Chapter 10: Advanced String Algorithms

Algorithms

Lexicographic order | Radix sort | Regular expressions | Finite state machines | Levenshtein distance | Similarity measures | Suffix arrays | Burrows–Wheeler transform

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_10_Advanced_String_Algorithms.html

Learning Outcomes

Sorting

Compare strings in lexicographic order; run LSD and MSD radix sort step by step and explain their $O(W \cdot (N + R))$ cost.

Parsing

Read and write regular expressions; design and draw a finite state machine (DFA) and run it on an input.

Similarity

Compute Levenshtein distance with a dynamic programming table and read the edit sequence back from it; compare it with Hamming, LCS and common-substring measures.

Transforms

Build a suffix array and use it for search; compute and invert the Burrows–Wheeler transform.

String Algorithms: Four Problem Families

Four families of algorithms that work on strings:

String Sorting

Lexicographic order: strings are arranged as in a dictionary by comparing character codes. **Radix sort**: groups strings without comparisons, treating each character as a digit.

Parsing

Regular expressions: strings that describe a search pattern. **Finite state machines** (FSM): automata that read the input one character at a time and change state.

Similarity

Levenshtein distance: the minimum number of single-character edits that turn one string into another. **LCS**: the longest common subsequence, in order but not necessarily contiguous.

Transformation

Suffix array: the starting indices of all suffixes in sorted order. **Burrows–Wheeler transform** (BWT): a reversible transform obtained by sorting rotations; the first stage of bzip2.

Lexicographic Order

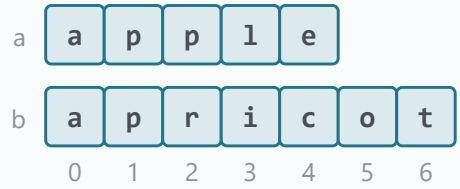
- **Lexicographic (alphabetical) order** arranges items by the **positions** of their characters in the alphabet or code table.
- Comparison starts with the **first characters**; if they are equal, move on to the **next** ones.
- The first **differing** character decides: "apple" < "banana" because 'a' < 'b'.
- If one string runs out, the **shorter one (the prefix) comes first**: "car" < "cart".

Comparison	Decided by
apple · banana	a < b (1st character)
apple · apricot	p < r (3rd character)
car · cart	"car" runs out
Zebra · apple	'Z' (90) < 'a' (97)

The computer's "alphabet" is the character code (ASCII / Unicode): every uppercase letter comes before every lowercase one, and ç, ğ, ı, ö, ş, ü come after z. For Turkish dictionary order use `java.text.Collator`.

Example: Comparing "apple" and "apricot"

```
▶ int compare(String a, String b) {
2   int n = Math.min(a.length(), b.length());
3   for (int k = 0; k < n; k++) {
4     char x = a.charAt(k), y = b.charAt(k);
5     if (x != y)
6       return x - y;   // first difference
7   }
8   return a.length() - b.length();
9 }
```



Call compare("apple", "apricot"). A negative result means a comes first, a positive one means b does (String.compareTo works the same way).

Step 1 / 12

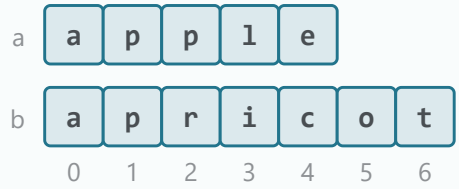
VARIABLES

a = "apple" b = "apricot" n = – k = – x = – y = –

OUTPUT

Example: Comparing "apple" and "apricot"

```
1 int compare(String a, String b) {
  int n = Math.min(a.length(), b.length());
3   for (int k = 0; k < n; k++) {
4     char x = a.charAt(k), y = b.charAt(k);
5     if (x != y)
6       return x - y;   // first difference
7   }
8   return a.length() - b.length();
9 }
```



At most as many characters as the shorter string can be compared: n = min(5, 7) = 5.

Step 2 / 12

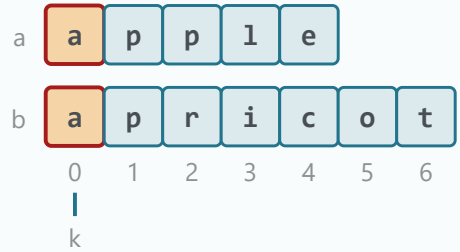
VARIABLES

a = "apple" | b = "apricot" | n = 5 | k = – | x = – | y = –

OUTPUT

Example: Comparing "apple" and "apricot"

```
1 int compare(String a, String b) {
2   int n = Math.min(a.length(), b.length());
▶   for (int k = 0; k < n; k++) {
4     char x = a.charAt(k), y = b.charAt(k);
5     if (x != y)
6       return x - y;   // first difference
7   }
8   return a.length() - b.length();
9 }
```



k = 0 < n: the loop continues with character 1.

Step 3 / 12

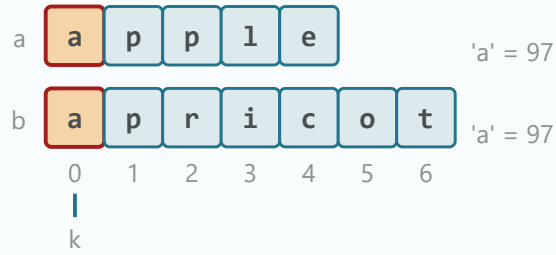
VARIABLES

a = "apple" | b = "apricot" | n = 5 | k = 0 | x = – | y = –

OUTPUT

Example: Comparing "apple" and "apricot"

```
1 int compare(String a, String b) {
2   int n = Math.min(a.length(), b.length());
3   for (int k = 0; k < n; k++) {
      char x = a.charAt(k), y = b.charAt(k);
5     if (x != y)
6       return x - y;   // first difference
7   }
8   return a.length() - b.length();
9 }
```



Read x = 'a' (97), y = 'a' (97).

Step 4 / 12

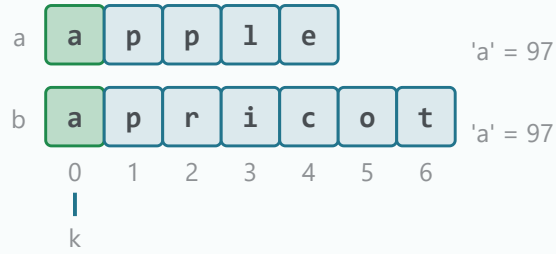
VARIABLES

a = "apple" | b = "apricot" | n = 5 | k = 0 | x = 'a' | y = 'a'

OUTPUT

Example: Comparing "apple" and "apricot"

```
1 int compare(String a, String b) {
2   int n = Math.min(a.length(), b.length());
3   for (int k = 0; k < n; k++) {
4     char x = a.charAt(k), y = b.charAt(k);
▶     if (x != y)
6       return x - y;   // first difference
7   }
8   return a.length() - b.length();
9 }
```



'a' == 'a': equal characters decide nothing; move on to the next one.

Step 5 / 12

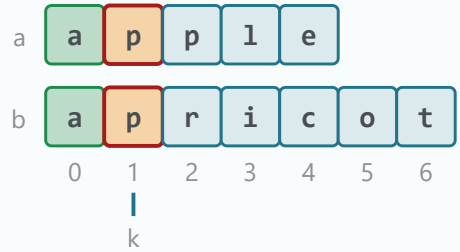
VARIABLES

a = "apple" | b = "apricot" | n = 5 | k = 0 | x = 'a' | y = 'a'

OUTPUT

Example: Comparing "apple" and "apricot"

```
1 int compare(String a, String b) {
2   int n = Math.min(a.length(), b.length());
▶   for (int k = 0; k < n; k++) {
4     char x = a.charAt(k), y = b.charAt(k);
5     if (x != y)
6       return x - y;   // first difference
7   }
8   return a.length() - b.length();
9 }
```



k = 1 < n: the loop continues with character 2.

Step 6 / 12

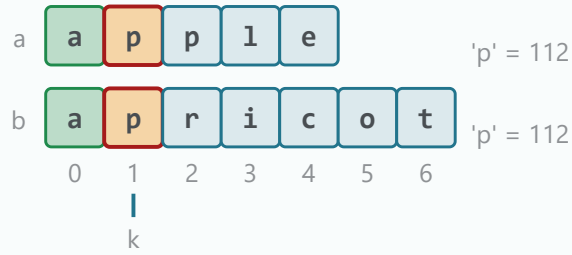
VARIABLES

a = "apple" | b = "apricot" | n = 5 | k = 1 | x = 'a' | y = 'a'

OUTPUT

Example: Comparing "apple" and "apricot"

```
1 int compare(String a, String b) {
2   int n = Math.min(a.length(), b.length());
3   for (int k = 0; k < n; k++) {
      char x = a.charAt(k), y = b.charAt(k);
5     if (x != y)
6       return x - y;   // first difference
7   }
8   return a.length() - b.length();
9 }
```



Read x = 'p' (112), y = 'p' (112).

Step 7 / 12

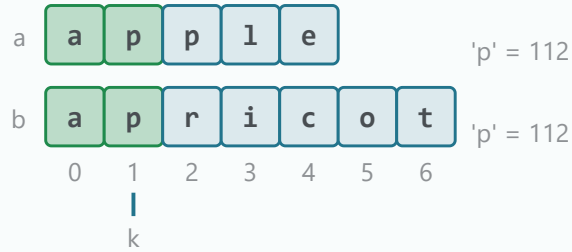
VARIABLES

a = "apple" | b = "apricot" | n = 5 | k = 1 | x = 'p' | y = 'p'

OUTPUT

Example: Comparing "apple" and "apricot"

```
1 int compare(String a, String b) {
2   int n = Math.min(a.length(), b.length());
3   for (int k = 0; k < n; k++) {
4     char x = a.charAt(k), y = b.charAt(k);
▶     if (x != y)
6       return x - y;   // first difference
7   }
8   return a.length() - b.length();
9 }
```



'p' == 'p': equal characters decide nothing; move on to the next one.

Step 8 / 12

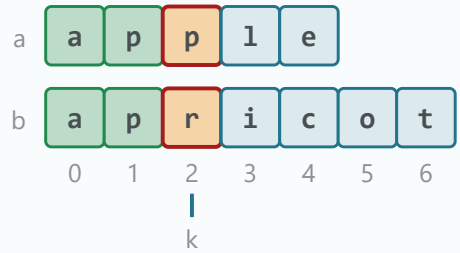
VARIABLES

a = "apple" | b = "apricot" | n = 5 | k = 1 | x = 'p' | y = 'p'

OUTPUT

Example: Comparing "apple" and "apricot"

```
1 int compare(String a, String b) {
2   int n = Math.min(a.length(), b.length());
▶   for (int k = 0; k < n; k++) {
4     char x = a.charAt(k), y = b.charAt(k);
5     if (x != y)
6       return x - y;   // first difference
7   }
8   return a.length() - b.length();
9 }
```



k = 2 < n: the loop continues with character 3.

Step 9 / 12

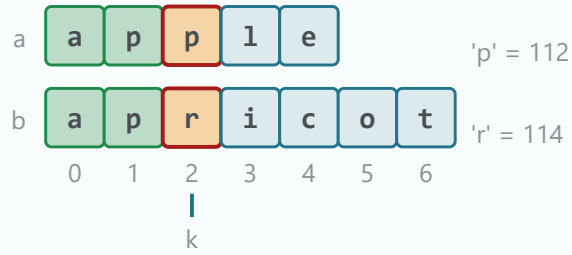
VARIABLES

a = "apple" | b = "apricot" | n = 5 | k = 2 | x = 'p' | y = 'p'

OUTPUT

Example: Comparing "apple" and "apricot"

```
1 int compare(String a, String b) {
2   int n = Math.min(a.length(), b.length());
3   for (int k = 0; k < n; k++) {
▶     char x = a.charAt(k), y = b.charAt(k);
5     if (x != y)
6       return x - y;   // first difference
7   }
8   return a.length() - b.length();
9 }
```



Read x = 'p' (112), y = 'r' (114).

Step 10 / 12

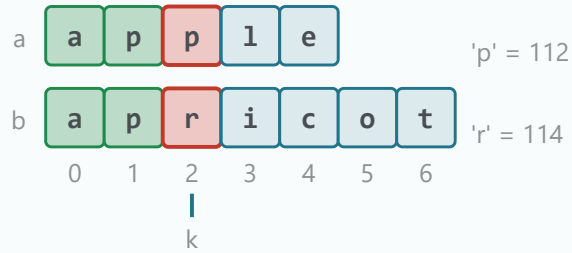
VARIABLES

a = "apple" | b = "apricot" | n = 5 | k = 2 | x = 'p' | y = 'r'

OUTPUT

Example: Comparing "apple" and "apricot"

```
1 int compare(String a, String b) {
2   int n = Math.min(a.length(), b.length());
3   for (int k = 0; k < n; k++) {
4     char x = a.charAt(k), y = b.charAt(k);
▶     if (x != y)
6       return x - y;   // first difference
7   }
8   return a.length() - b.length();
9 }
```



'p' ≠ 'r': first differing character found (character 3).

Step 11 / 12

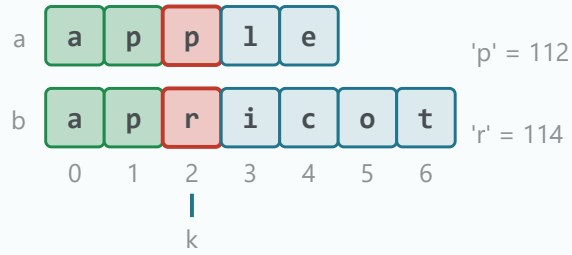
VARIABLES

a = "apple" | b = "apricot" | n = 5 | k = 2 | x = 'p' | y = 'r'

OUTPUT

Example: Comparing "apple" and "apricot"

```
1 int compare(String a, String b) {
2   int n = Math.min(a.length(), b.length());
3   for (int k = 0; k < n; k++) {
4     char x = a.charAt(k), y = b.charAt(k);
5     if (x != y)
▶       return x - y;   // first difference
7   }
8   return a.length() - b.length();
9 }
```



return 112 − 114 = **−2** < 0 → **apple comes before apricot**. The remaining characters are never examined.

Step 12 / 12

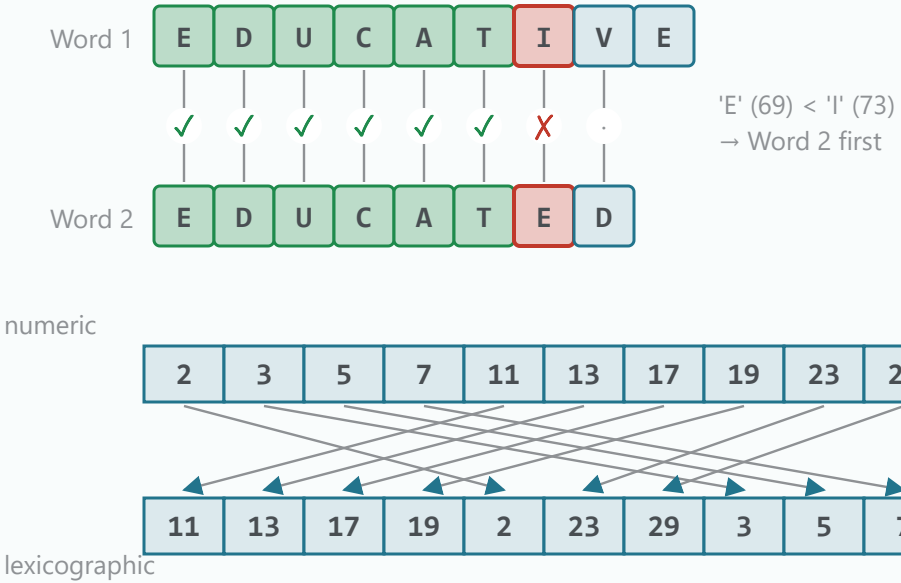
VARIABLES

`a = "apple"` `b = "apricot"` `n = 5` `k = 2` `x = 'p'` `y = 'r'`

OUTPUT

```
compare("apple", "apricot") → -2
```

Lexicographic Order: Words and Numbers



- Numbers sorted **as strings** are compared character by character from the left: "11" < "2" because '1' < '2'. Likewise, 123, 45, 6, 789 are already in lexicographic order.
- For numeric order, convert to integers or left-pad with zeros to equal length ("02" < "11").

Comparison-Based String Sorting

```
void lexSort(String[] a) {
  int n = a.length;
  for (int i = 0; i < n - 1; i++)
    for (int j = i + 1; j < n; j++)
      if (a[i].compareTo(a[j]) > 0) {
        String tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
      }
}
```

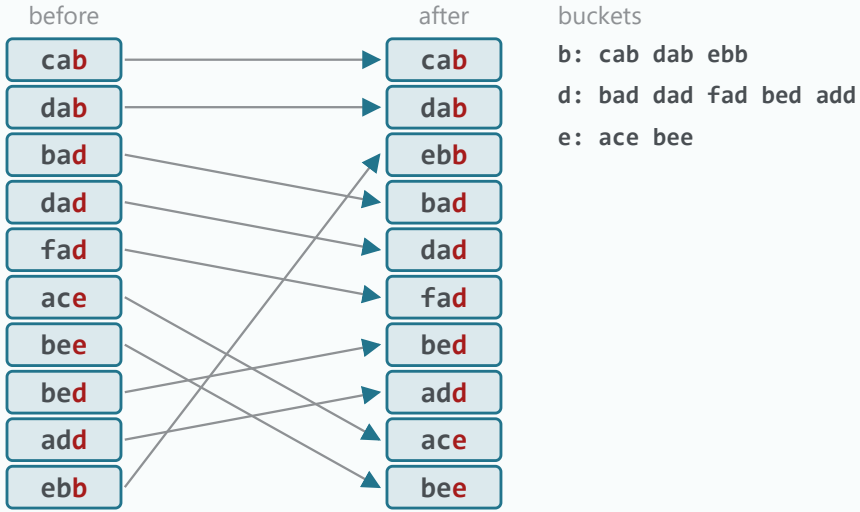
- Each position receives the lexicographically smallest of the remaining strings via swaps.
- `compareTo` reads at most W characters (W: longest string).
- Time: $O(N^2)$ comparisons × $O(W)$ = **$O(N^2 \cdot W)$**.
- With merge sort, $O(N \log N)$ comparisons → **$O(W \cdot N \log N)$**.

Input	Output
eabcdf, abcde, bcdea, cdeab, deabcg	abcde, bcdea, cdeab, deabcg, eabcdf
banana, orange, grape, apple, kiwi	apple, banana, grape, kiwi, orange

Comparison sorts need $\Omega(N \log N)$ comparisons. Radix sort makes no comparisons, so this lower bound does not apply to it.

LSD Radix Sort: Strings

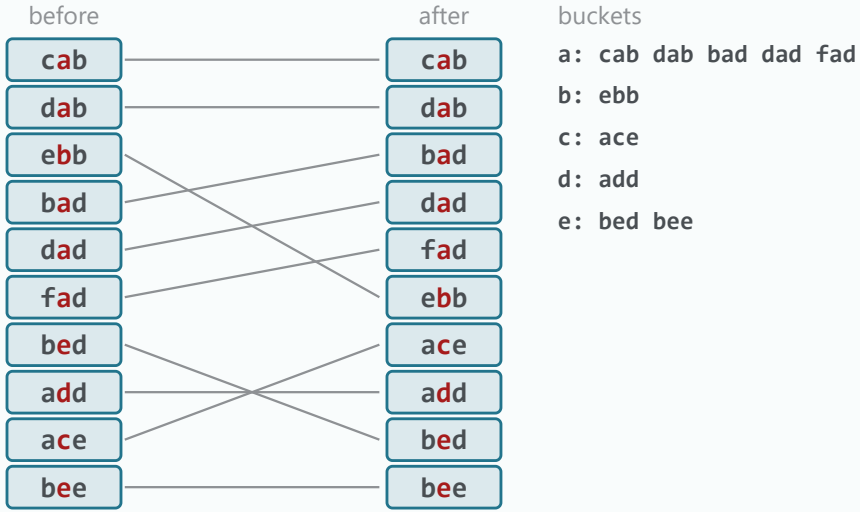
LSD: right to left, each pass a **stable** counting sort on that character (short strings get an empty character); equal keys keep the order of the previous pass. Example: **cab, dab, bad, dad, fad, ace, bee, bed, add, ebb** (W = 3).



Pass 1 (character 3): distribute into buckets by the last character; strings in the same bucket (cab, dab) keep their input order.

LSD Radix Sort: Strings

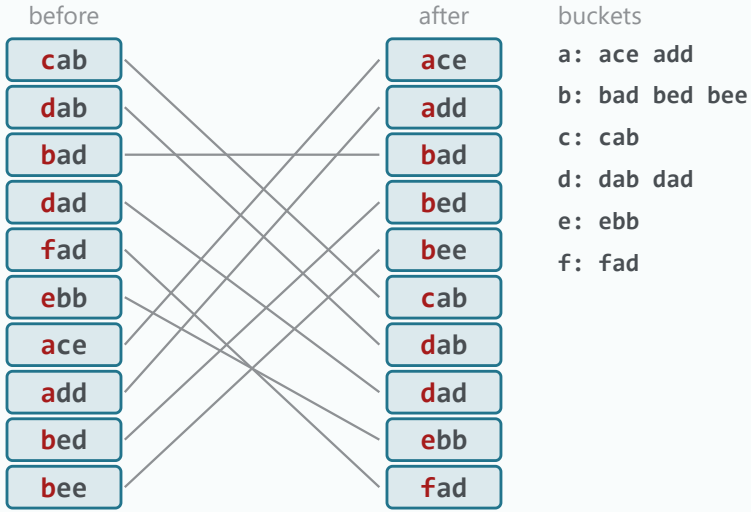
LSD: right to left, each pass a **stable** counting sort on that character (short strings get an empty character); equal keys keep the order of the previous pass. Example: **cab, dab, bad, dad, fad, ace, bee, bed, add, ebb** (W = 3).



Pass 2 (character 2): by the middle character; in bucket 'a', cab, dab, bad, dad, fad keep the previous order (b < d).

LSD Radix Sort: Strings

LSD: right to left, each pass a **stable** counting sort on that character (short strings get an empty character); equal keys keep the order of the previous pass. Example: **cab, dab, bad, dad, fad, ace, bee, bed, add, ebb** (W = 3).



Pass 3 (character 1): by the first character; the array is now in lexicographic order: ace, add, bad, bed, bee, cab, dab, dad, ebb, fad.

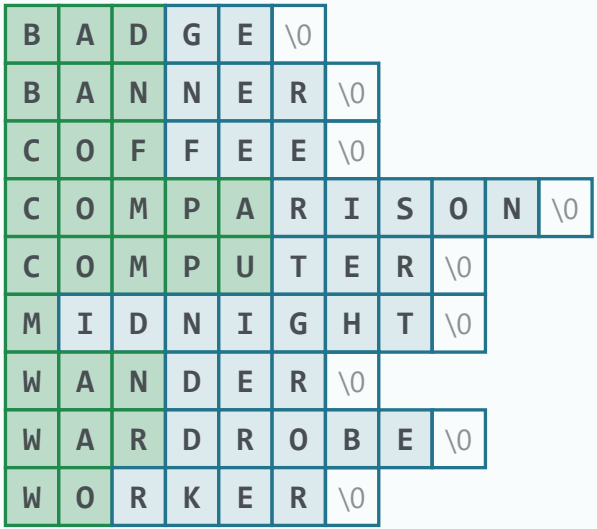
LSD Radix Sort: Java Code

```
void lsdSort(String[] a) {
  int W = maxLen(a);               // longest string
  for (int d = W - 1; d >= 0; d--)
    countSort(a, d);               // right to left
}
void countSort(String[] a, int d) {
  int[] cnt = new int[256];        // ASCII
  for (String s : a) cnt[ch(s, d)]++;
  for (int r = 1; r < 256; r++) cnt[r] += cnt[r - 1];
  String[] aux = new String[a.length];
  for (int i = a.length - 1; i >= 0; i--)   // stable
    aux[--cnt[ch(a[i], d)]] = a[i];
  System.arraycopy(aux, 0, a, 0, a.length);
}
int ch(String s, int d) {          // short string → 0
  return d < s.length() ? s.charAt(d) : 0;
}
```

- **Count:** `cnt[c]` is the number of strings whose d-th character is c.
- **Prefix sums:** `cnt[c]` now marks where bucket c **ends**.
- **Placement** runs from back to front, so equal keys keep their order (stability).
- `ch` returns 0 for a string that is too short: "car" comes before "cart".

One pass costs O(N + R), with alphabet size R = 256; W passes → O(W · (N + R)).

MSD Radix Sort: Left to Right



1. Split all strings into buckets by their **first character** (B, C, M, W).
2. Recursively sort each bucket on the **next character**.
3. Stop at a one-element bucket or at the end of a string (`\0`).

Green cells are the characters that **must be read** to fix the word's place (its distinguishing prefix). MSD examines only these: one character suffices for MIDNIGHT, five for COMPARISON / COMPUTER.

With long common prefixes MSD also reads every character. Since an R = 256 count array is expensive for small buckets, practical versions switch to insertion sort for small subarrays.

Time Complexity and Comparison

- LSD: W passes (W: length of the longest string), each pass over N strings + R buckets → **O(W · (N + R))**.
- With a fixed alphabet (R = 256) the cost is **O(W · N)**: longest string length × number of items.

Method	Time	Extra memory	Stable
Exchange sort + compareTo	$O(N^2 \cdot W)$	O(1)	no
Merge sort + compareTo	O(W · N log N)	O(N)	yes
LSD radix sort	O(W · (N + R))	O(N + R)	yes
MSD radix sort	O(N · W) worst	O(N + W · R)	yes

When to use radix sort? LSD is ideal for short, fixed-length keys (license plates, postal codes, IP addresses, 3-letter codes). When lengths vary widely, prefer MSD or a comparison sort.

Regular Expressions

- A **regular expression** (regex) is a string that describes a **search pattern**.
- Used to find, validate, extract or replace **every part** of a text that matches the pattern.
- It **automates** text processing: log analysis, form validation, data cleaning, lexical analysis (lexers) in compilers.
- Theory: every regular expression has an equivalent **finite state machine** (Kleene's theorem).

Pattern	Text	Matches
`colou?r`	color colour	color, colour
`gr(a|e)y`	gray grey groy	gray, grey
`\d+`	20.99 TL, 5 pcs	20, 99, 5
`\bcode\b`	code coding	only "code"

In Java: `java.util.regex` (`Pattern`, `Matcher`) and `String.matches`, `replaceAll`, `split`. Inside a Java string the backslash is doubled: `"\\d+"`.

Basic Operators and Concepts

Op.	Meaning	Example	Found in	Not in
`.`	Any single character (except newline)	`a.c`	abc, a7c	ac
`*`	Zero or more of the preceding item	`ab*c`	ac, abbc	adc
`+`	One or more of the preceding item	`ab+c`	abc, abbc	ac
`?`	Zero or one of the preceding item	`ab?c`	ac, abc	abbc
`[ ]`	One character from the set	`[ck]ar`	car, kar	bar
`^`	Start of the string	`^ab`	abc	cab
`$`	End of the string	`ab$`	cab	abc
`( )`	Grouping; defines a subexpression	`^(ab)+$`	ab, abab	aba

The columns show whether the string contains a part that matches the pattern (`Matcher.find`). `*` , `+` , `?` apply only to the item right before them (a character, set or group).

More Operators and Special Characters

Expression	Meaning
`a`	a single 'a' character
`abc`	exactly "abc"
`{ }`	a given number of repeats: `\d{4}`
`|`	alternatives: `cat|dog`
`\b`	word boundary (start or end)
`\d`	a digit: [0-9]
`\w`	a **word character**: [a-zA-Z0-9_]
`\s`	whitespace: space, tab, newline

- Uppercase forms are **complements**: `\D` a non-digit, `\W` a non-word character, `\S` a non-whitespace character.
- A special character itself must be **escaped**: `\.` a dot, `\+` a plus, `\(` a parenthesis.
- `\w` matches a single character; write `\w+` for a word.

In Java, `\w` covers only ASCII letters by default: the 'ş' in "şeker" does not match. For Turkish text use `\p{L}` or `Pattern.UNICODE_CHARACTER_CLASS`.

Character Classes

Character classes

`[abc]`	'a', 'b' or 'c'
`[a-z]`	any lowercase letter
`[A-Z]`	any uppercase letter
`[0-9]`	any digit
`[a-zA-Z_]`	a letter or underscore

Negated character classes

`[^abc]`	any character **except** 'a', 'b', 'c'
`[^0-9]`	a non-digit character
`[^0-9]+`	"ab12cd" → "ab", "cd"

- `^` means "not" **inside a bracket, at its start**; outside, it marks the start of the string.
- Inside a set, `.`, `*`, `+` are not special: `[.+]` matches a dot or a plus. A hyphen (`-`) denotes a range; to match it literally, put it first or last.
- The range `[a-z]` is based on character codes; ç, ğ, ı, ö, ş, ü are **not** in it.

Repetition and Grouping

Quantifiers

`a*`	zero or more 'a'
`a+`	one or more 'a'
`a?`	zero or one 'a'
`a{3}`	exactly three 'a'
`a{2,4}`	two to four 'a'
`a{2,}`	at least two 'a'

Grouping

`(abc)`	exactly "abc"; captured as a group
`(a|b|c)`	'a', 'b' or 'c'; captured as a group
`(ab)+`	"ab", "abab", ...

In Java a captured group is read with `m.group(1)`.

Greedy vs lazy: `a{2,4}` takes the longest possible piece of "aaaaa" (aaaa). A `?` after the quantifier picks the shortest match: in "x", `"<.+?>"` first takes only "", not the whole text.

Lookahead and Lookbehind

Pattern	Name	Matches when	"ab ac ba"
`a(?=b)`	lookahead	'a' **followed by** 'b'	ab ac ba
`a(?!b)`	negative lookahead	'a' **not followed by** 'b'	ab ac ba
`(?<=b)a`	lookbehind	'a' **preceded by** 'b'	ab ac ba
`(?<!b)a`	negative lookbehind	'a' **not preceded by** 'b'	ab ac ba

- Lookaround **tests** a position and **consumes** no characters: the match of `a(?=b)` is just "a"; the 'b' is not part of it.
- Example: `^(?=.*\d)(?=.*[A-Z]).{8,}$` validates a password of at least 8 characters containing at least one digit and one uppercase letter.

Example: Validation Expressions

Purpose	Regular expression
Email	`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`
Phone	`^(\+\d{1,3}[- ]?)?\d{10}$`
URL	`^https?:\/\/[^\s/$.?#].[^\s]*$`

Email

`[…]+` user name, `@`, domain, `\.` a literal dot, an extension of at least 2 letters. ✓ ali.veli@giresun.edu.tr ✗ ali@tr ✗ a b@x.com

Phone

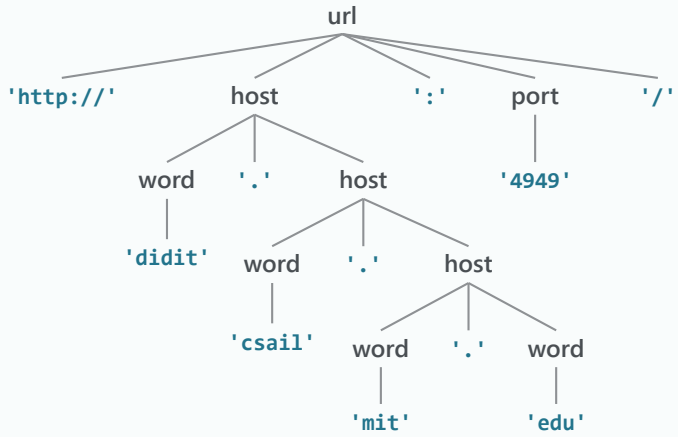
Optional `(…)?` country code: + and 1–3 digits, then a hyphen or space; then exactly 10 digits. ✓ +90 5321234567 ✓ 5321234567 ✗ 0532 123 45 67

URL

`s?` http or https; the first character cannot be whitespace, /, $, ., ?, #; then non-whitespace characters. ✓ https://giresun.edu.tr ✗ ftp://x.com ✗ "http:// x"

These expressions are practical approximations; they do not cover the full standards (RFC 5322 email, RFC 3986 URI).

Parse Tree: A URL Example



A grammar for `http://didit.csail.mit.edu:4949/` :

```
url  ::= 'http://' host
         ':' port '/'
host ::= word '.' host
       | word
word ::= [a-z]+
port ::= [0-9]+
```

The grammar looks recursive, but it is only tail recursion; the equivalent regex is `http://([a-z]+\.)*[a-z]+:[0-9]+/` . **Nested** structures such as balanced parentheses cannot be described by a regex; they need a context-free grammar and a parser.

Regular Expressions in Java: Pattern and Matcher

```
String input = "Name: Ali Veli, Age: 30, Email: ali.veli@example.com";
Pattern p = Pattern.compile("Age: (\\d+),");  // compile
Matcher m = p.matcher(input);
if (m.find()) {                           // first match
  int age = Integer.parseInt(m.group(1)); // group 1: "30"
  System.out.println("Age: " + age);
}
```

Pattern	group(1)	Explanation
`Name: (.*?),`	Ali Veli	lazy `.*?` : stops at the first comma
`Age: (\d+),`	30	one or more digits
`Email: (.*?)$`	ali.veli@example.com	up to the end of the line

- The same job can be done without regexes using `indexOf` and `substring` : find the start marker (e.g. "Age: ") and cut up to the end marker (",").
- `find()` tries to match **part** of the text, `matches()` the **whole** of it.

REGULAR EXPRESSIONS

Try It: Regex Tester

Pattern `\d+` ☐ case-insensitive

Text `Price 20.99 TL, qty 5, code A7`

\d+ | colou?r | email | a(?=b) | groups

Price 20.99 TL, qty 5, code A7

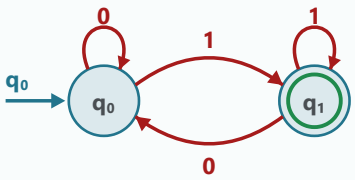
4 matches found.

Uses the browser's JavaScript regex engine; its syntax is largely the same as Java's. Matches are highlighted and groups are listed.

Finite State Machines

- A **finite state machine** (FSM) is a **mathematical model** that processes a string one character at a time.
- It has finitely many states; its only memory is the **current state**.
- Simple and easy to understand, yet it handles complex string analysis tasks (lexical analysis, protocols, pattern search).

Formal definition: **M = (Q, Σ, δ, q_0, F)**



Q = {q_0, q_1}, Σ = {0, 1}
F = {q_1} (double ring)
strings "ending in 1"

States Q
The distinct states the machine can be in.

Alphabet Σ
The set of input symbols.

Start state q_0
Where processing begins.

Transitions δ
δ(state, symbol) → new state.

Accept states F ⊆ Q
If the machine is in one of these when the string ends, the string is **accepted**.

Kinds of FSM: DFA, NFA, ε-NFA

DFA

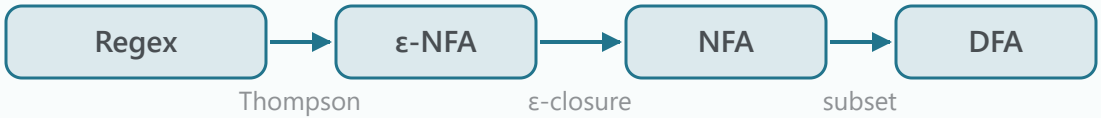
Deterministic finite automaton: every (state, symbol) pair has **exactly one** transition. Simple to run: O(n).

NFA

Nondeterministic finite automaton: a (state, symbol) pair may have **zero, one or several** transitions. The machine is in a set of states at once.

ε-NFA

An NFA that allows **empty transitions** (epsilon, ε) that read no symbol. Built directly from a regex (Thompson's construction).



- All three recognize **the same languages** (the regular languages): subset construction turns any NFA into a DFA, though the number of states can grow to 2^n in the worst case.
- grep, lex/flex and compiler lexers use this chain.

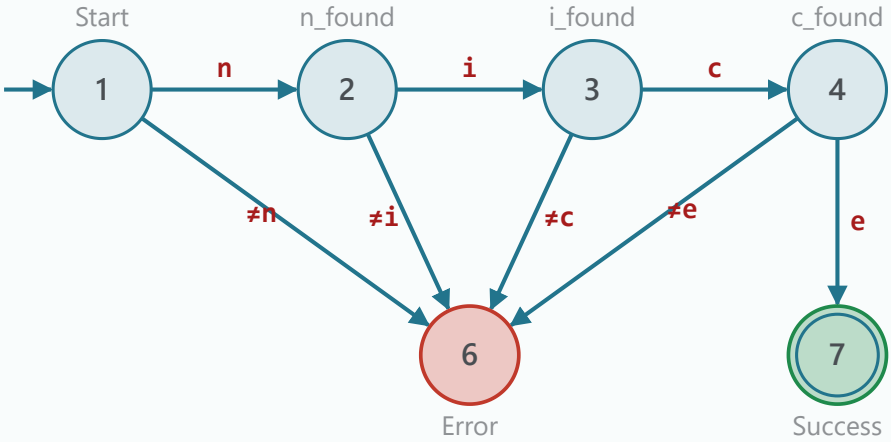
Using It: Simulating a DFA

1. Enter the **start** state.
2. For each character, update the state from the **transition table**.
3. When the string ends, check whether the final state is an **accept state**.

```
boolean accepts(int[][] delta, boolean[] accept,
                String s) {
  int state = 0;                    // start state
  for (char c : s.toCharArray())
    state = delta[state][c - '0'];  // one transition
  return accept[state];              // accepting?
}
```

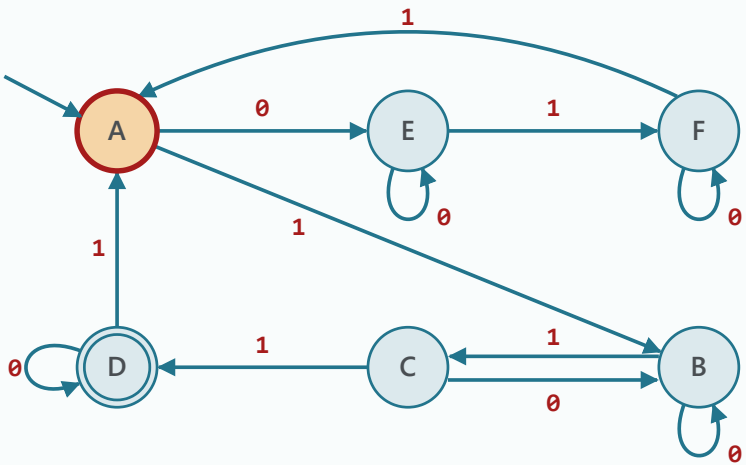
The transition table has size `int[|Q|][|Σ|]`. Each character costs one table lookup: a string of length n is processed in **O(n)** time, with no backtracking.

Example: A Machine That Recognizes "nice"



- A machine that recognizes "nice": each state stands for the **prefix read so far** (ε, n, ni, nic, nice).
- If the expected character does not arrive (≠n, ≠i, ...), the machine moves to **Error** and stays there (a trap state).
- "nice" → 1 → 2 → 3 → 4 → 7: accept. "nine" → 1 → 2 → 3 → 6: reject (the third character 'n' ≠ 'c').
- Any character after Success → Error ("nicer" is rejected); self-loops that are not drawn are implied.

Example: A DFA over a Binary Alphabet



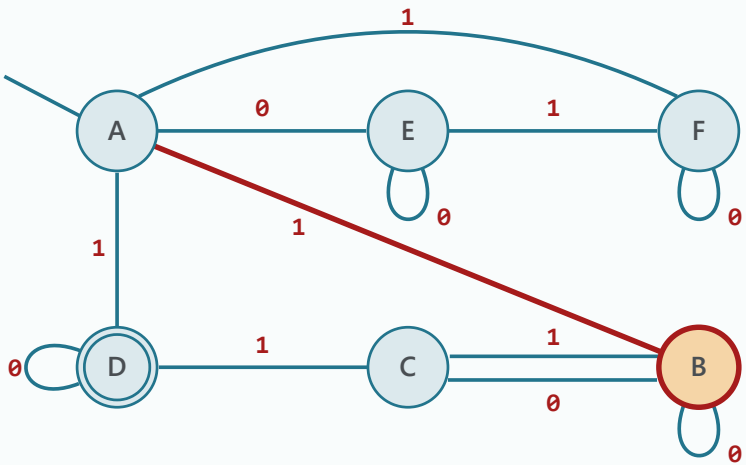
Start: state **A**. Input: **10011**.

Step 1 / 6

δ	0	1
→ A	E	B
B	B	C
C	B	D
D *	D	A
E	E	F
F	F	A

→ start, * accept. One state per cell: a **DFA**.

Example: A DFA over a Binary Alphabet

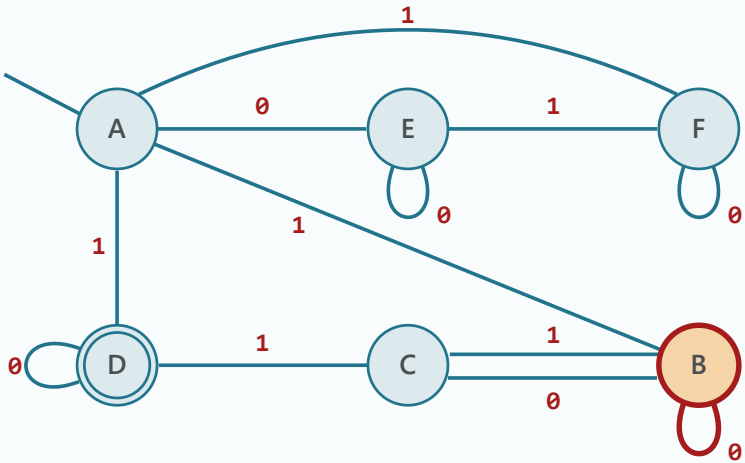


Symbol 1 is **1**: δ(A, 1) = **B**.

δ	0	1
→ A	E	B
B	B	C
C	B	D
D *	D	A
E	E	F
F	F	A

→ start, * accept. One state per cell: a **DFA**.

Example: A DFA over a Binary Alphabet

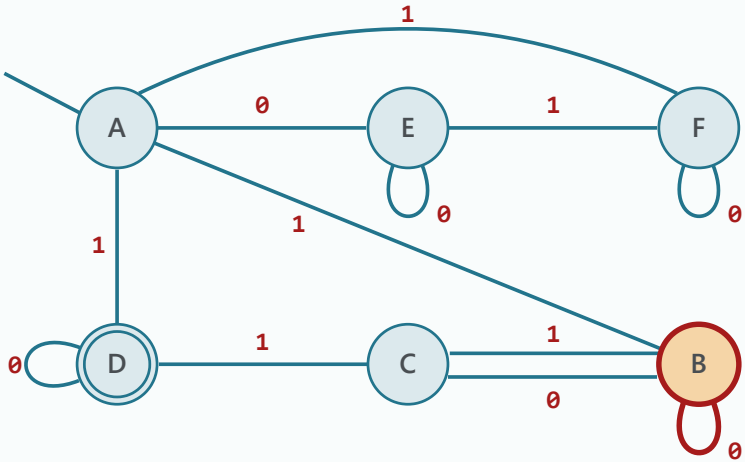


δ	0	1
→ A	E	B
B	B	C
C	B	D
D *	D	A
E	E	F
F	F	A

Symbol 2 is **0**: δ(B, 0) = **B**.

→ start, * accept. One state per cell: a **DFA**.

Example: A DFA over a Binary Alphabet

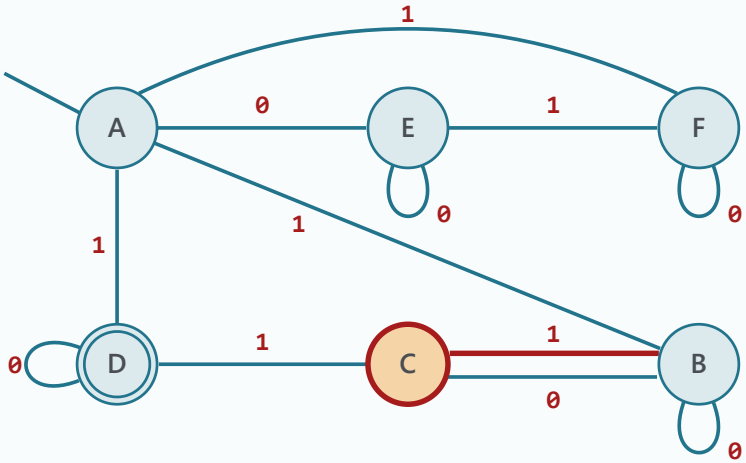


δ	0	1
→ A	E	B
B	B	C
C	B	D
D *	D	A
E	E	F
F	F	A

Symbol 3 is **0**: δ(B, 0) = **B**.

→ start, * accept. One state per cell: a **DFA**.

Example: A DFA over a Binary Alphabet

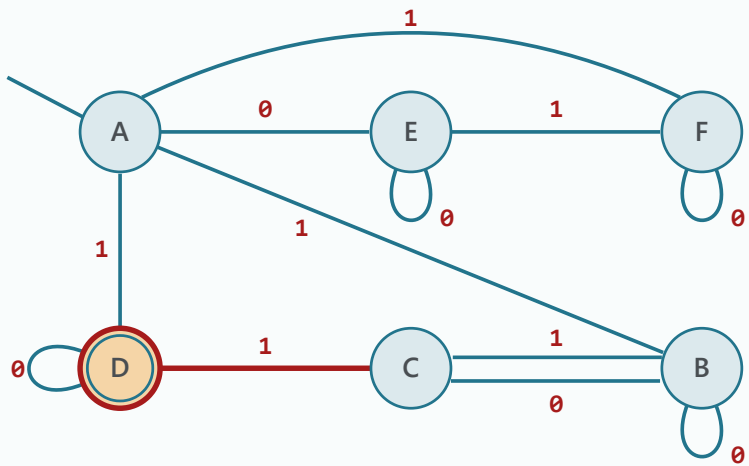


δ	0	1
→ A	E	B
B	B	C
C	B	D
D *	D	A
E	E	F
F	F	A

Symbol 4 is **1**: δ(B, 1) = **C**.

→ start, * accept. One state per cell: a **DFA**.

Example: A DFA over a Binary Alphabet

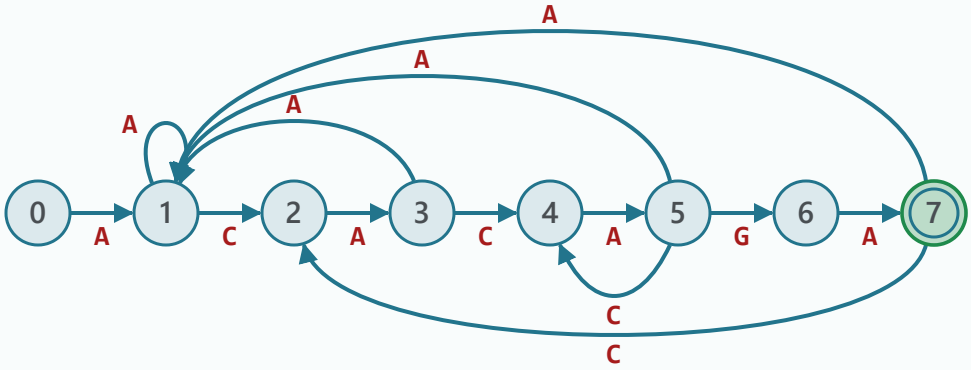


δ	0	1
→ A	E	B
B	B	C
C	B	D
D *	D	A
E	E	F
F	F	A

Symbol 5 is **1**: δ(C, 1) = **D**. Input exhausted; D is an accept state → **accept**.

→ start, * accept. One state per cell: a **DFA**.

Example: An Automaton That Searches for "ACACAGA"

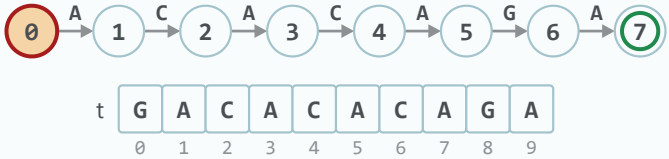


text	G	A	C	A	C	A	C	A	G	A
state	0	1	2	3	4	5	4	5	6	**7**

- State q = length of the **longest prefix** of the pattern that matches the end of the text read so far. All undrawn transitions go to 0. Reaching state 7 means a match (here at position 3).
- When the 7th character 'C' arrives, go back 5 → 4: the end of "ACACAC" can continue as the prefix "ACAC". This is the automaton form of KMP's prefix function: preprocessing $O(m \cdot |\Sigma|)$, search **O(n)**.

dfaSearch(dfa, "GACACACAGA", 7)

```
1 int dfaSearch(int[][] dfa, String t, int m)
  {
▶   int q = 0;                          // state
3   for (int i = 0; i < t.length(); i++) {
4     q = dfa[q][t.charAt(i)];
5     if (q == m)
6       return i - m + 1;               // match
7   }
8   return -1;
9 }
```



The automaton for ACACAGA is built in advance. State q = length of the longest pattern prefix that matches the end of the text. Only forward transitions are drawn here; the back transitions are on the previous slide. Initially q = 0.

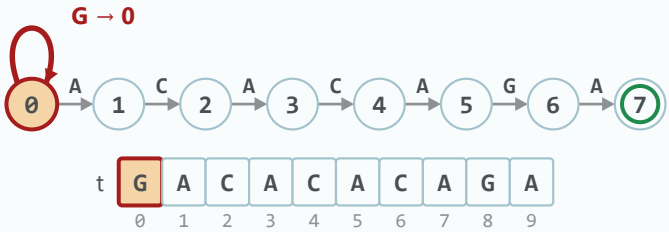
Step 1 / 11

VARIABLES

`m = 7` `i = –` `q = 0`

dfaSearch(dfa, "GACACACAGA", 7)

```
1 int dfaSearch(int[][] dfa, String t, int m)
  {
2   int q = 0;                          // state
    for (int i = 0; i < t.length(); i++) {
      q = dfa[q][t.charAt(i)];
      if (q == m)
6       return i - m + 1;               // match
7   }
8   return -1;
9 }
```



t[0] = 'G' extends no prefix of the pattern: q stays 0.

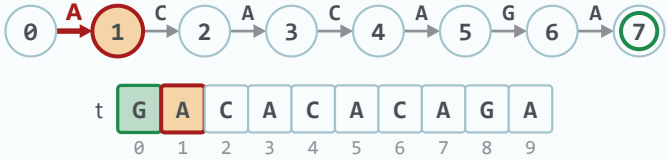
Step 2 / 11

VARIABLES

m = 7 | i = 0 | q = 0

dfaSearch(dfa, "GACACACAGA", 7)

```
1 int dfaSearch(int[][] dfa, String t, int m)
  {
2   int q = 0;                        // state
    for (int i = 0; i < t.length(); i++) {
      q = dfa[q][t.charAt(i)];
      if (q == m)
6       return i - m + 1;             // match
7   }
8   return -1;
9 }
```



t[1] = 'A' is the next pattern character: 0 → 1.

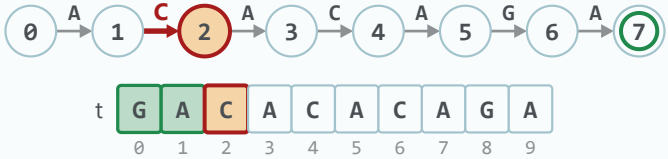
Step 3 / 11

VARIABLES

m = 7 i = 1 q = 1

dfaSearch(dfa, "GACACACAGA", 7)

```
1 int dfaSearch(int[][] dfa, String t, int m)
  {
2   int q = 0;                    // state
▶   for (int i = 0; i < t.length(); i++) {
▶     q = dfa[q][t.charAt(i)];
▶     if (q == m)
6       return i - m + 1;         // match
7   }
8   return -1;
9 }
```



t[2] = 'C' is the next pattern character: 1 → 2.

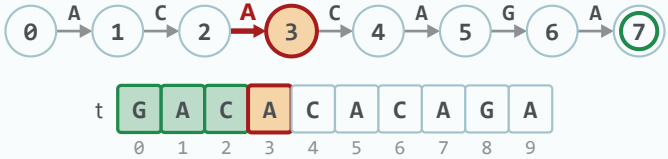
Step 4 / 11

VARIABLES

`m = 7` `i = 2` `q = 2`

dfaSearch(dfa, "GACACACAGA", 7)

```
1 int dfaSearch(int[][] dfa, String t, int m)
  {
2   int q = 0;                    // state
▶   for (int i = 0; i < t.length(); i++) {
▶     q = dfa[q][t.charAt(i)];
▶     if (q == m)
6       return i - m + 1;         // match
7   }
8   return -1;
9 }
```



t[3] = 'A' is the next pattern character: 2 → 3.

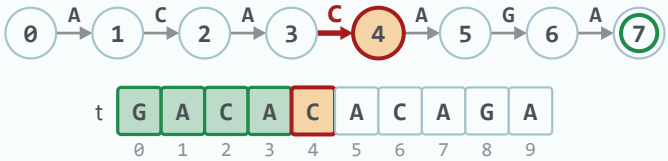
Step 5 / 11

VARIABLES

`m = 7` `i = 3` `q = 3`

dfaSearch(dfa, "GACACACAGA", 7)

```
1 int dfaSearch(int[][] dfa, String t, int m)
  {
2   int q = 0;                    // state
▶   for (int i = 0; i < t.length(); i++) {
▶     q = dfa[q][t.charAt(i)];
▶     if (q == m)
6       return i - m + 1;         // match
7   }
8   return -1;
9 }
```



t[4] = 'C' is the next pattern character: 3 → 4.

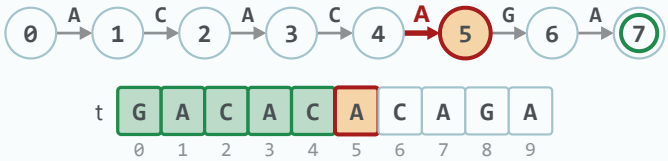
Step 6 / 11

VARIABLES

`m = 7` `i = 4` `q = 4`

dfaSearch(dfa, "GACACACAGA", 7)

```
1 int dfaSearch(int[][] dfa, String t, int m)
  {
2   int q = 0;                    // state
▶   for (int i = 0; i < t.length(); i++) {
▶     q = dfa[q][t.charAt(i)];
▶     if (q == m)
6       return i - m + 1;         // match
7   }
8   return -1;
9 }
```



t[5] = 'A' is the next pattern character: 4 → 5.

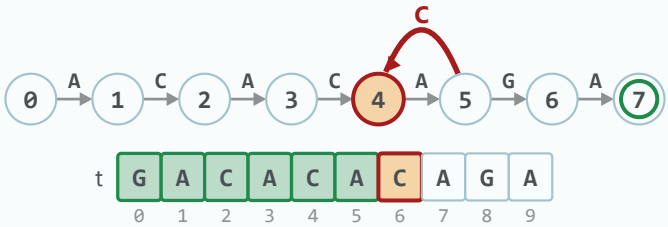
Step 7 / 11

VARIABLES

`m = 7` `i = 5` `q = 5`

dfaSearch(dfa, "GACACACAGA", 7)

```
1 int dfaSearch(int[][] dfa, String t, int m)
  {
2   int q = 0;                          // state
▶   for (int i = 0; i < t.length(); i++) {
▶     q = dfa[q][t.charAt(i)];
▶     if (q == m)
6       return i - m + 1;               // match
7   }
8   return -1;
9 }
```



t[6] = 'C' is not the expected 'G', but q is not reset: "ACACAC" ends with the prefix "ACAC" → 5 → **4**. No backing up in the text.

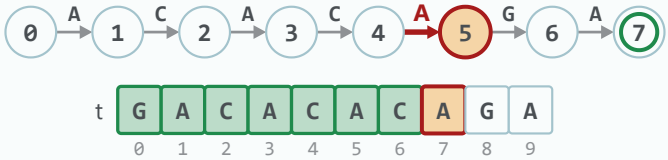
Step 8 / 11

VARIABLES

`m = 7` `i = 6` `q = 4`

dfaSearch(dfa, "GACACACAGA", 7)

```
1 int dfaSearch(int[][] dfa, String t, int m)
  {
2   int q = 0;                    // state
    for (int i = 0; i < t.length(); i++) {
      q = dfa[q][t.charAt(i)];
      if (q == m)
6       return i - m + 1;         // match
7   }
8   return -1;
9 }
```



t[7] = 'A' is the next pattern character: 4 → 5.

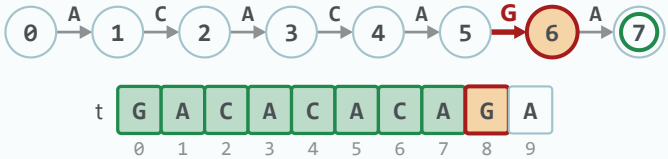
Step 9 / 11

VARIABLES

m = 7 | i = 7 | q = 5

dfaSearch(dfa, "GACACACAGA", 7)

```
1 int dfaSearch(int[][] dfa, String t, int m)
  {
2   int q = 0;                    // state
    for (int i = 0; i < t.length(); i++) {
      q = dfa[q][t.charAt(i)];
      if (q == m)
6       return i - m + 1;         // match
7   }
8   return -1;
9 }
```



t[8] = 'G' is the next pattern character: 5 → 6.

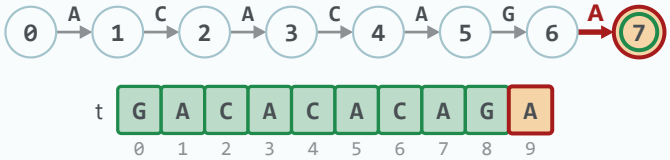
Step 10 / 11

VARIABLES

m = 7 | i = 8 | q = 6

dfaSearch(dfa, "GACACACAGA", 7)

```
1 int dfaSearch(int[][] dfa, String t, int m)
  {
2   int q = 0;                          // state
3   for (int i = 0; i < t.length(); i++) {
      q = dfa[q][t.charAt(i)];
      if (q == m)
        return i - m + 1;               // match
7   }
8   return -1;
9 }
```



'A' takes 6 → **7** = m: pattern found; return i − m + 1 = 9 − 7 + 1 = **3**.

Step 11 / 11

VARIABLES

m = **7** i = **9** q = **7**

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
▶    State d = State.START;
▶    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
5      boolean digit = Character.isDigit(c);
6      if (digit) {
7        if (d == State.START) d = State.DIGIT;
8        buf.append(c);
9      } else if (c == '.' && d != State.DECIMAL) {
10       d = State.DECIMAL;
11       buf.append(c);
12     } else if (d != State.START) {  // number
   ended
13       System.out.println(buf);
14       buf.setLength(0);
15       d = State.START;
16     }
17   }
18   if (d != State.START) System.out.println(buf);
19 }
```



START DIGIT DECIMAL

Input "$20.99 or 5.". State START, buffer empty.

Step 1 / 14

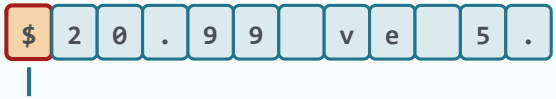
VARIABLES

`c = –` `digit = –` `d = START` `buf = ""`

OUTPUT

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
2    State d = State.START;
3    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
▶      boolean digit = Character.isDigit(c);
6      if (digit) {
7        if (d == State.START) d = State.DIGIT;
8        buf.append(c);
9      } else if (c == '.' && d != State.DECIMAL) {
10       d = State.DECIMAL;
11       buf.append(c);
12     } else if (d != State.START) {  // number
   ended
13       System.out.println(buf);
14       buf.setLength(0);
15       d = State.START;
16     }
17   }
18   if (d != State.START) System.out.println(buf);
19 }
```



START DIGIT DECIMAL

'$' is neither a digit nor a dot; state stays START, nothing happens.

Step 2 / 14

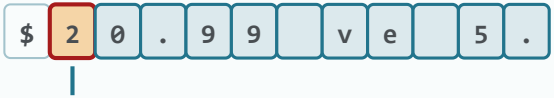
VARIABLES

`c = '$'` `digit = false` `d = START` `buf = ""`

OUTPUT

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
2    State d = State.START;
3    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
5      boolean digit = Character.isDigit(c);
▶      if (digit) {
▶        if (d == State.START) d = State.DIGIT;
▶        buf.append(c);
9      } else if (c == '.' && d != State.DECIMAL) {
10       d = State.DECIMAL;
11       buf.append(c);
12     } else if (d != State.START) {  // number
   ended
13       System.out.println(buf);
14       buf.setLength(0);
15       d = State.START;
16     }
17   }
18   if (d != State.START) System.out.println(buf);
19 }
```



START DIGIT DECIMAL

'2' is a digit and the state is START → **DIGIT**; appended to the buffer.

Step 3 / 14

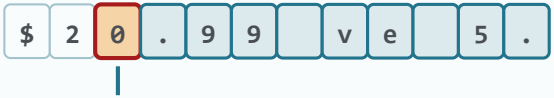
VARIABLES

`c = '2'` `digit = true` `d = DIGIT` `buf = "2"`

OUTPUT

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
2    State d = State.START;
3    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
5      boolean digit = Character.isDigit(c);
▶      if (digit) {
7        if (d == State.START) d = State.DIGIT;
▶        buf.append(c);
9      } else if (c == '.' && d != State.DECIMAL) {
10       d = State.DECIMAL;
11       buf.append(c);
12     } else if (d != State.START) {  // number
   ended
13       System.out.println(buf);
14       buf.setLength(0);
15       d = State.START;
16     }
17   }
18   if (d != State.START) System.out.println(buf);
19 }
```



START DIGIT DECIMAL

'0' is a digit: state stays DIGIT; append to the buffer.

Step 4 / 14

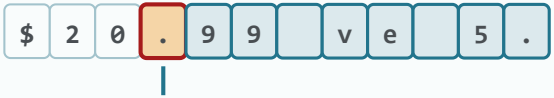
VARIABLES

`c = '0'` `digit = true` `d = DIGIT` `buf = "20"`

OUTPUT

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
2    State d = State.START;
3    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
5      boolean digit = Character.isDigit(c);
6      if (digit) {
7        if (d == State.START) d = State.DIGIT;
8        buf.append(c);
▶      } else if (c == '.' && d != State.DECIMAL) {
▶        d = State.DECIMAL;
▶        buf.append(c);
12     } else if (d != State.START) {  // number
   ended
13       System.out.println(buf);
14       buf.setLength(0);
15       d = State.START;
16     }
17   }
18   if (d != State.START) System.out.println(buf);
19 }
```



START DIGIT DECIMAL

'.' arrives → move to **DECIMAL**; the dot is appended to the buffer.

Step 5 / 14

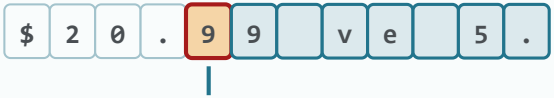
VARIABLES

`c = '.'` `digit = false` `d = DECIMAL` `buf = "20."`

OUTPUT

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
2    State d = State.START;
3    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
5      boolean digit = Character.isDigit(c);
▶      if (digit) {
7        if (d == State.START) d = State.DIGIT;
▶        buf.append(c);
9      } else if (c == '.' && d != State.DECIMAL) {
10       d = State.DECIMAL;
11       buf.append(c);
12     } else if (d != State.START) {  // number
   ended
13       System.out.println(buf);
14       buf.setLength(0);
15       d = State.START;
16     }
17   }
18   if (d != State.START) System.out.println(buf);
19 }
```



START DIGIT DECIMAL

'9' is a digit: state stays DECIMAL; append to the buffer.

Step 6 / 14

VARIABLES

`c = '9'` `digit = true` `d = DECIMAL` `buf = "20.9"`

OUTPUT

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
2    State d = State.START;
3    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
5      boolean digit = Character.isDigit(c);
▶      if (digit) {
7        if (d == State.START) d = State.DIGIT;
▶        buf.append(c);
9      } else if (c == '.' && d != State.DECIMAL) {
10       d = State.DECIMAL;
11       buf.append(c);
12     } else if (d != State.START) {  // number
   ended
13       System.out.println(buf);
14       buf.setLength(0);
15       d = State.START;
16     }
17   }
18   if (d != State.START) System.out.println(buf);
19 }
```



START DIGIT DECIMAL

'9' is a digit: state stays DECIMAL; append to the buffer.

Step 7 / 14

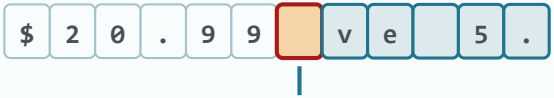
VARIABLES

`c = '9'` `digit = true` `d = DECIMAL` `buf = "20.99"`

OUTPUT

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
2    State d = State.START;
3    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
5      boolean digit = Character.isDigit(c);
6      if (digit) {
7        if (d == State.START) d = State.DIGIT;
8        buf.append(c);
9      } else if (c == '.' && d != State.DECIMAL) {
10       d = State.DECIMAL;
11       buf.append(c);
▶      } else if (d != State.START) {  // number ended
▶        System.out.println(buf);
▶        buf.setLength(0);
▶        d = State.START;
16     }
17   }
18   if (d != State.START) System.out.println(buf);
19 }
```



START DIGIT DECIMAL

' ' is not part of a number: the number ends, **20.99** is printed; back to START.

Step 8 / 14

VARIABLES

`c = ' '` `digit = false` `d = START` `buf = ""`

OUTPUT

```
20.99
```

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
2    State d = State.START;
3    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
▶      boolean digit = Character.isDigit(c);
6      if (digit) {
7        if (d == State.START) d = State.DIGIT;
8        buf.append(c);
9      } else if (c == '.' && d != State.DECIMAL) {
10       d = State.DECIMAL;
11       buf.append(c);
12     } else if (d != State.START) {  // number ended
13       System.out.println(buf);
14       buf.setLength(0);
15       d = State.START;
16     }
17   }
18   if (d != State.START) System.out.println(buf);
19 }
```



START DIGIT DECIMAL

'o' is neither a digit nor a dot; state stays START, nothing happens.

Step 9 / 14

VARIABLES

`c = 'o'` `digit = false` `d = START` `buf = ""`

OUTPUT

```
20.99
```

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
2    State d = State.START;
3    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
▶      boolean digit = Character.isDigit(c);
6      if (digit) {
7        if (d == State.START) d = State.DIGIT;
8        buf.append(c);
9      } else if (c == '.' && d != State.DECIMAL) {
10       d = State.DECIMAL;
11       buf.append(c);
12     } else if (d != State.START) {  // number
   ended
13       System.out.println(buf);
14       buf.setLength(0);
15       d = State.START;
16     }
17   }
18   if (d != State.START) System.out.println(buf);
19 }
```



START DIGIT DECIMAL

'r' is neither a digit nor a dot; state stays START, nothing happens.

Step 10 / 14

VARIABLES

`c = 'r'` `digit = false` `d = START` `buf = ""`

OUTPUT

```
20.99
```

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
2    State d = State.START;
3    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
▶      boolean digit = Character.isDigit(c);
6      if (digit) {
7        if (d == State.START) d = State.DIGIT;
8        buf.append(c);
9      } else if (c == '.' && d != State.DECIMAL) {
10       d = State.DECIMAL;
11       buf.append(c);
12     } else if (d != State.START) {  // number
   ended
13       System.out.println(buf);
14       buf.setLength(0);
15       d = State.START;
16     }
17   }
18   if (d != State.START) System.out.println(buf);
19 }
```



START DIGIT DECIMAL

' ' is neither a digit nor a dot; state stays START, nothing happens.

Step 11 / 14

VARIABLES

`c = ' '` `digit = false` `d = START` `buf = ""`

OUTPUT

```
20.99
```

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
2    State d = State.START;
3    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
5      boolean digit = Character.isDigit(c);
▶      if (digit) {
▶        if (d == State.START) d = State.DIGIT;
▶        buf.append(c);
9      } else if (c == '.' && d != State.DECIMAL) {
10       d = State.DECIMAL;
11       buf.append(c);
12     } else if (d != State.START) {  // number
   ended
13       System.out.println(buf);
14       buf.setLength(0);
15       d = State.START;
16     }
17   }
18   if (d != State.START) System.out.println(buf);
19 }
```



START | DIGIT | DECIMAL

'5' is a digit and the state is START → **DIGIT**; appended to the buffer.

Step 12 / 14

VARIABLES

`c = '5'` `digit = true` `d = DIGIT` `buf = "5"`

OUTPUT

```
20.99
```

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
2    State d = State.START;
3    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
5      boolean digit = Character.isDigit(c);
6      if (digit) {
7        if (d == State.START) d = State.DIGIT;
8        buf.append(c);
▶      } else if (c == '.' && d != State.DECIMAL) {
▶        d = State.DECIMAL;
▶        buf.append(c);
12     } else if (d != State.START) {  // number ended
13       System.out.println(buf);
14       buf.setLength(0);
15       d = State.START;
16     }
17   }
18   if (d != State.START) System.out.println(buf);
19 }
```



START DIGIT DECIMAL

'.' arrives → move to **DECIMAL**; the dot is appended to the buffer.

Step 13 / 14

VARIABLES

`c = '.'` `digit = false` `d = DECIMAL` `buf = "5."`

OUTPUT

```
20.99
```

Parsing Numbers with an FSM: Code Trace

```
1  void findNumbers(String s) {
2    State d = State.START;
3    StringBuilder buf = new StringBuilder();
4    for (char c : s.toCharArray()) {
5      boolean digit = Character.isDigit(c);
6      if (digit) {
7        if (d == State.START) d = State.DIGIT;
8        buf.append(c);
9      } else if (c == '.' && d != State.DECIMAL) {
10       d = State.DECIMAL;
11       buf.append(c);
12     } else if (d != State.START) {  // number ended
13       System.out.println(buf);
14       buf.setLength(0);
15       d = State.START;
16     }
17   }
▶  if (d != State.START) System.out.println(buf);
19 }
```

$	2	0	.	9	9		v	e		5	.

START DIGIT DECIMAL

Input exhausted; the state is DECIMAL, so the buffer is printed: **5.**. The trailing dot was included in the number (a flaw).

Step 14 / 14

VARIABLES

`c = –` `digit = –` `d = DECIMAL` `buf = "5."`

OUTPUT

```
20.99
5.
```

Levenshtein Distance (Edit Distance)

- **Levenshtein distance** (V. Levenshtein, 1965) is a metric that measures how different two strings are.
- It is the **minimum number of single-character edits** needed to turn one string into the other.
- Used in autocorrect and predictive text, spell checking, DNA sequence comparison and fuzzy search.
- Metric properties: d(a, a) = 0, d(a, b) = d(b, a), triangle inequality.

Insertion

Add a character: "sittin" → "sitting"

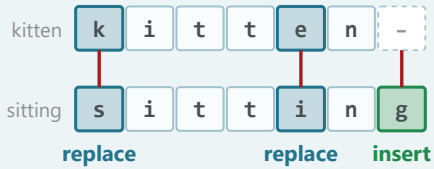
Deletion

Remove a character: "shine" → "shin"

Substitution

Replace one character with another: "**k**itten" → "**s**itten"

kitten → sitten (k→s) → sittin (e→i) → sitting (+g): **d = 3**. No shorter sequence exists; dynamic programming proves it.



Edit Examples

HONDA	H	O		N	D	A	
HYUNDAI	H	Y	U	N	D	A	I

HONDA → HYUNDAI: substitute O→Y, insert U, insert I: d = 3

rain → sain → shin → shine (a) 3

shine → rhine → raine → rain (b) 3

shine → tshine → trhine → traine → train (c) 4

substitution insertion deletion

- (a) rain → **s**ain → s**h**in → shin**e** · (b) shine → **r**hine → r**a**ine → rain (delete e) · (c) shine → **t**shine → t**r**hine → tr**a**ine → train (delete e). A struck-through orange letter is deleted.
- The distance is **symmetric**: (a) and (b) are inverses of each other; insertions and deletions swap roles.

Computing Levenshtein Distance

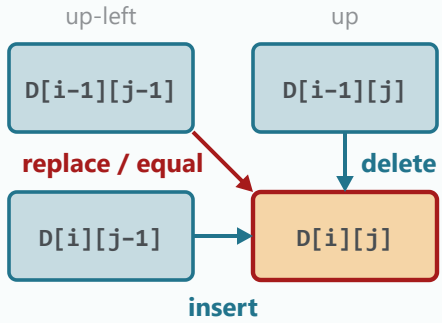
D[i][j] = minimum cost of turning the first **i** characters of a into the first **j** characters of b.

D[i][0] = i (i deletions) · D[0][j] = j (j insertions)

if $a_i = b_j$ then D[i][j] = D[i−1][j−1]

otherwise D[i][j] = 1 + min(D[i−1][j] delete, D[i][j−1] insert, D[i−1][j−1] replace)

- **Dynamic programming**: each subproblem is solved once and stored in an (m+1) × (n+1) table; the answer is in the bottom-right corner.
- A cell depends only on its **upper**, **left** and **upper-left** neighbors → fill the table row by row.
- Time **O(m · n)**; memory O(m · n), or **O(min(m, n))** with two rows if only the distance is needed.



DP Table: CARS → PAIRS

a = **CARS** (rows), b = **PAIRS** (columns)

	ε	P	A	I	R	S
ε	0	1	2	3	4	5
C	1					
A	2					
R	3					
S	4					

Start: first row 0..5 (ε → P, PA, … : insertions), first column 0..4 (C, CA, … → ε : deletions).

DP Table: CARS → PAIRS

a = **CARS** (rows), b = **PAIRS** (columns)

	ε	P	A	I	R	S
ε	0	1	2	3	4	5
C	1	1	2	3	4	5
A	2					
R	3					
S	4					

Row 1 (C) filled. C ≠ P: D[1][1] = 1 + min(0, 1, 1) = 1 (substitution).

DP Table: CARS → PAIRS

a = **CARS** (rows), b = **PAIRS** (columns)

	ε	P	A	I	R	S
ε	0	1	2	3	4	5
C	1	1	2	3	4	5
A	2	2	1	2	3	4
R	3					
S	4					

Row 2 (A) filled. A = A: D[2][2] = D[1][1] = 1, no added cost.

DP Table: CARS → PAIRS

a = **CARS** (rows), b = **PAIRS** (columns)

	ε	P	A	I	R	S
ε	0	1	2	3	4	5
C	1	1	2	3	4	5
A	2	2	1	2	3	4
R	3	3	2	2	2	3
S	4					

Row 3 (R) filled. R = R: D[3][4] = D[2][3] = 2. A ≠ I: D[2][3] = 1 + min(1, 2, 3) = 2.

DP Table: CARS → PAIRS

a = **CARS** (rows), b = **PAIRS** (columns)

	ε	P	A	I	R	S
ε	0	1	2	3	4	5
C	1	1	2	3	4	5
A	2	2	1	2	3	4
R	3	3	2	2	2	3
S	4	4	3	3	3	2

Row 4 (S) filled. S = S: D[4][5] = D[3][4] = 2.

DP Table: CARS → PAIRS

a = **CARS** (rows), b = **PAIRS** (columns)

	ε	P	A	I	R	S
ε	0	1	2	3	4	5
C	1	1	2	3	4	5
A	2	2	1	2	3	4
R	3	3	2	2	2	3
S	4	4	3	3	3	2

Result **D[4][5] = 2**. Reading back: substitute C→P, A equal, **insert I**, R and S equal: CARS → PARS → PAIRS.

Levenshtein: Java Code

```java
int levenshtein(String a, String b) {
  int m = a.length(), n = b.length();
  int[][] D = new int[m + 1][n + 1];
  for (int i = 0; i <= m; i++) D[i][0] = i;  // delete
  for (int j = 0; j <= n; j++) D[0][j] = j;  // insert
  for (int i = 1; i <= m; i++)
    for (int j = 1; j <= n; j++)
      if (a.charAt(i - 1) == b.charAt(j - 1))
        D[i][j] = D[i - 1][j - 1];            // equal
      else
        D[i][j] = 1 + Math.min(D[i - 1][j - 1], // replace
                  Math.min(D[i][j - 1],         // insert
                           D[i - 1][j]));       // delete
  return D[m][n];
}
```

- Table index i corresponds to string index i − 1: row 0 of D is for the **empty prefix**.
- Examples: kitten → sitting = 3, ercan → sercan = 1, apple → banana = 5.

levenshtein("kar", "ark")

```
1  int levenshtein(String a, String b) {
2    int m = a.length(), n = b.length();
3    int[][] D = new int[m + 1][n + 1];
▶    for (int i = 0; i <= m; i++)
▶      D[i][0] = i;            // delete
▶    for (int j = 0; j <= n; j++)
▶      D[0][j] = j;            // insert
8    for (int i = 1; i <= m; i++)
9      for (int j = 1; j <= n; j++)
10       if (a.charAt(i-1) == b.charAt(j-1))
11         D[i][j] = D[i-1][j-1];  // equal
12       else
13         D[i][j] = 1 + Math.min(D[i-1][j-1],
14             Math.min(D[i][j-1], D[i-1][j]));
15   return D[m][n];
16 }
```

	ε	a	r	k
ε	0	1	2	3
k	1			
a	2			
r	3			

a = "kar" (rows), b = "ark" (columns). Column 0 holds i deletions and row 0 holds j insertions: the cost of converting to or from the empty prefix.

Step 1 / 11

VARIABLES

m = 3 | n = 3 | i = – | j = –

levenshtein("kar", "ark")

```
1  int levenshtein(String a, String b) {
2    int m = a.length(), n = b.length();
3    int[][] D = new int[m + 1][n + 1];
4    for (int i = 0; i <= m; i++)
5      D[i][0] = i;            // delete
6    for (int j = 0; j <= n; j++)
7      D[0][j] = j;            // insert
8    for (int i = 1; i <= m; i++)
9      for (int j = 1; j <= n; j++)
▶        if (a.charAt(i-1) == b.charAt(j-1))
11         D[i][j] = D[i-1][j-1];  // equal
12       else
▶          D[i][j] = 1 + Math.min(D[i-1][j-1],
▶              Math.min(D[i][j-1], D[i-1][j]));
15   return D[m][n];
16 }
```

	ε	a	r	k
ε	0	1	2	3
k	1	1		
a	2			
r	3			

'k' ≠ 'a' → 1 + min(replace 0, insert 1, delete 1) = **1**.

Step 2 / 11

VARIABLES

m = 3 · n = 3 · i = 1 · j = 1

levenshtein("kar", "ark")

```
1  int levenshtein(String a, String b) {
2    int m = a.length(), n = b.length();
3    int[][] D = new int[m + 1][n + 1];
4    for (int i = 0; i <= m; i++)
5      D[i][0] = i;            // delete
6    for (int j = 0; j <= n; j++)
7      D[0][j] = j;            // insert
8    for (int i = 1; i <= m; i++)
9      for (int j = 1; j <= n; j++)
▶        if (a.charAt(i-1) == b.charAt(j-1))
11         D[i][j] = D[i-1][j-1];  // equal
12       else
▶          D[i][j] = 1 + Math.min(D[i-1][j-1],
▶              Math.min(D[i][j-1], D[i-1][j]));
15   return D[m][n];
16 }
```

	ε	a	r	k
ε	0	1	2	3
k	1	1	2	
a	2			
r	3			

'k' ≠ 'r' → 1 + min(replace 1, insert 1, delete 2) = **2**.

Step 3 / 11

VARIABLES

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

levenshtein("kar", "ark")

```
1  int levenshtein(String a, String b) {
2    int m = a.length(), n = b.length();
3    int[][] D = new int[m + 1][n + 1];
4    for (int i = 0; i <= m; i++)
5      D[i][0] = i;              // delete
6    for (int j = 0; j <= n; j++)
7      D[0][j] = j;              // insert
8    for (int i = 1; i <= m; i++)
9      for (int j = 1; j <= n; j++)
▶        if (a.charAt(i-1) == b.charAt(j-1))
▶          D[i][j] = D[i-1][j-1];  // equal
12       else
13         D[i][j] = 1 + Math.min(D[i-1][j-1],
14             Math.min(D[i][j-1], D[i-1][j]));
15   return D[m][n];
16 }
```

	ε	a	r	k
ε	0	1	2	3
k	1	1	2	2
a	2			
r	3			

a[0] = b[2] = 'k' → free: D[1][3] = up-left = **2**.

Step 4 / 11

VARIABLES

`m = 3` `n = 3` `i = 1` `j = 3`

levenshtein("kar", "ark")

```
1  int levenshtein(String a, String b) {
2    int m = a.length(), n = b.length();
3    int[][] D = new int[m + 1][n + 1];
4    for (int i = 0; i <= m; i++)
5      D[i][0] = i;              // delete
6    for (int j = 0; j <= n; j++)
7      D[0][j] = j;              // insert
8    for (int i = 1; i <= m; i++)
9      for (int j = 1; j <= n; j++)
▶        if (a.charAt(i-1) == b.charAt(j-1))
▶          D[i][j] = D[i-1][j-1];  // equal
12       else
13         D[i][j] = 1 + Math.min(D[i-1][j-1],
14             Math.min(D[i][j-1], D[i-1][j]));
15   return D[m][n];
16 }
```

	ε	a	r	k
ε	0	1	2	3
k	1	1	2	2
a	2	1		
r	3			

a[1] = b[0] = 'a' → free: D[2][1] = up-left = **1**.

Step 5 / 11

VARIABLES

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

levenshtein("kar", "ark")

```
1  int levenshtein(String a, String b) {
2    int m = a.length(), n = b.length();
3    int[][] D = new int[m + 1][n + 1];
4    for (int i = 0; i <= m; i++)
5      D[i][0] = i;              // delete
6    for (int j = 0; j <= n; j++)
7      D[0][j] = j;              // insert
8    for (int i = 1; i <= m; i++)
9      for (int j = 1; j <= n; j++)
▶        if (a.charAt(i-1) == b.charAt(j-1))
11         D[i][j] = D[i-1][j-1];  // equal
12       else
▶          D[i][j] = 1 + Math.min(D[i-1][j-1],
▶              Math.min(D[i][j-1], D[i-1][j]));
15   return D[m][n];
16 }
```

	ε	a	r	k
ε	0	1	2	3
k	1	1	2	2
a	2	1	2	
r	3			

'a' ≠ 'r' → 1 + min(replace 1, insert 1, delete 2) = **2**.

Step 6 / 11

VARIABLES

m = 3 | n = 3 | i = 2 | j = 2

levenshtein("kar", "ark")

```
1  int levenshtein(String a, String b) {
2    int m = a.length(), n = b.length();
3    int[][] D = new int[m + 1][n + 1];
4    for (int i = 0; i <= m; i++)
5      D[i][0] = i;            // delete
6    for (int j = 0; j <= n; j++)
7      D[0][j] = j;            // insert
8    for (int i = 1; i <= m; i++)
9      for (int j = 1; j <= n; j++)
▶        if (a.charAt(i-1) == b.charAt(j-1))
11         D[i][j] = D[i-1][j-1];  // equal
12       else
▶          D[i][j] = 1 + Math.min(D[i-1][j-1],
▶              Math.min(D[i][j-1], D[i-1][j]));
15   return D[m][n];
16 }
```

	ε	a	r	k
ε	0	1	2	3
k	1	1	2	2
a	2	1	2	3
r	3			

'a' ≠ 'k' → 1 + min(replace 2, insert 2, delete 2) = **3**.

Step 7 / 11

VARIABLES

m = 3 | n = 3 | i = 2 | j = 3

levenshtein("kar", "ark")

```
1  int levenshtein(String a, String b) {
2    int m = a.length(), n = b.length();
3    int[][] D = new int[m + 1][n + 1];
4    for (int i = 0; i <= m; i++)
5      D[i][0] = i;            // delete
6    for (int j = 0; j <= n; j++)
7      D[0][j] = j;            // insert
8    for (int i = 1; i <= m; i++)
9      for (int j = 1; j <= n; j++)
▶        if (a.charAt(i-1) == b.charAt(j-1))
11         D[i][j] = D[i-1][j-1];  // equal
12       else
▶          D[i][j] = 1 + Math.min(D[i-1][j-1],
▶              Math.min(D[i][j-1], D[i-1][j]));
15   return D[m][n];
16 }
```

	ε	a	r	k
ε	0	1	2	3
k	1	1	2	2
a	2	1	2	3
r	3	2		

'r' ≠ 'a' → 1 + min(replace 2, insert 3, delete 1) = **2**.

Step 8 / 11

VARIABLES

m = 3 · n = 3 · i = 3 · j = 1

levenshtein("kar", "ark")

```
1  int levenshtein(String a, String b) {
2    int m = a.length(), n = b.length();
3    int[][] D = new int[m + 1][n + 1];
4    for (int i = 0; i <= m; i++)
5      D[i][0] = i;              // delete
6    for (int j = 0; j <= n; j++)
7      D[0][j] = j;              // insert
8    for (int i = 1; i <= m; i++)
9      for (int j = 1; j <= n; j++)
▶        if (a.charAt(i-1) == b.charAt(j-1))
▶          D[i][j] = D[i-1][j-1];  // equal
12       else
13         D[i][j] = 1 + Math.min(D[i-1][j-1],
14             Math.min(D[i][j-1], D[i-1][j]));
15   return D[m][n];
16 }
```

	ε	a	r	k
ε	0	1	2	3
k	1	1	2	2
a	2	1	2	3
r	3	2	1	

a[2] = b[1] = 'r' → free: D[3][2] = up-left = **1**.

Step 9 / 11

VARIABLES

`m = 3` `n = 3` `i = 3` `j = 2`

levenshtein("kar", "ark")

```
1  int levenshtein(String a, String b) {
2    int m = a.length(), n = b.length();
3    int[][] D = new int[m + 1][n + 1];
4    for (int i = 0; i <= m; i++)
5      D[i][0] = i;            // delete
6    for (int j = 0; j <= n; j++)
7      D[0][j] = j;            // insert
8    for (int i = 1; i <= m; i++)
9      for (int j = 1; j <= n; j++)
▶        if (a.charAt(i-1) == b.charAt(j-1))
11         D[i][j] = D[i-1][j-1];  // equal
12       else
▶          D[i][j] = 1 + Math.min(D[i-1][j-1],
▶              Math.min(D[i][j-1], D[i-1][j]));
15   return D[m][n];
16 }
```

	ε	a	r	k
ε	0	1	2	3
k	1	1	2	2
a	2	1	2	3
r	3	2	1	2

'r' ≠ 'k' → 1 + min(replace 2, insert 1, delete 3) = **2**.

Step 10 / 11

VARIABLES

m = 3 | n = 3 | i = 3 | j = 3

levenshtein("kar", "ark")

```
1  int levenshtein(String a, String b) {
2    int m = a.length(), n = b.length();
3    int[][] D = new int[m + 1][n + 1];
4    for (int i = 0; i <= m; i++)
5      D[i][0] = i;            // delete
6    for (int j = 0; j <= n; j++)
7      D[0][j] = j;            // insert
8    for (int i = 1; i <= m; i++)
9      for (int j = 1; j <= n; j++)
10       if (a.charAt(i-1) == b.charAt(j-1))
11         D[i][j] = D[i-1][j-1];  // equal
12       else
13         D[i][j] = 1 + Math.min(D[i-1][j-1],
14             Math.min(D[i][j-1], D[i-1][j]));
▶    return D[m][n];
16 }
```

	ε	a	r	k
ε	0	1	2	3
k	1	1	2	2
a	2	1	2	3
r	3	2	1	2

D[3][3] = **2**: delete the leading k of "kar" (ar), then append k (ark). The green path shows these two edits.

Step 11 / 11

VARIABLES

m = 3 | n = 3 | i = 4 | j = 4

Comparing Similarity Measures

Measure	What it measures	Time	AGGTAB · GXTXAYB
Hamming	equal lengths; number of differing positions	O(n)	undefined (6 ≠ 7)
Levenshtein	number of inserts / deletes / substitutions	O(m · n)	4
LCS	longest common subsequence	O(m · n)	4 (GTAB)
Longest common substring	longest contiguous common piece	O(m · n)	1 (e.g. "G")

- The LCS is the longest common subsequence that keeps relative order but need not be contiguous; it is found with a DP table like Levenshtein's (diagonal + 1 on a match, otherwise max(up, left)).
- If only insertions and deletions are allowed, the distance is **m + n − 2 · LCS**: 6 + 7 − 8 = 5; Levenshtein, which also allows substitutions, brings it down to 4.
- Longest common substring: on a match L[i][j] = L[i−1][j−1] + 1, otherwise **0**; the answer is the largest value in the table (ercan · sercan → "ercan").
- A similarity ratio: 1 − d / max(m, n). kitten · sitting: 1 − 3/7 ≈ 0.57.

Suffix Arrays

- A **suffix array** holds the starting indices of all suffixes of a string, in **sorted order**.
- Each suffix runs from some position of the string to its end. A suffix array is a **compact representation** of all substrings: every substring is a prefix of some suffix.

i	suffix
0	`banana`
1	`anana`
2	`nana`
3	`ana`
4	`na`
5	`a`

rank	SA	sorted suffix
0	**5**	`a`
1	**3**	`ana`
2	**1**	`anana`
3	**0**	`banana`
4	**4**	`na`
5	**2**	`nana`

"banana" → SA = [5, 3, 1, 0, 4, 2]. Only n integers: O(n) memory versus the $O(n^2)$ nodes of a suffix trie. Uses: string search, sorting, genome sequence analysis, BWT and data compression.

Building a Suffix Array

Method	Idea	Time
Brute force	Generate all suffixes and sort them with `Arrays.sort`; each comparison is O(n)	$O(n^2 \log n)$
Manber–Myers (1990)	Prefix doubling: derive the order on 2k characters from the order on k characters **without comparing letters**, by radix sorting pairs of ranks	O(n log n)
Larsson–Sadakane (1999)	Same doubling; only groups not yet separated are re-sorted by comparing the ranks of later suffixes	O(n log n)
DC3 (2003), SA-IS (2009)	Recursively sort a subset of the suffixes and merge in the rest	O(n)

Prefix doubling runs in **O(n log n)** time, not linear. Linear-time construction came later, with algorithms such as DC3 and SA-IS.

Prefix Doubling

The Manber–Myers idea on "banana": the order on 2k characters comes from the order on k characters. $\log_2 n$ rounds, each O(n) with radix sort.

i	first 1 char	key	new rank
1	a	'a'	**0**
3	a	'a'	**0**
5	a	'a'	**0**
0	b	'b'	**1**
2	n	'n'	**2**
4	n	'n'	**2**

k = 1: sort the suffixes by their first character: a → 0, b → 1, n → 2. Equal ranks are not yet separated.

Prefix Doubling

The Manber–Myers idea on "banana": the order on 2k characters comes from the order on k characters. $\log_2 n$ rounds, each O(n) with radix sort.

i	first 2 chars	key	new rank
5	`a`	(0, −1)	**0**
1	`an`	(0, 2)	**1**
3	`an`	(0, 2)	**1**
0	`ba`	(1, 0)	**2**
2	`na`	(2, 0)	**3**
4	`na`	(2, 0)	**3**

k = 2: key = (rank[i], rank[i + 1]), or −1 if i + 1 ≥ n. No letters are compared; only pairs of numbers are sorted.

Prefix Doubling

The Manber–Myers idea on "banana": the order on 2k characters comes from the order on k characters. $\log_2 n$ rounds, each O(n) with radix sort.

i	first 4 chars	key	new rank
5	`a`	(0, −1)	**0**
3	`ana`	(1, 0)	**1**
1	`anan`	(1, 1)	**2**
0	`bana`	(2, 3)	**3**
4	`na`	(3, −1)	**4**
2	`nana`	(3, 3)	**5**

k = 4: key = (rank[i], rank[i + 2]), or −1 if i + 2 ≥ n. No letters are compared; only pairs of numbers are sorted. All ranks distinct → SA = **[5, 3, 1, 0, 4, 2]**.

Searching with a Suffix Array

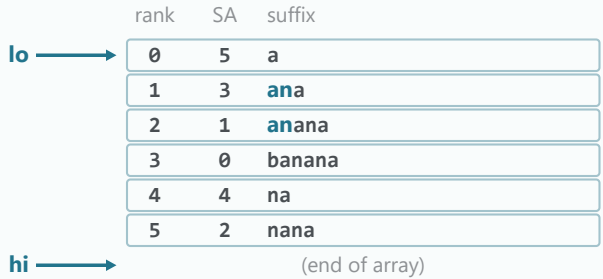
- Since the suffixes are sorted, those that **start with** the pattern P form a **contiguous range** in SA.
- Both ends of the range are found by **binary search**: O(log n) steps × O(m) per comparison = **O(m log n)**.
- Far less memory than a suffix trie: just n integers.
- Add the LCP array (longest common prefix of adjacent suffixes) and the longest repeated substring is found in O(n).

rank	SA	suffix
0	5	`a`
1	3	`ana`
2	1	`anana`
3	0	`banana`
4	4	`na`
5	2	`nana`

Example: P = "an" in "banana". The middle, rank 2 ("anana"), starts with "an"; the left- and right-boundary searches find the range [1, 2] → "an" occurs at **3 and 1**.

search("banana", sa, "an")

```
1  int[] search(String t, int[] sa, String p)
   {
▶    int lo = 0, hi = sa.length;
3    while (lo < hi) {   // first >= p
4      int mid = (lo + hi) / 2;
5      if (t.substring(sa[mid])
6            .compareTo(p) < 0) lo = mid + 1;
7      else hi = mid;
8    }
9    int first = lo;
10   hi = sa.length;
11   while (lo < hi) {   // not prefixed by p
12     int mid = (lo + hi) / 2;
13     if (t.startsWith(p, sa[mid]))
14       lo = mid + 1;
15     else hi = mid;
16   }
17   return new int[] { first, lo };
18 }
```



t = "banana", SA = [5, 3, 1, 0, 4, 2], p = "an". The suffixes starting with "an" form a contiguous range of the sorted array; two binary searches find its ends.

Step 1 / 8

VARIABLES

`lo = 0` `hi = 6` `mid = –` `first = –`

search("banana", sa, "an")

```
1  int[] search(String t, int[] sa, String p)
   {
2    int lo = 0, hi = sa.length;
3    while (lo < hi) {   // first >= p
       int mid = (lo + hi) / 2;
       if (t.substring(sa[mid])
6            .compareTo(p) < 0) lo = mid + 1;
       else hi = mid;
8    }
9    int first = lo;
10   hi = sa.length;
11   while (lo < hi) {   // not prefixed by p
12     int mid = (lo + hi) / 2;
13     if (t.startsWith(p, sa[mid]))
14       lo = mid + 1;
15     else hi = mid;
16   }
17   return new int[] { first, lo };
18 }
```

	rank	SA	suffix
lo →	0	5	a
	1	3	ana
	2	1	anana
hi →	3	0	banana
	4	4	na
	5	2	nana

mid = 3: "banana" ≥ "an" → hi = mid. Range [0, 3).

Step 2 / 8

VARIABLES

`lo = 0` `hi = 3` `mid = 3` `first = –`

search("banana", sa, "an")

```
1  int[] search(String t, int[] sa, String p)
   {
2    int lo = 0, hi = sa.length;
3    while (lo < hi) {   // first >= p
       int mid = (lo + hi) / 2;
       if (t.substring(sa[mid])
6            .compareTo(p) < 0) lo = mid + 1;
       else hi = mid;
8    }
9    int first = lo;
10   hi = sa.length;
11   while (lo < hi) {   // not prefixed by p
12     int mid = (lo + hi) / 2;
13     if (t.startsWith(p, sa[mid]))
14       lo = mid + 1;
15     else hi = mid;
16   }
17   return new int[] { first, lo };
18 }
```

	rank	SA	suffix
lo →	0	5	a
hi →	1	3	ana
	2	1	anana
	3	0	banana
	4	4	na
	5	2	nana

mid = 1: "ana" ≥ "an" → hi = mid. Range [0, 1).

Step 3 / 8

VARIABLES

`lo = 0` `hi = 1` `mid = 1` `first = –`

search("banana", sa, "an")

```
1  int[] search(String t, int[] sa, String p)
   {
2    int lo = 0, hi = sa.length;
3    while (lo < hi) {   // first >= p
       int mid = (lo + hi) / 2;
       if (t.substring(sa[mid])
             .compareTo(p) < 0) lo = mid + 1;
7      else hi = mid;
8    }
9    int first = lo;
10   hi = sa.length;
11   while (lo < hi) {   // not prefixed by p
12     int mid = (lo + hi) / 2;
13     if (t.startsWith(p, sa[mid]))
14       lo = mid + 1;
15     else hi = mid;
16   }
17   return new int[] { first, lo };
18 }
```

lo=hi →

rank	SA	suffix
0	5	a
1	3	ana
2	1	anana
3	0	banana
4	4	na
5	2	nana

mid = 0: "a" < "an" → lo = mid + 1. Range [1, 1).

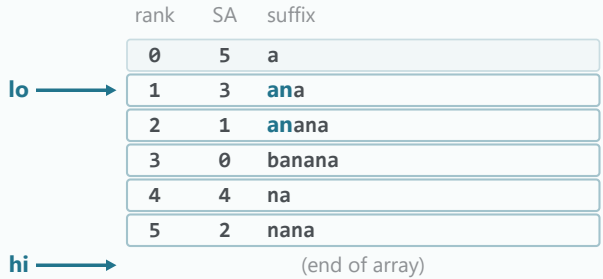
Step 4 / 8

VARIABLES

lo = 1 | hi = 1 | mid = 0 | first = –

search("banana", sa, "an")

```
1  int[] search(String t, int[] sa, String p)
   {
2    int lo = 0, hi = sa.length;
3    while (lo < hi) {   // first >= p
4      int mid = (lo + hi) / 2;
5      if (t.substring(sa[mid])
6            .compareTo(p) < 0) lo = mid + 1;
7      else hi = mid;
8    }
▶    int first = lo;
▶    hi = sa.length;
11   while (lo < hi) {   // not prefixed by p
12     int mid = (lo + hi) / 2;
13     if (t.startsWith(p, sa[mid]))
14       lo = mid + 1;
15     else hi = mid;
16   }
17   return new int[] { first, lo };
18 }
```



First loop done: the first suffix not less than "an" is at rank 1 → **first = 1**. For the second search, hi is reset to the end of the array.

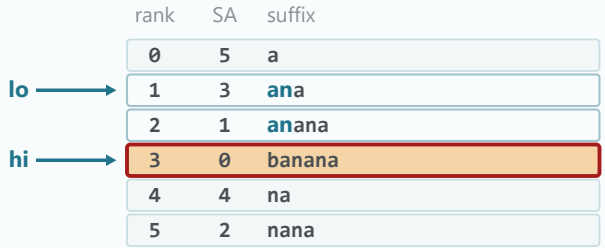
Step 5 / 8

VARIABLES

`lo = 1` `hi = 6` `mid = –` `first = 1`

search("banana", sa, "an")

```
1  int[] search(String t, int[] sa, String p)
   {
2    int lo = 0, hi = sa.length;
3    while (lo < hi) {   // first >= p
4      int mid = (lo + hi) / 2;
5      if (t.substring(sa[mid])
6            .compareTo(p) < 0) lo = mid + 1;
7      else hi = mid;
8    }
9    int first = lo;
10   hi = sa.length;
11   while (lo < hi) {   // not prefixed by p
▶      int mid = (lo + hi) / 2;
▶      if (t.startsWith(p, sa[mid]))
14       lo = mid + 1;
▶      else hi = mid;
16   }
17   return new int[] { first, lo };
18 }
```



mid = 3: "banana" does not start with "an" → hi = mid.

VARIABLES

lo = 1 | hi = 3 | mid = 3 | first = 1

search("banana", sa, "an")

```
1  int[] search(String t, int[] sa, String p)
   {
2    int lo = 0, hi = sa.length;
3    while (lo < hi) {   // first >= p
4      int mid = (lo + hi) / 2;
5      if (t.substring(sa[mid])
6            .compareTo(p) < 0) lo = mid + 1;
7      else hi = mid;
8    }
9    int first = lo;
10   hi = sa.length;
11   while (lo < hi) {   // not prefixed by p
▶      int mid = (lo + hi) / 2;
▶      if (t.startsWith(p, sa[mid]))
▶        lo = mid + 1;
15     else hi = mid;
16   }
17   return new int[] { first, lo };
18 }
```

lo=hi →

rank	SA	suffix	
0	5	a	
1	3	ana	first
2	1	anana	
3	0	banana	
4	4	na	
5	2	nana	

mid = 2: "anana" starts with "an" → lo = mid + 1.

Step 7 / 8

VARIABLES

lo = 3 | hi = 3 | mid = 2 | first = 1

search("banana", sa, "an")

```
1  int[] search(String t, int[] sa, String p)
   {
2    int lo = 0, hi = sa.length;
3    while (lo < hi) {   // first >= p
4      int mid = (lo + hi) / 2;
5      if (t.substring(sa[mid])
6            .compareTo(p) < 0) lo = mid + 1;
7      else hi = mid;
8    }
9    int first = lo;
10   hi = sa.length;
11   while (lo < hi) {   // not prefixed by p
12     int mid = (lo + hi) / 2;
13     if (t.startsWith(p, sa[mid]))
14       lo = mid + 1;
15     else hi = mid;
16   }
▶    return new int[] { first, lo };
18 }
```

	rank	SA	suffix	
	0	5	a	
	1	3	ana	first
	2	1	anana	
lo=hi →	3	0	banana	
	4	4	na	
	5	2	nana	

Result [1, 3): ranks 1 and 2 → SA values **3 and 1**. "an" occurs in banana at positions 3 and 1; 5 comparisons sufficed (each binary search takes at most $\lfloor \log_2 6 \rfloor + 1 = 3$).

Step 8 / 8

VARIABLES

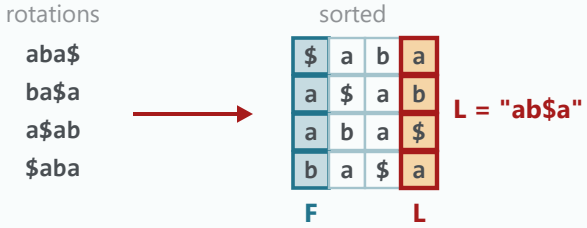
`lo = 3` `hi = 3` `mid = –` `first = 1`

The Burrows–Wheeler Transform (BWT)

- The **Burrows–Wheeler transform** (BWT, 1994) **rearranges** the input: it changes no characters, only their positions.
- It **brings equal characters together**: characters that precede similar contexts end up next to each other.
- It improves the performance of compression algorithms.
- It is **reversible**: the original data can be recovered from the transform.

Steps

1. Generate **all cyclic rotations** of the input string.
2. Sort the rotations **alphabetically**.
3. Form a new string from the **last characters** of the sorted rotations (column L).
4. Store the **row number** of the original string in the sorted list (or append a unique `$`).



The bzip2 pipeline: BWT → Move-to-Front (repeats become small numbers) → Run-Length Encoding → Huffman coding.

BWT Example: BANANA

Input: **BANANA**

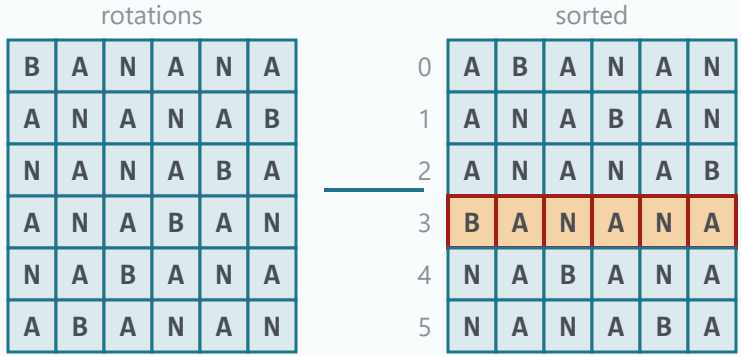
rotations

0	B	A	N	A	N	A
1	A	N	A	N	A	B
2	N	A	N	A	B	A
3	A	N	A	B	A	N
4	N	A	B	A	N	A
5	A	B	A	N	A	N

1. All rotations: BANANA, ANANAB, NANABA, ANABAN, NABANA, ABANAN (moving the first character to the end).

BWT Example: BANANA

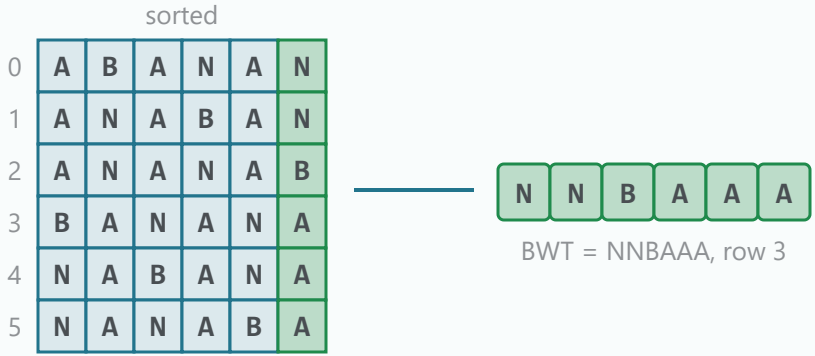
Input: **BANANA**



2. Sorted: ABANAN, ANABAN, ANANAB, BANANA, NABANA, NANABA. The original string is in **row 3** (counting from 0).

BWT Example: BANANA

Input: **BANANA**



3. Last characters (column L): **NNBAAA**. Output: (NNBAAA, 3). Equal letters are grouped: NN, AAA.

BWT Example: abraca

S = abraca (initial)

a	b	r	a	c	a
b	r	a	c	a	a
r	a	c	a	a	b
a	c	a	a	b	r
c	a	a	b	r	a
a	a	b	r	a	c

sorted matrix

a	a	b	r	a	c	← L
a	b	r	a	c	a	
a	c	a	a	b	r	
b	r	a	c	a	a	
c	a	a	b	r	a	
r	a	c	a	a	b	

- Last column of the sorted matrix: **L = caraab**.
- The original "abraca" is in **row 1** of the sorted matrix (counting from 0); this index is stored for the inverse transform.
- The first column (F = aaabcr) is just L's letters in sorted order; knowing L gives F.

The direct method generates n rotations with `substring` and sorts them with `Arrays.sort` → $O(n^2 \log n)$ time, $O(n^2)$ memory. In practice it is derived from the suffix array in O(n) via **L[i] = T[SA[i] − 1]** (T is the text with `$` appended; L[i] = $ if SA[i] = 0; the formula is for the $ version). The $ and cyclic BWTs differ: cyclic BANANA → NNBAAA, BANANA$ → ANNB$AA.

Inverting the BWT (LF Mapping)

i	F	…	L
0	**$**	`banan`	**a**
1	**a**	`$bana`	**n**
2	**a**	`na$ba`	**n**
3	**a**	`nana$`	**b**
4	**b**	`anana`	**$**
5	**n**	`a$ban`	**a**
6	**n**	`ana$b`	**a**

- T = banana**$** ($ smallest, unique) → BWT = **annb$aa**.
- **F** = sort(L) = $aaabnn.
- **LF rule:** the k-th 'c' in L is the same character as the k-th 'c' in F. In row i, L[i] immediately precedes F[i] in the text.
- Start at row 0, which begins with $; read L[i] and go to row LF(i):

rows 0 → 1 → 5 → 2 → 6 → 3 → 4
L read: a n a n a b → reverse → **banana**

With the ranks (each character's occurrence number in L) precomputed, inversion takes O(n) time. Without $, store the original row index instead (3, as in the BANANA example).

inverseBWT("annb$aa")

```
1  String inverseBWT(String L) {   // with $
2    int n = L.length();
3    int[] rank = new int[n];
4    int[] cnt = new int[256];
5    int[] C = new int[256];
▶    for (int i = 0; i < n; i++)
▶      rank[i] = cnt[L.charAt(i)]++;
8    for (int c = 1; c < 256; c++)
9      C[c] = C[c - 1] + cnt[c - 1];
10   StringBuilder s = new StringBuilder();
11   int i = 0;                    // $ row
12   for (int k = 0; k < n - 1; k++) {
13     char c = L.charAt(i);
14     s.insert(0, c);
15     i = C[c] + rank[i];         // LF(i)
16   }
17   return s.toString();
18 }
```

i	L	rank	F
0	a	0	$
1	n	0	a
2	n	1	a
3	b	0	a
4	$	0	b
5	a	1	n
6	a	2	n

	$	a	b	n
C[c]				

C[c]: # of chars smaller than c

s

L = "annb$aa" (the BWT of banana$). rank[i]: which occurrence of its character L[i] is in L (from 0). E.g. L[6] = a, the third a → 2.

Step 1 / 9

VARIABLES

n = 7 | k = – | i = – | c = –

inverseBWT("annb$aa")

```
1  String inverseBWT(String L) {   // with $
2    int n = L.length();
3    int[] rank = new int[n];
4    int[] cnt = new int[256];
5    int[] C = new int[256];
6    for (int i = 0; i < n; i++)
7      rank[i] = cnt[L.charAt(i)]++;
▶    for (int c = 1; c < 256; c++)
▶      C[c] = C[c - 1] + cnt[c - 1];
10   StringBuilder s = new StringBuilder();
11   int i = 0;                    // $ row
12   for (int k = 0; k < n - 1; k++) {
13     char c = L.charAt(i);
14     s.insert(0, c);
15     i = C[c] + rank[i];         // LF(i)
16   }
17   return s.toString();
18 }
```

i	L	rank	F
0	a	0	$
1	n	0	a
2	n	1	a
3	b	0	a
4	$	0	b
5	a	1	n
6	a	2	n

C[c]	$	a	b	n
	0	1	4	5

C[c]: # of chars smaller than c

s

C[c] = number of characters smaller than c = the row where c starts in column F: $ → 0, a → 1, b → 4, n → 5.

Step 2 / 9

VARIABLES

n = 7 | k = – | i = – | c = –

inverseBWT("annb$aa")

```
1  String inverseBWT(String L) {   // with $
2    int n = L.length();
3    int[] rank = new int[n];
4    int[] cnt = new int[256];
5    int[] C = new int[256];
6    for (int i = 0; i < n; i++)
7      rank[i] = cnt[L.charAt(i)]++;
8    for (int c = 1; c < 256; c++)
9      C[c] = C[c - 1] + cnt[c - 1];
10   StringBuilder s = new StringBuilder();
11   int i = 0;                    // $ row
12   for (int k = 0; k < n - 1; k++) {
▶      char c = L.charAt(i);
▶      s.insert(0, c);
▶      i = C[c] + rank[i];         // LF(i)
16   }
17   return s.toString();
18 }
```

i	L	rank	F
0	a	0	$
1	n	0	a
2	n	1	a
3	b	0	a
4	$	0	b
5	a	1	n
6	a	2	n

C[c]	$	a	b	n
	0	1	4	5

C[c]: # of chars smaller than c

s: [ ][ ][ ][ ][ ][a]

i = 0: L[0] = 'a' is prepended → s = "a". LF(0) = C['a'] + rank[0] = 1 + 0 = **1**.

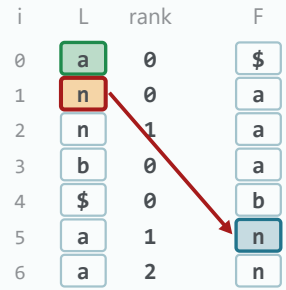
Step 3 / 9

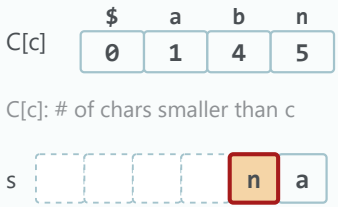
VARIABLES

n = 7 | k = 0 | i = 1 | c = 'a'

inverseBWT("annb$aa")

```
1  String inverseBWT(String L) {   // with $
2    int n = L.length();
3    int[] rank = new int[n];
4    int[] cnt = new int[256];
5    int[] C = new int[256];
6    for (int i = 0; i < n; i++)
7      rank[i] = cnt[L.charAt(i)]++;
8    for (int c = 1; c < 256; c++)
9      C[c] = C[c - 1] + cnt[c - 1];
10   StringBuilder s = new StringBuilder();
11   int i = 0;                    // $ row
12   for (int k = 0; k < n - 1; k++) {
▶      char c = L.charAt(i);
▶      s.insert(0, c);
▶      i = C[c] + rank[i];         // LF(i)
16   }
17   return s.toString();
18 }
```





i = 1: L[1] = 'n' is prepended → s = "na". LF(1) = C['n'] + rank[1] = 5 + 0 = **5**.

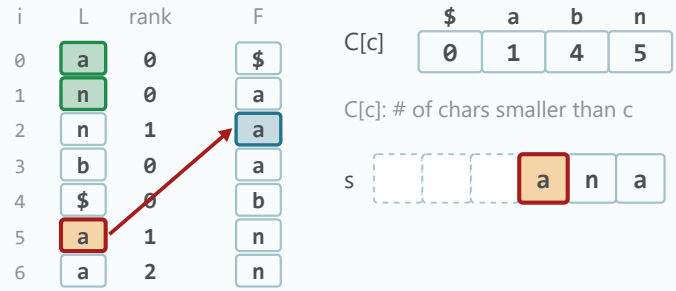
Step 4 / 9

VARIABLES

n = **7** k = **1** i = **5** c = **'n'**

inverseBWT("annb$aa")

```
1  String inverseBWT(String L) {   // with $
2    int n = L.length();
3    int[] rank = new int[n];
4    int[] cnt = new int[256];
5    int[] C = new int[256];
6    for (int i = 0; i < n; i++)
7      rank[i] = cnt[L.charAt(i)]++;
8    for (int c = 1; c < 256; c++)
9      C[c] = C[c - 1] + cnt[c - 1];
10   StringBuilder s = new StringBuilder();
11   int i = 0;                    // $ row
12   for (int k = 0; k < n - 1; k++) {
▶      char c = L.charAt(i);
▶      s.insert(0, c);
▶      i = C[c] + rank[i];         // LF(i)
16   }
17   return s.toString();
18 }
```



i = 5: L[5] = 'a' is prepended → s = "ana". LF(5) = C['a'] + rank[5] = 1 + 1 = **2**.

Step 5 / 9

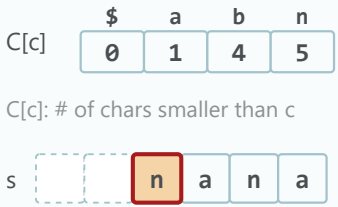
VARIABLES

`n = 7` `k = 2` `i = 2` `c = 'a'`

inverseBWT("annb$aa")

```
1  String inverseBWT(String L) {   // with $
2    int n = L.length();
3    int[] rank = new int[n];
4    int[] cnt = new int[256];
5    int[] C = new int[256];
6    for (int i = 0; i < n; i++)
7      rank[i] = cnt[L.charAt(i)]++;
8    for (int c = 1; c < 256; c++)
9      C[c] = C[c - 1] + cnt[c - 1];
10   StringBuilder s = new StringBuilder();
11   int i = 0;                      // $ row
12   for (int k = 0; k < n - 1; k++) {
▶      char c = L.charAt(i);
▶      s.insert(0, c);
▶      i = C[c] + rank[i];         // LF(i)
16   }
17   return s.toString();
18 }
```





i = 2: L[2] = 'n' is prepended → s = "nana". LF(2) = C['n'] + rank[2] = 5 + 1 = **6**.

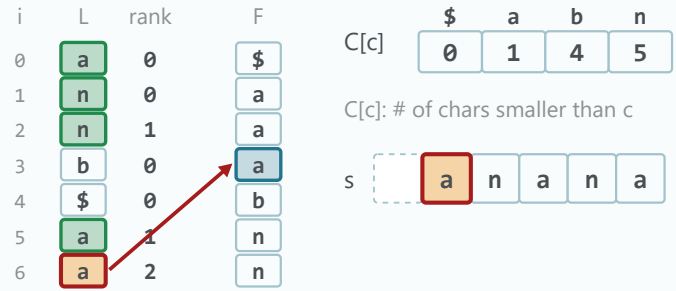
Step 6 / 9

VARIABLES

n = 7 | k = 3 | i = 6 | c = 'n'

inverseBWT("annb$aa")

```
1  String inverseBWT(String L) {   // with $
2    int n = L.length();
3    int[] rank = new int[n];
4    int[] cnt = new int[256];
5    int[] C = new int[256];
6    for (int i = 0; i < n; i++)
7      rank[i] = cnt[L.charAt(i)]++;
8    for (int c = 1; c < 256; c++)
9      C[c] = C[c - 1] + cnt[c - 1];
10   StringBuilder s = new StringBuilder();
11   int i = 0;                    // $ row
12   for (int k = 0; k < n - 1; k++) {
▶      char c = L.charAt(i);
▶      s.insert(0, c);
▶      i = C[c] + rank[i];         // LF(i)
16   }
17   return s.toString();
18 }
```



i = 6: L[6] = 'a' is prepended → s = "anana". LF(6) = C['a'] + rank[6] = 1 + 2 = **3**.

Step 7 / 9

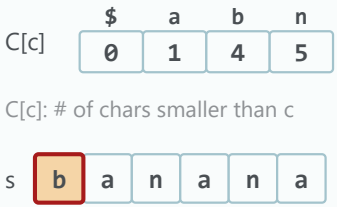
VARIABLES

n = 7 | k = 4 | i = 3 | c = 'a'

inverseBWT("annb$aa")

```
1  String inverseBWT(String L) {   // with $
2    int n = L.length();
3    int[] rank = new int[n];
4    int[] cnt = new int[256];
5    int[] C = new int[256];
6    for (int i = 0; i < n; i++)
7      rank[i] = cnt[L.charAt(i)]++;
8    for (int c = 1; c < 256; c++)
9      C[c] = C[c - 1] + cnt[c - 1];
10   StringBuilder s = new StringBuilder();
11   int i = 0;                    // $ row
12   for (int k = 0; k < n - 1; k++) {
▶      char c = L.charAt(i);
▶      s.insert(0, c);
▶      i = C[c] + rank[i];         // LF(i)
16   }
17   return s.toString();
18 }
```





i = 3: L[3] = 'b' is prepended → s = "banana". LF(3) = C['b'] + rank[3] = 4 + 0 = **4**.

Step 8 / 9

VARIABLES

n = 7 | k = 5 | i = 4 | c = 'b'

inverseBWT("annb$aa")

```
1  String inverseBWT(String L) {   // with $
2    int n = L.length();
3    int[] rank = new int[n];
4    int[] cnt = new int[256];
5    int[] C = new int[256];
6    for (int i = 0; i < n; i++)
7      rank[i] = cnt[L.charAt(i)]++;
8    for (int c = 1; c < 256; c++)
9      C[c] = C[c - 1] + cnt[c - 1];
10   StringBuilder s = new StringBuilder();
11   int i = 0;                    // $ row
12   for (int k = 0; k < n - 1; k++) {
13     char c = L.charAt(i);
14     s.insert(0, c);
15     i = C[c] + rank[i];         // LF(i)
16   }
▶    return s.toString();
18 }
```

i	L	rank	F
0	a	0	$
1	n	0	a
2	n	1	a
3	b	0	a
4	$	0	b
5	a	1	n
6	a	2	n

C[c]	$	a	b	n
	0	1	4	5

C[c]: # of chars smaller than c

s	b	a	n	a	n	a

Rows 0 → 1 → 5 → 2 → 6 → 3 → 4 were visited; the next row would end in $, so the loop stops after n − 1 = 6 rounds. Result: **banana**.

Step 9 / 9

VARIABLES

`n = 7` `k = 6` `i = 4` `c = 'b'`

Summary

Topic	Key points
Lexicographic order	the first differing character decides; a prefix comes first; code order ≠ Turkish alphabet
Radix sort	LSD: stable counting sort right to left, $O(W \cdot (N + R))$; MSD: from the left, recursing into buckets
Regular expressions	. * + ? [ ] ^ $ ( ) { } | \d \w \s; lookarounds consume no characters
FSM	$(Q, \Sigma, \delta, q_0, F)$; one transition per step in a DFA, O(n) scan; NFA ≡ DFA ≡ regex
Levenshtein	1 + min(delete, insert, replace), or up-left if equal; $O(m \cdot n)$
Similarity	Hamming on equal lengths O(n); LCS and common substring $O(m \cdot n)$; insert/delete-only distance = $m + n - 2 \cdot LCS$
Suffix array	sorted suffix indices; brute force $O(n^2 \log n)$, doubling $O(n \log n)$; search $O(m \log n)$
BWT	last column of the sorted rotations + row index; inverted via LF mapping; bzip2

SUMMARY

Common Mistakes

Mistaking code order for the alphabet

"Zebra" < "apple" and "şeker" > "zeytin". Case and Turkish letters need a `Collator` or normalization.

Sorting numbers as strings

"10" < "9". For numeric order, convert to integers or zero-pad to equal length.

An unstable sub-sort in LSD

Every pass must be stable, or the order from earlier passes is lost. In counting sort, place items from back to front.

Unescaped dot, greedy .*

`a.b` also matches "a+b"; `<.*>` swallows the whole line. Use `\.` and the lazy `.*?`. Expressions like `(a+)+$` cause exponential backtracking.

Off-by-one in the DP table

The table is (m+1) × (n+1); D[i][j] compares `charAt(i - 1)`. Not initializing the first row/column breaks the result.

Confusing subsequence and substring

LCS does not require contiguity; longest common substring does. In BWT, forgetting to store the row index or the `$` makes the transform irreversible.

SUMMARY

Sample Problems

1. Put the strings "10", "9", "100", "2" in lexicographic order.
2. Apply LSD radix sort to sob, bat, cab, bot, tab; write the array after each pass.
3. Which of "555-1234", "5551234", "55-12345" does `^\d{3}-\d{4}$` match?
4. What is Levenshtein("sunday", "saturday")?
5. What is Hamming("karolin", "kathrin")? Can the Levenshtein distance be larger?
6. Find the BWT of "kayak" and the original row index.

▶ **Show answers**

SUMMARY

Exercises

1. Write MSD radix sort recursively.
2. Write a regex that validates the Turkish mobile number format (0532 123 45 67).
3. Draw a DFA that accepts binary strings with an even number of 1s and write its transition table.
4. Compute Levenshtein with only two rows of memory, O(min(m, n)) extra space.
5. The number-parsing FSM prints "5." on the input "... or 5.". Fix the machine.
6. Find the suffix array of "mississippi".

▶ **Answers / hints**

References

- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011 (5.1 String Sorts, 5.4 Regular Expressions).
- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., 2022 (string-matching automata).
- Hopcroft, Motwani and Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed., 2006.
- V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals", *Soviet Physics Doklady* 10(8), 1966.
- R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem", *Journal of the ACM* 21(1), 1974.
- U. Manber and G. Myers, "Suffix Arrays: A New Method for On-Line String Searches", *SIAM Journal on Computing* 22(5), 1993.
- M. Burrows and D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm", DEC SRC Research Report 124, 1994.
- J. E. F. Friedl, *Mastering Regular Expressions*, 3rd ed., O'Reilly, 2006; Java `java.util.regex.Pattern` documentation.

The End

Chapter 10: Advanced String Algorithms

Previous: Chapter 9: String Matching and Compression · Next: Chapter 11: Tries