$ overall.
- **Space.** $O(|\Sigma| \cdot \lceil m/w \rceil)$ for the masks, plus a single register for the current column M. The full $m \times n$ matrix is *never materialised*: we only keep one column at a time.

> *Remark* 1.10.5. The Shift-And algorithm is particularly attractive in bioinformatics (DNA sequencing), where the alphabet is small ($|\Sigma| = 4$), patterns are short, and the text can be very long. The same dynamic programming framework extends naturally to *approximate matching* (allowing a bounded number of mismatches, insertions, or deletions)—a topic we will revisit later.

CHAPTER 2

Randomized Algorithms

CHAPTER 3

Data Compression