

Data Structures

Week 2 – Efficiency of algorithms, linear and binary search

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

Review

- **Algorithm:** a finite, well-defined sequence of steps
- **Data type:** set of values + operations + representation
- **Data structure:** elements + relationships + operations
- Access: **direct** (array) · **sequential** (list)

Questions for today

- How can we tell whether an algorithm is **fast** or **slow**?
- How do we find an element in an array – and can we do it **faster**?



Efficiency of algorithms

Why is a stopwatch not enough?

The running time depends on

- the machine (processor, memory),
- the programming language and the compiler,
- other running programs,
- **the size and content of the input.**

Therefore we do not measure in seconds; instead we examine how the **number of steps** grows as a function of the **input size (n)**.

Step (elementary operation): assignment, comparison, arithmetic operation, accessing an array element – each counts as one step.

Counting steps: an example

Pseudocode – sum

```
Function Sum(A, n)
    s ← 0                // 1
    For i ← 1 to n        // n times
        s ← s + A[i]      // n
    End for
    Return s              // 1
End function
```

- The assignment and the return: a **constant** number of steps
- The loop body runs **n** times
- In total about **$c_1 \cdot n + c_2$** steps

The constants (c_1, c_2) depend on the machine – what matters is that the number of steps grows **linearly** with n .

Best, worst and average case

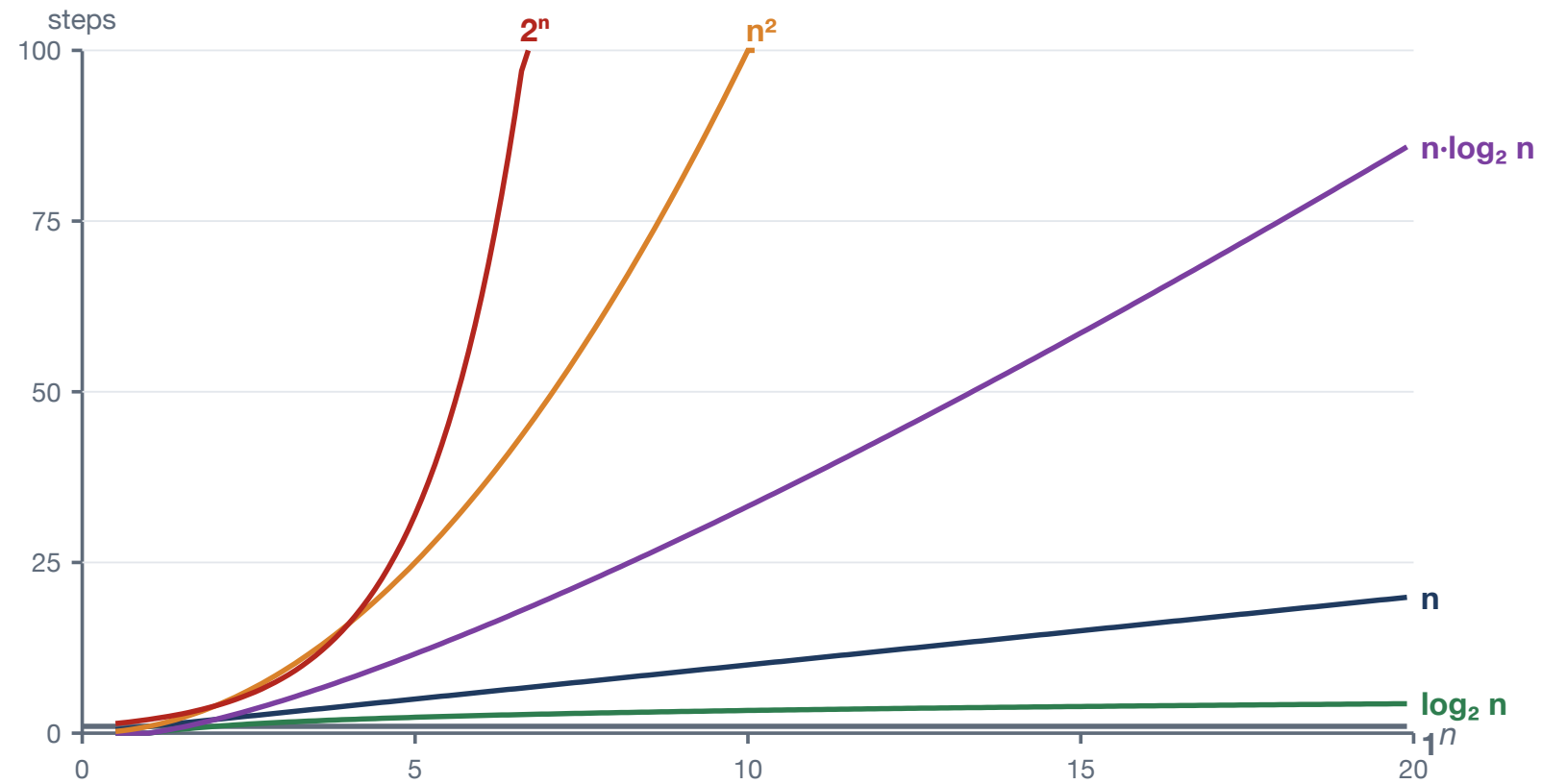
Even for inputs of the same size the number of steps may differ!

Example: we search for the value `x` in an `n`-element array, from the front.

Case	When?	Steps
Best	<code>x</code> is the first element	1
Worst	<code>x</code> is the last one, or not in the array	<code>n</code>
Average	<code>x</code> is equally likely anywhere	about $n / 2$

Usually we examine the **worst case**: it is a guarantee – it will certainly not be slower than that.

Growth rates



How long does this take in practice?

Assume the machine performs **10^9 steps** per second.

n	$\log_2 n$	n	$n \cdot \log_2 n$	n^2	2^n
10	3 ns	10 ns	33 ns	100 ns	1 μ s
1 000	10 ns	1 μ s	10 μ s	1 ms	∞ ($\approx 10^{284}$ years)
10^6	20 ns	1 ms	20 ms	17 minutes	∞
10^9	30 ns	1 s	30 s	32 years	∞

A faster machine hardly helps: a **10× faster** machine turns 32 years into 3.2 years. A **better algorithm** is the solution.

Big-O notation

Intuitively: $f(n) = O(g(n))$ if **f** grows at most as fast as **g** (ignoring a constant factor).

Precisely: $f(n) = O(g(n))$ if there are constants **c** > 0 and **n₀** such that

$$f(n) \leq c \cdot g(n) \text{ for every } n \geq n_0.$$

Rules:

- Constant factors can be dropped: $5n = O(n)$
- Only the highest-order term counts: $3n^2 + 10n + 7 = O(n^2)$
- The base of the logarithm does not matter: $\log_2 n = O(\log n)$

Related notations: Ω (lower bound), Θ (exact order).

Common orders of growth

Notation	Name	Example
$O(1)$	constant	accessing an array element by index, stack <code>push</code>
$O(\log n)$	logarithmic	binary search
$O(n)$	linear	linear search, sum, maximum
$O(n \log n)$	"n log n"	merge sort, quicksort (on average)
$O(n^2)$	quadratic	bubble sort, examining every pair
$O(2^n)$	exponential	examining every subset

Rule of thumb: for $n = 10^6$ elements, $O(n \log n)$ is perfectly fine, $O(n^2)$ is not.

Determining the order from code

```
// O(n)
for (int i = 0; i < n; i++)
    sum += a[i];

// O(n²)
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        count++;
```

```
// O(log n) – halving
while (n > 1)
    n = n / 2;

// O(n) + O(n²) = O(n²)
for (int i = 0; i < n; i++) ...
for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++) ...
```

- **In sequence:** the costs add up → the larger one counts
- **Nested:** the costs multiply
- **Halving:** if every step halves the problem → $\log n$

Space complexity

Not only time but **memory** is a resource too.

- **Space complexity:** the memory needed in addition to the input, as a function of n
- Linear and binary search: **$O(1)$** – only a few helper variables
- Making a copy of an array: **$O(n)$**

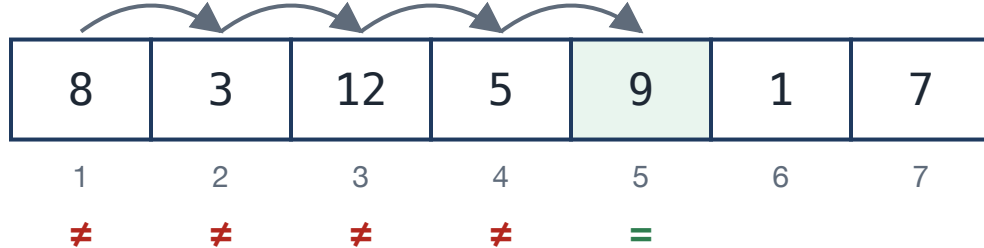
We often have to choose **between time and space**: with more memory (e.g. a hash table, week 11) searching can be faster.



Searching in an array

Linear (sequential) search

Search for $x = 9$



5 comparisons

worst case: n

- Going from the front, we compare every element with the value we are looking for
- The array does **not** need to be sorted
- Also works on a linked list (sequential access)

Linear search – algorithm

Pseudocode

```
Function LinearSearch(A, n, x)
  i ← 1
  While i ≤ n and A[i] ≠ x
    i ← i + 1
  End while
  If i ≤ n then
    Return i
  else
    Return 0      // not found
  End if
End function
```

Steps: $O(n)$ in the worst case

Java

```
static int linearSearch(int[] a, int x) {
    int i = 0;
    while (i < a.length && a[i] != x)
        i++;
    return i < a.length ? i : -1;
}
```

Go

```
func linearSearch(a []int, x int) int {
    for i, v := range a {
        if v == x {
            return i
        }
    }
    return -1
}
```

Attention: the order of the conditions!

```
While  $i \leq n$  and  $A[i] \neq x$ 
```

- If $i = n + 1$, then $A[i]$ is a **non-existent** element
- Thanks to **short-circuit evaluation** this is still correct: if $i \leq n$ is false, $A[i] \neq x$ is not evaluated

In the reverse order ($A[i] \neq x$ and $i \leq n$) we get an `ArrayIndexOutOfBoundsException` in Java and an `index out of range` error in Go.

Sentinel search

Idea: put the value we are looking for at the end of the array (position $n + 1$) – then we will certainly find it, and the index bound need not be checked.

Pseudocode

```
Function SentinelSearch(A, n, x)
  A[n + 1] ← x           // sentinel
  i ← 1
  While A[i] ≠ x
    i ← i + 1
  End while
  If i ≤ n then Return i
  else Return 0
End function
```

- **One** comparison per step instead of two
- Still **$O(n)$** , but about half as many comparisons
- Needs one **extra slot** at the end of the array

Variant	x = 4 not present, n = 7
simple	15 comparisons
sentinel	8 comparisons

We will use the sentinel idea again for the sentinel list in week 10.

Searching a sorted array – linearly

If the array is **sorted**, we can stop as soon as we reach an element **greater** than `x`.

```
Function SortedLinear(A, n, x)
  i ← 1
  While i ≤ n and A[i] < x
    i ← i + 1
  End while
  If i ≤ n and A[i] = x then Return i
  else Return 0
End function
```

- An unsuccessful search is faster on average
- But the worst case is still **O(n)** – can we do better?

Binary search – the idea

Precondition: the array is **sorted**.

1. Look at the **middle** element
2. If it is the one we want: done
3. If the value we want is **smaller**: continue in the left half
4. If the value we want is **larger**: continue in the right half

In every step the search range is **halved**.

Like looking up a word in a dictionary (or a name in a phone book): we do not turn every page, we open it in the middle and decide which way to go.

Binary search – example

Search for $x = 23$

step 1

1	2	3	4	5	6	7	8	9	10
2	5	8	12	16	23	38	56	72	91

lo mid hi

$A[5] = 16 < 23 \rightarrow lo \leftarrow 6$

step 2

2	5	8	12	16	23	38	56	72	91
---	---	---	----	----	----	----	----	----	----

lo mid hi

$A[8] = 56 > 23 \rightarrow hi \leftarrow 7$

step 3

2	5	8	12	16	23	38	56	72	91
---	---	---	----	----	----	----	----	----	----

lo=mid hi

$A[6] = 23 = x \checkmark$

Binary search – algorithm

Pseudocode

```
Function BinarySearch(A, n, x)
  lo ← 1
  hi ← n
  While lo ≤ hi
    k ← (lo + hi) div 2
    If A[k] = x then
      Return k
    else if A[k] < x then
      lo ← k + 1
    else
      hi ← k - 1
    End if
  End while
  Return 0
End function
```

Java

```
static int binarySearch(int[] a, int x) {
    int lo = 0, hi = a.length - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] == x) return mid;
        if (a[mid] < x) lo = mid + 1;
        else hi = mid - 1;
    }
    return -1;
}
```

Go

```
for lo <= hi {
    mid := lo + (hi-lo)/2
    switch {
    case a[mid] == x: return mid
    case a[mid] < x:  lo = mid + 1
    default:         hi = mid - 1
    }
}
```

Binary search – efficiency

After every step the size of the range is at most half: $n \rightarrow n/2 \rightarrow n/4 \rightarrow \dots \rightarrow 1$

At most $\lfloor \log_2 n \rfloor + 1$ steps $\rightarrow O(\log n)$

n	linear (worst case)	binary (worst case)
10	10	4
1 000	1 000	10
1 000 000	1 000 000	20
1 000 000 000	1 000 000 000	30

Finding a name in a sorted list of the whole population of China (1.4 billion): **at most 31 steps.**

Binary search – typical mistakes

- The array is **not sorted** → wrong result (but no error message!)
- `lo < hi` instead of `lo ≤ hi` → the one-element range is skipped
- `lo ← k` instead of `lo ← k + 1` → possible **infinite loop**
- `(lo + hi) / 2` → **overflow** for large indexes
instead: `lo + (hi - lo) / 2`

This bug was present for years in the `Arrays.binarySearch` function of the Java library (fixed in 2006).

Built-in versions: Java `Arrays.binarySearch(a, x)` · Go `sort.SearchInts(a, x)`

Linear or binary?

	Linear	Binary
Precondition	none	sorted array
Access	sequential is enough	direct access needed
Steps	$O(n)$	$O(\log n)$
On a linked list	✓	✗ (no direct access)

If we search the same data many times, it pays to **sort once** ($O(n \log n)$, week 13); after that every search is $O(\log n)$.

Summary

- Efficiency is measured by the **growth of the number of steps** as a function of the input size
- Usually we examine the **worst case**
- **Big-O:** constants and lower-order terms can be dropped
- Orders: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$
- **Linear search:** $O(n)$, on any array · with a sentinel, fewer comparisons
- **Binary search:** $O(\log n)$, only on a **sorted** array with direct access

Next week: arrays – one- and multidimensional arrays, storage, address calculation, dynamic arrays.

Review questions

1. What is the order (in big-O) of $3n^2 + 100n + 5$? And of $n + \log n$?
2. True or false: binary search is $O(n)$.
3. True or false: binary search also works in $O(\log n)$ steps on a linked list.
4. At most how many steps does binary search need to find an element in a 500-element array?
5. What happens if we run binary search on an unsorted array?
6. What is the order of the following code?

```
for (i = 1; i < n; i = i * 2) for (j = 0; j < n; j++) ...
```



Practice

Demo programs

Program	Java	Go	What does it show?
Linear search	<code>LinearSearch.java</code>	<code>w02/linearsearch</code>	simple, sentinel and sorted variants; number of comparisons
Binary search	<code>BinarySearch.java</code>	<code>w02/binarysearch</code>	printing the steps, table of worst cases
Timing	<code>Timing.java</code>	<code>w02/timing</code>	$O(n)$ vs. $O(\log n)$ vs. $O(n^2)$ in reality

Run the `Timing` program and observe: when n grows tenfold, how do the times of linear and binary search change? When n doubles, how does the time of the $O(n^2)$ algorithm change?

Exercises

1. **Maximum:** return the index of the largest element of the array! How many steps does it take?
2. **Counting:** how many elements of the array are equal to `x` ?
3. **First occurrence:** in a sorted array where a value may occur **several times**, find the **first** occurrence of `x` in $O(\log n)$ steps! (`[1,2,2,2,5]` , `x=2` $\rightarrow$ `1`)
4. **Insertion position:** return the index where `x` can be inserted so that the array stays sorted!
5. **Integer square root:** using binary search, find the largest `r` with $r^2 \leq n$!
6. **Guess the number:** the computer thinks of a number between 1 and 1000, we get “smaller/larger” answers. At most how many questions are needed?

Solutions: `Exercises02.java` , `w02/exercises`

Homework

1. Give the order (in big-O)!

a) `for i ← 1 to n: for j ← 1 to 10: ...`

b) `for i ← 1 to n: for j ← 1 to i: ...`

c) `i ← n; While i > 0: i ← i div 3`

2. Run binary search on paper on the array `[3, 8, 15, 21, 30, 42, 57, 66, 81]` for `x = 57` and `x = 10` ! Write down the values of `lo`, `hi` and `k` in every step!

3. Write a function that counts how many times `x` occurs in a **sorted** array, in $O(\log n)$ steps!