Data Structures

Week 4 – Strings: representation, encoding, operations, pattern matching

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

Review

- An array is **homogeneous, static, contiguous, with direct access**
- Address calculation → access **O(1)** · insertion/deletion in the middle **O(n)**
- **Dynamic array:** doubling the capacity → appending is amortized O(1)

Questions for today

- How does the machine store text – and how can `len("世界") == 6`?
- Why is `+` concatenation in a loop slow?
- How do we find a word in a text?

The concept of a string

A **string** (character string, text) is a finite sequence of characters.

- Essentially a **character array** – plus text-specific operations
- **Length:** the number of characters · **empty string:** `""` (length 0)
- **Substring:** a contiguous section of characters (`"data"` is part of `"database"`)

Property	String
Elements	homogeneous (characters)
Storage	contiguous
Access	direct (by index)
Mutability	Java, Go: **immutable**

Ways of representation

C: null-terminated char array



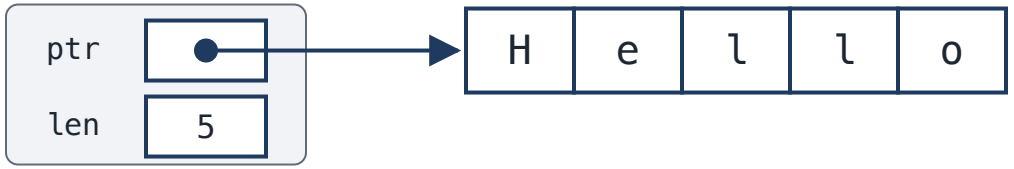
length is O(n): count until \0

Pascal: length prefix



length is O(1), but at most 255 chars

Java / Go: header (length + pointer) + char array



length O(1), contents immutable

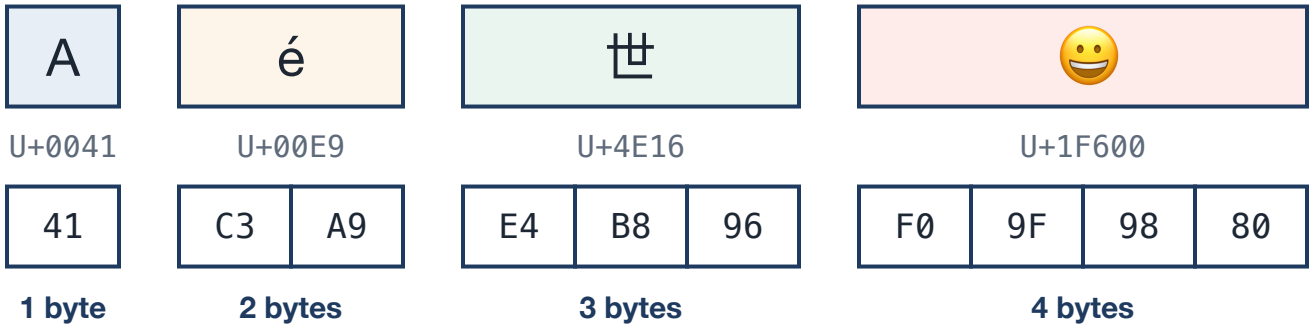
Character encoding

ASCII, Unicode, UTF-8

- **ASCII** (1963): 7 bits, 128 characters – English alphabet, digits, punctuation
- **Unicode:** every character of every writing system gets a number, the **code point**
  `A` = U+0041 · `é` = U+00E9 · `世` = U+4E16 · `😀` = U+1F600
- The code point must be converted into bytes: **encoding**

Encoding	Size of one character	Used by
UTF-8	1–4 bytes (ASCII = 1 byte)	the web, Linux, **Go**
UTF-16	2 or 4 bytes	**Java**, Windows, JavaScript
UTF-32	always 4 bytes	rarely
GB 2312 / GBK / GB 18030	1–4 bytes	Chinese standards

UTF-8: variable-length encoding




```
Go: len("Aé世😀") = 10 bytes · 4 runes Java: "Aé世😀".length() = 5 chars (UTF-16) · 4 code point
```

The answer to “how many characters?” depends on what we count: **bytes**, **UTF-16 units** or **code points**.

Characters in Java and Go

Java – UTF-16

```
String s = "Aé世😀";
s.length();                    // 5 (char)
s.codePointCount(0, s.length()); // 4
s.charAt(2);                   // '世'
s.getBytes(UTF_8).length;      // 10

// iterating by code point:
s.codePoints().forEach(cp -> ...);
```

- `char` = 16 bits → an emoji takes **two** `char`s (surrogate pair)
- Most Chinese characters take **one** `char`

Go – UTF-8

```
s := "Aé世😀"
len(s)                         // 10 (bytes!)
utf8.RuneCountInString(s)      // 4
s[2]                           // 0xA9 — a single byte!
[]rune(s)[2]                   // '世'

for i, r := range s {          // rune by rune
    fmt.Printf("%d %c\n", i, r)
}
```

- `s[i]` gives a **byte**, `range` gives a **rune**
- For character-by-character work: `[]rune(s)`

Character arithmetic

The codes of the English letters and of the digits are **consecutive**:

	Code	Consequence
`'0'` … `'9'`	48 … 57	value of a digit: `c - '0'`
`'A'` … `'Z'`	65 … 90	position of a letter: `c - 'A'`
`'a'` … `'z'`	97 … 122	lower case → upper case: `c - 32`

```
int digit = '7' - '0';                    // 7
char upper = (char) ('g' - ('a' - 'A')); // 'G'
int[] freq = new int[26];
freq[c - 'a']++;                          // letter frequency
```

A 26-element array is a simple way to count letter frequencies, but it only works for the English alphabet. In general we use a **hash table** (week 11).

String operations

Immutable strings

In Java and Go the content of a string **cannot change after creation**.

```
String s = "hello";
s.toUpperCase();          // returns a new string, s does not change!
s = s.toUpperCase();      // this is the way
```

Advantages:

- Can be shared safely (between variables, between threads)
- Can be used as a key in a hash table
- Java: the **string pool** stores identical literals only once

Disadvantage: every “modification” creates a new string → **copying**

Concatenation in a loop: O(n²)!

```java
// SLOW — O(n²)
String s = "";
for (int i = 0; i < n; i++)
    s += "x";   // new string + copying
```

```java
// FAST — O(n)
StringBuilder sb = new StringBuilder();
for (int i = 0; i < n; i++)
    sb.append("x");
String s = sb.toString();
```

Go: `strings.Builder` · `sb.WriteString("x")`

n	+=	Builder
10 000	10 ms	0.05 ms
40 000	46 ms	0.08 ms
160 000	624 ms	0.4 ms

(Go, measured values)

Internally, `StringBuilder` is a **dynamic array** (week 3) – amortized O(1) append.

Comparison

Lexicographic (dictionary) order: character by character from the left, the first difference decides; if one is a prefix of the other, the shorter one is smaller.

```
Function Compare(S, T)                  // −1, 0 or 1
  i ← 1
  While i ≤ length(S) and i ≤ length(T) and S[i] = T[i]
    i ← i + 1
  End while
  If i ≤ length(S) and i ≤ length(T) then
    If S[i] < T[i] then Return −1 else Return 1
  else if length(S) = length(T) then Return 0
  else if length(S) < length(T) then Return −1
  else Return 1
End function
```

`"apple" < "apply"` · `"data" < "database"` · `"Zebra" < "apple"` (an upper-case letter has a smaller code!)

A common mistake in Java: `==` vs `equals`

```
String a = "data";
String b = "data";                // the same object (string pool)
String c = new String("data");    // a new object

a == b          // true  — by accident!
a == c          // false — a different object
a.equals(c)     // true  — the contents are equal
```

In Java, strings are **always compared with** `equals`. `==` compares the **references** (week 1).

In Go, `==` compares the **contents**: `a == c` → `true`

Operations: Java and Go

Operation	Java	Go	Steps
length	`s.length()`	`len(s)` (bytes)	O(1)
i-th character	`s.charAt(i)`	`s[i]` (byte)	O(1)
substring	`s.substring(a, b)`	`s[a:b]`	O(b–a) / O(1)*
concatenation	`s + t`, `s.concat(t)`	`s + t`	O(n + m)
equality	`s.equals(t)`	`s == t`	O(n)
comparison	`s.compareTo(t)`	`s < t`, `strings.Compare`	O(n)
search	`s.indexOf(t)`	`strings.Index(s, t)`	O(n · m)**
splitting	`s.split(",")`	`strings.Split(s, ",")`	O(n)
replacement	`s.replace(a, b)`	`strings.ReplaceAll(s, a, b)`	O(n)
lower / upper case	`toLowerCase()`, `toUpperCase()`	`strings.ToLower`, `ToUpper`	O(n)

* In Go a substring does not copy, it refers to the shared bytes. ** Libraries also use smarter algorithms.

A classic task: palindrome

Palindrome: reads the same backwards – `"radar"`, `"上海自来水来自海上"`

Pseudocode

```
Function Palindrome(S, n)
  i ← 1 ; j ← n
  While i < j and S[i] = S[j]
    i ← i + 1
    j ← j − 1
  End while
  Return i ≥ j
End function
```

Two indexes move towards each other – **O(n)**, O(1) space.

Go – Unicode-safe

```go
func isPalindrome(s string) bool {
    r := []rune(s)
    for i, j := 0, len(r)-1; i < j;
            i, j = i+1, j-1 {
        if r[i] != r[j] {
            return false
        }
    }
    return true
}
```

Checked byte by byte in Go, the Chinese text would **not** be a palindrome!

Pattern matching

The pattern matching problem

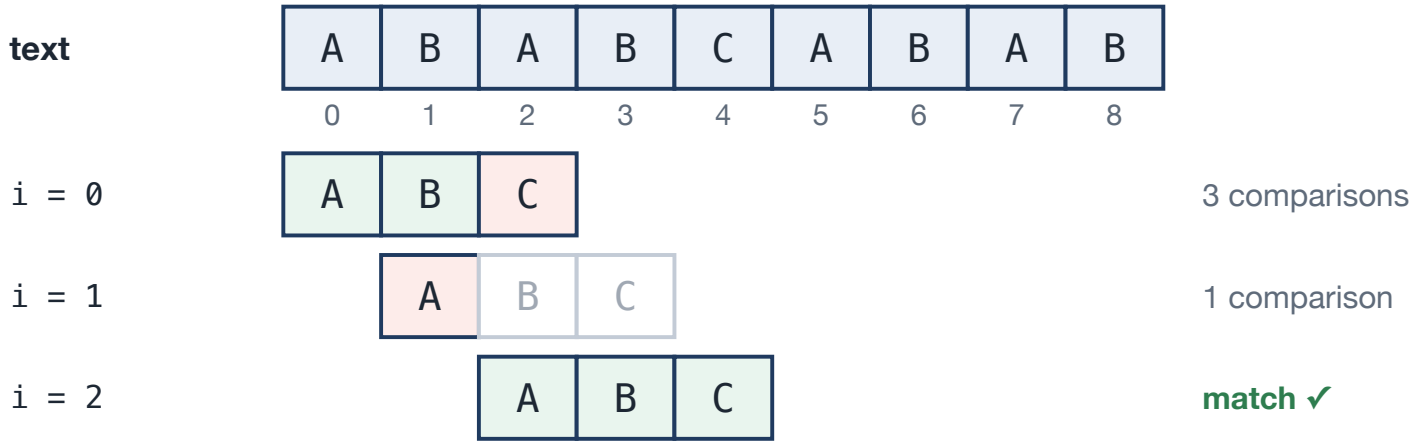
Given: a **text** (S) of length **n** and a **pattern** (P) of length **m**, m ≤ n.
Task: where does P first occur in S?

Applications: the “find” function of text editors, `grep`, virus scanners, analysing **DNA sequences**, plagiarism detection

Naive (brute-force) algorithm:

- Try every possible starting position of the pattern: i = 1, 2, …, n – m + 1
- At each position compare the pattern with the text character by character
- At the first mismatch move on to the next position

Naive pattern matching – example



7 character comparisons in total

Naive pattern matching – algorithm

Pseudocode

```
Function NaiveSearch(S, n, P, m)
  For i ← 1 to n − m + 1
    j ← 1
    While j ≤ m and S[i + j − 1] = P[j]
      j ← j + 1
    End while
    If j > m then Return i
  End for
  Return 0
End function
```

Java

```java
static int naiveSearch(String s, String p) {
    int n = s.length(), m = p.length();
    for (int i = 0; i <= n − m; i++) {
        int j = 0;
        while (j < m &&
               s.charAt(i + j) == p.charAt(j))
            j++;
        if (j == m) return i;
    }
    return −1;
}
```

Efficiency of the naive algorithm

- **Best case:** the pattern is at the beginning → O(m)
- **Worst case:** at every position it almost matches to the end
  S = `aaaa…aab` (n = 10 001), P = `aa…ab` (m = 101)
  → **1 000 001** comparisons ≈ n · m → **O(n · m)**
- In natural text a mismatch is usually found within the first 1–2 characters → fast in practice

More efficient algorithms (mentioned only)

- **Knuth–Morris–Pratt (KMP):** O(n + m) – learns from the structure of the pattern, never steps back in the text
- **Boyer–Moore:** compares from right to left, makes big jumps – very fast in practice
- **Rabin–Karp:** compares hash values (week 11)

Summary

- A string is a **sequence of characters**, stored contiguously, with direct access
- Encoding: **Unicode** code point → **UTF-8** (Go, web) or **UTF-16** (Java)
- “Length” can mean bytes, UTF-16 units or code points – in Go `len(s)` gives bytes!
- Java and Go strings are **immutable** → in loops use `StringBuilder` / `strings.Builder`
- In Java use `equals`, not `==`!
- **Naive pattern matching:** O(n · m) in the worst case; better: KMP, Boyer–Moore

Next week: sorting I – bubble sort, selection sort and insertion sort.

Review questions

1. What is the value of `len("中国")` in Go, and of `"中国".length()` in Java?
2. True or false: in Java, after calling `s.toUpperCase()`, `s` is in upper case.
3. Why is the following code $O(n^2)$? `for (...) s += "x";`
4. Which is lexicographically smaller: `"data"` or `"Data"`? Why?
5. How many character comparisons does the naive algorithm make for `S = "AAAAA"` and `P = "AAB"`?
6. What does Java print: `new String("go") == "go"`?

Practice

✎ Demo programs

Program	Java	Go	What does it show?
Basics	`StringBasics.java`	`w04/stringbasics`	operations, `==` vs `equals`, comparison
Encoding	`Encoding.java`	`w04/encoding`	bytes, `char`, code points, UTF-8 bytes
Builder	`Builder.java`	`w04/builder`	`+=` vs `StringBuilder` timing
Pattern matching	`PatternSearch.java`	`w04/patternsearch`	naive search, all occurrences, worst case

✎ Exercises

1. **Palindrome:** decide whether a sentence is a palindrome (ignoring case and punctuation)! (`"A man, a plan, a canal: Panama"` ✓)
2. **Word count:** how many words are in a text, if the words are separated by one or more spaces?
3. **Letter frequency:** count the occurrences of the English letters with a 26-element array!
4. **Caesar cipher:** shift the letters by **k** positions (`"Hello" → "Khoor"`, k = 3), and decode it!
5. **Anagram:** do two words consist of the same letters? (`listen` – `silent`)
6. **Compression:** `"aaabbcdddd" → "a3b2c1d4"`
7. **Reversing words:** `"I love data structures" → "structures data love I"`

Solutions: `Exercises04.java`, `w04/exercises`

✎ Homework

1. Write lexicographic comparison **without library functions** (returning `-1`, `0`, `1`)!
2. Write a function that returns the starting index of **every** occurrence of a pattern – including overlapping ones! (in `"aaaa"`, `"aa"`: 0, 1, 2)
3. Write a program that prints the **most frequent** word of a text! (Hint: Java `HashMap<String,Integer>`, Go `map[string]int`.)
4. In Go: write a function that reverses a string so that Chinese characters and emojis are displayed correctly too!