Chapter 9: String Matching and Compression

Algorithms

Brute force | KMP and the LPS table | Boyer-Moore | Rabin-Karp | Comparison | RLE | LZW

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/algorithms/slides/Chapter_09_String_Matching_and_Compression.html

Learning Outcomes

Concepts

Define text, pattern, shift, prefix and suffix; state the string matching problem formally.

Matching

Run brute force, KMP, Boyer-Moore and Rabin-Karp step by step on an example and implement them in Java.

Analysis

Explain what the LPS table, the bad-character table and the rolling hash save; compare best, average and worst cases.

Compression

Compress losslessly with RLE and LZW, decompress the result, and compute the compression ratio.

String Algorithms

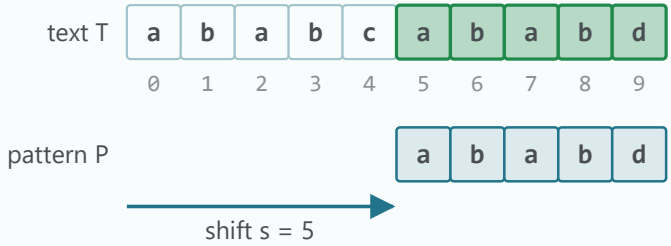
- Text is everywhere: **email, chat messages, social media posts, news articles**...
- Our computers handle enormous amounts of text every day. **How do we organize and analyze it?**
- **String algorithms** perform operations such as **searching**, **replacing** and **comparing** text.
- Two core problems: **string matching** (finding a pattern in a text) and **string compression** (storing text in less space).

Where?

Ctrl+F in editors, `grep`, search engines, virus signature scanning, plagiarism detection, gene search in DNA, file formats such as ZIP/GIF/PNG.

Basic Concepts and Notation

- **Alphabet** Σ: the set of characters in use (e.g. {a, b, …, z}, {A, C, G, T}).
- **Text** T[0..n−1] and **pattern** P[0..m−1], usually m ≤ n.
- P occurs in T **with shift s** ⇔ T[s..s+m−1] = P. Valid shifts: 0 ≤ s ≤ n − m.
- **Prefix**: P[0..k−1]; **suffix**: P[m−k..m−1]. A **proper** prefix/suffix is one not equal to the whole string.
- Example: for "ababd" the prefixes are a, ab, aba, abab; the suffixes are d, bd, abd, babd.



The string matching problem: find all (or the first) shifts s at which P occurs.

Chapter Map

Brute Force
Checks every position of the text for a match with the pattern. Makes the most comparisons.

Knuth-Morris-Pratt (KMP)
Builds a table (LPS) up front, then scans the text without ever moving backwards, saving comparisons.

Boyer-Moore
Compares right to left; the bad-character and good-suffix rules allow long jumps. Very effective for long patterns.

Rabin-Karp
Compares hash values; as the window slides, the hash is updated in O(1). Fast with high probability.

RLE (Run-Length Encoding)
Each run of repeated values is stored as a single value plus a repeat count.

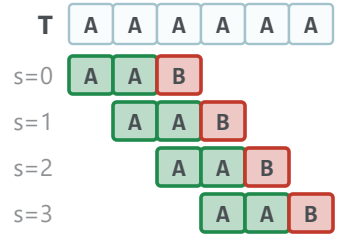
LZW (Lempel-Ziv-Welch)
Dictionary-based compression: repeated patterns become short codes in a dictionary that grows on the fly (GIF).

The Brute-Force (Naive) Algorithm

- The **simplest** way to search for a pattern in a text, hence the name "naive".
- **How it works:**
  1. For each shift s = 0, 1, …, n − m, align the pattern under the text.
  2. Compare characters **left to right**; stop at the first mismatch.
  3. If all m characters match, **report** s; otherwise slide the pattern **one position**.
- **No** preprocessing and no extra memory.

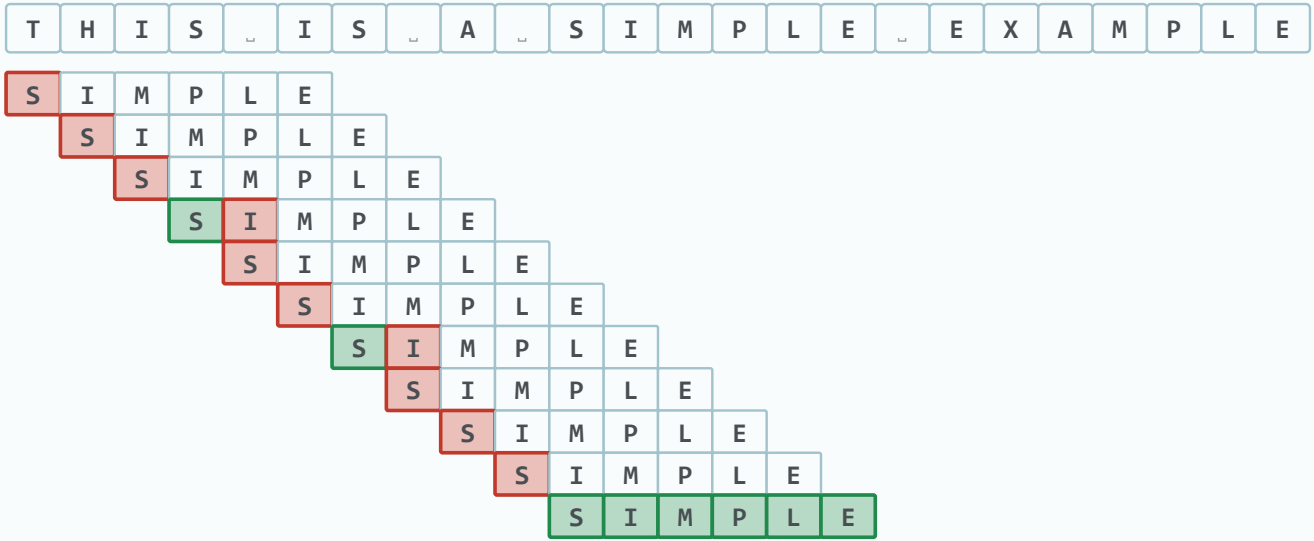
Complexity

Worst case: (n − m + 1) · m comparisons → **O(n · m)**.
Best case: the first character mismatches at every shift → O(n).
n: text length, m: pattern length.



4 shifts × 3 = 12 comparisons

Example: "THIS IS A SIMPLE EXAMPLE"



Each row is one shift: green = match, red = mismatch. SIMPLE is found at the 11th shift (s = 10) after **18** comparisons in total (n = 24, m = 6).

"LO" in "HELLO WORLD": Code Trace

```
1  int bruteForce(String t, String p) {
▶    int n = t.length(), m = p.length();
3    for (int i = 0; i <= n - m; i++) {
4      int j = 0;
5      while (j < m &&
6        t.charAt(i + j) == p.charAt(j))
7        j++;
8      if (j == m) return i;
9    }
10   return -1;
11 }
```



n = 11, m = 2. The pattern will be tried at shifts i = 0, 1, …, 9.

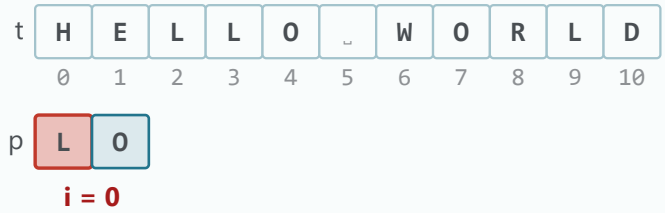
Step 1 / 8

VARIABLES

n = 11 | m = 2 | i = – | j = – | cmp = 0

"LO" in "HELLO WORLD": Code Trace

```
1  int bruteForce(String t, String p) {
2    int n = t.length(), m = p.length();
3    for (int i = 0; i <= n - m; i++) {
4      int j = 0;
▶      while (j < m &&
▶        t.charAt(i + j) == p.charAt(j))
7        j++;
8      if (j == m) return i;
9    }
10   return -1;
11 }
```



t[0] = 'H' ≠ p[0] = 'L' → mismatch; slide the pattern one to the right.

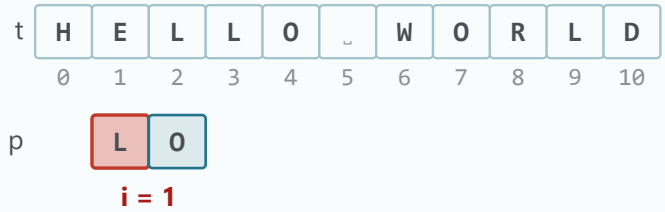
Step 2 / 8

VARIABLES

`n = 11` `m = 2` `i = 0` `j = 0` `cmp = 1`

"LO" in "HELLO WORLD": Code Trace

```
1  int bruteForce(String t, String p) {
2    int n = t.length(), m = p.length();
3    for (int i = 0; i <= n - m; i++) {
4      int j = 0;
▶      while (j < m &&
▶        t.charAt(i + j) == p.charAt(j))
7        j++;
8      if (j == m) return i;
9    }
10   return -1;
11 }
```



t[1] = 'E' ≠ p[0] = 'L' → mismatch; slide the pattern one to the right.

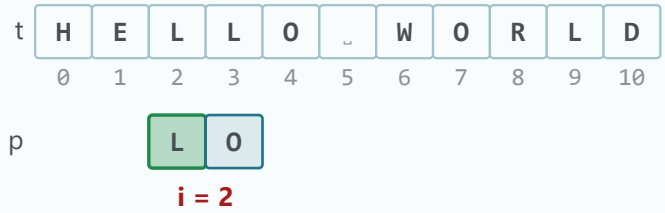
Step 3 / 8

VARIABLES

`n = 11` `m = 2` `i = 1` `j = 0` `cmp = 2`

"LO" in "HELLO WORLD": Code Trace

```
1  int bruteForce(String t, String p) {
2    int n = t.length(), m = p.length();
3    for (int i = 0; i <= n - m; i++) {
4      int j = 0;
▶      while (j < m &&
▶        t.charAt(i + j) == p.charAt(j))
▶        j++;
8      if (j == m) return i;
9    }
10   return -1;
11 }
```



t[2] = 'L' matches p[0] = 'L' → j++.

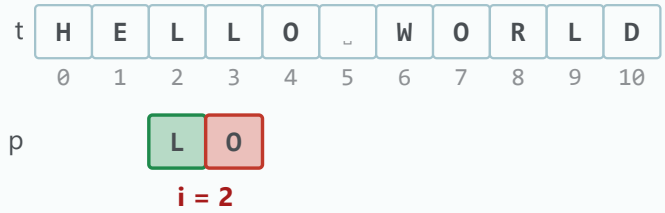
Step 4 / 8

VARIABLES

`n = 11` `m = 2` `i = 2` `j = 1` `cmp = 3`

"LO" in "HELLO WORLD": Code Trace

```
1  int bruteForce(String t, String p) {
2    int n = t.length(), m = p.length();
3    for (int i = 0; i <= n - m; i++) {
4      int j = 0;
▶      while (j < m &&
▶        t.charAt(i + j) == p.charAt(j))
7        j++;
8      if (j == m) return i;
9    }
10   return -1;
11 }
```



t[3] = 'L' ≠ p[1] = 'O' → mismatch; slide the pattern one to the right.

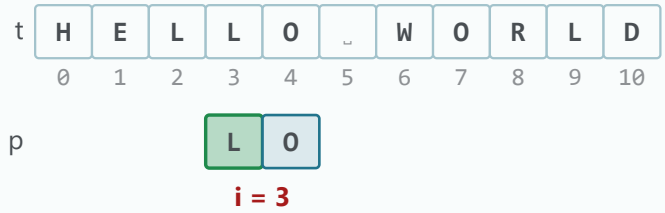
Step 5 / 8

VARIABLES

`n = 11` `m = 2` `i = 2` `j = 1` `cmp = 4`

"LO" in "HELLO WORLD": Code Trace

```
1  int bruteForce(String t, String p) {
2    int n = t.length(), m = p.length();
3    for (int i = 0; i <= n - m; i++) {
4      int j = 0;
▶      while (j < m &&
▶        t.charAt(i + j) == p.charAt(j))
▶        j++;
8      if (j == m) return i;
9    }
10   return -1;
11 }
```



t[3] = 'L' matches p[0] = 'L' → j++.

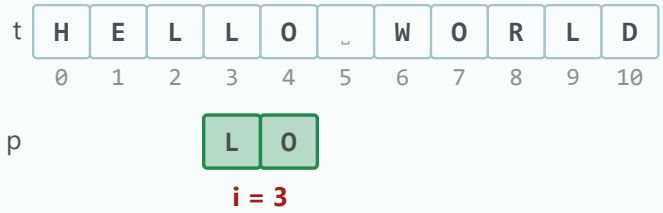
Step 6 / 8

VARIABLES

`n = 11` `m = 2` `i = 3` `j = 1` `cmp = 5`

"LO" in "HELLO WORLD": Code Trace

```
1  int bruteForce(String t, String p) {
2    int n = t.length(), m = p.length();
3    for (int i = 0; i <= n - m; i++) {
4      int j = 0;
▶      while (j < m &&
▶        t.charAt(i + j) == p.charAt(j))
▶        j++;
8      if (j == m) return i;
9    }
10   return -1;
11 }
```



t[4] = 'O' matches p[1] = 'O' → j++.

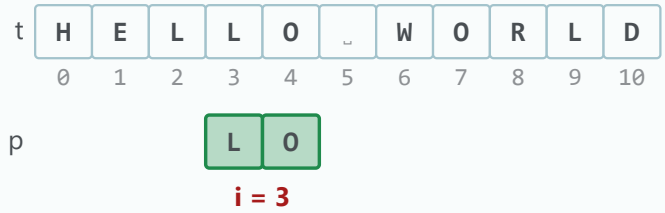
Step 7 / 8

VARIABLES

`n = 11` `m = 2` `i = 3` `j = 2` `cmp = 6`

"LO" in "HELLO WORLD": Code Trace

```
1  int bruteForce(String t, String p) {
2    int n = t.length(), m = p.length();
3    for (int i = 0; i <= n - m; i++) {
4      int j = 0;
5      while (j < m &&
6        t.charAt(i + j) == p.charAt(j))
7        j++;
▶      if (j == m) return i;
9    }
10   return -1;
11 }
```



j == m: pattern found at **i = 3**. 6 comparisons in total.

Step 8 / 8

VARIABLES

`n = 11` `m = 2` `i = 3` `j = 2` `cmp = 6`

Brute Force: Best, Average, Worst

Case	When?	Comparisons	Time
Best	P[0] matches no character of the text (e.g. P = "xyz", T = "aaaa…")	$n - m + 1$	O(n)
Average	Random text, large alphabet: a mismatch usually comes within the first 1–2 characters	$\approx n \cdot \sigma/(\sigma-1)$	O(n)
Worst	T = "aaaaaaaaaa", P = "aaab": m comparisons at every shift	$(n - m + 1) \cdot m$	O(n · m)

Numeric example: n = 10, m = 4, T = a^{10}, P = aaab → 7 shifts × 4 = **28** comparisons. The mismatch always comes at the last character.

The real flaw: what a mismatch teaches us (the characters that matched) is **thrown away**; the algorithm backs up in the text and rereads the same characters. KMP puts this information to use.

The Knuth-Morris-Pratt (KMP) Algorithm

- Finds occurrences of a pattern in a text.
- Developed by **Donald Knuth** and **Vaughan Pratt** and, independently, by **James H. Morris**; published jointly in 1977.
- Key idea: on a mismatch, use what we already know about the **part of the pattern that has matched**; the text pointer **never moves back**.
- That knowledge is precomputed from the pattern and stored in the **LPS table** (the prefix function π).

Complexity

Preprocessing: O(m)

Search: O(n)

Total: **O(n + m)**, even in the worst case

Extra memory: O(m)

KMP: How It Works

1. Preprocessing

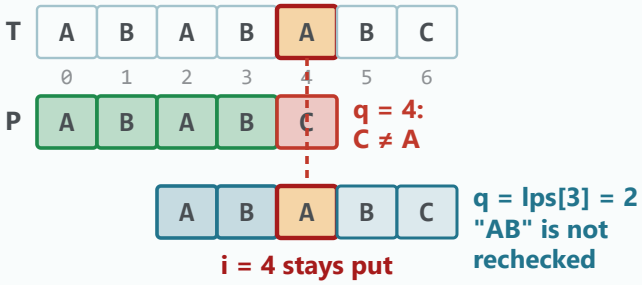
For each position of the pattern, record the length of the **longest proper prefix that is also a suffix** (LPS). On a mismatch, this table tells us where to resume in the pattern.

2. Pattern search

Scan the text once, left to right. On a mismatch, the pattern pointer q falls back to the value read from the table (q = lps[q − 1]); the text pointer i stays put.

3. Match check

When q reaches m, every character of the pattern has matched: report position i − m + 1. To find all occurrences, continue with q = lps[m − 1].



Why is this correct? If P[0..q−1] has matched the text and P[q] mismatches, the next possible alignment is the one that overlaps the longest prefix of P[0..q−1] that is also its suffix. No shift in between can **possibly** match.

The LPS Table (Prefix Function)

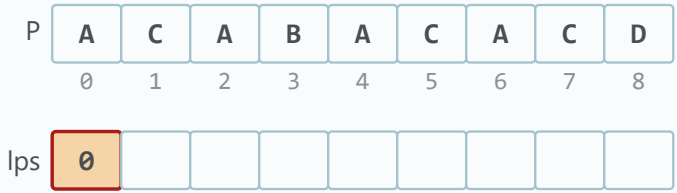
lps[q] = length of the **longest proper prefix** of P[0..q] that is also a suffix of P[0..q].

- lps[0] = 0 (a single character has no proper prefix).
- Example: text "ababcababcababababd", pattern **"ababd"**.
- Pattern table: a: 0, b: 0, a: 1, b: 2, d: 0.

q	P[q]	lps[q]	Reason (P[0..q])
0	`a`	**0**	"a" — no proper prefix
1	`b`	**0**	"ab" — a ≠ b
2	`a`	**1**	"aba" — "a" is both prefix and suffix
3	`b`	**2**	"abab" — "ab" is both prefix and suffix
4	`d`	**0**	"ababd" — no prefix ends in d

Building the LPS Table: "ACABACACD"

```
1  int[] computeLPS(String p) {
▶    int m = p.length(), k = 0;
▶    int[] lps = new int[m];
4    for (int q = 1; q < m; q++) {
5      while (k > 0 &&
6        p.charAt(k) != p.charAt(q))
7        k = lps[k - 1];
8      if (p.charAt(k) == p.charAt(q))
9        k++;
10     lps[q] = k;
11   }
12   return lps;
13 }
```



lps[0] = 0 always. k = 0: length of the prefix matched so far.

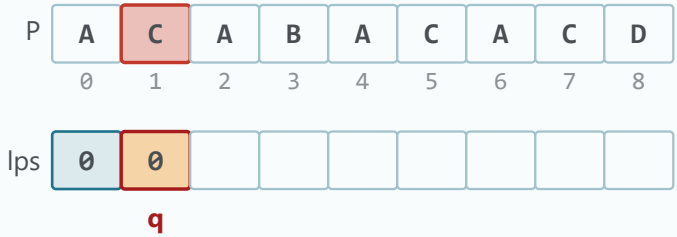
Step 1 / 13

VARIABLES

m = 9 | q = – | k = 0

Building the LPS Table: "ACABACACD"

```
1  int[] computeLPS(String p) {
2    int m = p.length(), k = 0;
3    int[] lps = new int[m];
4    for (int q = 1; q < m; q++) {
5      while (k > 0 &&
6        p.charAt(k) != p.charAt(q))
7        k = lps[k - 1];
▶      if (p.charAt(k) == p.charAt(q))
9        k++;
▶      lps[q] = k;
11   }
12   return lps;
13 }
```



p[0] = 'A' ≠ p[1] = 'C' → k = 0, lps[1] = 0.

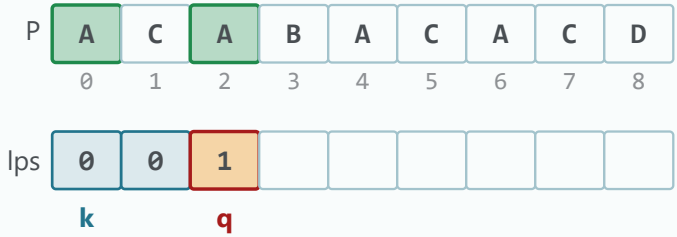
Step 2 / 13

VARIABLES

m = 9 q = 1 k = 0

Building the LPS Table: "ACABACACD"

```
1  int[] computeLPS(String p) {
2    int m = p.length(), k = 0;
3    int[] lps = new int[m];
4    for (int q = 1; q < m; q++) {
5      while (k > 0 &&
6        p.charAt(k) != p.charAt(q))
7        k = lps[k - 1];
▶      if (p.charAt(k) == p.charAt(q))
▶        k++;
▶      lps[q] = k;
11   }
12   return lps;
13 }
```



p[0] = 'A' = p[2] = 'A' → k = 1, lps[2] = 1.

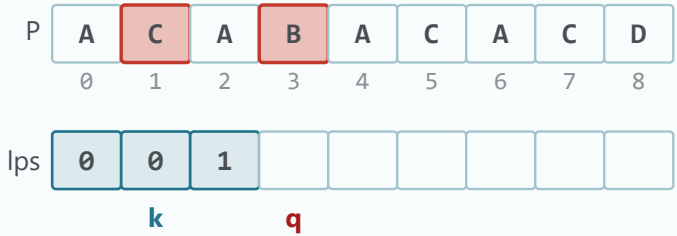
Step 3 / 13

VARIABLES

m = 9 q = 2 k = 1

Building the LPS Table: "ACABACACD"

```
1  int[] computeLPS(String p) {
2    int m = p.length(), k = 0;
3    int[] lps = new int[m];
4    for (int q = 1; q < m; q++) {
▶      while (k > 0 &&
▶        p.charAt(k) != p.charAt(q))
▶        k = lps[k - 1];
8      if (p.charAt(k) == p.charAt(q))
9        k++;
10     lps[q] = k;
11   }
12   return lps;
13 }
```



p[1] = 'C' ≠ p[3] = 'B' → fall back: k = lps[0] = 0.

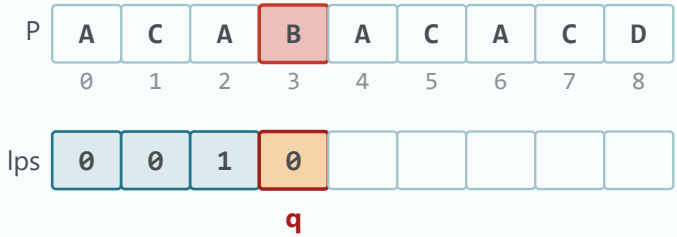
Step 4 / 13

VARIABLES

m = 9 q = 3 k = 0

Building the LPS Table: "ACABACACD"

```
1  int[] computeLPS(String p) {
2    int m = p.length(), k = 0;
3    int[] lps = new int[m];
4    for (int q = 1; q < m; q++) {
5      while (k > 0 &&
6        p.charAt(k) != p.charAt(q))
7        k = lps[k - 1];
▶      if (p.charAt(k) == p.charAt(q))
9        k++;
▶      lps[q] = k;
11   }
12   return lps;
13 }
```



p[0] = 'A' ≠ p[3] = 'B' → k = 0, lps[3] = 0.

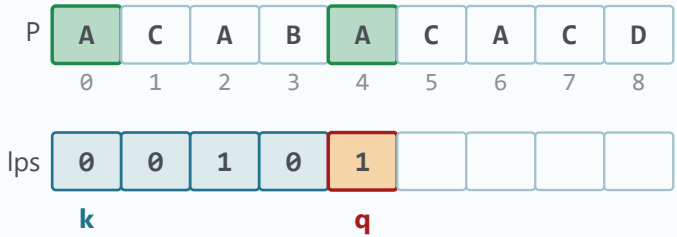
Step 5 / 13

VARIABLES

m = 9 q = 3 k = 0

Building the LPS Table: "ACABACACD"

```
1  int[] computeLPS(String p) {
2    int m = p.length(), k = 0;
3    int[] lps = new int[m];
4    for (int q = 1; q < m; q++) {
5      while (k > 0 &&
6        p.charAt(k) != p.charAt(q))
7        k = lps[k - 1];
▶      if (p.charAt(k) == p.charAt(q))
▶        k++;
▶      lps[q] = k;
11   }
12   return lps;
13 }
```



p[0] = 'A' = p[4] = 'A' → k = 1, lps[4] = 1.

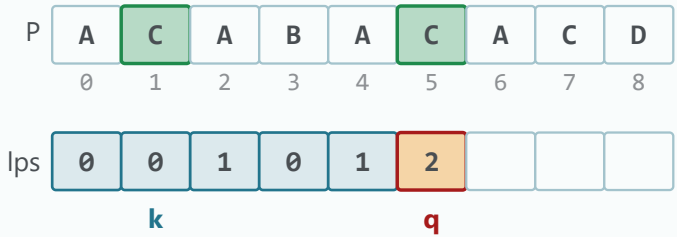
Step 6 / 13

VARIABLES

m = 9 q = 4 k = 1

Building the LPS Table: "ACABACACD"

```
1  int[] computeLPS(String p) {
2    int m = p.length(), k = 0;
3    int[] lps = new int[m];
4    for (int q = 1; q < m; q++) {
5      while (k > 0 &&
6        p.charAt(k) != p.charAt(q))
7        k = lps[k - 1];
▶      if (p.charAt(k) == p.charAt(q))
▶        k++;
▶      lps[q] = k;
11   }
12   return lps;
13 }
```



p[1] = 'C' = p[5] = 'C' → k = 2, lps[5] = 2. "AC" is both prefix and suffix.

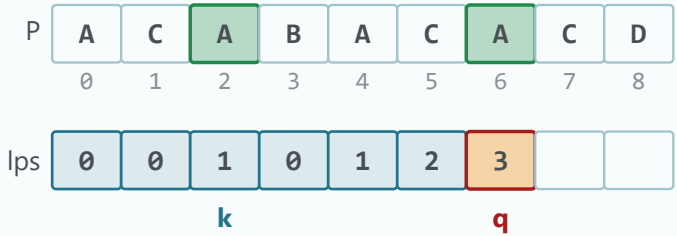
Step 7 / 13

VARIABLES

m = 9 q = 5 k = 2

Building the LPS Table: "ACABACACD"

```
1  int[] computeLPS(String p) {
2    int m = p.length(), k = 0;
3    int[] lps = new int[m];
4    for (int q = 1; q < m; q++) {
5      while (k > 0 &&
6        p.charAt(k) != p.charAt(q))
7        k = lps[k - 1];
▶      if (p.charAt(k) == p.charAt(q))
▶        k++;
▶      lps[q] = k;
11   }
12   return lps;
13 }
```



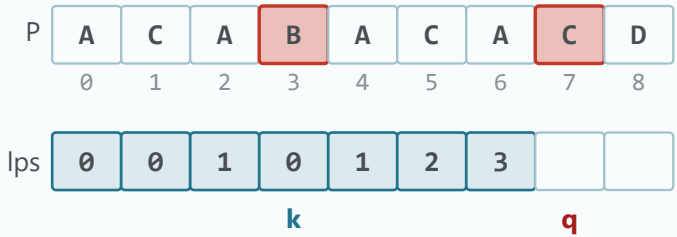
p[2] = 'A' = p[6] = 'A' → k = 3, lps[6] = 3. "ACA" is both prefix and suffix.

VARIABLES

m = 9 q = 6 k = 3

Building the LPS Table: "ACABACACD"

```
1  int[] computeLPS(String p) {
2    int m = p.length(), k = 0;
3    int[] lps = new int[m];
4    for (int q = 1; q < m; q++) {
▶      while (k > 0 &&
▶        p.charAt(k) != p.charAt(q))
▶        k = lps[k - 1];
8      if (p.charAt(k) == p.charAt(q))
9        k++;
10     lps[q] = k;
11   }
12   return lps;
13 }
```



p[3] = 'B' ≠ p[7] = 'C' → fall back: k = lps[2] = 1.

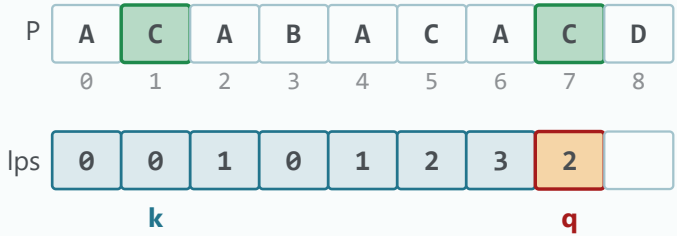
Step 9 / 13

VARIABLES

m = 9 q = 7 k = 1

Building the LPS Table: "ACABACACD"

```
1  int[] computeLPS(String p) {
2    int m = p.length(), k = 0;
3    int[] lps = new int[m];
4    for (int q = 1; q < m; q++) {
5      while (k > 0 &&
6        p.charAt(k) != p.charAt(q))
7        k = lps[k - 1];
▶      if (p.charAt(k) == p.charAt(q))
▶        k++;
▶      lps[q] = k;
11   }
12   return lps;
13 }
```



p[1] = 'C' = p[7] = 'C' → k = 2, lps[7] = 2. "AC" is both prefix and suffix.

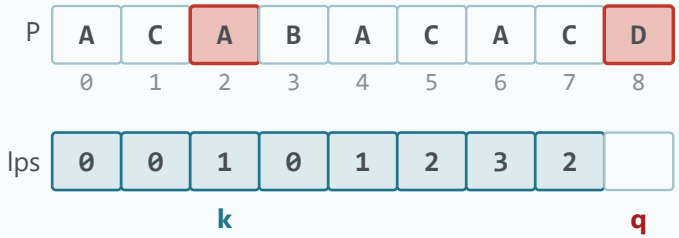
Step 10 / 13

VARIABLES

m = 9 q = 7 k = 2

Building the LPS Table: "ACABACACD"

```
1  int[] computeLPS(String p) {
2    int m = p.length(), k = 0;
3    int[] lps = new int[m];
4    for (int q = 1; q < m; q++) {
▶      while (k > 0 &&
▶        p.charAt(k) != p.charAt(q))
▶        k = lps[k - 1];
8      if (p.charAt(k) == p.charAt(q))
9        k++;
10     lps[q] = k;
11   }
12   return lps;
13 }
```



p[2] = 'A' ≠ p[8] = 'D' → fall back: k = lps[1] = 0.

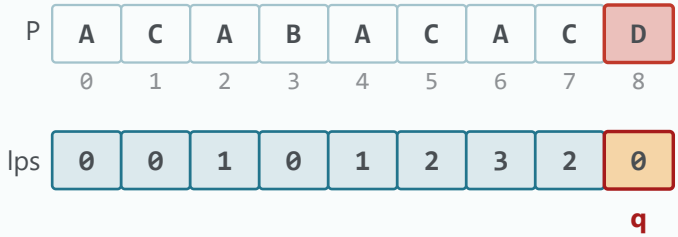
Step 11 / 13

VARIABLES

m = 9 q = 8 k = 0

Building the LPS Table: "ACABACACD"

```
1  int[] computeLPS(String p) {
2    int m = p.length(), k = 0;
3    int[] lps = new int[m];
4    for (int q = 1; q < m; q++) {
5      while (k > 0 &&
6        p.charAt(k) != p.charAt(q))
7        k = lps[k - 1];
▶      if (p.charAt(k) == p.charAt(q))
9        k++;
▶      lps[q] = k;
11   }
12   return lps;
13 }
```



p[0] = 'A' ≠ p[8] = 'D' → k = 0, lps[8] = 0.

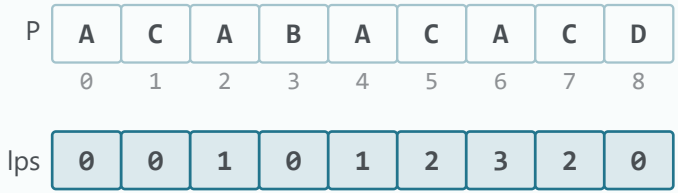
Step 12 / 13

VARIABLES

m = 9 q = 8 k = 0

Building the LPS Table: "ACABACACD"

```
1  int[] computeLPS(String p) {
2    int m = p.length(), k = 0;
3    int[] lps = new int[m];
4    for (int q = 1; q < m; q++) {
5      while (k > 0 &&
6        p.charAt(k) != p.charAt(q))
7        k = lps[k - 1];
8      if (p.charAt(k) == p.charAt(q))
9        k++;
10     lps[q] = k;
11   }
▶    return lps;
13 }
```



Result: lps = [0, 0, 1, 0, 1, 2, 3, 2, 0]. Constant amortized work per q → O(m).

Step 13 / 13

VARIABLES

`m = 9` `q = 9` `k = 0`

The Three Phases of a KMP Search

Phase 1 · Find the first character

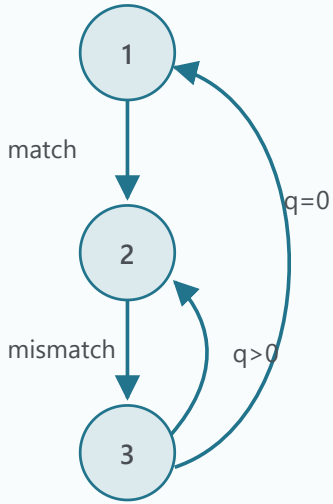
Advance the text pointer (i) until it matches the first symbol of the pattern.

Phase 2 · Advance together

While symbols match, both pointers advance. On reaching the end of the pattern, return (text position − pattern position).

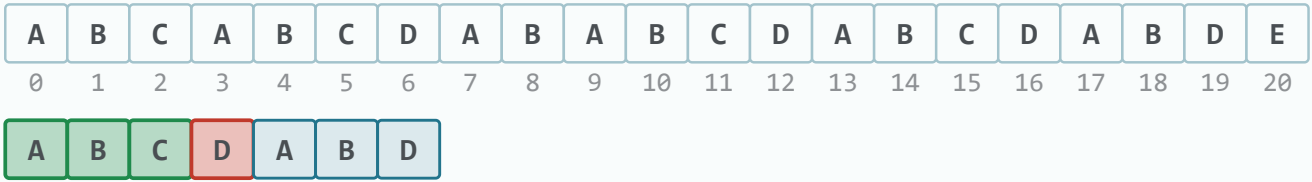
Phase 3 · Fall back via LPS

On a mismatch only the pattern pointer (q) falls back via LPS; after a match or on reaching 0, return to Phase 2 (Phase 1 if q = 0).



KMP Example: Searching for "ABCDABD"

lps("ABCDABD") = [0, 0, 0, 0, 1, 2, 0]. green = match, red = mismatch, blue = known from LPS (not compared again).



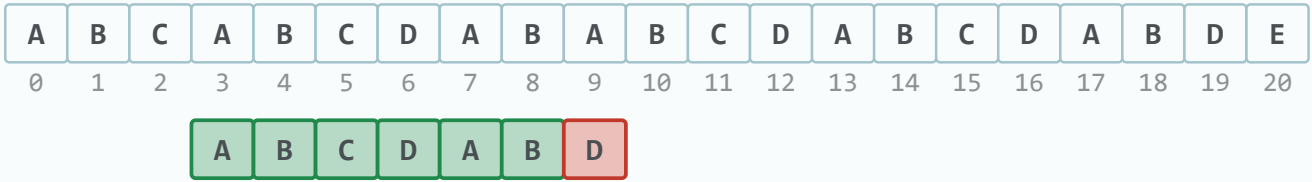
s = 0: ABC matched, t[3] = 'A' ≠ p[3] = 'D'. q = 3 → q = lps[2] = **0**; the pattern slides 3 and t[3] is compared with p[0].

Step 1 / 5

24 character comparisons in total (n = 21); brute force makes 37 on the same text.

KMP Example: Searching for "ABCDABD"

lps("ABCDABD") = [0, 0, 0, 0, 1, 2, 0]. green = match, red = mismatch, blue = known from LPS (not compared again).



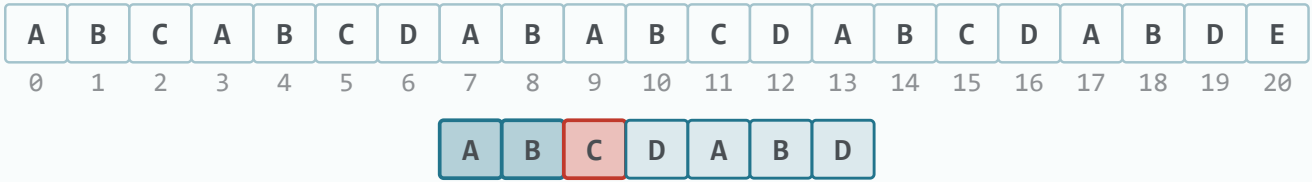
s = 3: ABCDAB matched, t[9] = 'A' ≠ 'D'. q = 6 → q = lps[5] = **2**: "AB" is known to match; the pattern slides 4.

Step 2 / 5

24 character comparisons in total (n = 21); brute force makes 37 on the same text.

KMP Example: Searching for "ABCDABD"

lps("ABCDABD") = [0, 0, 0, 0, 1, 2, 0]. green = match, red = mismatch, blue = known from LPS (not compared again).



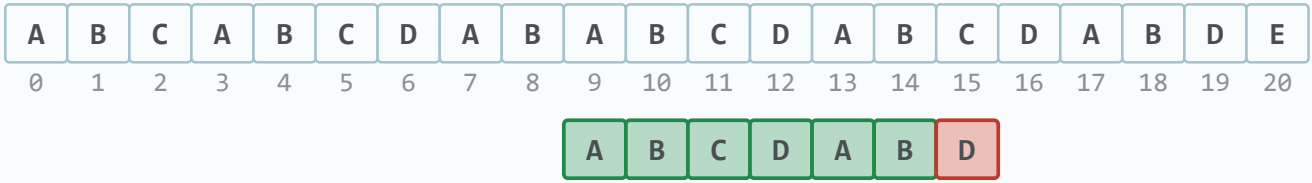
s = 7: AB is not compared again. t[9] = 'A' ≠ p[2] = 'C' → q = lps[1] = **0**.

Step 3 / 5

24 character comparisons in total (n = 21); brute force makes 37 on the same text.

KMP Example: Searching for "ABCDABD"

lps("ABCDABD") = [0, 0, 0, 0, 1, 2, 0]. green = match, red = mismatch, blue = known from LPS (not compared again).



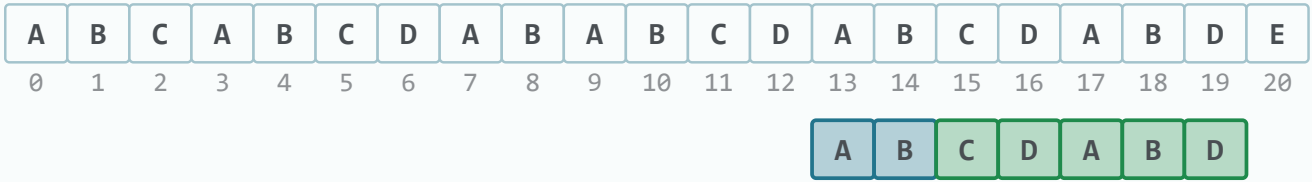
s = 9: ABCDAB matched, t[15] = 'C' ≠ 'D'. q = 6 → q = lps[5] = **2**; the pattern slides 4.

Step 4 / 5

24 character comparisons in total (n = 21); brute force makes 37 on the same text.

KMP Example: Searching for "ABCDABD"

lps("ABCDABD") = [0, 0, 0, 0, 1, 2, 0]. green = match, red = mismatch, blue = known from LPS (not compared again).



s = 13: AB is known; CDABD matches too → pattern found **at position 13**. The text pointer never moved back.

Step 5 / 5

24 character comparisons in total (n = 21); brute force makes 37 on the same text.

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
▶    int n = t.length(), m = p.length();
▶    int[] lps = computeLPS(p);
▶    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
9      if (p.charAt(q) == t.charAt(i))
10       q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```

t	a	b	a	b	c	a	b	a	b	c	a	b	a	b	a	b	d
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

lps = [0, 0, 1, 2, 0]

n = 17, m = 5, lps = [0, 0, 1, 2, 0], q = 0.

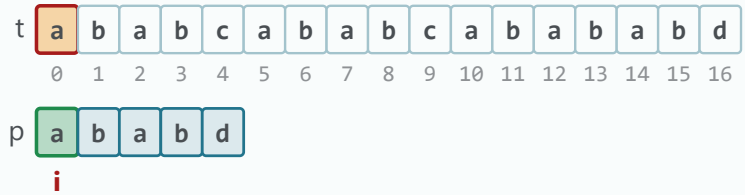
Step 1 / 24

VARIABLES

n = 17 | m = 5 | i = – | q = 0

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[0] = t[0] = 'a' → q = 1.

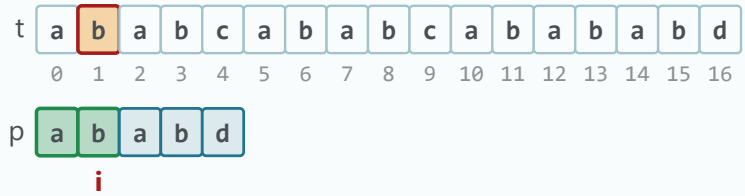
Step 2 / 24

VARIABLES

n = 17 | m = 5 | i = 0 | q = 1

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[1] = t[1] = 'b' → q = 2.

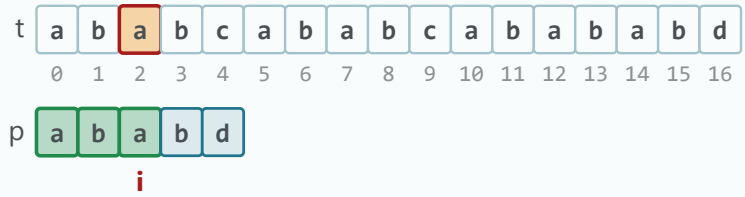
Step 3 / 24

VARIABLES

n = 17 | m = 5 | i = 1 | q = 2

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[2] = t[2] = 'a' → q = 3.

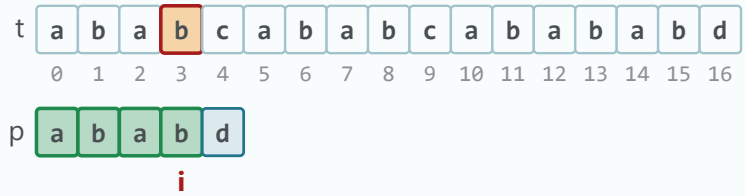
Step 4 / 24

VARIABLES

n = 17 | m = 5 | i = 2 | q = 3

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[3] = t[3] = 'b' → q = 4.

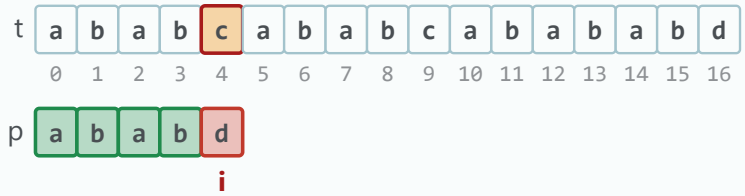
Step 5 / 24

VARIABLES

n = 17 | m = 5 | i = 3 | q = 4

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
▶      while (q > 0 &&
▶        p.charAt(q) != t.charAt(i))
▶        q = lps[q - 1];
9      if (p.charAt(q) == t.charAt(i))
10       q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[4] = 'd' ≠ t[4] = 'c' → q = lps[3] = 2 (i stays put).

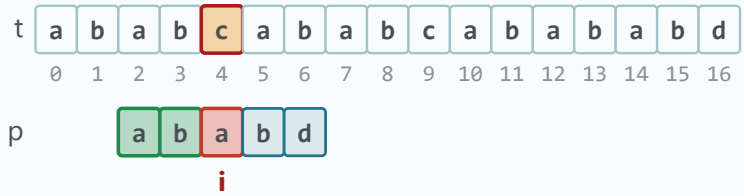
Step 6 / 24

VARIABLES

n = 17 | m = 5 | i = 4 | q = 2

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
▶      while (q > 0 &&
▶        p.charAt(q) != t.charAt(i))
▶        q = lps[q - 1];
9      if (p.charAt(q) == t.charAt(i))
10       q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[2] = 'a' ≠ t[4] = 'c' → q = lps[1] = 0 (i stays put).

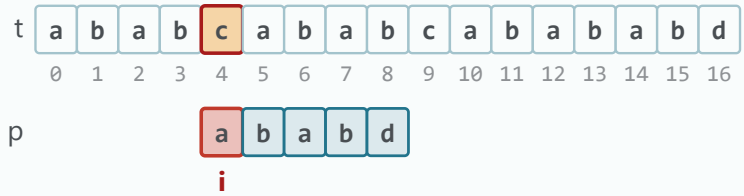
Step 7 / 24

VARIABLES

n = 17 | m = 5 | i = 4 | q = 0

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
10       q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[0] = 'a' ≠ t[4] = 'c' → q stays 0, i advances.

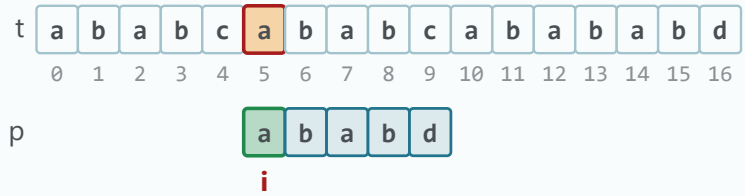
Step 8 / 24

VARIABLES

n = 17 | m = 5 | i = 4 | q = 0

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[0] = t[5] = 'a' → q = 1.

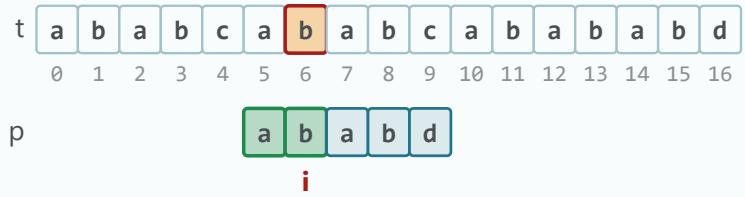
Step 9 / 24

VARIABLES

n = 17 | m = 5 | i = 5 | q = 1

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[1] = t[6] = 'b' → q = 2.

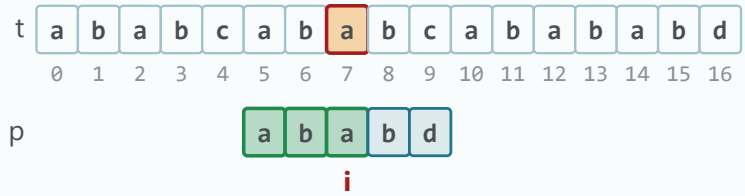
Step 10 / 24

VARIABLES

n = 17 | m = 5 | i = 6 | q = 2

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[2] = t[7] = 'a' → q = 3.

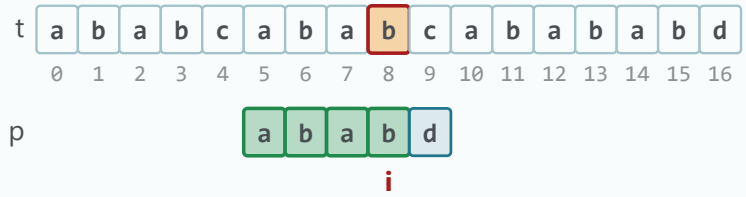
Step 11 / 24

VARIABLES

n = 17 | m = 5 | i = 7 | q = 3

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[3] = t[8] = 'b' → q = 4.

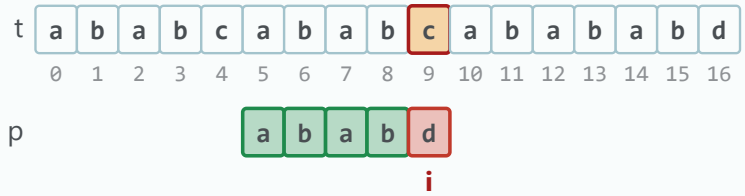
Step 12 / 24

VARIABLES

n = 17 | m = 5 | i = 8 | q = 4

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
▶      while (q > 0 &&
▶        p.charAt(q) != t.charAt(i))
▶        q = lps[q - 1];
9      if (p.charAt(q) == t.charAt(i))
10       q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[4] = 'd' ≠ t[9] = 'c' → q = lps[3] = 2 (i stays put).

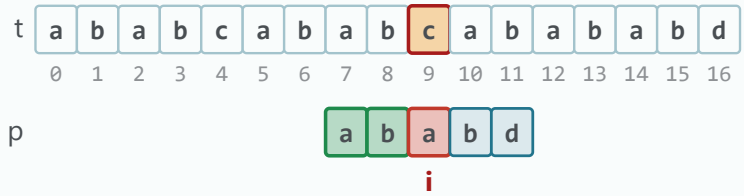
Step 13 / 24

VARIABLES

n = 17 | m = 5 | i = 9 | q = 2

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
▶      while (q > 0 &&
▶        p.charAt(q) != t.charAt(i))
▶        q = lps[q - 1];
9      if (p.charAt(q) == t.charAt(i))
10       q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[2] = 'a' ≠ t[9] = 'c' → q = lps[1] = 0 (i stays put).

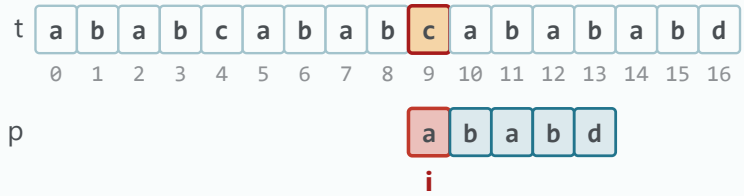
Step 14 / 24

VARIABLES

n = 17 | m = 5 | i = 9 | q = 0

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
10       q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[0] = 'a' ≠ t[9] = 'c' → q stays 0, i advances.

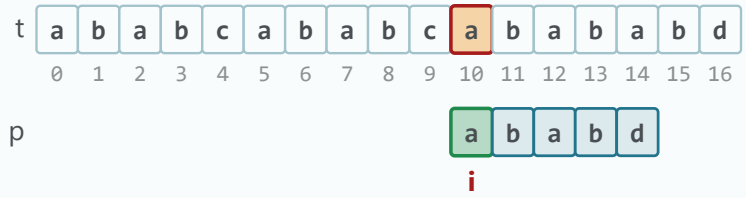
Step 15 / 24

VARIABLES

n = 17 | m = 5 | i = 9 | q = 0

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[0] = t[10] = 'a' → q = 1.

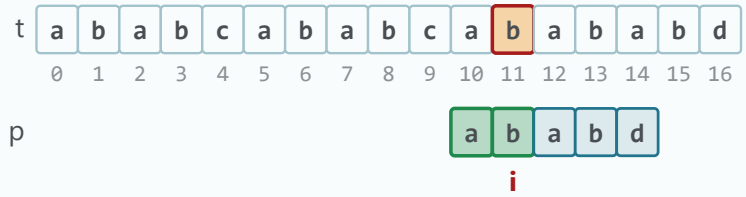
Step 16 / 24

VARIABLES

n = 17 | m = 5 | i = 10 | q = 1

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[1] = t[11] = 'b' → q = 2.

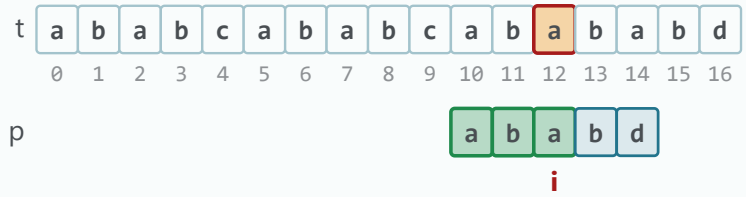
Step 17 / 24

VARIABLES

n = 17 | m = 5 | i = 11 | q = 2

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[2] = t[12] = 'a' → q = 3.

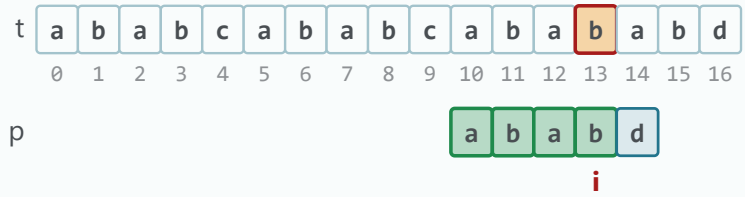
Step 18 / 24

VARIABLES

n = 17 | m = 5 | i = 12 | q = 3

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[3] = t[13] = 'b' → q = 4.

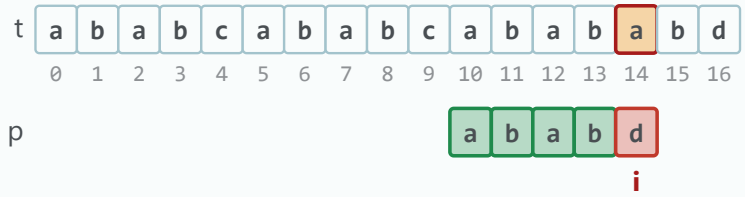
Step 19 / 24

VARIABLES

n = 17 | m = 5 | i = 13 | q = 4

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
▶      while (q > 0 &&
▶        p.charAt(q) != t.charAt(i))
▶        q = lps[q - 1];
9      if (p.charAt(q) == t.charAt(i))
10       q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[4] = 'd' ≠ t[14] = 'a' → q = lps[3] = 2 (i stays put).

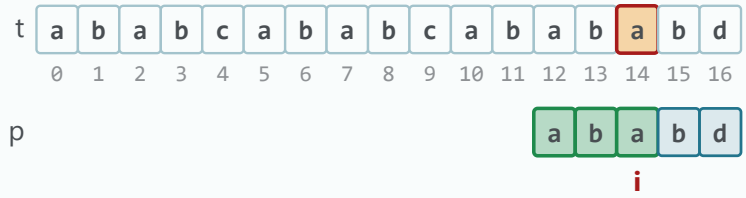
Step 20 / 24

VARIABLES

n = 17 | m = 5 | i = 14 | q = 2

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[2] = t[14] = 'a' → q = 3.

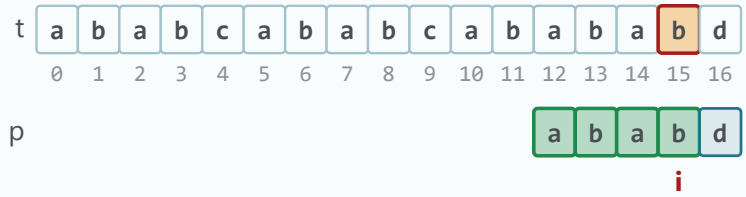
Step 21 / 24

VARIABLES

n = 17 | m = 5 | i = 14 | q = 3

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[3] = t[15] = 'b' → q = 4.

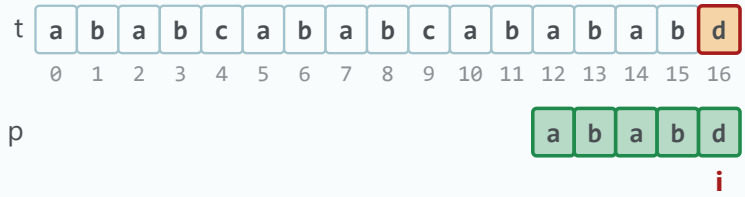
Step 22 / 24

VARIABLES

n = 17 | m = 5 | i = 15 | q = 4

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
▶      if (p.charAt(q) == t.charAt(i))
▶        q++;
11     if (q == m)
12       return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

p[4] = t[16] = 'd' → q = 5.

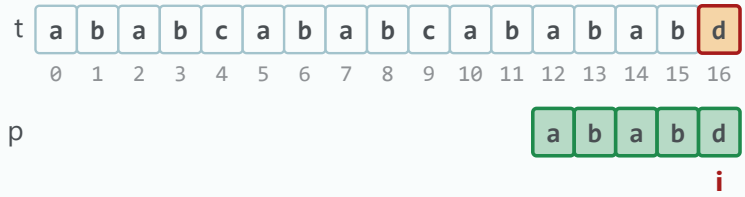
Step 23 / 24

VARIABLES

n = 17 | m = 5 | i = 16 | q = 5

kmpSearch("ababcababcababababd", "ababd")

```
1  int kmpSearch(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] lps = computeLPS(p);
4    int q = 0;  // matched length
5    for (int i = 0; i < n; i++) {
6      while (q > 0 &&
7        p.charAt(q) != t.charAt(i))
8        q = lps[q - 1];
9      if (p.charAt(q) == t.charAt(i))
10       q++;
▶      if (q == m)
▶        return i - m + 1;
13   }
14   return -1;
15 }
```



lps = [0, 0, 1, 2, 0]

q == m → **i − m + 1 = 12** is returned. The text was scanned only once.

Step 24 / 24

VARIABLES

`n = 17` `m = 5` `i = 16` `q = 5`

KMP: Why O(n + m)?

- The outer loop increments i **n times**; i never decreases.
- q grows by **at most 1** per step → total increase ≤ n.
- Each fallback in the `while` loop **decreases q by at least 1**; q cannot go negative → total fallbacks ≤ total increase ≤ n.
- Search: at most **2n** comparisons → O(n). The same amortized argument gives O(m) for LPS.

Pros

+ Linear even in the worst case
+ Never backs up in the text: suits streaming data

Cons

– Usually slower than Boyer-Moore in practice
– Little to skip over with a small alphabet (DNA)

The Boyer-Moore Algorithm

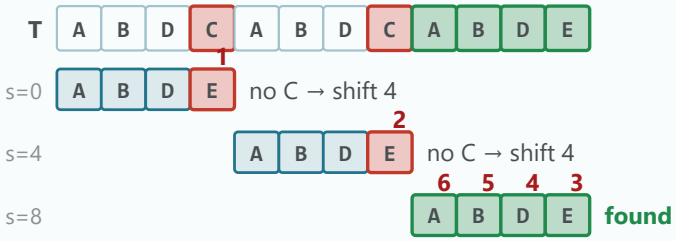
- Searches for a pattern in a text; developed by **Robert S. Boyer** and **J Strother Moore** in 1977.
- The pattern slides left to right, but characters are compared **right to left**.
- On a mismatch, shift by the larger of two rules: **bad character** and **good suffix**.
- Most text characters can be skipped **without ever being read**: very fast for long patterns over large alphabets (natural language); `grep` and editor search belong to this family.

Complexity

Best: O(n/m) — the first comparison fails at every attempt and we skip m.
Average: sublinear, faster than O(n) in practice.
Worst: O(n · m) with bad character alone; O(n + m) with good suffix + Galil's rule.

Boyer-Moore: How It Works

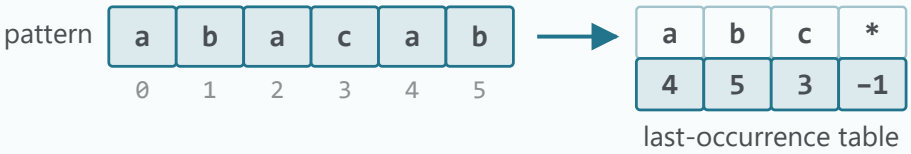
1. **Preprocessing:** build shift tables from the pattern: the **last occurrence** of each character in the pattern (bad character) and a shift amount for each suffix (good suffix).
2. **Search:** align the pattern under the text; compare starting **from the last character and moving left**.
3. **Match check:** if all characters match, report the position.
4. **Bad-character rule:** shift according to the **rightmost** position in the pattern of the mismatched text character.
5. **Good-suffix rule:** shift according to whether the matched suffix occurs elsewhere in the pattern.



Red numbers give the comparison order: only 6 comparisons on a 12-character text.

Shift = max(bad-character shift, good-suffix shift). Both are "safe": no skipped alignment can contain a match.

The Bad-Character Rule

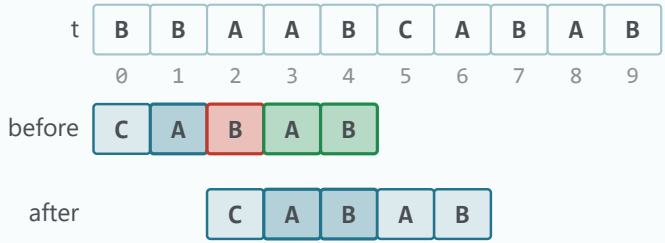


- Let p[j] ≠ t[s + j]; c = t[s + j] is the **bad character**.
- If c occurs in the pattern **left of j**: shift so that the rightmost c lines up under t[s + j].
- If c **does not occur** in the pattern (last = −1): shift the pattern entirely past c (j + 1).
- Formula: **s += max(1, j − last[c])**. If c last occurs right of j, then j − last[c] ≤ 0; hence the minimum of 1.

```
int[] lastOccurrence(String p) {
  int[] last = new int[256];
  Arrays.fill(last, -1);
  for (int i = 0; i < p.length(); i++)
    last[p.charAt(i)] = i;
  return last;
}
```

The Good-Suffix Rule

- The suffix u = P[j+1..m−1] matched the text before the mismatch: the **"good suffix"**.
- **Case 1:** if u occurs elsewhere in the pattern (preceded by a different character), shift so that this occurrence lines up under u in the text.
- **Case 2:** otherwise, align the longest **prefix** of P that equals a suffix of u.
- **Case 3:** if neither exists, shift the pattern by m.
- The table is precomputed in O(m) time (similar to KMP's LPS idea, but for suffixes).



"AB" matched, t[2] = 'A' ≠ p[2] = 'B'. Bad character: last[A] = 3 > j → only 1. Good suffix: "AB" also occurs at p[1..2] (preceded by 'C' ≠ 'B') → shift **2**.

Example: "HELLO" in "LO LOELLO O HELLO"

c	H	E	L	O	other
last[c]	0	1	3	4	−1

Rule: s += max(1, j − last[c]). green = match, red = bad character.



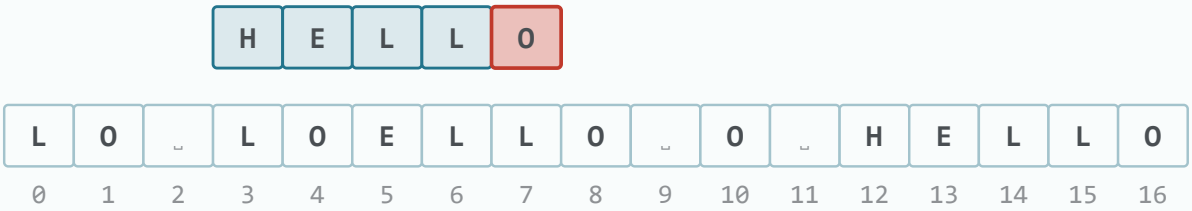


Alignment 1 (s = 0): 2 characters matched, t[2] = '␣' ≠ p[2] = 'L'. '␣' is not in the pattern → j − (−1) = 3 → **shift 3**.

Example: "HELLO" in "LO LOELLO O HELLO"

c	H	E	L	O	other
last[c]	0	1	3	4	−1

Rule: s += max(1, j − last[c]). green = match, red = bad character.

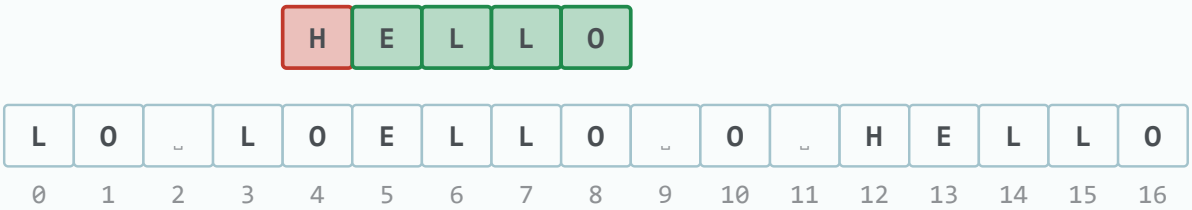


Alignment 2 (s = 3): t[7] = 'L' ≠ p[4] = 'O'. last['L'] = 3 → j − 3 = 1 → **shift 1**.

Example: "HELLO" in "LO LOELLO O HELLO"

c	H	E	L	O	other
last[c]	0	1	3	4	−1

Rule: s += max(1, j − last[c]). green = match, red = bad character.

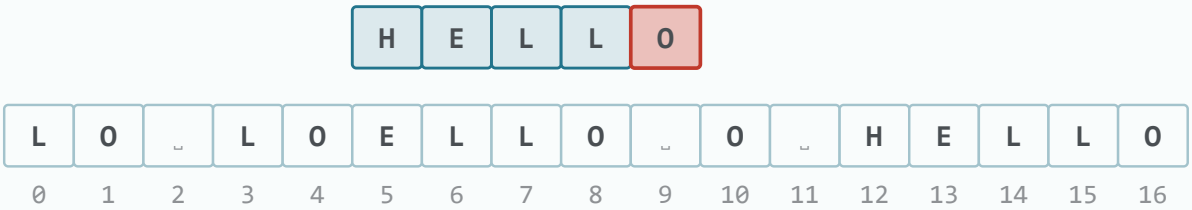


Alignment 3 (s = 4): 4 characters matched, t[4] = 'O' ≠ p[0] = 'H'. last['O'] = 4 ≥ j → at least 1 → **shift 1**.

Example: "HELLO" in "LO LOELLO O HELLO"

c	H	E	L	O	other
last[c]	0	1	3	4	−1

Rule: s += max(1, j − last[c]). green = match, red = bad character.



Alignment 4 (s = 5): t[9] = '␣' ≠ p[4] = 'O'. '␣' is not in the pattern → j − (−1) = 5 → **shift 5**.

Example: "HELLO" in "LO LOELLO O HELLO"

c	H	E	L	O	other
last[c]	0	1	3	4	−1

Rule: s += max(1, j − last[c]). green = match, red = bad character.



Alignment 5 (s = 10): t[14] = 'L' ≠ p[4] = 'O'. last['L'] = 3 → j − 3 = 1 → **shift 1**.

Example: "HELLO" in "LO LOELLO O HELLO"

c	H	E	L	O	other
last[c]	0	1	3	4	−1

Rule: s += max(1, j − last[c]). green = match, red = bad character.



Alignment 6 (s = 11): t[15] = 'L' ≠ p[4] = 'O'. last['L'] = 3 → j − 3 = 1 → **shift 1**.

Example: "HELLO" in "LO LOELLO O HELLO"

c	H	E	L	O	other
last[c]	0	1	3	4	−1

Rule: s += max(1, j − last[c]). green = match, red = bad character.



Alignment 7 (s = 12): all 5 characters match right to left → **HELLO found at s = 12**. 7 alignments, 17 comparisons in total.

Horspool's Simplification and the Shift Table

- Horspool (1980): wherever the mismatch occurs, shift according to the text character under the **last position of the window**.
- Shift table: shift[c] = m − 1 − (rightmost position of c in P[0..m−2]); **m** if c does not occur there.
- HELLO: H → 4, E → 3, L → 1, **O → 5** (the last character is excluded; O occurs nowhere else), other → 5.
- The same example needs only 5 alignments (7 with the basic bad-character rule).

Alignment	s	Comparison	last char	shift
1	0	t[2] = '␣' ≠ p[2]	'O'	5
2	5	t[9] = '␣' ≠ p[4]	'␣'	5
3	10	t[14] = 'L' ≠ p[4]	'L'	1
4	11	t[15] = 'L' ≠ p[4]	'L'	1
5	12	all match	—	**found**

horspool("LO LOELLO O HELLO", "HELLO")

```
1  int horspool(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] shift = new int[256];
4    Arrays.fill(shift, m);
5    for (int i = 0; i < m - 1; i++)
6      shift[p.charAt(i)] = m - 1 - i;
7    int s = 0;
8    while (s <= n - m) {
9      int j = m - 1;
10     while (j >= 0 &&
11       p.charAt(j) == t.charAt(s + j))
12       j--;
13     if (j < 0) return s;
14     s += shift[t.charAt(s + m - 1)];
15   }
16   return -1;
17 }
```

t	L	O	␣	L	O	E	L	L	O	␣	O	␣	H	E	L	L	O
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

	H	E	L	O	other
shift	5	5	5	5	5

n = 17, m = 5. All 256 cells of `shift` start at m = 5: on a character not in the pattern, the pattern jumps entirely past it.

Step 1 / 11

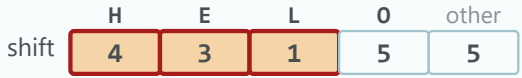
VARIABLES

`n = 17` `m = 5` `s = –` `j = –`

horspool("LO LOELLO O HELLO", "HELLO")

```
1  int horspool(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] shift = new int[256];
4    Arrays.fill(shift, m);
▶    for (int i = 0; i < m - 1; i++)
▶      shift[p.charAt(i)] = m - 1 - i;
7    int s = 0;
8    while (s <= n - m) {
9      int j = m - 1;
10     while (j >= 0 &&
11       p.charAt(j) == t.charAt(s + j))
12       j--;
13     if (j < 0) return s;
14     s += shift[t.charAt(s + m - 1)];
15   }
16   return -1;
17 }
```





Scan P[0..3], excluding the last character: H → 4, E → 3, L first 2 then **1** (the rightmost occurrence wins). O occurs only at the end, so it stays 5.

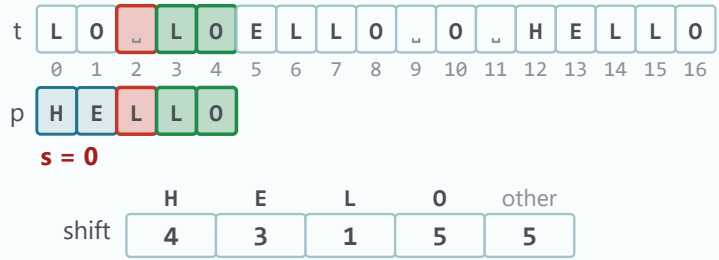
Step 2 / 11

VARIABLES

`n = 17` `m = 5` `s = –` `j = –`

horspool("LO LOELLO O HELLO", "HELLO")

```
1  int horspool(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] shift = new int[256];
4    Arrays.fill(shift, m);
5    for (int i = 0; i < m - 1; i++)
6      shift[p.charAt(i)] = m - 1 - i;
7    int s = 0;
8    while (s <= n - m) {
▶      int j = m - 1;
▶      while (j >= 0 &&
▶        p.charAt(j) == t.charAt(s + j))
▶        j--;
13     if (j < 0) return s;
14     s += shift[t.charAt(s + m - 1)];
15   }
16   return -1;
17 }
```



s = 0: 2 characters match from the right, then t[2] = '␣' ≠ p[2] = 'L' → mismatch.

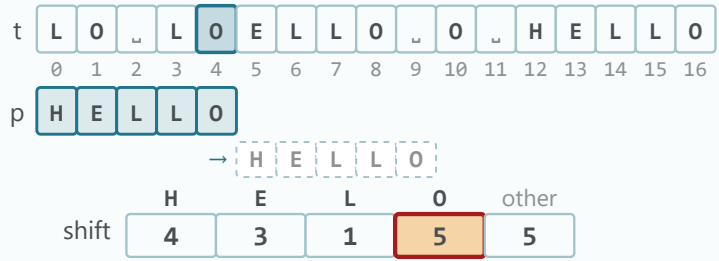
Step 3 / 11

VARIABLES

`n = 17` `m = 5` `s = 0` `j = 2`

horspool("LO LOELLO O HELLO", "HELLO")

```
1  int horspool(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] shift = new int[256];
4    Arrays.fill(shift, m);
5    for (int i = 0; i < m - 1; i++)
6      shift[p.charAt(i)] = m - 1 - i;
7    int s = 0;
8    while (s <= n - m) {
9      int j = m - 1;
10     while (j >= 0 &&
11       p.charAt(j) == t.charAt(s + j))
12       j--;
13     if (j < 0) return s;
▶      s += shift[t.charAt(s + m - 1)];
15   }
16   return -1;
17 }
```



The shift looks at the **last** character of the window, not at the mismatch: t[4] = 'O' → shift = 5 → s = 5.

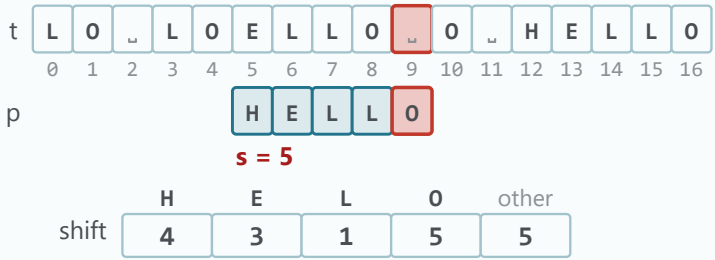
Step 4 / 11

VARIABLES

n = 17 | m = 5 | s = 5 | j = 2

horspool("LO LOELLO O HELLO", "HELLO")

```
1  int horspool(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] shift = new int[256];
4    Arrays.fill(shift, m);
5    for (int i = 0; i < m - 1; i++)
6      shift[p.charAt(i)] = m - 1 - i;
7    int s = 0;
8    while (s <= n - m) {
▶      int j = m - 1;
▶      while (j >= 0 &&
▶        p.charAt(j) == t.charAt(s + j))
▶        j--;
13     if (j < 0) return s;
14     s += shift[t.charAt(s + m - 1)];
15   }
16   return -1;
17 }
```



s = 5: t[9] = '␣' ≠ p[4] = 'O' → mismatch.

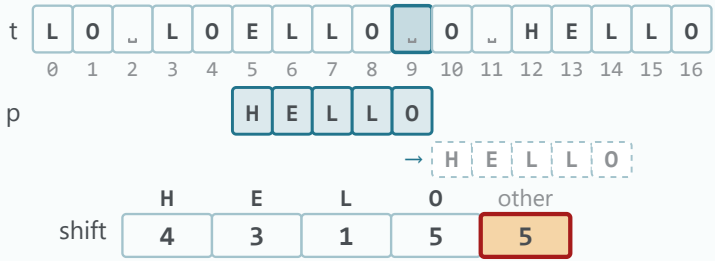
Step 5 / 11

VARIABLES

`n = 17` `m = 5` `s = 5` `j = 4`

horspool("LO LOELLO O HELLO", "HELLO")

```
1  int horspool(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] shift = new int[256];
4    Arrays.fill(shift, m);
5    for (int i = 0; i < m - 1; i++)
6      shift[p.charAt(i)] = m - 1 - i;
7    int s = 0;
8    while (s <= n - m) {
9      int j = m - 1;
10     while (j >= 0 &&
11       p.charAt(j) == t.charAt(s + j))
12       j--;
13     if (j < 0) return s;
▶      s += shift[t.charAt(s + m - 1)];
15   }
16   return -1;
17 }
```



The window's last character t[9] = '␣' is not in the pattern → shift = 5: the pattern jumps entirely past it, s = 10.

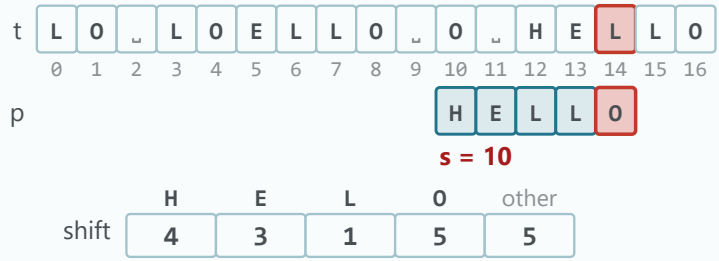
Step 6 / 11

VARIABLES

n = 17 | m = 5 | s = 10 | j = 4

horspool("LO LOELLO O HELLO", "HELLO")

```
1  int horspool(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] shift = new int[256];
4    Arrays.fill(shift, m);
5    for (int i = 0; i < m - 1; i++)
6      shift[p.charAt(i)] = m - 1 - i;
7    int s = 0;
8    while (s <= n - m) {
▶      int j = m - 1;
▶      while (j >= 0 &&
▶        p.charAt(j) == t.charAt(s + j))
▶        j--;
13     if (j < 0) return s;
14     s += shift[t.charAt(s + m - 1)];
15   }
16   return -1;
17 }
```



s = 10: t[14] = 'L' ≠ p[4] = 'O' → mismatch.

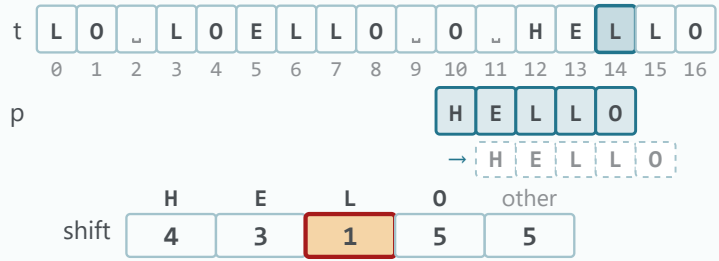
Step 7 / 11

VARIABLES

`n = 17` `m = 5` `s = 10` `j = 4`

horspool("LO LOELLO O HELLO", "HELLO")

```
1  int horspool(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] shift = new int[256];
4    Arrays.fill(shift, m);
5    for (int i = 0; i < m - 1; i++)
6      shift[p.charAt(i)] = m - 1 - i;
7    int s = 0;
8    while (s <= n - m) {
9      int j = m - 1;
10     while (j >= 0 &&
11       p.charAt(j) == t.charAt(s + j))
12       j--;
13     if (j < 0) return s;
▶      s += shift[t.charAt(s + m - 1)];
15   }
16   return -1;
17 }
```



The shift looks at the **last** character of the window, not at the mismatch: t[14] = 'L' → shift = 1 → s = 11.

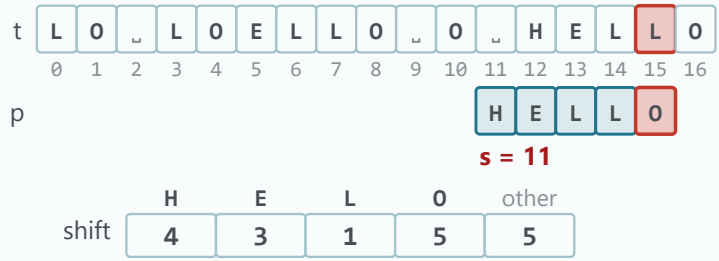
Step 8 / 11

VARIABLES

n = 17 | m = 5 | s = 11 | j = 4

horspool("LO LOELLO O HELLO", "HELLO")

```
1  int horspool(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] shift = new int[256];
4    Arrays.fill(shift, m);
5    for (int i = 0; i < m - 1; i++)
6      shift[p.charAt(i)] = m - 1 - i;
7    int s = 0;
8    while (s <= n - m) {
▶      int j = m - 1;
▶      while (j >= 0 &&
▶        p.charAt(j) == t.charAt(s + j))
▶        j--;
13     if (j < 0) return s;
14     s += shift[t.charAt(s + m - 1)];
15   }
16   return -1;
17 }
```



s = 11: t[15] = 'L' ≠ p[4] = 'O' → mismatch.

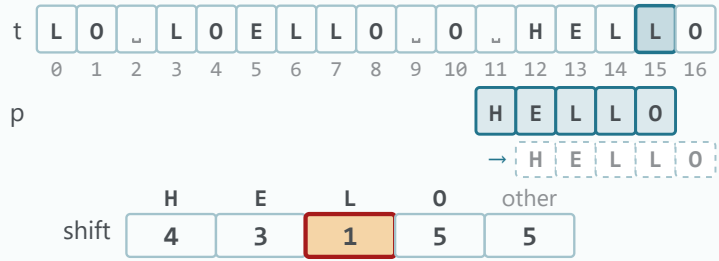
Step 9 / 11

VARIABLES

`n = 17` `m = 5` `s = 11` `j = 4`

horspool("LO LOELLO O HELLO", "HELLO")

```
1  int horspool(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] shift = new int[256];
4    Arrays.fill(shift, m);
5    for (int i = 0; i < m - 1; i++)
6      shift[p.charAt(i)] = m - 1 - i;
7    int s = 0;
8    while (s <= n - m) {
9      int j = m - 1;
10     while (j >= 0 &&
11       p.charAt(j) == t.charAt(s + j))
12       j--;
13     if (j < 0) return s;
▶      s += shift[t.charAt(s + m - 1)];
15   }
16   return -1;
17 }
```



The shift looks at the **last** character of the window, not at the mismatch: t[15] = 'L' → shift = 1 → s = 12.

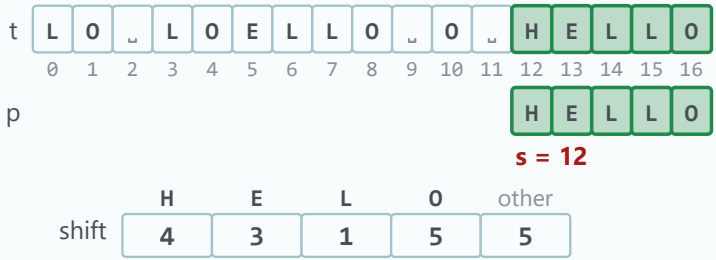
Step 10 / 11

VARIABLES

n = 17 | m = 5 | s = 12 | j = 4

horspool("LO LOELLO O HELLO", "HELLO")

```
1  int horspool(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] shift = new int[256];
4    Arrays.fill(shift, m);
5    for (int i = 0; i < m - 1; i++)
6      shift[p.charAt(i)] = m - 1 - i;
7    int s = 0;
8    while (s <= n - m) {
9      int j = m - 1;
10     while (j >= 0 &&
11       p.charAt(j) == t.charAt(s + j))
12       j--;
▶      if (j < 0) return s;
14     s += shift[t.charAt(s + m - 1)];
15   }
16   return -1;
17 }
```



All five characters match right to left, j = −1 → return **s = 12**. 5 alignments, 11 comparisons in total.

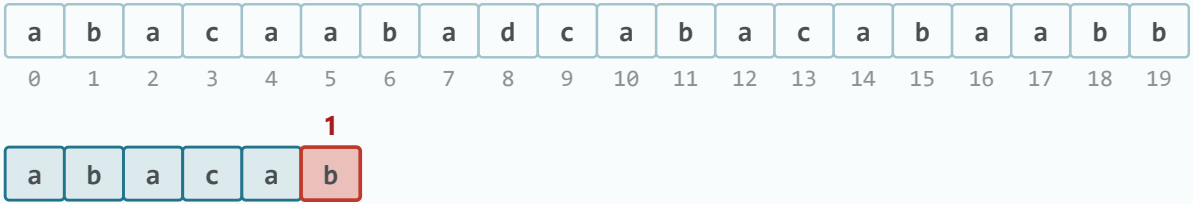
Step 11 / 11

VARIABLES

`n = 17` `m = 5` `s = 12` `j = −1`

Example: Searching with the Last-Occurrence Table

Pattern **abacab**, last occurrence: a = 4, b = 5, c = 3, * = −1. The number above a cell is the comparison counter.



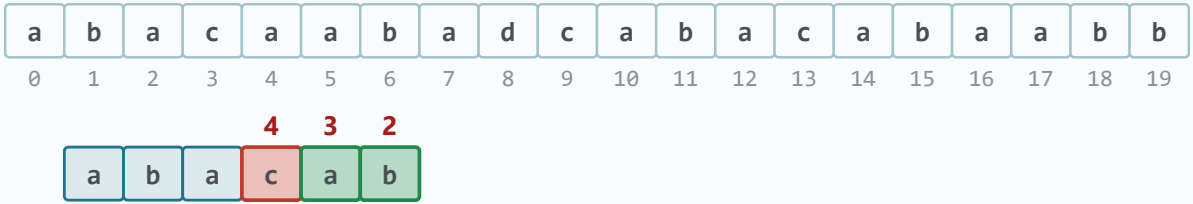
Alignment 1 (s = 0): t[5] = 'a' ≠ p[5] = 'b'. j − last['a'] = 5 − 4 = 1 → shift 1.

Step 1 / 10

10 alignments, 20 comparisons in total (n = 20, m = 6): a character not in the pattern, such as 'd', yields a 6-position jump.

Example: Searching with the Last-Occurrence Table

Pattern **abacab**, last occurrence: a = 4, b = 5, c = 3, * = −1. The number above a cell is the comparison counter.

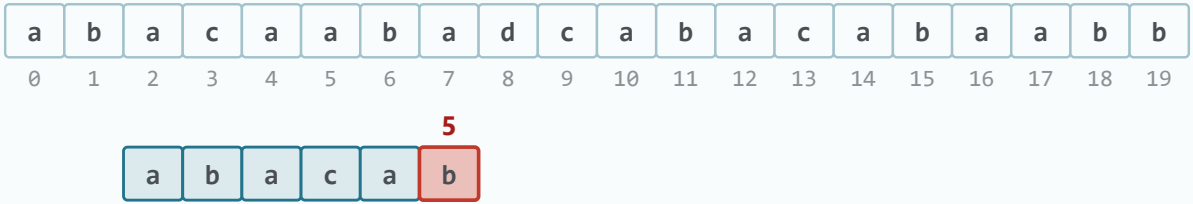


Alignment 2 (s = 1): t[4] = 'a' ≠ p[3] = 'c'. last['a'] = 4 ≥ j = 3 → 1 → shift 1.

10 alignments, 20 comparisons in total (n = 20, m = 6): a character not in the pattern, such as 'd', yields a 6-position jump.

Example: Searching with the Last-Occurrence Table

Pattern **abacab**, last occurrence: a = 4, b = 5, c = 3, * = −1. The number above a cell is the comparison counter.

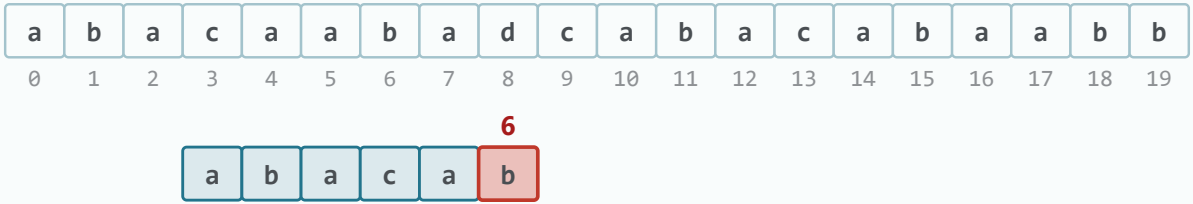


Alignment 3 (s = 2): t[7] = 'a' ≠ p[5] = 'b'. j − last['a'] = 5 − 4 = 1 → shift 1.

10 alignments, 20 comparisons in total (n = 20, m = 6): a character not in the pattern, such as 'd', yields a 6-position jump.

Example: Searching with the Last-Occurrence Table

Pattern **abacab**, last occurrence: a = 4, b = 5, c = 3, * = −1. The number above a cell is the comparison counter.

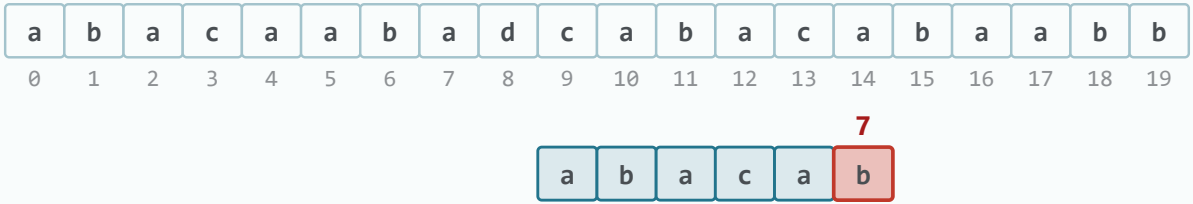


Alignment 4 (s = 3): t[8] = 'd' ≠ p[5] = 'b'. 'd' is not in the pattern → j + 1 = 6 → shift 6.

10 alignments, 20 comparisons in total (n = 20, m = 6): a character not in the pattern, such as 'd', yields a 6-position jump.

Example: Searching with the Last-Occurrence Table

Pattern **abacab**, last occurrence: a = 4, b = 5, c = 3, * = −1. The number above a cell is the comparison counter.

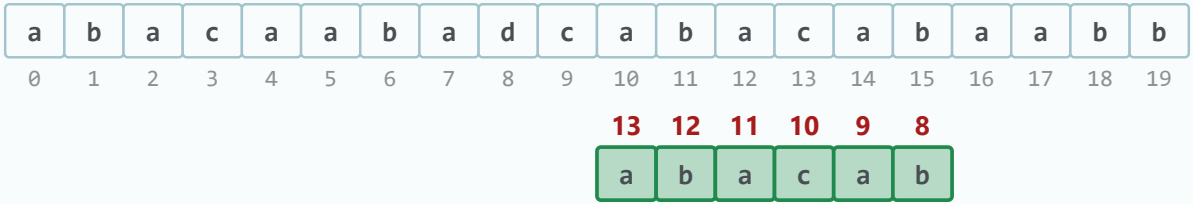


Alignment 5 (s = 9): t[14] = 'a' ≠ p[5] = 'b'. j − last['a'] = 5 − 4 = 1 → shift 1.

10 alignments, 20 comparisons in total (n = 20, m = 6): a character not in the pattern, such as 'd', yields a 6-position jump.

Example: Searching with the Last-Occurrence Table

Pattern **abacab**, last occurrence: a = 4, b = 5, c = 3, * = −1. The number above a cell is the comparison counter.

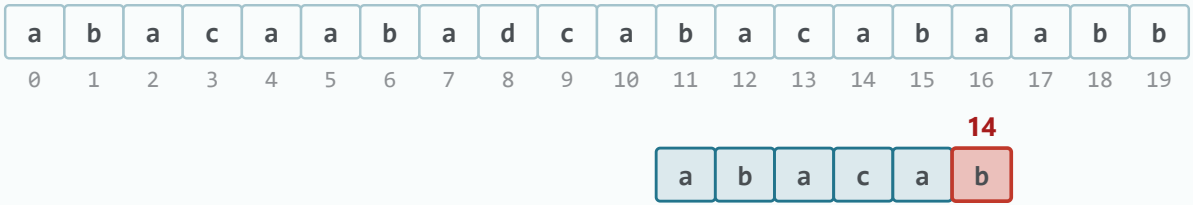


Alignment 6 (s = 10): all 6 comparisons match → **match at s = 10**. We want all occurrences, so shift 1 and continue.

10 alignments, 20 comparisons in total (n = 20, m = 6): a character not in the pattern, such as 'd', yields a 6-position jump.

Example: Searching with the Last-Occurrence Table

Pattern **abacab**, last occurrence: a = 4, b = 5, c = 3, * = −1. The number above a cell is the comparison counter.

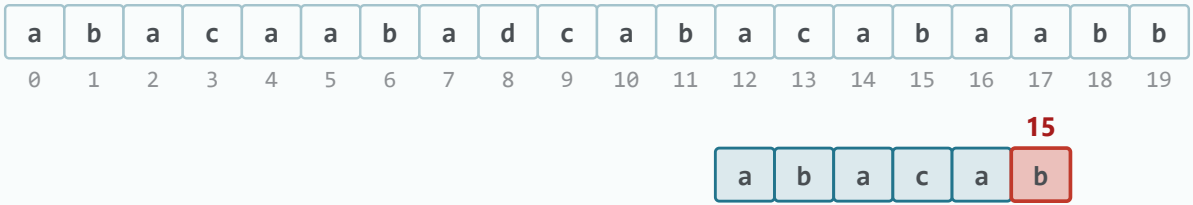


Alignment 7 (s = 11): t[16] = 'a' ≠ p[5] = 'b'. j − last['a'] = 5 − 4 = 1 → shift 1.

10 alignments, 20 comparisons in total (n = 20, m = 6): a character not in the pattern, such as 'd', yields a 6-position jump.

Example: Searching with the Last-Occurrence Table

Pattern **abacab**, last occurrence: a = 4, b = 5, c = 3, * = −1. The number above a cell is the comparison counter.

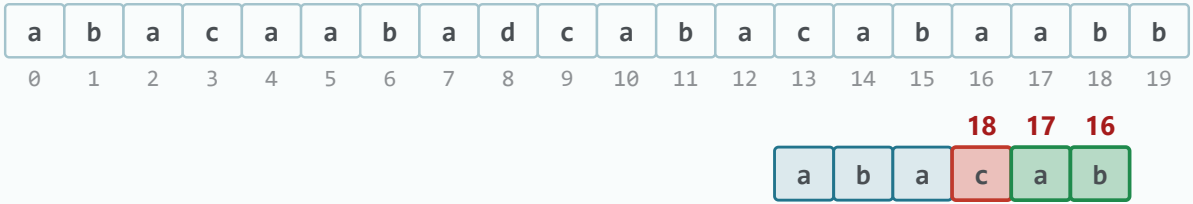


Alignment 8 (s = 12): t[17] = 'a' ≠ p[5] = 'b'. j − last['a'] = 5 − 4 = 1 → shift 1.

10 alignments, 20 comparisons in total (n = 20, m = 6): a character not in the pattern, such as 'd', yields a 6-position jump.

Example: Searching with the Last-Occurrence Table

Pattern **abacab**, last occurrence: a = 4, b = 5, c = 3, * = −1. The number above a cell is the comparison counter.

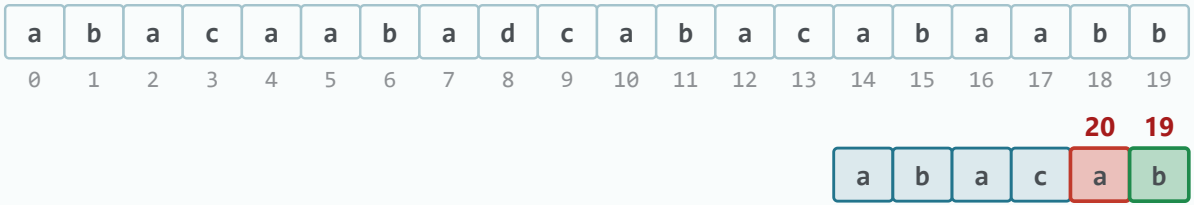


Alignment 9 (s = 13): t[16] = 'a' ≠ p[3] = 'c'. last['a'] = 4 ≥ j = 3 → 1 → shift 1.

10 alignments, 20 comparisons in total (n = 20, m = 6): a character not in the pattern, such as 'd', yields a 6-position jump.

Example: Searching with the Last-Occurrence Table

Pattern **abacab**, last occurrence: a = 4, b = 5, c = 3, * = −1. The number above a cell is the comparison counter.



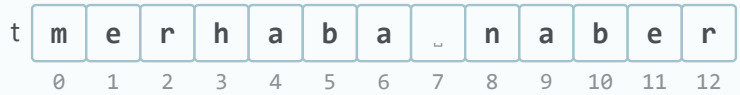
Alignment 10 (s = 14): t[18] = 'b' ≠ p[4] = 'a'. last['b'] = 5 ≥ j = 4 → 1 → shift 1.

Step 10 / 10

10 alignments, 20 comparisons in total (n = 20, m = 6): a character not in the pattern, such as 'd', yields a 6-position jump.

boyerMoore("merhaba naber", "abe")

```
1  int boyerMoore(String t, String p) {
▶    int n = t.length(), m = p.length();
▶    int[] last = lastOccurrence(p);
▶    int s = 0;
5    while (s <= n - m) {
6      int j = m - 1;
7      while (j >= 0 &&
8        p.charAt(j) == t.charAt(s + j))
9        j--;
10     if (j < 0) return s;
11     s += Math.max(1,
12       j - last[t.charAt(s + j)]);
13   }
14   return -1;
15 }
```



last: a = 0, b = 1, e = 2, other = −1

n = 13, m = 3. Last-occurrence table built; s = 0.

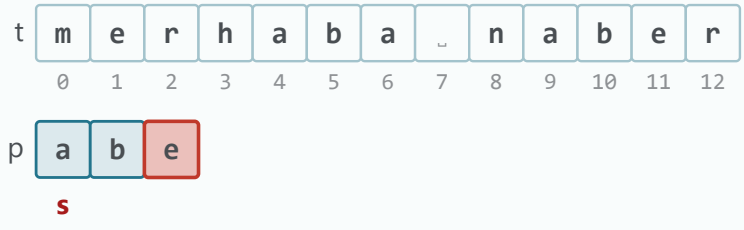
Step 1 / 9

VARIABLES

n = 13 | m = 3 | s = 0 | j = –

boyerMoore("merhaba naber", "abe")

```
1  int boyerMoore(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] last = lastOccurrence(p);
4    int s = 0;
5    while (s <= n - m) {
6      int j = m - 1;
7      while (j >= 0 &&
8        p.charAt(j) == t.charAt(s + j))
9        j--;
10     if (j < 0) return s;
▶      s += Math.max(1,
▶        j - last[t.charAt(s + j)]);
13   }
14   return -1;
15 }
```



last: a = 0, b = 1, e = 2, other = −1

p[2] = 'e' ≠ t[2] = 'r'; last['r'] = -1 → s += max(1, 2 − (-1)) = 3.

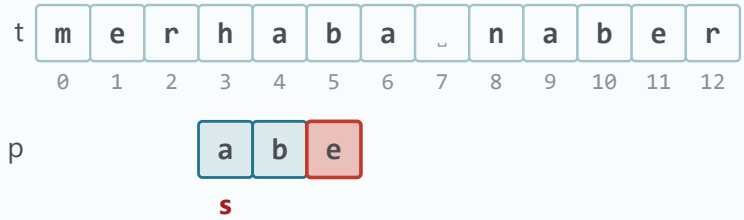
Step 2 / 9

VARIABLES

`n = 13` `m = 3` `s = 3` `j = 2`

boyerMoore("merhaba naber", "abe")

```
1  int boyerMoore(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] last = lastOccurrence(p);
4    int s = 0;
5    while (s <= n - m) {
6      int j = m - 1;
7      while (j >= 0 &&
8        p.charAt(j) == t.charAt(s + j))
9        j--;
10     if (j < 0) return s;
▶      s += Math.max(1,
▶        j - last[t.charAt(s + j)]);
13   }
14   return -1;
15 }
```



last: a = 0, b = 1, e = 2, other = −1

p[2] = 'e' ≠ t[5] = 'b'; last['b'] = 1 → s += max(1, 2 − (1)) = 1.

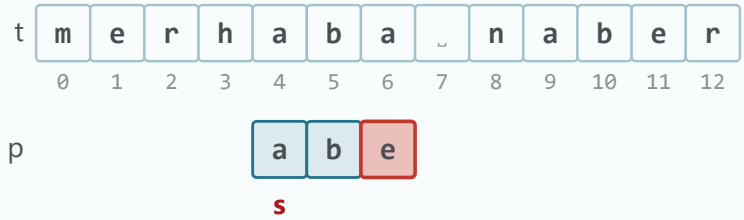
Step 3 / 9

VARIABLES

`n = 13` `m = 3` `s = 4` `j = 2`

boyerMoore("merhaba naber", "abe")

```
1  int boyerMoore(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] last = lastOccurrence(p);
4    int s = 0;
5    while (s <= n - m) {
6      int j = m - 1;
7      while (j >= 0 &&
8        p.charAt(j) == t.charAt(s + j))
9        j--;
10     if (j < 0) return s;
▶      s += Math.max(1,
▶        j - last[t.charAt(s + j)]);
13   }
14   return -1;
15 }
```



last: a = 0, b = 1, e = 2, other = −1

p[2] = 'e' ≠ t[6] = 'a'; last['a'] = 0 → s += max(1, 2 − (0)) = 2.

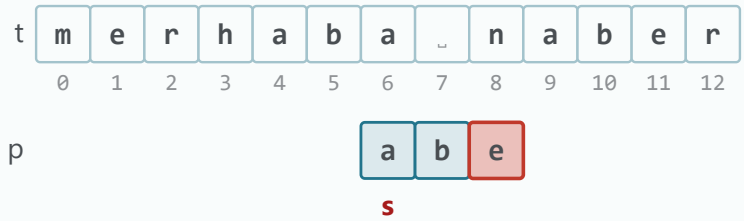
Step 4 / 9

VARIABLES

n = 13 | m = 3 | s = 6 | j = 2

boyerMoore("merhaba naber", "abe")

```
1  int boyerMoore(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] last = lastOccurrence(p);
4    int s = 0;
5    while (s <= n - m) {
6      int j = m - 1;
7      while (j >= 0 &&
8        p.charAt(j) == t.charAt(s + j))
9        j--;
10     if (j < 0) return s;
▶      s += Math.max(1,
▶        j - last[t.charAt(s + j)]);
13   }
14   return -1;
15 }
```



last: a = 0, b = 1, e = 2, other = −1

p[2] = 'e' ≠ t[8] = 'n'; last['n'] = -1 → s += max(1, 2 − (-1)) = 3.

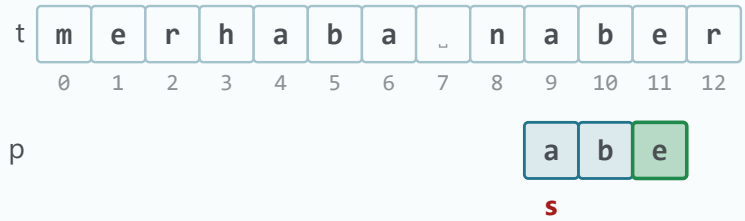
Step 5 / 9

VARIABLES

n = 13 | m = 3 | s = 9 | j = 2

boyerMoore("merhaba naber", "abe")

```
1  int boyerMoore(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] last = lastOccurrence(p);
4    int s = 0;
5    while (s <= n - m) {
6      int j = m - 1;
▶      while (j >= 0 &&
▶        p.charAt(j) == t.charAt(s + j))
▶        j--;
10     if (j < 0) return s;
11     s += Math.max(1,
12       j - last[t.charAt(s + j)]);
13   }
14   return -1;
15 }
```



last: a = 0, b = 1, e = 2, other = −1

p[2] = t[11] = 'e' → j-- .

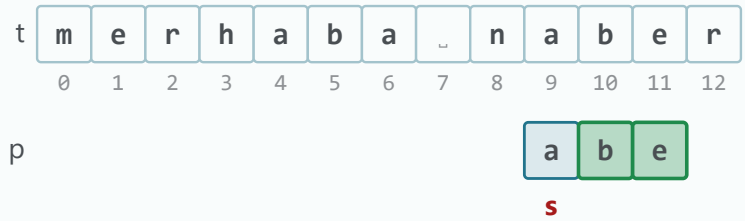
Step 6 / 9

VARIABLES

n = 13 | m = 3 | s = 9 | j = 1

boyerMoore("merhaba naber", "abe")

```
1  int boyerMoore(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] last = lastOccurrence(p);
4    int s = 0;
5    while (s <= n - m) {
6      int j = m - 1;
▶      while (j >= 0 &&
▶        p.charAt(j) == t.charAt(s + j))
▶        j--;
10     if (j < 0) return s;
11     s += Math.max(1,
12       j - last[t.charAt(s + j)]);
13   }
14   return -1;
15 }
```



last: a = 0, b = 1, e = 2, other = −1

p[1] = t[10] = 'b' → j-- .

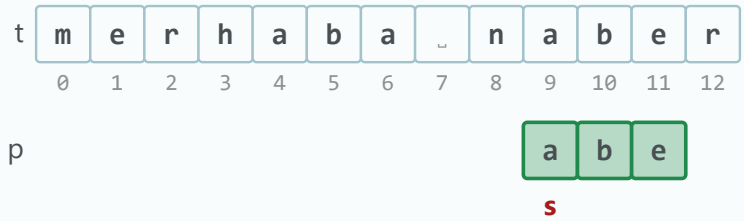
Step 7 / 9

VARIABLES

n = 13 | m = 3 | s = 9 | j = 0

boyerMoore("merhaba naber", "abe")

```
1  int boyerMoore(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] last = lastOccurrence(p);
4    int s = 0;
5    while (s <= n - m) {
6      int j = m - 1;
▶      while (j >= 0 &&
▶        p.charAt(j) == t.charAt(s + j))
▶        j--;
10     if (j < 0) return s;
11     s += Math.max(1,
12       j - last[t.charAt(s + j)]);
13   }
14   return -1;
15 }
```



last: a = 0, b = 1, e = 2, other = −1

p[0] = t[9] = 'a' → j-- .

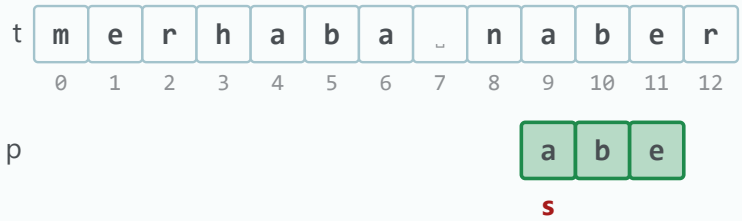
Step 8 / 9

VARIABLES

n = 13 | m = 3 | s = 9 | j = −1

boyerMoore("merhaba naber", "abe")

```
1  int boyerMoore(String t, String p) {
2    int n = t.length(), m = p.length();
3    int[] last = lastOccurrence(p);
4    int s = 0;
5    while (s <= n - m) {
6      int j = m - 1;
7      while (j >= 0 &&
8        p.charAt(j) == t.charAt(s + j))
9        j--;
▶      if (j < 0) return s;
11     s += Math.max(1,
12       j - last[t.charAt(s + j)]);
13   }
14   return -1;
15 }
```



last: a = 0, b = 1, e = 2, other = −1

j < 0 → pattern found at **s = 9** ("n**aber**").

Step 9 / 9

VARIABLES

`n = 13` `m = 3` `s = 9` `j = -1`

Boyer-Moore: Analysis and Example

- **Best:** text characters do not occur in the pattern → 1 comparison per alignment, skip m: ≈ n/m comparisons.
- **Worst (bad character only):** T = aaaa…a, P = baaa…a → m comparisons per alignment, shift 1: O(n · m).
- With a large alphabet and frequent mismatches, **longer patterns give longer shifts** (best ≈ n/m).
- With a small alphabet (DNA, binary data) the bad-character rule is weak; the good-suffix rule matters more.

Example

Text: `abccbaabccbaabcbcabbabcabc`
Pattern: `abcbcabbabcabc` (m = 14)
Last-occurrence table: **a: 11, b: 12, c: 13**
Result: the pattern is found **at position 12**.

The Rabin-Karp Algorithm

- Finds occurrences of a pattern in a text; developed by **Michael O. Rabin** and **Richard M. Karp** in 1987.
- Instead of comparing characters one by one, it compares the **hash** of the pattern with the hash of each length-m window of the text.
- **Rolling hash**: when the window slides one step right, the new hash is computed from the old one in **O(1)**.
- Equal hashes do not guarantee a match: a **spurious hit** is possible → verify character by character.

Complexity

Best: hashes rarely collide → O(n + m)
Average: O(n + m) in practice
Worst: a collision (spurious hit) at every window, constant verification → O(n · m)

Rabin-Karp: How It Works and the Rolling Hash

1. **Hash:** compute the hash of the pattern and of the first text window.
2. **Match check:** substrings whose hash matches are candidates.
3. **Verification:** compare a candidate window with the pattern character by character.
4. **Slide and rehash:** if there is no match, slide the window one step right and update the hash.
5. **Repeat:** continue across the whole text.

Polynomial hash (base B, prime modulus Q):
$h(t[s..s+m-1]) = (t[s]\cdot B^{m-1} + t[s+1]\cdot B^{m-2} + \ldots + t[s+m-1]) \bmod Q$
Slide: $h_{s+1} = (B \cdot (h_s - t[s] \cdot B^{m-1}) + t[s+m]) \bmod Q$ — subtract the outgoing character, shift the rest one digit left, add the incoming character.

A Simple Example: Sum Hash and Spurious Hits

Toy hash: the sum of the letter values. Searching for "TS" in "VUATS".



5 + 1 = 6

pattern "TS"

h = 2 + 3 = 5

U=1 T=2 S=3 A=4 V=5

Window "VU": 5 + 1 = 6 ≠ 5 → hashes differ, no character comparison.

Step 1 / 4

The sum hash is weak (it ignores letter order: "ST" also gives 5). That is why a position-weighted **polynomial hash** with a large prime Q is used.

A Simple Example: Sum Hash and Spurious Hits

Toy hash: the sum of the letter values. Searching for "TS" in "VUATS".



1 + 4 = 5

pattern "TS"

h = 2 + 3 = 5

U=1 T=2 S=3 A=4 V=5

Window "UA": 1 + 4 = **5** = 5 → equal hashes, but "UA" ≠ "TS": a **spurious hit**; the verification is wasted.

The sum hash is weak (it ignores letter order: "ST" also gives 5). That is why a position-weighted **polynomial hash** with a large prime Q is used.

A Simple Example: Sum Hash and Spurious Hits

Toy hash: the sum of the letter values. Searching for "TS" in "VUATS".



4 + 2 = 6

pattern "TS"

h = 2 + 3 = 5

U=1 T=2 S=3 A=4 V=5

Window "AT": 4 + 2 = 6 ≠ 5 → skip. New hash = old − outgoing (U=1) + incoming (T=2): O(1).

The sum hash is weak (it ignores letter order: "ST" also gives 5). That is why a position-weighted **polynomial hash** with a large prime Q is used.

A Simple Example: Sum Hash and Spurious Hits

Toy hash: the sum of the letter values. Searching for "TS" in "VUATS".



pattern "TS"

h = 2 + 3 = 5

2 + 3 = 5

U=1 T=2 S=3 A=4 V=5

Window "TS": 2 + 3 = 5 → equal hashes, and the characters agree too: **match** (s = 3).

Step 4 / 4

The sum hash is weak (it ignores letter order: "ST" also gives 5). That is why a position-weighted **polynomial hash** with a large prime Q is used.

rabinKarp("abracadabra", "cad")

```
1  int rabinKarp(String t, String p) {
2    int n = t.length(), m = p.length();
3    int B = 256, Q = 101, h = 1;
4    for (int i = 0; i < m - 1; i++)
5      h = (h * B) % Q;  // B^(m-1) mod Q
6    int ph = 0, th = 0;
7    for (int i = 0; i < m; i++) {
8      ph = (B * ph + p.charAt(i)) % Q;
9      th = (B * th + t.charAt(i)) % Q;
10   }
11   for (int s = 0; s <= n - m; s++) {
12     if (ph == th && matchAt(t, p, s))
13       return s;
14     if (s < n - m) {
15       th = (B * (th - t.charAt(s) * h)
16             + t.charAt(s + m)) % Q;
17       if (th < 0) th += Q;
18     }
19   }
20   return -1;
21 }
```

t	a	b	r	a	c	a	d	a	b	r	a
	0	1	2	3	4	5	6	7	8	9	10

a=97 b=98 c=99 d=100 r=114

n = 11, m = 3, B = 256, Q = 101. h = 256^2 mod 101 = 88 (weight of the outgoing character).

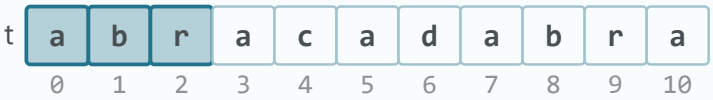
Step 1 / 7

VARIABLES

n = **11** m = **3** h = **88** ph = **–** th = **–** s = **–**

rabinKarp("abracadabra", "cad")

```
1  int rabinKarp(String t, String p) {
2    int n = t.length(), m = p.length();
3    int B = 256, Q = 101, h = 1;
4    for (int i = 0; i < m - 1; i++)
5      h = (h * B) % Q;  // B^(m-1) mod Q
▶    int ph = 0, th = 0;
▶    for (int i = 0; i < m; i++) {
▶      ph = (B * ph + p.charAt(i)) % Q;
▶      th = (B * th + t.charAt(i)) % Q;
10   }
11   for (int s = 0; s <= n - m; s++) {
12     if (ph == th && matchAt(t, p, s))
13       return s;
14     if (s < n - m) {
15       th = (B * (th - t.charAt(s) * h)
16             + t.charAt(s + m)) % Q;
17       if (th < 0) th += Q;
18     }
19   }
20   return -1;
21 }
```



window s = 0

ph = h("cad") = 11, th = h("abr") = 4

ph = (99·256² + 97·256 + 100) mod 101 = **11**; for the first window "abr", th = **4**.

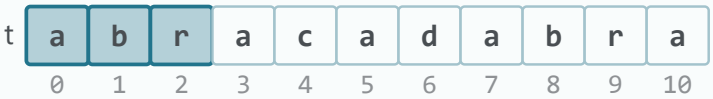
Step 2 / 7

VARIABLES

`n = 11` `m = 3` `h = 88` `ph = 11` `th = 4` `s = –`

rabinKarp("abracadabra", "cad")

```
1  int rabinKarp(String t, String p) {
2    int n = t.length(), m = p.length();
3    int B = 256, Q = 101, h = 1;
4    for (int i = 0; i < m - 1; i++)
5      h = (h * B) % Q;  // B^(m-1) mod Q
6    int ph = 0, th = 0;
7    for (int i = 0; i < m; i++) {
8      ph = (B * ph + p.charAt(i)) % Q;
9      th = (B * th + t.charAt(i)) % Q;
10   }
11   for (int s = 0; s <= n - m; s++) {
▶      if (ph == th && matchAt(t, p, s))
13       return s;
▶      if (s < n - m) {
▶        th = (B * (th - t.charAt(s) * h)
▶              + t.charAt(s + m)) % Q;
▶        if (th < 0) th += Q;
18     }
19   }
20   return -1;
21 }
```



window s = 0

th("abr") = 4 ≠ ph = 11

Window "abr": th = 4 ≠ 11 → characters not checked.
Slide: 'a' out, 'a' in → new th = **30**.

Step 3 / 7

VARIABLES

`n = 11` `m = 3` `h = 88` `ph = 11` `th = 30` `s = 0`

rabinKarp("abracadabra", "cad")

```
1  int rabinKarp(String t, String p) {
2    int n = t.length(), m = p.length();
3    int B = 256, Q = 101, h = 1;
4    for (int i = 0; i < m - 1; i++)
5      h = (h * B) % Q;  // B^(m-1) mod Q
6    int ph = 0, th = 0;
7    for (int i = 0; i < m; i++) {
8      ph = (B * ph + p.charAt(i)) % Q;
9      th = (B * th + t.charAt(i)) % Q;
10   }
11   for (int s = 0; s <= n - m; s++) {
▶      if (ph == th && matchAt(t, p, s))
13       return s;
▶      if (s < n - m) {
▶        th = (B * (th - t.charAt(s) * h)
▶              + t.charAt(s + m)) % Q;
▶        if (th < 0) th += Q;
18     }
19   }
20   return -1;
21 }
```

t	a	b	r	a	c	a	d	a	b	r	a
	0	1	2	3	4	5	6	7	8	9	10

window s = 1

th("bra") = 30 ≠ ph = 11

Window "bra": th = 30 ≠ 11 → characters not checked. Slide: 'b' out, 'c' in → new th = **17**.

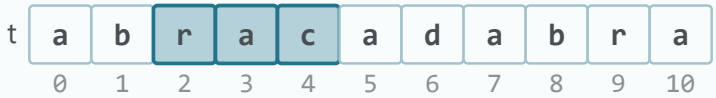
Step 4 / 7

VARIABLES

n = **11** m = **3** h = **88** ph = **11** th = **17** s = **1**

rabinKarp("abracadabra", "cad")

```
1  int rabinKarp(String t, String p) {
2    int n = t.length(), m = p.length();
3    int B = 256, Q = 101, h = 1;
4    for (int i = 0; i < m - 1; i++)
5      h = (h * B) % Q;  // B^(m-1) mod Q
6    int ph = 0, th = 0;
7    for (int i = 0; i < m; i++) {
8      ph = (B * ph + p.charAt(i)) % Q;
9      th = (B * th + t.charAt(i)) % Q;
10   }
11   for (int s = 0; s <= n - m; s++) {
▶      if (ph == th && matchAt(t, p, s))
13       return s;
▶      if (s < n - m) {
▶        th = (B * (th - t.charAt(s) * h)
▶              + t.charAt(s + m)) % Q;
▶        if (th < 0) th += Q;
18     }
19   }
20   return -1;
21 }
```



window s = 2

th("rac") = 17 ≠ ph = 11

Window "rac": th = 17 ≠ 11 → characters not checked. Slide: 'r' out, 'a' in → new th = **41**.

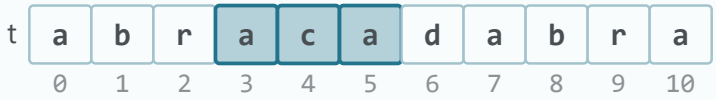
Step 5 / 7

VARIABLES

n = 11 | m = 3 | h = 88 | ph = 11 | th = 41 | s = 2

rabinKarp("abracadabra", "cad")

```
 1 int rabinKarp(String t, String p) {
 2   int n = t.length(), m = p.length();
 3   int B = 256, Q = 101, h = 1;
 4   for (int i = 0; i < m - 1; i++)
 5     h = (h * B) % Q;  // B^(m-1) mod Q
 6   int ph = 0, th = 0;
 7   for (int i = 0; i < m; i++) {
 8     ph = (B * ph + p.charAt(i)) % Q;
 9     th = (B * th + t.charAt(i)) % Q;
10   }
11   for (int s = 0; s <= n - m; s++) {
▶    if (ph == th && matchAt(t, p, s))
13     return s;
▶    if (s < n - m) {
▶      th = (B * (th - t.charAt(s) * h)
▶            + t.charAt(s + m)) % Q;
▶      if (th < 0) th += Q;
18   }
19   }
20   return -1;
21 }
```



window s = 3

th("aca") = 41 ≠ ph = 11

Window "aca": th = 41 ≠ 11 → characters not checked. Slide: 'a' out, 'd' in → new th = **11**.

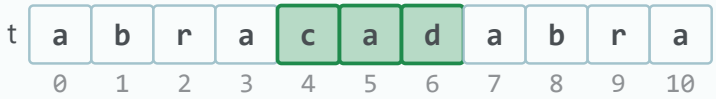
Step 6 / 7

VARIABLES

`n = 11` `m = 3` `h = 88` `ph = 11` `th = 11` `s = 3`

rabinKarp("abracadabra", "cad")

```
1  int rabinKarp(String t, String p) {
2    int n = t.length(), m = p.length();
3    int B = 256, Q = 101, h = 1;
4    for (int i = 0; i < m - 1; i++)
5      h = (h * B) % Q;  // B^(m-1) mod Q
6    int ph = 0, th = 0;
7    for (int i = 0; i < m; i++) {
8      ph = (B * ph + p.charAt(i)) % Q;
9      th = (B * th + t.charAt(i)) % Q;
10   }
11   for (int s = 0; s <= n - m; s++) {
▶      if (ph == th && matchAt(t, p, s))
▶        return s;
14     if (s < n - m) {
15       th = (B * (th - t.charAt(s) * h)
16             + t.charAt(s + m)) % Q;
17       if (th < 0) th += Q;
18     }
19   }
20   return -1;
21 }
```



window s = 4

th = 11 = ph → compare characters

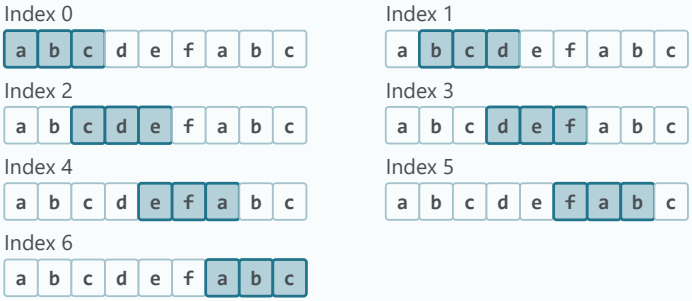
th = 11 = ph → character check with `matchAt`: "cad" = "cad" → **return s = 4**.

Step 7 / 7

VARIABLES

`n = 11` `m = 3` `h = 88` `ph = 11` `th = 11` `s = 4`

The Sliding Window and Analysis



n − m + 1 = 7 windows; each computed from the previous one in O(1).

- Hashing cost: O(m) + (n − m) · O(1).
- With a large prime Q, a spurious hit has probability ≈ 1/Q → expected time **O(n + m)**.
- Bad Q or adversarial input: every window is a candidate → O(n · m).
- **Strength: many patterns** of the same length can be searched in one pass (put their hashes in a set): plagiarism detection, document similarity.
- It also generalizes to 2D pattern search (a block within an image).

MATCHING SUMMARY

Comparing String Matching Algorithms

Algorithm	Key idea	Preprocessing	Average	Worst	Extra memory
Brute force	Tries every position	none	O(n)*	O(n · m)	O(1)
KMP	LPS table, never backs up in the text	O(m)	O(n + m)	O(n + m)	O(m)
Boyer-Moore	Right to left, bad character + good suffix	O(m + σ)	sublinear	O(n · m)**	O(m + σ)
Rabin-Karp	Compares rolling hashes	O(m)	O(n + m)	O(n · m)	O(1)

* for random text over a large alphabet; on regular (repetitive) text it approaches O(n · m). ** with the bad-character rule alone; O(n + m) with good suffix + Galil's rule. σ: alphabet size.

Need a guarantee

KMP: streaming data, where the worst-case bound matters.

Long pattern, natural language

Boyer-Moore / Horspool: fastest in practice.

Many patterns

Rabin-Karp: many patterns in a single pass.

String Compression Algorithms

- **Goal:** store or transmit text (or data) using fewer bits.
- **Lossless** compression: decompression gives back the **original data exactly**. RLE and LZW are lossless (text, programs, PNG/GIF).
- **Lossy** compression: perceptually unimportant detail is discarded (JPEG, MP3); not suitable for text.
- **Compression ratio** = compressed size / original size (smaller is better).

RLE

Consecutive repeats of the same value are stored as one value + a count.

LZW

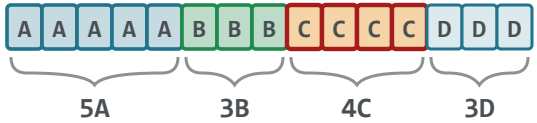
Repeated patterns become short codes in a dictionary that grows on the fly.

Run-Length Encoding (RLE)

- Run-length encoding replaces each sequence of repeated characters (a **run**) with a **count + character** pair.
- **Steps:** scan the text left to right → count consecutive repeats → combine count and character → store the result as the compressed text.
- The text is scanned once, so the time is **O(n)**.
- Simple and effective, but **useless** or even harmful when there is little repetition: "ABCD" → "1A1B1C1D" (twice the size!).
- Used in: fax (black-and-white lines), BMP/PCX/TIFF, simple icons.

Example

Text: `AAAAABBBCCCCDDD` (15 characters)



Compressed: `5A3B4C3D` (8 characters)

Ratio: 8/15 ≈ 0.53

rleCompress("AAAAABBBCCCCDDD")

```
1  String rleCompress(String s) {
▶    StringBuilder result = new
   StringBuilder();
▶    int n = s.length();
4    for (int i = 0; i < n; i++) {
5      char c = s.charAt(i);
6      int count = 1;
7      while (i + 1 < n &&
8        s.charAt(i + 1) == c) {
9        count++;
10       i++;
11     }
12     result.append(count).append(c);
13   }
14   return result.toString();
15 }
```

A	A	A	A	A	B	B	B	C	C	C	C	D	D	D
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

n = 15, result is empty.

Step 1 / 10

VARIABLES

`n = 15` `i = 0` `c = –` `count = –`

OUTPUT

rleCompress("AAAAABBBCCCCDDD")

```
1  String rleCompress(String s) {
2    StringBuilder result = new
   StringBuilder();
3    int n = s.length();
4    for (int i = 0; i < n; i++) {
▶      char c = s.charAt(i);
▶      int count = 1;
7      while (i + 1 < n &&
8        s.charAt(i + 1) == c) {
9        count++;
10       i++;
11     }
12     result.append(count).append(c);
13   }
14   return result.toString();
15 }
```

A	A	A	A	A	B	B	B	C	C	C	C	D	D	D
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

i

New run: c = 'A', count = 1 (i = 0).

Step 2 / 10

VARIABLES

n = 15 | i = 0 | c = A | count = 1

OUTPUT

rleCompress("AAAAABBBCCCCDDD")

```
1  String rleCompress(String s) {
2    StringBuilder result = new
   StringBuilder();
3    int n = s.length();
4    for (int i = 0; i < n; i++) {
5      char c = s.charAt(i);
6      int count = 1;
▶      while (i + 1 < n &&
▶        s.charAt(i + 1) == c) {
▶        count++;
▶        i++;
11     }
▶      result.append(count).append(c);
13   }
14   return result.toString();
15 }
```



The while loop ran 4 times: count = 5, i = 4. Append "5A" to result.

Step 3 / 10

VARIABLES

n = 15 | i = 4 | c = A | count = 5

OUTPUT

```
5A
```

rleCompress("AAAAABBBCCCCDDD")

```
1  String rleCompress(String s) {
2    StringBuilder result = new
   StringBuilder();
3    int n = s.length();
4    for (int i = 0; i < n; i++) {
▶      char c = s.charAt(i);
▶      int count = 1;
7      while (i + 1 < n &&
8        s.charAt(i + 1) == c) {
9        count++;
10       i++;
11     }
12     result.append(count).append(c);
13   }
14   return result.toString();
15 }
```

A	A	A	A	A	B	B	B	C	C	C	C	D	D	D
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

i

New run: c = 'B', count = 1 (i = 5).

Step 4 / 10

VARIABLES

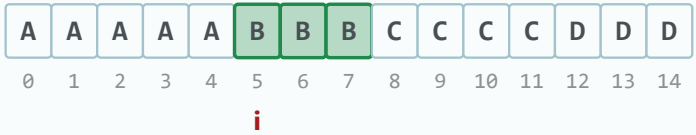
n = 15 | i = 5 | c = B | count = 1

OUTPUT

5A

rleCompress("AAAAABBBCCCCDDD")

```
1  String rleCompress(String s) {
2    StringBuilder result = new
   StringBuilder();
3    int n = s.length();
4    for (int i = 0; i < n; i++) {
5      char c = s.charAt(i);
6      int count = 1;
▶      while (i + 1 < n &&
▶        s.charAt(i + 1) == c) {
▶        count++;
▶        i++;
11     }
▶      result.append(count).append(c);
13   }
14   return result.toString();
15 }
```



The while loop ran 2 times: count = 3, i = 7.
Append "3B" to result.

Step 5 / 10

VARIABLES

n = 15 | i = 7 | c = B | count = 3

OUTPUT

5A3B

rleCompress("AAAAABBBCCCCDDD")

```
1  String rleCompress(String s) {
2    StringBuilder result = new
   StringBuilder();
3    int n = s.length();
4    for (int i = 0; i < n; i++) {
▶      char c = s.charAt(i);
▶      int count = 1;
7      while (i + 1 < n &&
8        s.charAt(i + 1) == c) {
9        count++;
10       i++;
11     }
12     result.append(count).append(c);
13   }
14   return result.toString();
15 }
```

A	A	A	A	A	B	B	B	C	C	C	C	D	D	D
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

i

New run: c = 'C', count = 1 (i = 8).

Step 6 / 10

VARIABLES

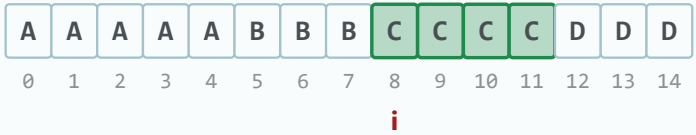
n = 15 | i = 8 | c = C | count = 1

OUTPUT

5A3B

rleCompress("AAAAABBBCCCCDDD")

```
1  String rleCompress(String s) {
2    StringBuilder result = new
   StringBuilder();
3    int n = s.length();
4    for (int i = 0; i < n; i++) {
5      char c = s.charAt(i);
6      int count = 1;
▶      while (i + 1 < n &&
▶        s.charAt(i + 1) == c) {
▶        count++;
▶        i++;
11     }
▶      result.append(count).append(c);
13   }
14   return result.toString();
15 }
```



The while loop ran 3 times: count = 4, i = 11. Append "4C" to result.

Step 7 / 10

VARIABLES

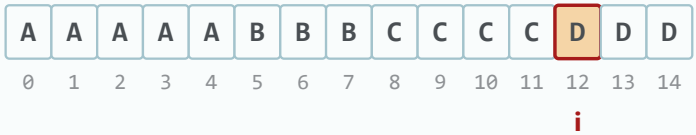
n = 15 | i = 11 | c = C | count = 4

OUTPUT

5A3B4C

rleCompress("AAAAABBBCCCCDDD")

```
1  String rleCompress(String s) {
2    StringBuilder result = new
   StringBuilder();
3    int n = s.length();
4    for (int i = 0; i < n; i++) {
▶      char c = s.charAt(i);
▶      int count = 1;
7      while (i + 1 < n &&
8        s.charAt(i + 1) == c) {
9        count++;
10       i++;
11     }
12     result.append(count).append(c);
13   }
14   return result.toString();
15 }
```



New run: c = 'D', count = 1 (i = 12).

Step 8 / 10

VARIABLES

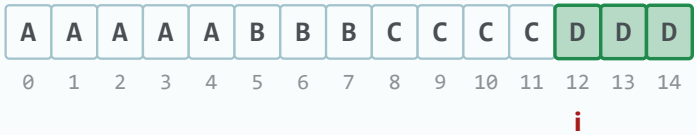
`n = 15` `i = 12` `c = D` `count = 1`

OUTPUT

```
5A3B4C
```

rleCompress("AAAAABBBCCCCDDD")

```
1  String rleCompress(String s) {
2    StringBuilder result = new
   StringBuilder();
3    int n = s.length();
4    for (int i = 0; i < n; i++) {
5      char c = s.charAt(i);
6      int count = 1;
▶      while (i + 1 < n &&
▶        s.charAt(i + 1) == c) {
▶        count++;
▶        i++;
11     }
▶      result.append(count).append(c);
13   }
14   return result.toString();
15 }
```



The while loop ran 2 times: count = 3, i = 14.
Append "3D" to result.

Step 9 / 10

VARIABLES

n = 15 | i = 14 | c = D | count = 3

OUTPUT

```
5A3B4C3D
```

rleCompress("AAAAABBBCCCCDDD")

```
1  String rleCompress(String s) {
2    StringBuilder result = new
   StringBuilder();
3    int n = s.length();
4    for (int i = 0; i < n; i++) {
5      char c = s.charAt(i);
6      int count = 1;
7      while (i + 1 < n &&
8        s.charAt(i + 1) == c) {
9        count++;
10       i++;
11     }
12     result.append(count).append(c);
13   }
▶    return result.toString();
15 }
```

A	A	A	A	A	B	B	B	C	C	C	C	D	D	D
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Result: **"5A3B4C3D"** — 8 characters instead of 15.

Step 10 / 10

VARIABLES

`n = 15` `i = 15` `c = D` `count = 3`

OUTPUT

```
5A3B4C3D
```

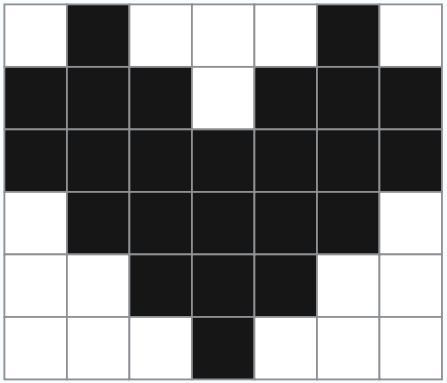
Byte-Level RLE Encoding

Text: `aaaaaaaaaabbbbbbeceeceeceecdddddddddddddddecb` (**46 bytes**)

Count–character pairs	`(10, a)(6, b)(1, e)(1, c)(1, e)(1, c)(1, e)(1, c)(1, e)(1, c)(1, e)(1, c)(1, e)(1, c)(15, d)(1, e)(1, c)(1, b)`
RLE without flag (2 bytes per run)	`10 97 06 98 01 101 01 99 01 101 01 99 01 101 01 99 01 101 01 99 01 101 01 99 01 101 01 99 15 100 01 101 01 99 01 98` → **36 bytes**
RLE with flag (flag = 255)	`255 10 97 255 06 98 101 99 101 99 101 99 101 99 101 99 101 99 255 15 100 101 99 98` → **24 bytes**

In plain RLE, non-repeating stretches such as "ececec…" cost 2 bytes per character. RLE with a flag writes only **long runs** as a `255 count char` triple and leaves other characters as they are (the value 255 itself must be escaped).

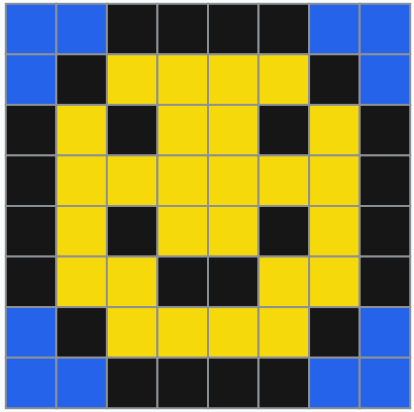
RLE on a Binary (Black-and-White) Image



```
W1B1W3B1W1
B3W1B3
B7
W1B5W1
W2B3W2
W3B1W3
```

- Each row is scanned left to right: W = white, B = black, followed by the run length.
- 42 pixels → 18 runs (each row is encoded separately). With only two colors, the color need not be written at all: runs simply alternate white/black (the fax standard uses this idea).
- Very effective for images with large flat areas (drawings, scanned text); useless for photographs.

Pixel-Level RLE (RGB)



Count	R G B	Count	R G B	Count	R G B
2	0 0 255	4	0 0 0	3	0 0 255
1	0 0 0	4	255 255 0	1	0 0 0
1	0 0 255	1	0 0 0	1	255 255 0
1	0 0 0	2	255 255 0	1	0 0 0
1	255 255 0	2	0 0 0	6	255 255 0
2	0 0 0	1	255 255 0	1	0 0 0
2	255 255 0	1	0 0 0	1	255 255 0
2	0 0 0	2	255 255 0	2	0 0 0
2	255 255 0	1	0 0 0	1	0 0 255
1	0 0 0	4	255 255 0	1	0 0 0
3	0 0 255	4	0 0 0	2	0 0 255

Blue (0, 0, 255), black (0, 0, 0), yellow (255, 255, 0). Raw: 64 × 3 = **192 bytes**. RLE: 33 runs × (1 count + 3 color) = **132 bytes**.

Read the table row by row (left to right, then the next row); if the same color continues at the end of an image row, the run carries over to the next row (e.g. 3 blue).

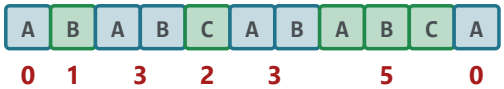
Lempel-Ziv-Welch (LZW) Compression

- **Dictionary-based** lossless compression: Terry Welch (1984) refined Lempel and Ziv's LZ78 method (1978).
- Identifies repeated strings and represents them by **short codes** (indices) into a dictionary.
- The dictionary is **dynamic**: it grows during compression and is **never transmitted**; the decompressor rebuilds the same dictionary on its own.
- Used in formats such as GIF, TIFF, PDF and Unix `compress`.
- As the dictionary grows, code width (bits) and memory grow and performance drops: in practice the dictionary is capped (GIF: 4096 codes = 12 bits) and reset when full.

Example

Text: `ABABCABABCA`

Initial dictionary: A = 0, B = 1, C = 2



Compressed: **0 1 3 2 3 5 0**

7 codes instead of 11 symbols.

lzw("ABABCABABCA")

```
1  List<Integer> lzw(String s) {
▶    Map<String, Integer> dict =
▶        initDict(s);  // A=0, B=1, C=2
▶    String w = "";
▶    List<Integer> codes = new ArrayList<>
     ();
6    for (char c : s.toCharArray()) {
7      String wc = w + c;
8      if (dict.containsKey(wc)) {
9        w = wc;
10     } else {
11       codes.add(dict.get(w));
12       dict.put(wc, dict.size());
13       w = "" + c;
14     }
15   }
16   if (!w.isEmpty())
17     codes.add(dict.get(w));
18   return codes;
19 }
```

A	B	A	B	C	A	B	A	B	C	A
0	1	2	3	4	5	6	7	8	9	10

0	A
1	B
2	C

The dictionary starts with the characters of the text: A = 0, B = 1, C = 2. w = "".

Step 1 / 13

VARIABLES

c = – w = "" wc = –

OUTPUT

lzw("ABABCABABCA")

```
1  List<Integer> lzw(String s) {
2    Map<String, Integer> dict =
3        initDict(s);  // A=0, B=1, C=2
4    String w = "";
5    List<Integer> codes = new ArrayList<>
     ();
6    for (char c : s.toCharArray()) {
▶      String wc = w + c;
▶      if (dict.containsKey(wc)) {
▶        w = wc;
10     } else {
11       codes.add(dict.get(w));
12       dict.put(wc, dict.size());
13       w = "" + c;
14     }
15   }
16   if (!w.isEmpty())
17     codes.add(dict.get(w));
18   return codes;
19 }
```

A	B	A	B	C	A	B	A	B	C	A
0	1	2	3	4	5	6	7	8	9	10

c

0	A
1	B
2	C

"A" is in dict (0) → w = "A"; look for a longer match.

Step 2 / 13

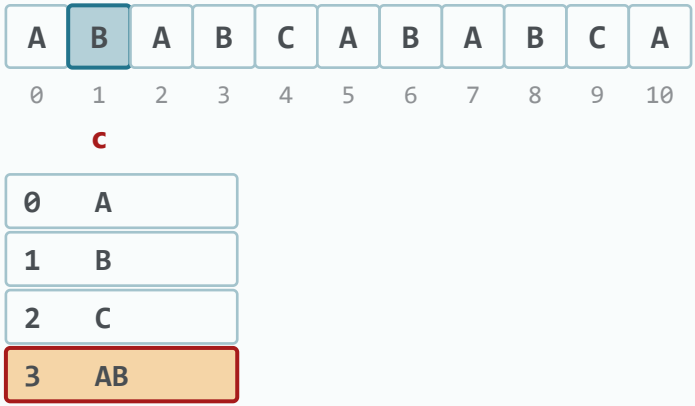
VARIABLES

`c = A` `w = "A"` `wc = "A"`

OUTPUT

lzw("ABABCABABCA")

```
1  List<Integer> lzw(String s) {
2    Map<String, Integer> dict =
3        initDict(s);  // A=0, B=1, C=2
4    String w = "";
5    List<Integer> codes = new ArrayList<>
     ();
6    for (char c : s.toCharArray()) {
7      String wc = w + c;
8      if (dict.containsKey(wc)) {
9        w = wc;
10     } else {
▶        codes.add(dict.get(w));
▶        dict.put(wc, dict.size());
▶        w = "" + c;
14     }
15   }
16   if (!w.isEmpty())
17     codes.add(dict.get(w));
18   return codes;
19 }
```



"AB" not in dict → output code of w = "A": **0**; add "AB" = 3 to dict; w = "B".

Step 3 / 13

VARIABLES

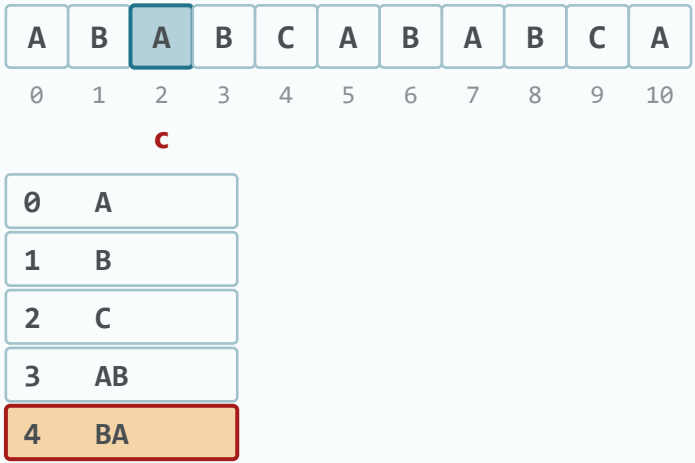
`c = B` `w = "B"` `wc = "AB"`

OUTPUT

```
0
```

lzw("ABABCABABCA")

```
1  List<Integer> lzw(String s) {
2    Map<String, Integer> dict =
3        initDict(s);  // A=0, B=1, C=2
4    String w = "";
5    List<Integer> codes = new ArrayList<>
   ();
6    for (char c : s.toCharArray()) {
7      String wc = w + c;
8      if (dict.containsKey(wc)) {
9        w = wc;
10     } else {
▶        codes.add(dict.get(w));
▶        dict.put(wc, dict.size());
▶        w = "" + c;
14     }
15   }
16   if (!w.isEmpty())
17     codes.add(dict.get(w));
18   return codes;
19 }
```



"BA" not in dict → output code of w = "B": **1**; add "BA" = 4 to dict; w = "A".

Step 4 / 13

VARIABLES

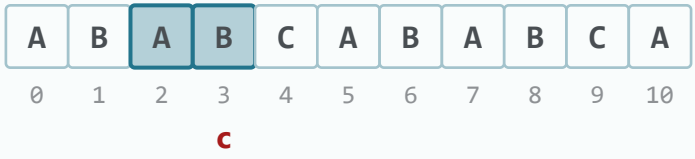
`c = A` `w = "A"` `wc = "BA"`

OUTPUT

```
0 1
```

lzw("ABABCABABCA")

```
1  List<Integer> lzw(String s) {
2    Map<String, Integer> dict =
3        initDict(s);  // A=0, B=1, C=2
4    String w = "";
5    List<Integer> codes = new ArrayList<>
     ();
6    for (char c : s.toCharArray()) {
▶      String wc = w + c;
▶      if (dict.containsKey(wc)) {
▶        w = wc;
10     } else {
11       codes.add(dict.get(w));
12       dict.put(wc, dict.size());
13       w = "" + c;
14     }
15   }
16   if (!w.isEmpty())
17     codes.add(dict.get(w));
18   return codes;
19 }
```



0	A
1	B
2	C
3	AB
4	BA

"AB" is in dict (3) → w = "AB"; look for a longer match.

Step 5 / 13

VARIABLES

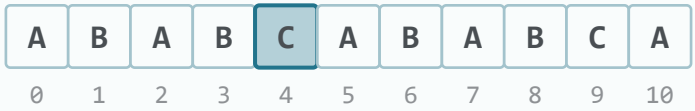
`c = B` `w = "AB"` `wc = "AB"`

OUTPUT

```
0 1
```

lzw("ABABCABABCA")

```
1  List<Integer> lzw(String s) {
2    Map<String, Integer> dict =
3        initDict(s);  // A=0, B=1, C=2
4    String w = "";
5    List<Integer> codes = new ArrayList<>
   ();
6    for (char c : s.toCharArray()) {
7      String wc = w + c;
8      if (dict.containsKey(wc)) {
9        w = wc;
10     } else {
▶        codes.add(dict.get(w));
▶        dict.put(wc, dict.size());
▶        w = "" + c;
14     }
15   }
16   if (!w.isEmpty())
17     codes.add(dict.get(w));
18   return codes;
19 }
```



0	A
1	B
2	C
3	AB
4	BA
5	ABC

"ABC" not in dict → output code of w = "AB": **3**; add "ABC" = 5 to dict; w = "C".

Step 6 / 13

VARIABLES

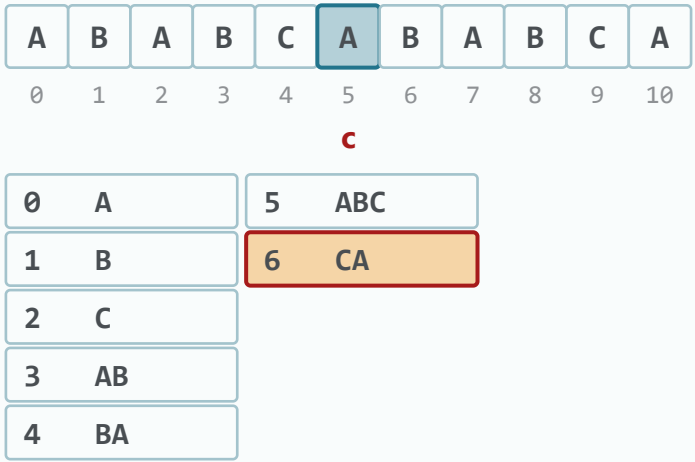
`c = C` `w = "C"` `wc = "ABC"`

OUTPUT

```
0 1 3
```

lzw("ABABCABABCA")

```
1  List<Integer> lzw(String s) {
2    Map<String, Integer> dict =
3        initDict(s);  // A=0, B=1, C=2
4    String w = "";
5    List<Integer> codes = new ArrayList<>
   ();
6    for (char c : s.toCharArray()) {
7      String wc = w + c;
8      if (dict.containsKey(wc)) {
9        w = wc;
10     } else {
▶        codes.add(dict.get(w));
▶        dict.put(wc, dict.size());
▶        w = "" + c;
14     }
15   }
16   if (!w.isEmpty())
17     codes.add(dict.get(w));
18   return codes;
19 }
```



"CA" not in dict → output code of w = "C": **2**; add "CA" = 6 to dict; w = "A".

Step 7 / 13

VARIABLES

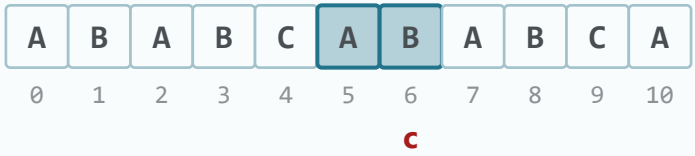
`c = A` `w = "A"` `wc = "CA"`

OUTPUT

```
0 1 3 2
```

lzw("ABABCABABCA")

```
1  List<Integer> lzw(String s) {
2    Map<String, Integer> dict =
3        initDict(s);  // A=0, B=1, C=2
4    String w = "";
5    List<Integer> codes = new ArrayList<>
   ();
6    for (char c : s.toCharArray()) {
▶      String wc = w + c;
▶      if (dict.containsKey(wc)) {
▶        w = wc;
10     } else {
11       codes.add(dict.get(w));
12       dict.put(wc, dict.size());
13       w = "" + c;
14     }
15   }
16   if (!w.isEmpty())
17     codes.add(dict.get(w));
18   return codes;
19 }
```



0	A	5	ABC
1	B	6	CA
2	C		
3	AB		
4	BA		

"AB" is in dict (3) → w = "AB"; look for a longer match.

Step 8 / 13

VARIABLES

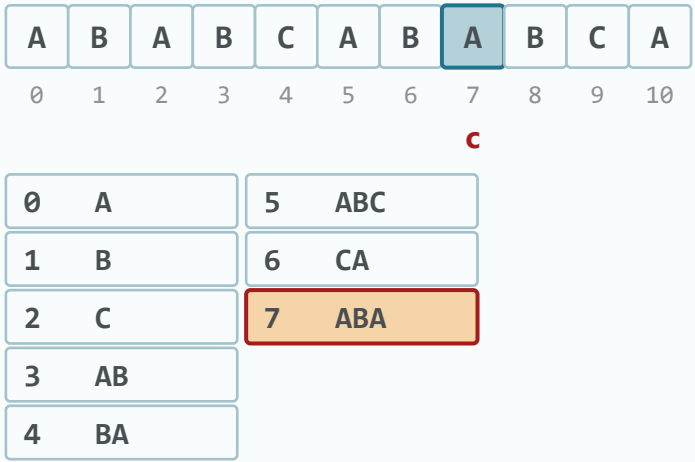
`c = B` `w = "AB"` `wc = "AB"`

OUTPUT

```
0 1 3 2
```

lzw("ABABCABABCA")

```
1  List<Integer> lzw(String s) {
2    Map<String, Integer> dict =
3        initDict(s);  // A=0, B=1, C=2
4    String w = "";
5    List<Integer> codes = new ArrayList<>
   ();
6    for (char c : s.toCharArray()) {
7      String wc = w + c;
8      if (dict.containsKey(wc)) {
9        w = wc;
10     } else {
▶        codes.add(dict.get(w));
▶        dict.put(wc, dict.size());
▶        w = "" + c;
14     }
15   }
16   if (!w.isEmpty())
17     codes.add(dict.get(w));
18   return codes;
19 }
```



"ABA" not in dict → output code of w = "AB": **3**; add "ABA" = 7 to dict; w = "A".

Step 9 / 13

VARIABLES

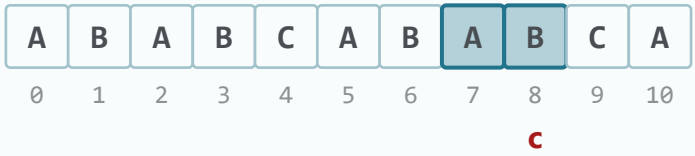
`c = A` `w = "A"` `wc = "ABA"`

OUTPUT

```
0 1 3 2 3
```

lzw("ABABCABABCA")

```
1  List<Integer> lzw(String s) {
2    Map<String, Integer> dict =
3        initDict(s);  // A=0, B=1, C=2
4    String w = "";
5    List<Integer> codes = new ArrayList<>
   ();
6    for (char c : s.toCharArray()) {
▶      String wc = w + c;
▶      if (dict.containsKey(wc)) {
▶        w = wc;
10     } else {
11       codes.add(dict.get(w));
12       dict.put(wc, dict.size());
13       w = "" + c;
14     }
15   }
16   if (!w.isEmpty())
17     codes.add(dict.get(w));
18   return codes;
19 }
```



0	A	5	ABC
1	B	6	CA
2	C	7	ABA
3	AB		
4	BA		

"AB" is in dict (3) → w = "AB"; look for a longer match.

Step 10 / 13

VARIABLES

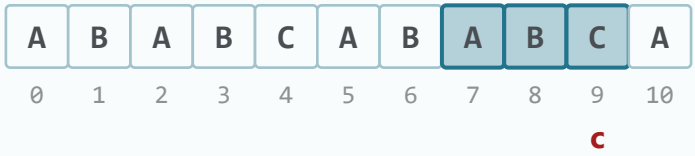
`c = B` `w = "AB"` `wc = "AB"`

OUTPUT

```
0 1 3 2 3
```

lzw("ABABCABABCA")

```
1  List<Integer> lzw(String s) {
2    Map<String, Integer> dict =
3        initDict(s);  // A=0, B=1, C=2
4    String w = "";
5    List<Integer> codes = new ArrayList<>
   ();
6    for (char c : s.toCharArray()) {
▶      String wc = w + c;
▶      if (dict.containsKey(wc)) {
▶        w = wc;
10     } else {
11       codes.add(dict.get(w));
12       dict.put(wc, dict.size());
13       w = "" + c;
14     }
15   }
16   if (!w.isEmpty())
17     codes.add(dict.get(w));
18   return codes;
19 }
```



0	A	5	ABC
1	B	6	CA
2	C	7	ABA
3	AB		
4	BA		

"ABC" is in dict (5) → w = "ABC"; look for a longer match.

Step 11 / 13

VARIABLES

`c = C` `w = "ABC"` `wc = "ABC"`

OUTPUT

```
0 1 3 2 3
```

lzw("ABABCABABCA")

```
1  List<Integer> lzw(String s) {
2    Map<String, Integer> dict =
3        initDict(s);  // A=0, B=1, C=2
4    String w = "";
5    List<Integer> codes = new ArrayList<>
   ();
6    for (char c : s.toCharArray()) {
7      String wc = w + c;
8      if (dict.containsKey(wc)) {
9        w = wc;
10     } else {
▶        codes.add(dict.get(w));
▶        dict.put(wc, dict.size());
▶        w = "" + c;
14     }
15   }
16   if (!w.isEmpty())
17     codes.add(dict.get(w));
18   return codes;
19 }
```

A	B	A	B	C	A	B	A	B	C	A
0	1	2	3	4	5	6	7	8	9	10
										c

0	A	5	ABC
1	B	6	CA
2	C	7	ABA
3	AB	8	ABCA
4	BA		

"ABCA" not in dict → output code of w = "ABC": **5**; add "ABCA" = 8 to dict; w = "A".

Step 12 / 13

VARIABLES

`c = A` `w = "A"` `wc = "ABCA"`

OUTPUT

```
0 1 3 2 3 5
```

lzw("ABABCABABCA")

```
1  List<Integer> lzw(String s) {
2    Map<String, Integer> dict =
3        initDict(s);  // A=0, B=1, C=2
4    String w = "";
5    List<Integer> codes = new ArrayList<>
   ();
6    for (char c : s.toCharArray()) {
7      String wc = w + c;
8      if (dict.containsKey(wc)) {
9        w = wc;
10     } else {
11       codes.add(dict.get(w));
12       dict.put(wc, dict.size());
13       w = "" + c;
14     }
15   }
▶  if (!w.isEmpty())
▶    codes.add(dict.get(w));
18   return codes;
19 }
```

A	B	A	B	C	A	B	A	B	C	A
0	1	2	3	4	5	6	7	8	9	10

c

0	A	5	ABC
1	B	6	CA
2	C	7	ABA
3	AB	8	ABCA
4	BA		

Loop done: output the code of the leftover w = "A", 0.
Output: **0 1 3 2 3 5 0**.

Step 13 / 13

VARIABLES

`c = A` `w = "A"` `wc = –`

OUTPUT

```
0 1 3 2 3 5 0
```

LZW Decompression

- The decompressor knows only the **initial dictionary**; it builds the rest itself, in the same order as the compressor.
- For each new code k: write e = dict[k]; add **w + e[0]** to the dictionary; set w = e.
- The decompressor lags **one step behind**. If k is not yet defined (k = the next code), e must be **w + w[0]** (the cScSc case).
- Output: A·B·AB·C·AB·ABC·A = **ABABCABABCA** ✓

Code	Output	Added
0	`A`	—
1	`B`	3 = AB
3	`AB`	4 = BA
2	`C`	5 = ABC
3	`AB`	6 = CA
5	`ABC`	7 = ABA
0	`A`	8 = ABCA

LZW Example: A DNA Sequence

Text: `ATGATCATGAG` (11 symbols)

Initial: **A = 0, T = 1, G = 2, C = 3**

Output: **0 1 2 4 3 4 6 2** (8 codes)

- Each row: the new string added to the dictionary and the code output at that moment.
- The second time "AT" appears, it is written as a single code (4); likewise "GA" (6).
- The final w = "G" is output after the loop (2).

Code	Added to dictionary	Output
4	`AT`	0
5	`TG`	1
6	`GA`	2
7	`ATC`	4
8	`CA`	3
9	`ATG`	4
10	`GAG`	6
—	(final w = G)	2

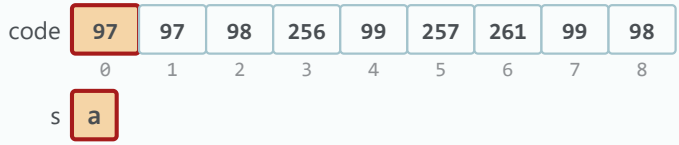
LZW with an ASCII Dictionary and the Special Case

- In practice the initial dictionary holds the 256 single-byte values (0–255; the first 128 are ASCII; a = 97, b = 98, c = 99); new strings are numbered from **256**.
- Text: `aabaacababacb` (13 characters) → 9 codes: **97 97 98 256 99 257 261 99 98**.
- "aba" = 261 is added while "ab" (257) is written and is used **right in the next code**.
- When the decompressor reads 261, its dictionary only goes up to 260 → **special case**: e = w + w[0] = "ab" + "a" = "aba".

Output	Added to dictionary
97	256 = `aa`
97	257 = `ab`
98	258 = `ba`
256	259 = `aac`
99	260 = `ca`
257	261 = `aba`
261	262 = `abac`
99	263 = `cb`
98	(final w = b)

lzwDecode([97 97 98 256 99 257 261 99 98])

```
1  String lzwDecode(List<Integer> codes) {
▶    List<String> dict =
▶        initDict();      // 0..255: byte
   values
▶    String w = dict.get(codes.get(0));
▶    StringBuilder s = new StringBuilder(w);
6    for (int i = 1; i < codes.size(); i++) {
7      int k = codes.get(i);
8      String e = k < dict.size()
9          ? dict.get(k)
10         : w + w.charAt(0); // special case
11     s.append(e);
12     dict.add(w + e.charAt(0));
13     w = e;
14   }
15   return s.toString();
16 }
```



dict 256 and up: still empty

The decompressor knows only the 256 single-byte values. The first code, 97, is always a single character: w = "a", output "a". Nothing from 256 up exists yet.

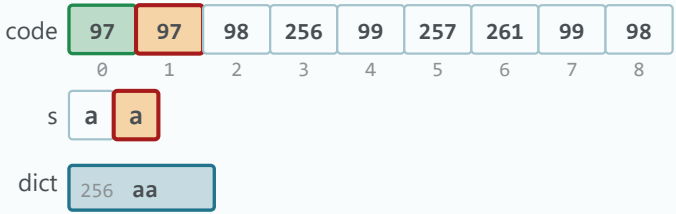
Step 1 / 10

VARIABLES

`i = –` `k = –` `w = "a"` `e = –`

lzwDecode([97 97 98 256 99 257 261 99 98])

```
1  String lzwDecode(List<Integer> codes) {
2    List<String> dict =
3        initDict();      // 0..255: byte
   values
4    String w = dict.get(codes.get(0));
5    StringBuilder s = new StringBuilder(w);
6    for (int i = 1; i < codes.size(); i++) {
       int k = codes.get(i);
       String e = k < dict.size()
           ? dict.get(k)
10         : w + w.charAt(0); // special case
       s.append(e);
       dict.add(w + e.charAt(0));
       w = e;
14   }
15   return s.toString();
16 }
```



k = 97 (ASCII) → write e = "a"; add w + e[0] = "a" + "a" = **256 = "aa"** to dict.

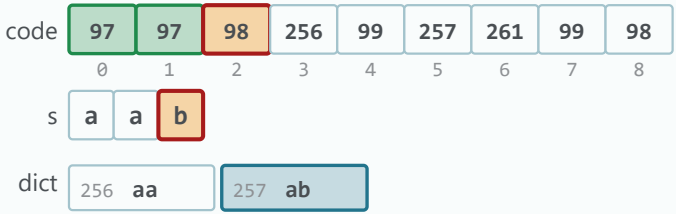
Step 2 / 10

VARIABLES

`i = 1` `k = 97` `w = "a"` `e = "a"`

lzwDecode([97 97 98 256 99 257 261 99 98])

```
1  String lzwDecode(List<Integer> codes) {
2    List<String> dict =
3        initDict();      // 0..255: byte
   values
4    String w = dict.get(codes.get(0));
5    StringBuilder s = new StringBuilder(w);
6    for (int i = 1; i < codes.size(); i++) {
▶      int k = codes.get(i);
▶      String e = k < dict.size()
▶          ? dict.get(k)
10         : w + w.charAt(0); // special case
▶      s.append(e);
▶      dict.add(w + e.charAt(0));
▶      w = e;
14   }
15   return s.toString();
16 }
```



k = 98 (ASCII) → write e = "b"; add w + e[0] = "a" + "b" = **257 = "ab"** to dict.

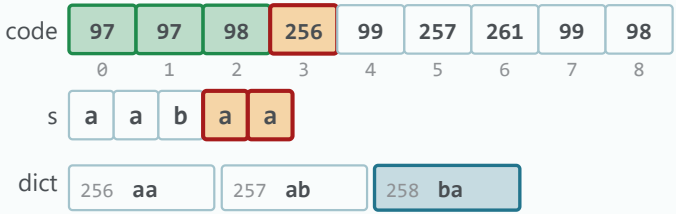
Step 3 / 10

VARIABLES

`i = 2` `k = 98` `w = "b"` `e = "b"`

lzwDecode([97 97 98 256 99 257 261 99 98])

```
1  String lzwDecode(List<Integer> codes) {
2    List<String> dict =
3        initDict();      // 0..255: byte
   values
4    String w = dict.get(codes.get(0));
5    StringBuilder s = new StringBuilder(w);
6    for (int i = 1; i < codes.size(); i++) {
▶      int k = codes.get(i);
▶      String e = k < dict.size()
▶          ? dict.get(k)
10         : w + w.charAt(0); // special case
▶      s.append(e);
▶      dict.add(w + e.charAt(0));
▶      w = e;
14   }
15   return s.toString();
16 }
```



k = 256 (in dict: 256 = "aa") → write e = "aa"; add w + e[0] = "b" + "a" = **258 = "ba"** to dict.

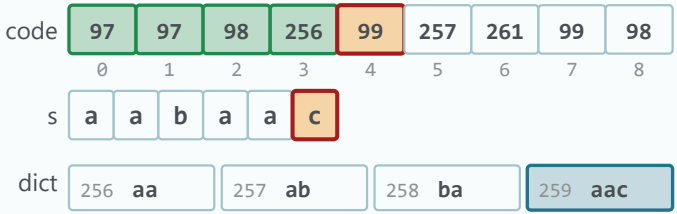
Step 4 / 10

VARIABLES

`i = 3` `k = 256` `w = "aa"` `e = "aa"`

lzwDecode([97 97 98 256 99 257 261 99 98])

```
1  String lzwDecode(List<Integer> codes) {
2    List<String> dict =
3        initDict();      // 0..255: byte
   values
4    String w = dict.get(codes.get(0));
5    StringBuilder s = new StringBuilder(w);
6    for (int i = 1; i < codes.size(); i++) {
       int k = codes.get(i);
       String e = k < dict.size()
           ? dict.get(k)
10         : w + w.charAt(0); // special case
       s.append(e);
       dict.add(w + e.charAt(0));
       w = e;
14   }
15   return s.toString();
16 }
```



k = 99 (ASCII) → write e = "c"; add w + e[0] = "aa" + "c" = **259 = "aac"** to dict.

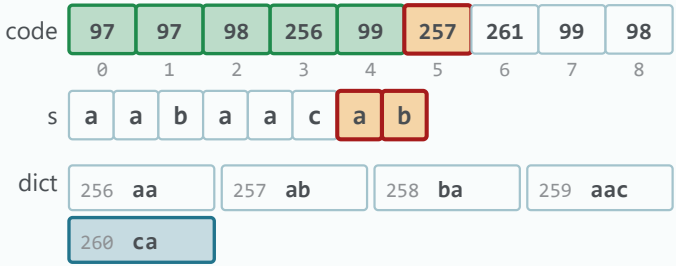
Step 5 / 10

VARIABLES

`i = 4` `k = 99` `w = "c"` `e = "c"`

lzwDecode([97 97 98 256 99 257 261 99 98])

```
1  String lzwDecode(List<Integer> codes) {
2    List<String> dict =
3        initDict();      // 0..255: byte
   values
4    String w = dict.get(codes.get(0));
5    StringBuilder s = new StringBuilder(w);
6    for (int i = 1; i < codes.size(); i++) {
       int k = codes.get(i);
       String e = k < dict.size()
           ? dict.get(k)
10         : w + w.charAt(0); // special case
       s.append(e);
       dict.add(w + e.charAt(0));
       w = e;
14   }
15   return s.toString();
16 }
```



k = 257 (in dict: 257 = "ab") → write e = "ab"; add w + e[0] = "c" + "a" = **260 = "ca"** to dict.

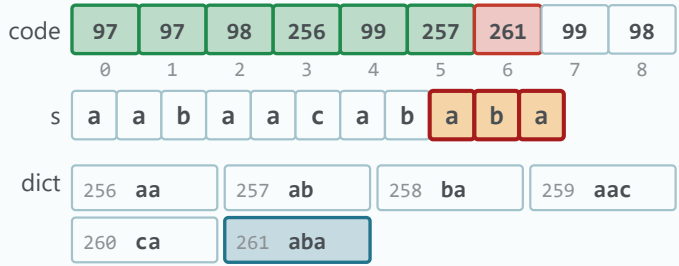
Step 6 / 10

VARIABLES

`i = 5` `k = 257` `w = "ab"` `e = "ab"`

lzwDecode([97 97 98 256 99 257 261 99 98])

```
1  String lzwDecode(List<Integer> codes) {
2    List<String> dict =
3        initDict();      // 0..255: byte
   values
4    String w = dict.get(codes.get(0));
5    StringBuilder s = new StringBuilder(w);
6    for (int i = 1; i < codes.size(); i++) {
       int k = codes.get(i);
       String e = k < dict.size()
9          ? dict.get(k)
           : w + w.charAt(0); // special case
       s.append(e);
       dict.add(w + e.charAt(0));
       w = e;
14   }
15   return s.toString();
16 }
```



k = 261, but the largest code in the dictionary is 260: **special case**. e = w + w[0] = "ab" + "a" = "aba"; 261 = "aba" is added too. The compressor used this code the moment it added it.

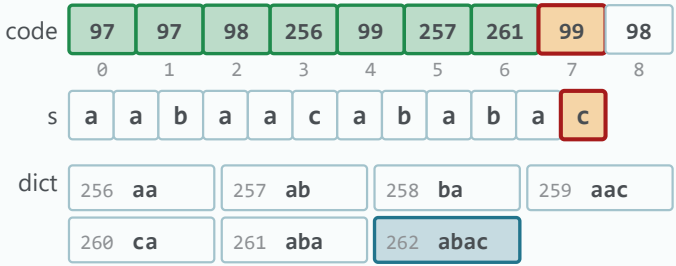
Step 7 / 10

VARIABLES

`i = 6` `k = 261` `w = "aba"` `e = "aba"`

lzwDecode([97 97 98 256 99 257 261 99 98])

```
1  String lzwDecode(List<Integer> codes) {
2    List<String> dict =
3        initDict();      // 0..255: byte
   values
4    String w = dict.get(codes.get(0));
5    StringBuilder s = new StringBuilder(w);
6    for (int i = 1; i < codes.size(); i++) {
       int k = codes.get(i);
       String e = k < dict.size()
           ? dict.get(k)
10         : w + w.charAt(0); // special case
       s.append(e);
       dict.add(w + e.charAt(0));
       w = e;
14   }
15   return s.toString();
16 }
```



k = 99 (ASCII) → write e = "c"; add w + e[0] = "aba" + "c" = **262 = "abac"** to dict.

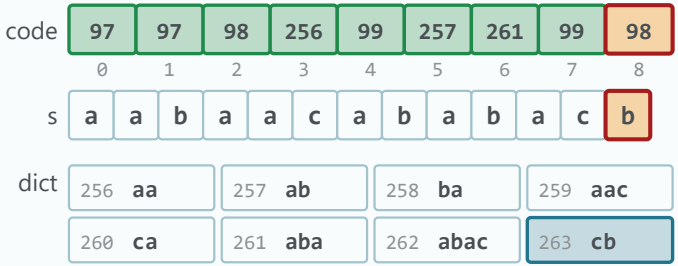
Step 8 / 10

VARIABLES

`i = 7` `k = 99` `w = "c"` `e = "c"`

lzwDecode([97 97 98 256 99 257 261 99 98])

```
1  String lzwDecode(List<Integer> codes) {
2    List<String> dict =
3        initDict();      // 0..255: byte
   values
4    String w = dict.get(codes.get(0));
5    StringBuilder s = new StringBuilder(w);
6    for (int i = 1; i < codes.size(); i++) {
▶      int k = codes.get(i);
▶      String e = k < dict.size()
▶          ? dict.get(k)
10         : w + w.charAt(0); // special case
▶      s.append(e);
▶      dict.add(w + e.charAt(0));
▶      w = e;
14   }
15   return s.toString();
16 }
```



k = 98 (ASCII) → write e = "b"; add w + e[0] = "c" + "b" = **263 = "cb"** to dict.

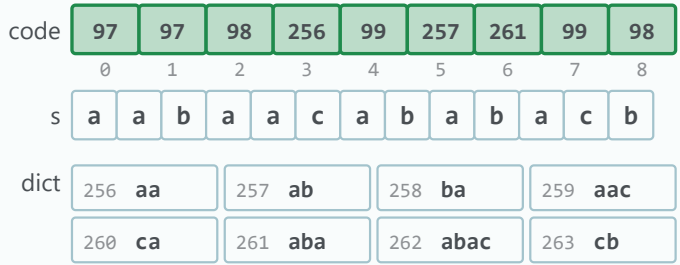
Step 9 / 10

VARIABLES

`i = 8` `k = 98` `w = "b"` `e = "b"`

lzwDecode([97 97 98 256 99 257 261 99 98])

```
1  String lzwDecode(List<Integer> codes) {
2    List<String> dict =
3        initDict();      // 0..255: byte
   values
4    String w = dict.get(codes.get(0));
5    StringBuilder s = new StringBuilder(w);
6    for (int i = 1; i < codes.size(); i++) {
7      int k = codes.get(i);
8      String e = k < dict.size()
9          ? dict.get(k)
10         : w + w.charAt(0); // special case
11     s.append(e);
12     dict.add(w + e.charAt(0));
13     w = e;
14   }
▶  return s.toString();
16 }
```



The 9 codes turned back into the 13-character text **aabaacababacb**; the decompressor's entries 256–263 match the compressor's.

Step 10 / 10

VARIABLES

`i = 9` `k = 98` `w = "b"` `e = "b"`

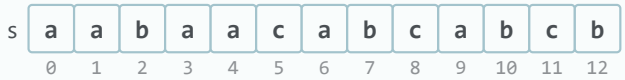
LZ78: The Ancestor of LZW

- LZ78 starts with an **empty** dictionary; its output is **(index, character)** pairs: the index of the longest dictionary match (0 if none) + the character that follows it.
- Text: `aabaacabcabcb` → (0,a) (1,b) (1,a) (0,c) (2,c) (5,b).
- Welch's change (LZW): start the dictionary with every single character → the output consists of **codes** only; no characters need to be written.
- The LZ77 (sliding window) family underlies gzip/ZIP (DEFLATE) and PNG.

Output	Dictionary
(0, a)	1 = `a`
(1, b)	2 = `ab`
(1, a)	3 = `aa`
(0, c)	4 = `c`
(2, c)	5 = `abc`
(5, b)	6 = `abcb`

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
▶    Map<String, Integer> dict =
▶        new HashMap<>();
▶    List<String> out = new ArrayList<>();
▶    String w = "";
6    for (char c : s.toCharArray()) {
7      if (dict.containsKey(w + c)) {
8        w = w + c;               // extend
9      } else {
10       int k = w.isEmpty() ? 0
11                           : dict.get(w);
12       out.add("(" + k + ", " + c + ")");
13       dict.put(w + c, dict.size() + 1);
14       w = "";
15     }
16   }
17   if (!w.isEmpty())         // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



dict empty (index 0 = empty string)

output

The LZ78 dictionary starts **empty** (unlike LZW). w holds the longest piece found in the dictionary so far; for now it is empty.

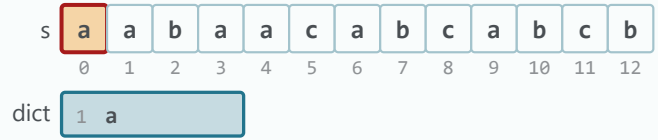
Step 1 / 14

VARIABLES

c = – w = "" k = –

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
2    Map<String, Integer> dict =
3        new HashMap<>();
4    List<String> out = new ArrayList<>();
5    String w = "";
6    for (char c : s.toCharArray()) {
▶      if (dict.containsKey(w + c)) {
8        w = w + c;               // extend
9      } else {
▶        int k = w.isEmpty() ? 0
▶                            : dict.get(w);
▶        out.add("(" + k + ", " + c + ")");
▶        dict.put(w + c, dict.size() + 1);
▶        w = "";
15     }
16   }
17   if (!w.isEmpty())          // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



output `(0,a)`

w is empty and 'a' is not in dict → **(0, a)** (0 = empty prefix); add 1 = "a".

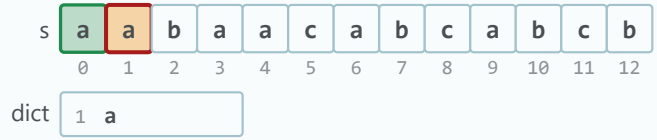
Step 2 / 14

VARIABLES

`c = 'a'` `w = ""` `k = 0`

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
2    Map<String, Integer> dict =
3        new HashMap<>();
4    List<String> out = new ArrayList<>();
5    String w = "";
6    for (char c : s.toCharArray()) {
▶      if (dict.containsKey(w + c)) {
▶        w = w + c;                // extend
9      } else {
10       int k = w.isEmpty() ? 0
11                           : dict.get(w);
12       out.add("(" + k + ", " + c + ")");
13       dict.put(w + c, dict.size() + 1);
14       w = "";
15     }
16   }
17   if (!w.isEmpty())          // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



output `(0,a)`

w + c = "a" is in dict → w = "a"; no output yet, try a longer piece with the next character.

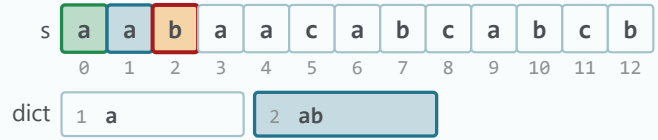
Step 3 / 14

VARIABLES

`c = 'a'` `w = "a"` `k = –`

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
2    Map<String, Integer> dict =
3        new HashMap<>();
4    List<String> out = new ArrayList<>();
5    String w = "";
6    for (char c : s.toCharArray()) {
       if (dict.containsKey(w + c)) {
8        w = w + c;               // extend
9      } else {
         int k = w.isEmpty() ? 0
                             : dict.get(w);
         out.add("(" + k + ", " + c + ")");
         dict.put(w + c, dict.size() + 1);
         w = "";
15     }
16   }
17   if (!w.isEmpty())          // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



output (0,a) (1,b)

"ab" not in dict → write w = "a" (index 1) and c = 'b': **(1, b)**; add 2 = "ab", reset w.

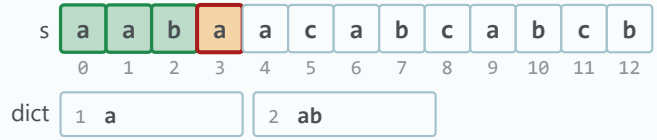
Step 4 / 14

VARIABLES

c = 'b' w = "" k = 1

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
2    Map<String, Integer> dict =
3        new HashMap<>();
4    List<String> out = new ArrayList<>();
5    String w = "";
6    for (char c : s.toCharArray()) {
▶      if (dict.containsKey(w + c)) {
▶        w = w + c;                // extend
9      } else {
10       int k = w.isEmpty() ? 0
11                           : dict.get(w);
12       out.add("(" + k + ", " + c + ")");
13       dict.put(w + c, dict.size() + 1);
14       w = "";
15     }
16   }
17   if (!w.isEmpty())          // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



output (0,a) (1,b)

w + c = "a" is in dict → w = "a"; no output yet, try a longer piece with the next character.

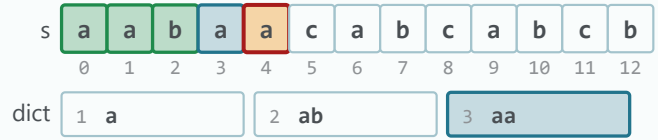
Step 5 / 14

VARIABLES

c = 'a' w = "a" k = –

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
2    Map<String, Integer> dict =
3        new HashMap<>();
4    List<String> out = new ArrayList<>();
5    String w = "";
6    for (char c : s.toCharArray()) {
       if (dict.containsKey(w + c)) {
8        w = w + c;               // extend
9      } else {
         int k = w.isEmpty() ? 0
                             : dict.get(w);
         out.add("(" + k + ", " + c + ")");
         dict.put(w + c, dict.size() + 1);
         w = "";
15     }
16   }
17   if (!w.isEmpty())          // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



output (0,a) (1,b) (1,a)

"aa" not in dict → write w = "a" (index 1) and c = 'a': **(1, a)**; add 3 = "aa", reset w.

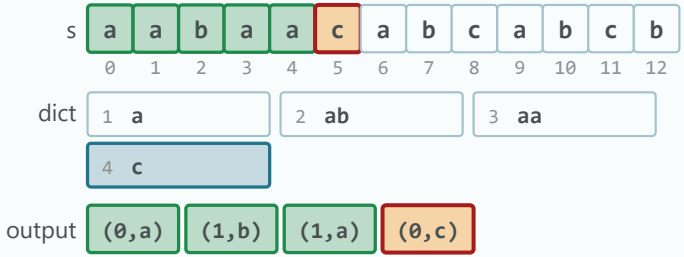
Step 6 / 14

VARIABLES

c = 'a' w = "" k = 1

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
2    Map<String, Integer> dict =
3        new HashMap<>();
4    List<String> out = new ArrayList<>();
5    String w = "";
6    for (char c : s.toCharArray()) {
       if (dict.containsKey(w + c)) {
8        w = w + c;               // extend
9      } else {
         int k = w.isEmpty() ? 0
                             : dict.get(w);
         out.add("(" + k + ", " + c + ")");
         dict.put(w + c, dict.size() + 1);
         w = "";
15     }
16   }
17   if (!w.isEmpty())          // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



w is empty and 'c' is not in dict → **(0, c)** (0 = empty prefix); add 4 = "c".

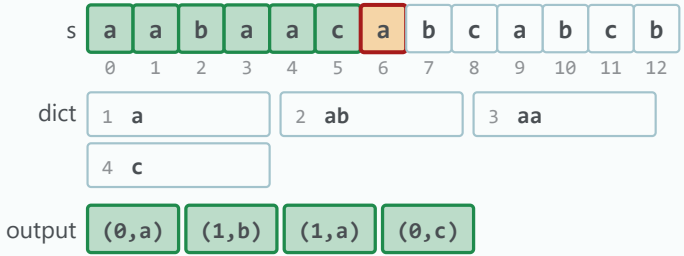
Step 7 / 14

VARIABLES

`c = 'c'` `w = ""` `k = 0`

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
2    Map<String, Integer> dict =
3        new HashMap<>();
4    List<String> out = new ArrayList<>();
5    String w = "";
6    for (char c : s.toCharArray()) {
▶      if (dict.containsKey(w + c)) {
▶        w = w + c;                // extend
9      } else {
10       int k = w.isEmpty() ? 0
11                           : dict.get(w);
12       out.add("(" + k + ", " + c + ")");
13       dict.put(w + c, dict.size() + 1);
14       w = "";
15     }
16   }
17   if (!w.isEmpty())          // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



w + c = "a" is in dict → w = "a"; no output yet, try a longer piece with the next character.

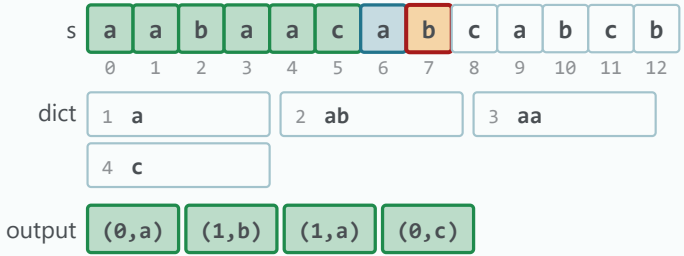
Step 8 / 14

VARIABLES

`c = 'a'` `w = "a"` `k = –`

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
2    Map<String, Integer> dict =
3        new HashMap<>();
4    List<String> out = new ArrayList<>();
5    String w = "";
6    for (char c : s.toCharArray()) {
▶      if (dict.containsKey(w + c)) {
▶        w = w + c;                // extend
9      } else {
10       int k = w.isEmpty() ? 0
11                           : dict.get(w);
12       out.add("(" + k + ", " + c + ")");
13       dict.put(w + c, dict.size() + 1);
14       w = "";
15     }
16   }
17   if (!w.isEmpty())          // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



w + c = "ab" is in dict → w = "ab"; no output yet, try a longer piece with the next character.

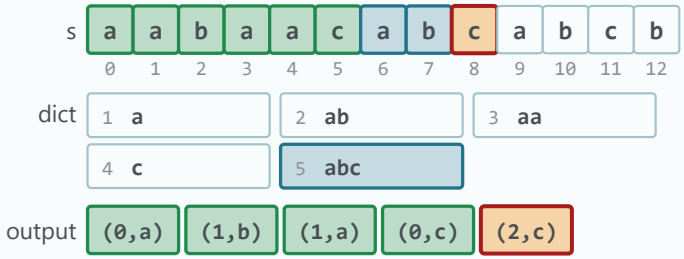
Step 9 / 14

VARIABLES

`c = 'b'` `w = "ab"` `k = –`

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
2    Map<String, Integer> dict =
3        new HashMap<>();
4    List<String> out = new ArrayList<>();
5    String w = "";
6    for (char c : s.toCharArray()) {
       if (dict.containsKey(w + c)) {
8        w = w + c;               // extend
9      } else {
         int k = w.isEmpty() ? 0
                             : dict.get(w);
         out.add("(" + k + ", " + c + ")");
         dict.put(w + c, dict.size() + 1);
         w = "";
15     }
16   }
17   if (!w.isEmpty())          // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



"abc" not in dict → write w = "ab" (index 2) and c = 'c': **(2, c)**; add 5 = "abc", reset w.

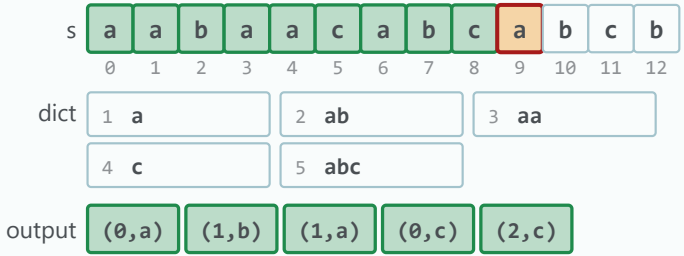
Step 10 / 14

VARIABLES

`c = 'c'` `w = ""` `k = 2`

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
2    Map<String, Integer> dict =
3        new HashMap<>();
4    List<String> out = new ArrayList<>();
5    String w = "";
6    for (char c : s.toCharArray()) {
▶      if (dict.containsKey(w + c)) {
▶        w = w + c;                  // extend
9      } else {
10       int k = w.isEmpty() ? 0
11                           : dict.get(w);
12       out.add("(" + k + ", " + c + ")");
13       dict.put(w + c, dict.size() + 1);
14       w = "";
15     }
16   }
17   if (!w.isEmpty())          // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



w + c = "a" is in dict → w = "a"; no output yet, try a longer piece with the next character.

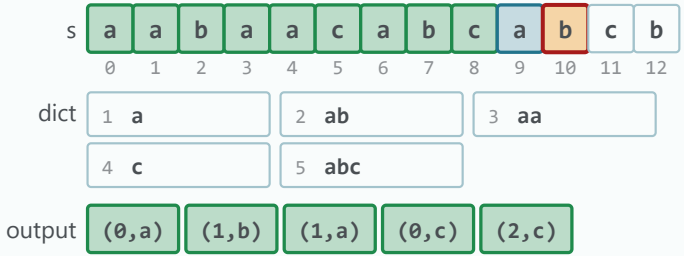
Step 11 / 14

VARIABLES

`c = 'a'` `w = "a"` `k = –`

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
2    Map<String, Integer> dict =
3        new HashMap<>();
4    List<String> out = new ArrayList<>();
5    String w = "";
6    for (char c : s.toCharArray()) {
▶      if (dict.containsKey(w + c)) {
▶        w = w + c;                // extend
9      } else {
10       int k = w.isEmpty() ? 0
11                           : dict.get(w);
12       out.add("(" + k + ", " + c + ")");
13       dict.put(w + c, dict.size() + 1);
14       w = "";
15     }
16   }
17   if (!w.isEmpty())          // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



w + c = "ab" is in dict → w = "ab"; no output yet, try a longer piece with the next character.

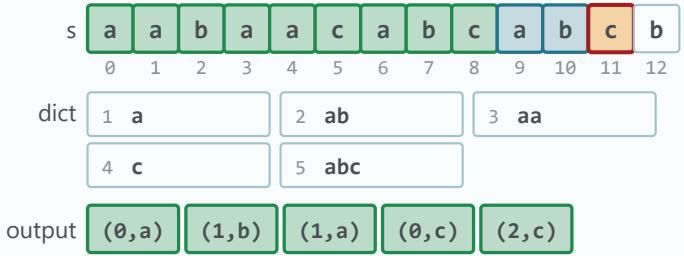
Step 12 / 14

VARIABLES

`c = 'b'` `w = "ab"` `k = –`

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
2    Map<String, Integer> dict =
3        new HashMap<>();
4    List<String> out = new ArrayList<>();
5    String w = "";
6    for (char c : s.toCharArray()) {
▶      if (dict.containsKey(w + c)) {
▶        w = w + c;                // extend
9      } else {
10       int k = w.isEmpty() ? 0
11                           : dict.get(w);
12       out.add("(" + k + ", " + c + ")");
13       dict.put(w + c, dict.size() + 1);
14       w = "";
15     }
16   }
17   if (!w.isEmpty())          // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



w + c = "abc" is in dict → w = "abc"; no output yet, try a longer piece with the next character.

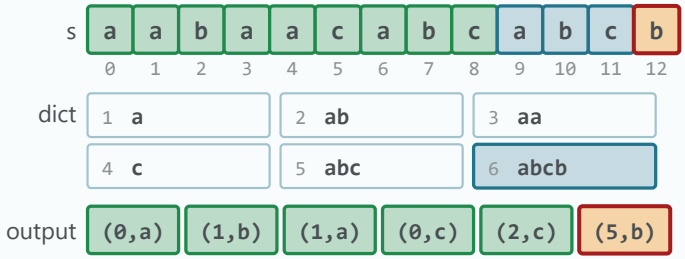
Step 13 / 14

VARIABLES

`c = 'c'` `w = "abc"` `k = –`

lz78("aabaacabcabcb")

```
1  List<String> lz78(String s) {
2    Map<String, Integer> dict =
3        new HashMap<>();
4    List<String> out = new ArrayList<>();
5    String w = "";
6    for (char c : s.toCharArray()) {
▶      if (dict.containsKey(w + c)) {
8        w = w + c;               // extend
9      } else {
▶        int k = w.isEmpty() ? 0
▶                            : dict.get(w);
▶        out.add("(" + k + ", " + c + ")");
▶        dict.put(w + c, dict.size() + 1);
▶        w = "";
15     }
16   }
17   if (!w.isEmpty())          // last piece
18     out.add("(" + dict.get(w) + ")");
19   return out;
20 }
```



"abcb" not in dict → write w = "abc" (index 5) and c = 'b': **(5, b)**; add 6 = "abcb", reset w. End of text: 6 pairs.

Step 14 / 14

VARIABLES

`c = 'b'` `w = ""` `k = 5`

RLE vs LZW

	RLE	LZW
Redundancy captured	consecutive repeats of one symbol	repeated **strings** (even if not consecutive)
Extra structure	none	dictionary (hash table / trie)
Time	O(n)	O(n) (O(1) average dictionary access with a trie or a (code, char)-keyed dictionary, not String concatenation)
Worst case	output doubles if nothing repeats	little gain on short/random data, code width grows
Used in	fax, BMP/PCX, simple graphics	GIF, TIFF, PDF, Unix `compress`

Both are **lossless**: decompression returns the original data exactly. No lossless method can shrink **every** input (pigeonhole principle).

Summary

Topic	Key points
Brute force	try every shift left to right; no preprocessing — worst case O(n · m)
KMP	lps[q] = longest proper prefix that is also a suffix; on a mismatch q = lps[q − 1], i never backs up — O(n + m)
Boyer-Moore	right to left; s += max(1, j − last[c]) plus good suffix; best O(n/m), fastest in practice
Rabin-Karp	polynomial rolling hash, O(1) update; verify on equal hashes; expected O(n + m)
RLE	run → (count, symbol); O(n); expands data without repeats; improved by a flag byte
LZW	if w + c is in the dictionary, extend; else output code(w) and add w + c; the decompressor rebuilds the dictionary (special case: w + w[0])

SUMMARY

Common Mistakes

Loop bound

Writing `i < n - m` in brute force skips the last shift (s = n − m); it should be `i <= n - m`.

Misdefining LPS

The pattern itself does not count: lps("aaaa") = [0, 1, 2, 3], not [1, 2, 3, 4]. On a KMP mismatch use **lps[q − 1]**, not lps[q].

Negative shift

In Boyer-Moore, j − last[c] can be negative (c right of j). Forget the `max(1, …)` and the pattern moves back or loops forever.

Trusting equal hashes

In Rabin-Karp, equal hashes do not mean a match: always verify. In Java, `%` can return a negative value: `if (th < 0) th += Q`.

Multi-digit counts in RLE

Decoding "12A" by reading the count as one digit is wrong (one '2'?). Bound the count or store it as a byte.

LZW's final w and special case

Forgetting to output the code of the leftover w after the loop; not handling a not-yet-defined code (w + w[0]) when decompressing.

SUMMARY

Sample Problems

1. Find the LPS table of the pattern "aabaaab".
2. How many comparisons does brute force make for T = "aaaaaaaa" (n = 8), P = "aab"?
3. Boyer-Moore (bad character only): T = "GCAATGCCTATGTGACC", P = "TATGTG". What is the first shift?
4. Base 10, Q = 13, characters are digits: T = "31415", m = 3. Given h("314"), use the rolling formula to find h("141").
5. Encode "WWWWBBBWWWWWWB" with RLE and compute the ratio.
6. Compress "ABABABA" with LZW using the dictionary A = 0, B = 1.

▶ **Show answers**

Exercises

1. Modify KMP search to return **all** occurrences in a list.
2. Find the last-occurrence table of the pattern "abcbcabbabcabc" and the first shift in the text "abccbaabccbaabcbcabbabcabc".
3. Design a Rabin-Karp method that searches for k patterns of the same length in a single pass.
4. Write `rleDecode(String s)`, an RLE decoder that also handles multi-digit counts correctly.
5. Decompress the codes 0 1 3 2 3 5 0 (A = 0, B = 1, C = 2) with LZW.
6. Give a pair T, P (n = 12, m = 4) that produces brute force's worst case; state the number of comparisons.

▶ **Answers / hints**

References

- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., 2022 (Chapter 32: String Matching).
- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011 (Section 5.3 Substring Search, 5.5 Data Compression).
- Goodrich, Tamassia and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014 (Chapter 13).
- D. E. Knuth, J. H. Morris, V. R. Pratt, "Fast Pattern Matching in Strings", *SIAM J. Computing* 6(2), 1977.
- R. S. Boyer, J S. Moore, "A Fast String Searching Algorithm", *Communications of the ACM* 20(10), 1977.
- R. M. Karp, M. O. Rabin, "Efficient Randomized Pattern-Matching Algorithms", *IBM J. Res. Dev.* 31(2), 1987.
- T. A. Welch, "A Technique for High-Performance Data Compression", *IEEE Computer* 17(6), 1984.

The End

Chapter 9: String Matching and Compression

Previous: Chapter 8: Network Flow · Next: Chapter 10: Advanced String Algorithms