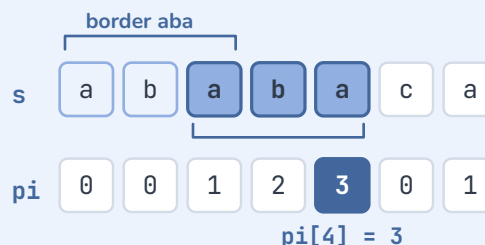


Selected string techniques

Chapter 4.5 · Overall level: Advanced



Searching for a word in a long text sounds easy, until the word keeps *almost* matching. *After a near miss, how far can you safely slide the word forward without ever checking a letter twice?* The answer is stored in one small array, the prefix function.

This chapter builds that array, uses it for exact pattern matching (the Knuth–Morris–Pratt method), and then reads periods and repeated blocks straight from it. It ends with a careful look at hashing, a popular shortcut that can be wrong.

4.5.1 Prefixes, suffixes and borders

Strings in this chapter are numbered from 0, so $s[0..i]$ is the first $i + 1$ characters. A *prefix* of s is a beginning part of it, a *suffix* is an end part, and a *substring* is any contiguous part. For $s = \text{ababa}$:

Kind	Examples for ababa
prefixes	a, ab, aba, abab, ababa (and the empty string)
suffixes	a, ba, aba, baba, ababa (and the empty string)
substrings	every contiguous piece, such as bab or ab
proper prefixes	a, ab, aba, abab: the whole string is excluded

A **border** of a string is a *proper* prefix that is also a suffix. The word “proper” matters: the whole string is always both a prefix and a suffix of itself, so allowing it would make every string its own border. The borders of ababa are a and aba (and the empty one). Borders may overlap themselves: the prefix aba uses positions 0–2 and the suffix aba uses positions 2–4.

Two occurrences of a pattern in a text *overlap* when they share positions. The pattern aba occurs in the text ababa starting at positions 1 and 3 (counting from 1), and both occurrences use the middle a. Overlaps and borders are linked: two occurrences of a pattern of length m can start $d < m$ places apart only if the pattern has a border of length $m - d$, because the end of the first copy must equal the start of the second.

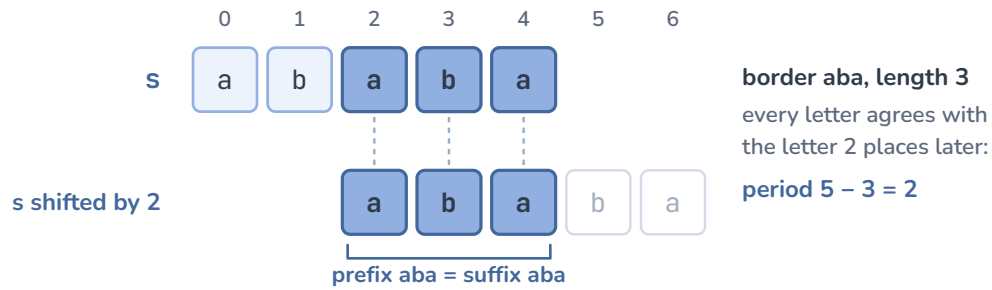


Figure 4.5.1 — A border is a part that lines up with itself after a shift. A border of length b gives a shift of $n - b$ under which every overlapping letter agrees.

4.5.2 The prefix function

For every position i , the **prefix function** stores the length of the longest border of the prefix $s[0..i]$:

$pi[i]$ = length of the longest proper prefix of $s[0..i]$
that is also a suffix of $s[0..i]$

For the chapter's example $s = \text{ababaca}$:

i	Prefix $s[0..i]$	Longest border	$pi[i]$
0	a	(empty)	0
1	ab	(empty)	0
2	aba	a	1
3	abab	ab	2
4	ababa	aba	3
5	ababac	(empty)	0
6	ababaca	a	1

Note that $pi[0] = 0$ always: a one-letter string has only the empty proper prefix. Checking every length of every prefix directly would take about n^2 comparisons of whole pieces, far too slow for a million characters. The next section computes the whole array in $O(n)$ by reusing what is already known.

4.5.3 Reusing the previous border

Suppose $\text{pi}[0..i-1]$ is known and we want $\text{pi}[i]$. Take a border of $s[0..i]$ of length $k \geq 1$. Remove its last character from both copies: what is left is a border of $s[0..i-1]$ of length $k-1$, followed by the character $s[i]$ in both places. So every nonempty border of the new prefix is *an old border extended by one character*, and the old border of length j can be extended exactly when $s[j] = s[i]$ (the character right after the prefix copy must match the new character).

So we try the borders of `s[0..i - 1]` from the longest down, and take the first that extends. The longest is `j = pi[i - 1]`. The key fact is how to find the **next shorter** one:

Border of a border. If j is a border length of a string, the next shorter border length is $\text{pi}[j - 1]$.

Why: a shorter border is a prefix of the string and also a suffix of it. Being shorter than the border of length j , it is a prefix and a suffix of that border too, so it is a border of $s[0..j-1]$. Conversely, every border of $s[0..j-1]$ is a border of the whole string. So the borders shorter than j are exactly the borders of $s[0..j-1]$, and the longest of them has length $\text{pi}[j-1]$. This is the *fallback* step: on a mismatch, set $j \leftarrow \text{pi}[j-1]$ and try again.

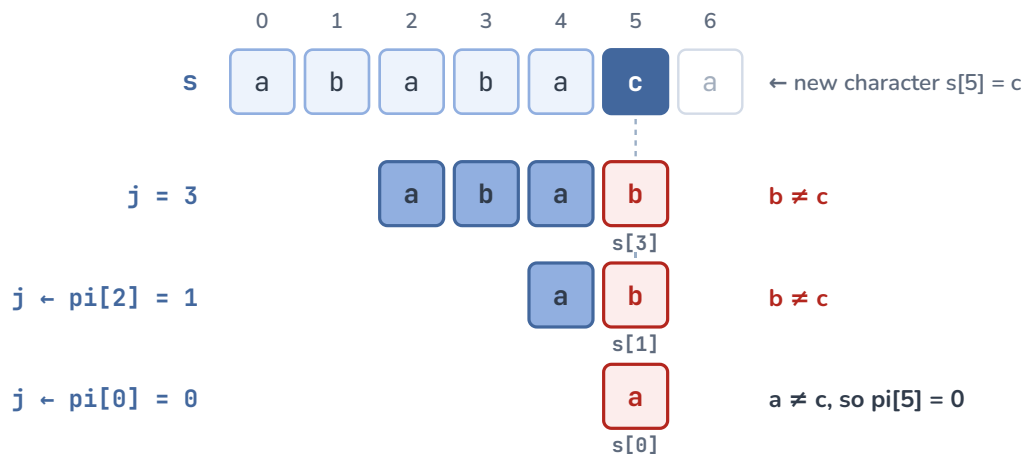


Figure 4.5.2 — Fallback keeps only a suffix that is also a valid prefix, avoiding comparisons known to fail. Each row aligns the candidate border with the end of ababa; the dotted guide marks the compared column.

Index $i = 6$ then starts from $j = pi[5] = 0$: $s[6] = a = s[0]$, so $pi[6] = 1$.

Pseudocode

Prefix function: $O(n)$ time, $O(n)$ memory

```

pi[0] ← 0
for i from 1 to n - 1:
    j ← pi[i - 1]           # longest border of s[0..i - 1]
    while j > 0 and s[i] ≠ s[j]:
        j ← pi[j - 1]       # next shorter border
    if s[i] = s[j]:
        j ← j + 1
    pi[i] ← j

```

The loop tries every border of $s[0..i - 1]$ in decreasing length, and stops at the first that extends, so the result is the **longest** border of $s[0..i]$. If none extends, j reaches 0 and the final check tests the one-letter border, $s[0] = s[i]$.

Why the loop is linear overall

A single step can fall back several times, so it is not obvious that the total is $O(n)$. Watch the value of j as it is carried from one i to the next. Each i increases it by **at most 1**, so over the whole run it rises by at most $n - 1$. Every fallback lowers it by at least 1, and it never goes below 0. A value cannot fall more than it has risen, so there are at most $n - 1$ fallbacks in total. This kind of counting, where expensive steps are paid for by earlier cheap ones, is called *amortized* analysis.

The complete trace for ababaca shows the single expensive step, at $i = 5$, paid for by the three extensions before it:

i	$s[i]$	Start $j = \text{pi}[i - 1]$	Fallbacks and final check	$\text{pi}[i]$
1	b	0	none; $b \neq s[0] = a$	0
2	a	0	none; $a = s[0]$, extend	1
3	b	1	none; $b = s[1]$, extend	2
4	a	2	none; $a = s[2]$, extend	3
5	c	3	$3 \rightarrow 1 \rightarrow 0$; $c \neq s[0]$	0
6	a	0	none; $a = s[0]$, extend	1

4.5.4 Finding a pattern with KMP

To find every occurrence of a pattern p (length m) in a text t , compute the prefix function of p and then scan the text once. The state j is the length of the longest prefix of p that ends at the current text position. Each new text character $t[i]$ either extends that prefix or causes the same fallbacks as before.

Pseudocode

KMP search: $O(m + |t|)$ after the prefix function of p

```

j ← 0
for i from 0 to |t| - 1:
    while j > 0 and t[i] ≠ p[j]:
        j ← pi[j - 1]
    if t[i] = p[j]:
        j ← j + 1
    if j = m:
        report start i - m + 1      # p ends at text index i
        j ← pi[j - 1]             # 0-based; add 1 for 1-based output
                                   # keep the overlap

```

Why it works. The scan keeps the invariant “ j is the length of the longest prefix of p that is a suffix of the text read so far”. The argument is the same as for the prefix function: the candidates for the new state are the borders of the old matched part, tried in decreasing length. A match ends at i exactly when j reaches m . By the same amortized count the cost is $O(m + |t|)$. This is the Knuth–Morris–Pratt (KMP) method.

Overlapping matches

After a full match, do not reset j to 0. The end of the match may be the start of the next one, so fall back to the longest border of the whole pattern. Searching for aba in ababa:

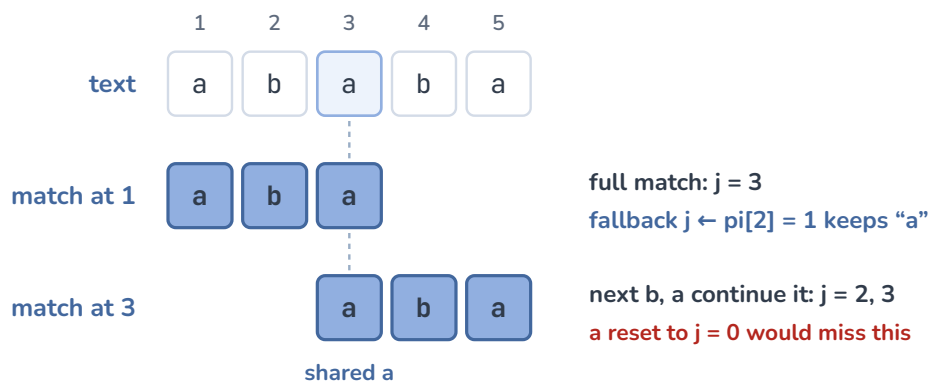


Figure 4.5.3 — A completed match may contain the beginning of the next match, so the state does not always reset to zero.

i	0	1	2	3	4
t[i]	a	b	a	b	a
j after step	1	2	3 → match at 1, $j \leftarrow 1$	2	3 → match at 3, $j \leftarrow 1$

Another common way to get the same result is to compute the prefix function of $p + \# + t$, with a separator that occurs in neither string, and report every position where the value equals m .

Strings and indices in four languages

Text input is read as one token, and characters are compared by their codes. Each language has its own way to reach character i :

Language	Reading s	Character i	Length
Python	<code>sys.stdin.buffer.read().split()[0]</code>	<code>s[i]</code> (an int: bytes)	<code>len(s)</code>
C++	<code>string s; cin >> s;</code>	<code>s[i]</code> (char)	<code>(int) s.size()</code>
Java	<code>in.readLine().trim().toArray()</code>	<code>s[i]</code> (char)	<code>s.length</code>
JavaScript	<code>readFileSync(0, "utf8").trim()</code>	<code>s.charCodeAt(i)</code>	<code>s.length</code>

Two details catch people. First, `s.size()` in C++ is unsigned, so store it in an `int` before computing $n - 1$ or similar. Second, the arrays are 0-based while statements usually number positions from 1: a match ending at 0-based index i starts at 0-based $i - m + 1$, which is position $i - m + 2$ in 1-based output. Build long outputs once (with `join` or a `StringBuilder`) instead of printing a million separate pieces.

4.5.5 Periods and repeating blocks

A string of length n has *period* q (with $1 \leq q \leq n$) if $s[x] = s[x + q]$ whenever both positions exist. Figure 4.5.1 shows the link: **a border of length b is the same thing as a period $n - b$** , because sliding the string by $n - b$ places lines its prefix of length b up with its suffix. So the longest border gives the **smallest** period:

$$\text{smallest period } q = n - \text{pi}[n - 1]$$

A *repeating block* is a prefix whose copies, placed end to end, form exactly the string. A block of length k works precisely when k is a period **and** k divides n . For `ababab`, `pi[5] = 4`, so $q = 6 - 4 = 2$, which divides 6: the block is `ab`, used three times. For `ababa`, $q = 5 - 3 = 2$, but 2 does not divide 5, and `ab`, `ab`, `a` is not whole copies.

Could a longer block still work when q does not divide n ? No. Periods have a useful property: **if q and k are periods with $q + k \leq n$, then $k - q$ is also a period** (for $q < k$). To check a position x : if $x + k$ exists, then $s[x] = s[x + k] = s[x + k - q]$; otherwise $x \geq n - k \geq q$, so $s[x] = s[x - q] = s[x - q + k]$. Repeating this subtraction, as in Euclid's algorithm, shows that $\text{gcd}(q, k)$ is a period. A block length $k < n$ that divides n is at most $n/2$, and $q \leq k$, so $q + k \leq n$. Then $\text{gcd}(q, k)$ is a period no larger than the smallest one, so it equals q , and q divides k and therefore n .

All borders: following the chain

The border-of-a-border fact also lists every border of the whole string, not just the longest. Start with $k = \text{pi}[n - 1]$ and repeat $k \leftarrow \text{pi}[k - 1]$ until $k = 0$. The values visited are all the nonempty border lengths, in decreasing order, each exactly once, because each step moves to the next shorter border. For *ababa*: $\text{pi}[4] = 3$, then $\text{pi}[2] = 1$, then $\text{pi}[0] = 0$. The borders have lengths 3 and 1, that is *aba* and *a*.

Be careful not to confuse this chain with the whole *pi* array. The array describes borders of every prefix; only the chain starting at the last entry describes borders of the full string.

4.5.6 An optional comparison: hashing

Another well-known way to compare strings quickly turns each string into a number, a *hash*. A common choice treats letters as digits $a = 1, \dots, z = 26$ in base B and keeps the remainder modulo M :

$$h(c_0 c_1 \dots c_k) = (c_0 \cdot B^k + c_1 \cdot B^{(k-1)} + \dots + c_k) \bmod M$$

With prefix hashes (the idea of Chapter 2.1), the hash of any substring is available in $O(1)$, so two substrings can be *compared by hash* in $O(1)$. It is fast, but it has a real limit: **equal hashes do not prove equal strings**.

The reason is counting. A hash takes at most M different values, while there are 26^L strings of length L . With $M = 1,000,000,007$ and $L = 7$, there are $26^7 = 8,031,810,176$ strings for about a billion values, so some different strings **must** share a value. A tiny example: with $B = 31$ and $M = 101$,

$$h(ab) = 1 \cdot 31 + 2 = 33 \qquad h(dj) = 4 \cdot 31 + 10 = 134 \equiv 33 \pmod{101}$$

A fixed base and modulus can also be attacked on purpose: test data can be built so that many different strings collide. A hash comparison is therefore at best a quick filter, and a real answer still needs a letter-by-letter check of each candidate. KMP and the prefix function need no luck at all: every answer they give is proved correct, and their running time is guaranteed. That is why all the practice problems in this chapter are meant to be solved with them.

4.5.7 Code in four languages

Each program solves the first practice problem, **Prefix border lengths**. The input is one nonempty lowercase string s of up to 1,000,000 characters. The program prints the n prefix-function values in order, separated by spaces. Value number k belongs to the prefix of length k , so no conversion between 0-based and 1-based numbering is needed.

Sample input	Sample output
ababaca	0 0 1 2 3 0 1

Python 3

```
import sys

def main():
    s = sys.stdin.buffer.read().split()[0] # bytes: s[i] is a character code
    n = len(s)
    pi = [0] * n
    for i in range(1, n):
        j = pi[i - 1] # longest border of s[0..i-1]
        while j > 0 and s[i] != s[j]:
            j = pi[j - 1] # fall back to the next shorter border
        if s[i] == s[j]:
            j += 1
        pi[i] = j
    print(" ".join(map(str, pi)))

main()
```

C++17

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    string s;
    cin >> s;
    int n = s.size();
    vector<int> pi(n, 0);
    for (int i = 1; i < n; i++) {
        int j = pi[i - 1]; // longest border of s[0..i-1]
        while (j > 0 && s[i] != s[j])
            j = pi[j - 1]; // fall back to the next shorter border
        if (s[i] == s[j]) j++;
        pi[i] = j;
    }
    for (int i = 0; i < n; i++) cout << pi[i] << (i + 1 < n ? ' ' : '\n');
    return 0;
}
```

Java 17

```
import java.io.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        char[] s = in.readLine().trim().toCharArray();
        int n = s.length;
        int[] pi = new int[n];
        for (int i = 1; i < n; i++) {
            int j = pi[i - 1];           // longest border of s[0..i-1]
            while (j > 0 && s[i] != s[j])
                j = pi[j - 1];          // fall back to the next shorter border
            if (s[i] == s[j]) j++;
            pi[i] = j;
        }
        StringBuilder out = new StringBuilder();
        for (int i = 0; i < n; i++) {
            if (i > 0) out.append(' ');
            out.append(pi[i]);
        }
        System.out.println(out);
    }
}
```

JavaScript (Node)

```
const s = require("fs").readFileSync(0, "utf8").trim();
const n = s.length;
const pi = new Int32Array(n);
for (let i = 1; i < n; i++) {
    let j = pi[i - 1];           // longest border of s[0..i-1]
    while (j > 0 && s.charCodeAt(i) !== s.charCodeAt(j)) {
        j = pi[j - 1];          // fall back to the next shorter border
    }
    if (s.charCodeAt(i) === s.charCodeAt(j)) j++;
    pi[i] = j;
}
console.log(pi.join(" "));
```

4.5.8 Cost

Building the prefix function takes $O(n)$ time, at most about $2n$ character comparisons by the amortized count, and $O(n)$ memory for the array. KMP adds $O(|t|)$ time and needs no copy of the text beyond what is read. All values are lengths at most 10^6 , so 32-bit integers are enough. The output, up to a million numbers, is the largest part of the work in practice, which is why it is built in one piece.

4.5.9 Common mistakes

- **Resetting to zero after a full match.** Searching for aba in ababa then finds only position 1. Fall back with $j \leftarrow \text{pi}[j - 1]$.
- **Using $\text{pi}[j]$ instead of $\text{pi}[j - 1]$.** The matched part has length j , so its last index is $j - 1$. The entry $\text{pi}[j]$ describes a prefix one character longer that has not been matched.
- **Treating equal hashes as equal strings.** Different strings can share a hash value, as ab and dj do for base 31 modulo 101.
- **Forgetting that a border is proper.** $\text{pi}[i]$ is never $i + 1$; for a string of one letter it is 0.
- **Mixing 0-based and 1-based positions.** A match ending at index i starts at 1-based position $i - m + 2$.
- **Using the whole pi array as the list of borders.** Only the chain from $\text{pi}[n - 1]$ lists the borders of the full string.

4.5.10 Where these tools stop

The prefix function answers questions about one pattern against one text, and about the structure of a single string. Searching for many patterns at once, finding the longest common substring of two texts, or sorting all suffixes needs other structures, such as automata or suffix arrays, which are beyond this library. The method compares only equality of symbols, so it works unchanged on arrays of numbers. Questions that only need counts of letters, not their order, are often easier with the frequency tables of Chapter 1.4.

Exercises

Difficulty: ● quick check, ●● needs an extra idea. Try each one before reading the answers.

1. ● What does the word “proper” exclude in “proper prefix”?
2. ● Why is a hash match not a proof that two strings are equal?
3. ● Compute the prefix function of aabaaab. Which index needs a fallback?
4. ●● Run KMP for the pattern aa on the text aaaa. List the 1-based match positions and the value of j right after each match.
5. ●● For $s = \text{abcabcbab}$, $\text{pi} = [0, 0, 0, 1, 2, 3, 4, 5]$. List every border of s with the chain, give the smallest period, and decide whether a block shorter than s repeats to form it.

Answers and explanations

Read these after attempting the exercises.

1. It excludes the **whole string** itself. Every string is a prefix of itself, so without “proper” the longest border would always be the full string.
2. A hash maps many strings onto fewer values, so **different strings can share a value** (for example `ab` and `dj` with base 31 modulo 101). Equal hashes say “possibly equal”; only a direct comparison proves it.
3. `pi = [0, 1, 0, 1, 2, 2, 3]`. Index 5 needs a fallback: it starts from `j = pi[4] = 2`, finds `s[2] = b ≠ a`, falls back to `j = pi[1] = 1`, and then `s[1] = a` matches, giving 2.
4. `pi` for `aa` is `[0, 1]`. Matches at **1, 2 and 3**. After each match `j = 2` falls back to `pi[1] = 1`, so the next a completes another match at once.
5. Chain: `pi[7] = 5`, then `pi[4] = 2`, then `pi[1] = 0`, so the borders are **abcab** and **ab**. The smallest period is $8 - 5 = 3$. 3 does not divide 8, so **no** shorter block repeats exactly; the only block is the whole string.

Practice this topic

Four coding problems for this chapter are in the **Practice** tab. Open any of them in any order; the label shows difficulty within this topic.

Prefix border lengths

Apply

For every prefix `s[1..i]` of a lowercase string, print the length of its longest proper prefix that is also a suffix. A proper prefix is shorter than the whole current prefix.

All pattern starts

Connect

Find all starting positions where pattern `p` occurs in text `t`. Include overlapping matches. Positions are 1-based.

Shortest repeating block

Reason

Find the minimum positive length `k` such that the string is an integer number of copies of its first `k` characters. One copy of the whole string is allowed.

Every full-string border

Stretch

Print all lengths `k` with $1 \leq k < |s|$ such that the first `k` characters equal the last `k` characters. Output lengths increasingly; the empty border and whole string are excluded.