Tutorial 1

Algorithm, its description, its complexity

Jan Bures

18YZALG – Basics of Algorithmization, Summer Semester 2026

Instructor & contact

Jan Bures

buresj11@cvut.cz

Room T-111

buresj.cz

Feel free to email questions or setup after-class consultations.

Basic info

- T-214, Monday, 12PM
- 2 unexcused absences are allowed.
- From the 3rd absence onward a valid justification is required.

Course rhythm

- Lecture: concepts + demonstrations + general ideas
- Tutorials: try to put things into practice, not necessarily 100% 1-to-1 with lecture, think of it as augmentation
- Tutorials alternate: one week you relax / the other week I relax
- You will test your knowledge with group assignments
- Group assignments accumulate (50% of final grade)
- Fail to hand in an assignment = 0% from that assignment

Course rhythm

- Lecture: concepts + demonstrations + general ideas
- Tutorials: try to put things into practice, not necessarily 100% 1-to-1 with lecture, think of it as augmentation
- Tutorials alternate: one week you relax / the other week I relax
- You will test your knowledge with group assignments
- Group assignments accumulate (50% of final grade)
- Fail to hand in an assignment = 0% from that assignment

Course rhythm

- Lecture: concepts + demonstrations + general ideas
- Tutorials: try to put things into practice, not necessarily 100% 1-to-1 with lecture, think of it as augmentation
- Tutorials alternate: one week you relax / the other week I relax
- You will test your knowledge with group assignments
- Group assignments accumulate (50% of final grade)
- Fail to hand in an assignment = 0% from that assignment

Course rhythm

- Lecture: concepts + demonstrations + general ideas
- Tutorials: try to put things into practice, not necessarily 100% 1-to-1 with lecture, think of it as augmentation
- Tutorials alternate: one week you relax / the other week I relax
- You will test your knowledge with group assignments
- Group assignments accumulate (50% of final grade)
- Fail to hand in an assignment = 0% from that assignment

Course rhythm

- Lecture: concepts + demonstrations + general ideas
- Tutorials: try to put things into practice, not necessarily 100% 1-to-1 with lecture, think of it as augmentation
- Tutorials alternate: one week you relax / the other week I relax
- You will test your knowledge with group assignments
- Group assignments accumulate (50% of final grade)
- Fail to hand in an assignment = 0% from that assignment

Course rhythm

- Lecture: concepts + demonstrations + general ideas
- Tutorials: try to put things into practice, not necessarily 100% 1-to-1 with lecture, think of it as augmentation
- Tutorials alternate: one week you relax / the other week I relax
- You will test your knowledge with group assignments
- Group assignments accumulate (50% of final grade)
- Fail to hand in an assignment = 0% from that assignment

Today

- What an algorithm is (and is not)
- How to describe an algorithm clearly
- What algorithmic complexity means in practice
- Two short case studies: searching and duplicates

Today

- What an algorithm is (and is not)
- How to describe an algorithm clearly
- What algorithmic complexity means in practice
- Two short case studies: searching and duplicates

Today

- What an algorithm is (and is not)
- How to describe an algorithm clearly
- What algorithmic complexity means in practice
- Two short case studies: searching and duplicates

Today

- What an algorithm is (and is not)
- How to describe an algorithm clearly
- What algorithmic complexity means in practice
- Two short case studies: searching and duplicates

A familiar situation

- You implement a solution and test it on a toy dataset ($\sim 10^2$ items).
- Then the real input arrives ($\sim 10^5$ items) and everything slows down or crashes.
- Typical symptoms: minutes instead of seconds, high CPU, memory spikes, timeouts.
- The root cause is usually **algorithmic scaling** (not “bad luck” or just a slow machine).

Takeaway

Performance is about **how runtime grows with** n, not just whether it runs on small inputs.

A familiar situation

- You implement a solution and test it on a toy dataset ($\sim 10^2$ items).
- Then the real input arrives ($\sim 10^5$ items) and everything slows down or crashes.
- Typical symptoms: minutes instead of seconds, high CPU, memory spikes, timeouts.
- The root cause is usually **algorithmic scaling** (not "bad luck" or just a slow machine).

Takeaway

Performance is about **how runtime grows with** n, not just whether it runs on small inputs.

A familiar situation

- You implement a solution and test it on a toy dataset ($\sim 10^2$ items).
- Then the real input arrives ($\sim 10^5$ items) and everything slows down or crashes.
- Typical symptoms: minutes instead of seconds, high CPU, memory spikes, timeouts.
- The root cause is usually **algorithmic scaling** (not "bad luck" or just a slow machine).

Takeaway

Performance is about **how runtime grows with** n, not just whether it runs on small inputs.

A familiar situation

- You implement a solution and test it on a toy dataset ($\sim 10^2$ items).
- Then the real input arrives ($\sim 10^5$ items) and everything slows down or crashes.
- Typical symptoms: minutes instead of seconds, high CPU, memory spikes, timeouts.
- The root cause is usually **algorithmic scaling** (not "bad luck" or just a slow machine).

Takeaway

Performance is about **how runtime grows with** n, not just whether it runs on small inputs.

A familiar situation

- You implement a solution and test it on a toy dataset ($\sim 10^2$ items).
- Then the real input arrives ($\sim 10^5$ items) and everything slows down or crashes.
- Typical symptoms: minutes instead of seconds, high CPU, memory spikes, timeouts.
- The root cause is usually **algorithmic scaling** (not “bad luck” or just a slow machine).

Takeaway

Performance is about **how runtime grows with** n, not just whether it runs on small inputs.

A familiar situation

- You implement a solution and test it on a toy dataset ($\sim 10^2$ items).
- Then the real input arrives ($\sim 10^5$ items) and everything slows down or crashes.
- Typical symptoms: minutes instead of seconds, high CPU, memory spikes, timeouts.
- The root cause is usually **algorithmic scaling** (not "bad luck" or just a slow machine).

Takeaway

Performance is about **how runtime grows with** n, not just whether it runs on small inputs.

Goal of algorithmization

- **Predict behavior before you run:** how runtime and memory grow with input size n.
- **Design intentionally:** choose an algorithm and data structure that match the task.
- **Make correctness explicit:** specify inputs/outputs and state what "correct" means.
- **Reason about cost:** identify the bottleneck and estimate complexity ($O(\cdot)$) with constants in mind.
- **Communicate and defend decisions:** explain why your solution is appropriate under given constraints (time limits, memory limits, data properties).

In one sentence

Algorithmization means turning an idea into a **correct, efficient, and explainable procedure**.

Goal of algorithmization

- **Predict behavior before you run:** how runtime and memory grow with input size n.
- **Design intentionally:** choose an algorithm and data structure that match the task.
- **Make correctness explicit:** specify inputs/outputs and state what "correct" means.
- **Reason about cost:** identify the bottleneck and estimate complexity ($O(\cdot)$) with constants in mind.
- **Communicate and defend decisions:** explain why your solution is appropriate under given constraints (time limits, memory limits, data properties).

In one sentence

Algorithmization means turning an idea into a **correct, efficient, and explainable procedure**.

Goal of algorithmization

- **Predict behavior before you run:** how runtime and memory grow with input size n.
- **Design intentionally:** choose an algorithm and data structure that match the task.
- **Make correctness explicit:** specify inputs/outputs and state what "correct" means.
- **Reason about cost:** identify the bottleneck and estimate complexity ($O(\cdot)$) with constants in mind.
- **Communicate and defend decisions:** explain why your solution is appropriate under given constraints (time limits, memory limits, data properties).

In one sentence

Algorithmization means turning an idea into a **correct, efficient, and explainable procedure**.

Goal of algorithmization

- **Predict behavior before you run:** how runtime and memory grow with input size n.
- **Design intentionally:** choose an algorithm and data structure that match the task.
- **Make correctness explicit:** specify inputs/outputs and state what "correct" means.
- **Reason about cost:** identify the bottleneck and estimate complexity ($O(\cdot)$) with constants in mind.
- **Communicate and defend decisions:** explain why your solution is appropriate under given constraints (time limits, memory limits, data properties).

In one sentence

Algorithmization means turning an idea into a **correct, efficient, and explainable procedure**.

Goal of algorithmization

- **Predict behavior before you run:** how runtime and memory grow with input size n.
- **Design intentionally:** choose an algorithm and data structure that match the task.
- **Make correctness explicit:** specify inputs/outputs and state what "correct" means.
- **Reason about cost:** identify the bottleneck and estimate complexity ($O(\cdot)$) with constants in mind.
- **Communicate and defend decisions:** explain why your solution is appropriate under given constraints (time limits, memory limits, data properties).

In one sentence

Algorithmization means turning an idea into a **correct, efficient, and explainable procedure**.

Goal of algorithmization

- **Predict behavior before you run:** how runtime and memory grow with input size n.
- **Design intentionally:** choose an algorithm and data structure that match the task.
- **Make correctness explicit:** specify inputs/outputs and state what "correct" means.
- **Reason about cost:** identify the bottleneck and estimate complexity ($O(\cdot)$) with constants in mind.
- **Communicate and defend decisions:** explain why your solution is appropriate under given constraints (time limits, memory limits, data properties).

In one sentence

Algorithmization means turning an idea into a **correct, efficient, and explainable procedure**.

Goal of algorithmization

- **Predict behavior before you run:** how runtime and memory grow with input size n.
- **Design intentionally:** choose an algorithm and data structure that match the task.
- **Make correctness explicit:** specify inputs/outputs and state what "correct" means.
- **Reason about cost:** identify the bottleneck and estimate complexity ($O(\cdot)$) with constants in mind.
- **Communicate and defend decisions:** explain why your solution is appropriate under given constraints (time limits, memory limits, data properties).

In one sentence

Algorithmization means turning an idea into a **correct, efficient, and explainable procedure**.

Working definition

Algorithm

A **finite**, **precise** procedure that solves a **class of problems**.

Key properties

- Has well-defined **inputs** and **outputs**
- Consists of explicit **steps** (a recipe you can follow)
- Is **language-independent**

Working definition

Algorithm

A **finite**, **precise** procedure that solves a **class of problems**.

Key properties

- Has well-defined **inputs** and **outputs**
- Consists of explicit **steps** (a recipe you can follow)
- Is **language-independent**

Working definition

Algorithm

A **finite**, **precise** procedure that solves a **class of problems**.

Key properties

- Has well-defined **inputs** and **outputs**
- Consists of explicit **steps** (a recipe you can follow)
- Is **language-independent**

Working definition

Algorithm

A **finite**, **precise** procedure that solves a **class of problems**.

Key properties

- Has well-defined **inputs** and **outputs**
- Consists of explicit **steps** (a recipe you can follow)
- Is **language-independent**

Three levels

1) Problem

What do we want? Define inputs/outputs, constraints, and what "correct" means.

2) Algorithm

The method. A language-independent procedure: the key idea + data structures + steps.

3) Implementation

Concrete code. One specific realization in a programming language, with engineering details (I/O, libraries, edge cases, performance tuning).

Executor matters

- Humans can fill gaps (but slowly)
- CPUs follow instructions literally (but fast)
- So algorithm descriptions must be unambiguous

Executor matters

- Humans can fill gaps (but slowly)
- CPUs follow instructions literally (but fast)
- So algorithm descriptions must be unambiguous

Executor matters

- Humans can fill gaps (but slowly)
- CPUs follow instructions literally (but fast)
- So algorithm descriptions must be unambiguous

Quick question

- Name an algorithm from everyday life
- What are its inputs and outputs?

Description template

1. Specification: inputs, outputs, assumptions
2. Small example + edge cases
3. Procedure: pseudocode
4. Complexity: time + memory
5. Correctness note: key argument or invariant

Description template

1. Specification: inputs, outputs, assumptions
2. Small example + edge cases
3. Procedure: pseudocode
4. Complexity: time + memory
5. Correctness note: key argument or invariant

Description template

1. Specification: inputs, outputs, assumptions
2. Small example + edge cases
3. Procedure: pseudocode
4. Complexity: time + memory
5. Correctness note: key argument or invariant

Description template

1. Specification: inputs, outputs, assumptions
2. Small example + edge cases
3. Procedure: pseudocode
4. Complexity: time + memory
5. Correctness note: key argument or invariant

Description template

1. Specification: inputs, outputs, assumptions
2. Small example + edge cases
3. Procedure: pseudocode
4. Complexity: time + memory
5. Correctness note: key argument or invariant

Clarity checklist

- Avoid: phrases like “do it efficiently” without details
- Name the data structure you use
- State what changes each step (progress)
- Make termination obvious

Clarity checklist

- Avoid: phrases like "do it efficiently" without details
- Name the data structure you use
- State what changes each step (progress)
- Make termination obvious

Clarity checklist

- Avoid: phrases like “do it efficiently” without details
- Name the data structure you use
- State what changes each step (progress)
- Make termination obvious

Clarity checklist

- Avoid: phrases like "do it efficiently" without details
- Name the data structure you use
- State what changes each step (progress)
- Make termination obvious

Worked example

- Problem: maximum of a non-empty list
- We will do: spec $\rightarrow$ pseudocode $\rightarrow$ correctness $\rightarrow$ complexity

Maximum

Specification

```
Problem MAXIMUM
Input: non-empty list A of numbers
Output: max value in A
```

Pseudocode

```
Algorithm MAXIMUM(A):
   m = A[0]
   for each x in A[1:]:
      if x > m:
         m = x
   return m
```

Maximum

Specification

```
Problem MAXIMUM
Input: non-empty list A of numbers
Output: max value in A
```

Pseudocode

```
Algorithm MAXIMUM(A):
   m = A[0]
   for each x in A[1:]:
      if x > m:
         m = x
   return m
```

Maximum (Python)

Implementation

```
def maximum(A):
    """Return the maximum element of a non-empty list A."""
    m = A[0]
    for x in A[1:]:
        if x > m:
            m = x
    return m
```

Correctness

- Correct for all valid inputs (not only examples)
- We want a short argument, not hand-waving
- Tool for loop-based algorithms: **loop invariant**

Loop invariant (practical meaning)

A **loop invariant** is a statement that is true **before** the loop starts, stays true **after every iteration**, and together with the loop ending condition implies the result is correct.

Correctness

- Correct for all valid inputs (not only examples)
- We want a short argument, not hand-waving
- Tool for loop-based algorithms: **loop invariant**

Loop invariant (practical meaning)

A **loop invariant** is a statement that is true **before** the loop starts, stays true **after every iteration**, and together with the loop ending condition implies the result is correct.

Correctness

- Correct for all valid inputs (not only examples)
- We want a short argument, not hand-waving
- Tool for loop-based algorithms: **loop invariant**

Loop invariant (practical meaning)

A **loop invariant** is a statement that is true **before** the loop starts, stays true **after every iteration**, and together with the loop ending condition implies the result is correct.

Correctness

- Correct for all valid inputs (not only examples)
- We want a short argument, not hand-waving
- Tool for loop-based algorithms: **loop invariant**

Loop invariant (practical meaning)

A **loop invariant** is a statement that is true **before** the loop starts, stays true **after every iteration**, and together with the loop ending condition implies the result is correct.

Correctness

- Correct for all valid inputs (not only examples)
- We want a short argument, not hand-waving
- Tool for loop-based algorithms: **loop invariant**

Loop invariant (practical meaning)

A **loop invariant** is a statement that is true **before** the loop starts, stays true **after every iteration**, and together with the loop ending condition implies the result is correct.

Invariant for maximum

Loop invariant in our maximum case

After processing k elements, m equals the maximum of those k elements.

Two questions

- Is it correct?

Correctness via loop invariant (example: maximum in an array)

Invariant: after processing k elements, $m = \max(A[0], \ldots, A[k-1])$.

- **Initialization:** set $m \leftarrow A[0] \Rightarrow$ true for $k = 1$.
- **Maintenance:** for next element x: if $x \leq m$, keep m; else set $m \leftarrow x \Rightarrow$ invariant stays true.
- **Termination:** after n elements, invariant gives $m = \max(A[0], \ldots, A[n-1])$.

- How expensive is it as n grows?

Two questions

- Is it correct?

Correctness via loop invariant (example: maximum in an array)

Invariant: after processing k elements, $m = \max(A[0], \ldots, A[k-1])$.

- **Initialization:** set $m \leftarrow A[0] \Rightarrow$ true for $k = 1$.
- **Maintenance:** for next element x: if $x \leq m$, keep m; else set $m \leftarrow x \Rightarrow$ invariant stays true.
- **Termination:** after n elements, invariant gives $m = \max(A[0], \ldots, A[n-1])$.

- How expensive is it as n grows?

Two questions

- Is it correct?

Correctness via loop invariant (example: maximum in an array)

Invariant: after processing k elements, $m = \max(A[0], \ldots, A[k-1])$.

- **Initialization:** set $m \leftarrow A[0] \Rightarrow$ true for $k = 1$.
- **Maintenance:** for next element x: if $x \leq m$, keep m; else set $m \leftarrow x \Rightarrow$ invariant stays true.
- **Termination:** after n elements, invariant gives $m = \max(A[0], \ldots, A[n-1])$.

- How expensive is it as n grows?

Two questions

- Is it correct?

Correctness via loop invariant (example: maximum in an array)

Invariant: after processing k elements, $m = \max(A[0], \ldots, A[k-1])$.

- **Initialization:** set $m \leftarrow A[0] \Rightarrow$ true for $k = 1$.
- **Maintenance:** for next element x: if $x \leq m$, keep m; else set $m \leftarrow x \Rightarrow$ invariant stays true.
- **Termination:** after n elements, invariant gives $m = \max(A[0], \ldots, A[n-1])$.

- How expensive is it as n grows?

Big-O in one sentence

- Big-O describes growth rate, not exact runtime

Example

Nested loop:

```
Pair comparisons:
  for i in 0..n-1:
    for j in i+1..n-1:
      compare
```

Total comparisons $= (n-1) + (n-2) + ... + 1 = n(n-1)/2$

Big-O in one sentence

- Big-O describes growth rate, not exact runtime

Example

Nested loop:

```
Pair comparisons:
   for i in 0..n-1:
      for j in i+1..n-1:
         compare
```

Total comparisons $= (n-1) + (n-2) + ... + 1 = n(n-1)/2$

Big-O in one sentence

- Big-O describes growth rate, not exact runtime

Example

Nested loop:

```
Pair comparisons:
   for i in 0..n-1:
      for j in i+1..n-1:
         compare
```

Total comparisons $= (n-1) + (n-2) + \ldots + 1 = n(n-1)/2$

Big-O in one sentence

- Big-O describes growth rate, not exact runtime

Example

Nested loop:

```
Pair comparisons:
   for i in 0..n-1:
      for j in i+1..n-1:
         compare
```

Total comparisons $= (n-1) + (n-2) + \ldots + 1 = n(n-1)/2$

Maximum: time complexity

Maximum: pseudocode

```
Algorithm MAXIMUM(A):
   m = A[0]
   for each x in A[1:]:
      if x > m:
         m = x
   return m
```

- One pass over n elements
- About $n - 1$ comparisons
- Time: $\mathcal{O}(n)$

Maximum: time complexity

Maximum: pseudocode

```
Algorithm MAXIMUM(A):
   m = A[0]
   for each x in A[1:]:
      if x > m:
         m = x
   return m
```

- One pass over n elements
- About $n - 1$ comparisons
- Time: $\mathcal{O}(n)$

Maximum: time complexity

Maximum: pseudocode

```
Algorithm MAXIMUM(A):
   m = A[0]
   for each x in A[1:]:
      if x > m:
         m = x
   return m
```

- One pass over n elements
- About $n - 1$ comparisons
- Time: $\mathcal{O}(n)$

Memory complexity

- Extra memory beyond the input
- Maximum uses only a few variables $\rightarrow \mathcal{O}(1)$
- Set-based methods often use $\mathcal{O}(n)$ memory

Memory complexity

- Extra memory beyond the input
- Maximum uses only a few variables $\rightarrow \mathcal{O}(1)$
- Set-based methods often use $\mathcal{O}(n)$ memory

Memory complexity

- Extra memory beyond the input
- Maximum uses only a few variables $\rightarrow \mathcal{O}(1)$
- Set-based methods often use $\mathcal{O}(n)$ memory

Different cases

- Best-case: easiest inputs
- Worst-case: hardest inputs
- Average-case: typical behavior (needs a distribution to determine, what inputs are "typical")

Different cases

- Best-case: easiest inputs
- Worst-case: hardest inputs
- Average-case: typical behavior (needs a distribution to determine, what inputs are "typical")

Different cases

- Best-case: easiest inputs
- Worst-case: hardest inputs
- Average-case: typical behavior (needs a distribution to determine, what inputs are "typical")

Vocabulary (optional)

- $\mathcal{O}(f(n))$: upper bound (grows no faster than)
- $\Theta(f(n))$: tight bound (grows like)
- We will mostly use Big-O for practicality

Vocabulary (optional)

- $\mathcal{O}(f(n))$: upper bound (grows no faster than)
- $\Theta(f(n))$: tight bound (grows like)
- We will mostly use Big-O for practicality

Vocabulary (optional)

- $\mathcal{O}(f(n))$: upper bound (grows no faster than)
- $\Theta(f(n))$: tight bound (grows like)
- We will mostly use Big-O for practicality

Case study: membership

- Input: collection of n items
- Query: value x
- Output: True if x is contained, else False

Case study: membership

- Input: collection of n items
- Query: value x
- Output: True if x is contained, else False

Case study: membership

- Input: collection of n items
- Query: value x
- Output: True if x is contained, else False

Linear search

- Scan left to right
- Stop if found
- Worst-case $\mathcal{O}(n)$

Linear search

- Scan left to right
- Stop if found
- Worst-case $\mathcal{O}(n)$

Linear search

- Scan left to right
- Stop if found
- Worst-case $\mathcal{O}(n)$

Binary search idea

- Requires sorted data
- Compare to the middle
- Discard half of the remaining range

Binary search idea

- Requires sorted data
- Compare to the middle
- Discard half of the remaining range

Binary search idea

- Requires sorted data
- Compare to the middle
- Discard half of the remaining range

When does sorting pay off?

- Sort once: $\mathcal{O}(n \log n)$
- Then q queries: $q \cdot \mathcal{O}(\log n)$
- Compare to $q \cdot \mathcal{O}(n)$ for repeated linear scans

When does sorting pay off?

- Sort once: $\mathcal{O}(n \log n)$
- Then q queries: $q \cdot \mathcal{O}(\log n)$
- Compare to $q \cdot \mathcal{O}(n)$ for repeated linear scans

When does sorting pay off?

- Sort once: $\mathcal{O}(n \log n)$
- Then q queries: $q \cdot \mathcal{O}(\log n)$
- Compare to $q \cdot \mathcal{O}(n)$ for repeated linear scans

Python tools: bisect and set membership

Code

```
import bisect
# Binary search in Python uses bisect on a sorted list
i = bisect.bisect_left(S_sorted, x)
found = (i < len(S_sorted) and S_sorted[i] == x)

# Set membership is average O(1)
found2 = (x in S_set)
```

Duplicates = repeated membership

- Scan items one by one
- Ask: have I seen this already?
- This suggests a set-based approach

Duplicates = repeated membership

- Scan items one by one
- Ask: have I seen this already?
- This suggests a set-based approach

Duplicates = repeated membership

- Scan items one by one
- Ask: have I seen this already?
- This suggests a set-based approach

Method 1: pairwise

- Compare all pairs ($i < j$)
- Time $\mathcal{O}(n^2)$
- Extra memory $\mathcal{O}(1)$

Pseudocode

```
Algorithm HAS-DUPLICATE-PAIRS(A):
   for i in 0..n-1:
      for j in i+1..n-1:
         if A[i] == A[j]:
            return True
         return False
```

Method 1: pairwise

- Compare all pairs ($i < j$)
- Time $\mathcal{O}(n^2)$
- Extra memory $\mathcal{O}(1)$

Pseudocode

```
Algorithm HAS-DUPLICATE-PAIRS(A):
   for i in 0..n-1:
      for j in i+1..n-1:
         if A[i] == A[j]:
            return True
         return False
```

Method 1: pairwise

- Compare all pairs ($i < j$)
- Time $\mathcal{O}(n^2)$
- Extra memory $\mathcal{O}(1)$

Pseudocode

```
Algorithm HAS-DUPLICATE-PAIRS(A):
   for i in 0..n-1:
      for j in i+1..n-1:
         if A[i] == A[j]:
            return True
         return False
```

Method 2: sort + scan

- Sort a copy of the list
- Duplicates become adjacent
- Time $\mathcal{O}(n \log n)$
- Extra memory often $\mathcal{O}(n)$

Pseudocode

```
Algorithm HAS-DUPLICATE-SORT(A):
   B = sorted copy of A
   for k in 1..n-1:
      if B[k] == B[k-1]:
         return True
      return False
```

Method 2: sort + scan

- Sort a copy of the list
- Duplicates become adjacent
- Time $\mathcal{O}(n \log n)$
- Extra memory often $\mathcal{O}(n)$

Pseudocode

```
Algorithm HAS-DUPLICATE-SORT(A):
   B = sorted copy of A
   for k in 1..n-1:
      if B[k] == B[k-1]:
         return True
      return False
```

Method 2: sort + scan

- Sort a copy of the list
- Duplicates become adjacent
- Time $\mathcal{O}(n \log n)$
- Extra memory often $\mathcal{O}(n)$

Pseudocode

```
Algorithm HAS-DUPLICATE-SORT(A):
   B = sorted copy of A
   for k in 1..n-1:
      if B[k] == B[k-1]:
         return True
      return False
```

Method 2: sort + scan

- Sort a copy of the list
- Duplicates become adjacent
- Time $\mathcal{O}(n \log n)$
- Extra memory often $\mathcal{O}(n)$

Pseudocode

```
Algorithm HAS-DUPLICATE-SORT(A):
   B = sorted copy of A
   for k in 1..n-1:
      if B[k] == B[k-1]:
         return True
      return False
```

Method 3: set scan

- seen = empty set
- Scan: if $x \in$ seen $\to$ duplicate
- Average time $\mathcal{O}(n)$
- Extra memory $\mathcal{O}(n)$

Pseudocode

```
Algorithm HAS-DUPLICATE-SET(A):
   seen = empty set
   for x in A:
      if x in seen:
         return True
      add x to seen
   return False
```

Method 3: set scan

- seen = empty set
- Scan: if $x \in$ seen $\rightarrow$ duplicate
- Average time $\mathcal{O}(n)$
- Extra memory $\mathcal{O}(n)$

Pseudocode

```
Algorithm HAS-DUPLICATE-SET(A):
   seen = empty set
   for x in A:
      if x in seen:
         return True
      add x to seen
   return False
```

Method 3: set scan

- `seen` = empty set
- Scan: if $x \in$ `seen` $\to$ duplicate
- Average time $\mathcal{O}(n)$
- Extra memory $\mathcal{O}(n)$

Pseudocode

```
Algorithm HAS-DUPLICATE-SET(A):
   seen = empty set
   for x in A:
      if x in seen:
         return True
      add x to seen
   return False
```

Method 3: set scan

- `seen` = empty set
- Scan: if $x \in$ `seen` $\rightarrow$ duplicate
- Average time $\mathcal{O}(n)$
- Extra memory $\mathcal{O}(n)$

Pseudocode

```
Algorithm HAS-DUPLICATE-SET(A):
   seen = empty set
   for x in A:
      if x in seen:
         return True
      add x to seen
   return False
```

Choosing an approach

- If memory is tight: consider sort+scan
- If you need speed and can store a set: use set-based
- If n is tiny: simplicity may win

Choosing an approach

- If memory is tight: consider sort+scan
- If you need speed and can store a set: use set-based
- If n is tiny: simplicity may win

Choosing an approach

- If memory is tight: consider sort+scan
- If you need speed and can store a set: use set-based
- If n is tiny: simplicity may win

Benchmarking rules of thumb

- Measure across multiple n (scaling)
- Repeat runs; use median or best-of-k
- Separate data generation from algorithm timing
- Avoid printing in timed sections

Benchmarking rules of thumb

- Measure across multiple n (scaling)
- Repeat runs; use median or best-of-k
- Separate data generation from algorithm timing
- Avoid printing in timed sections

Benchmarking rules of thumb

- Measure across multiple n (scaling)
- Repeat runs; use median or best-of-k
- Separate data generation from algorithm timing
- Avoid printing in timed sections

Benchmarking rules of thumb

- Measure across multiple n (scaling)
- Repeat runs; use median or best-of-k
- Separate data generation from algorithm timing
- Avoid printing in timed sections

A minimal timing harness

Code

```
import time

def time_one(func, data, repeats=5):
    best = float('inf')
    for _ in range(repeats):
        t0 = time.perf_counter()
        func(data)
        t1 = time.perf_counter()
        best = min(best, t1 - t0)
    return best
```

Mini-quiz: complexity reading (1/3)

Snippet 1

```
for i in range(n):
    for j in range(10):
        do_constant_work()
```

Question: What is the time complexity in terms of n?

Mini-quiz: complexity reading (2/3)

Snippet 2

```
k = n
while k > 1:
    k //= 2
    do_constant_work()
```

Question: What is the time complexity in terms of n?

Mini-quiz: complexity reading (3/3)

Snippet 3

```
for i in range(n):
    for j in range(i, n):
        do_constant_work()
```

Question: What is the time complexity in terms of n?

Summary

- Algorithm: precise method for a class of problems
- Describe with a template: spec $\rightarrow$ procedure $\rightarrow$ complexity $\rightarrow$ correctness
- Complexity predicts scaling; data structures matter
- Tutorial: implement, benchmark, then mini-projects

Summary

- Algorithm: precise method for a class of problems
- Describe with a template: spec → procedure → complexity → correctness
- Complexity predicts scaling; data structures matter
- Tutorial: implement, benchmark, then mini-projects

Summary

- Algorithm: precise method for a class of problems
- Describe with a template: spec $\rightarrow$ procedure $\rightarrow$ complexity $\rightarrow$ correctness
- Complexity predicts scaling; data structures matter
- Tutorial: implement, benchmark, then mini-projects

Summary

- Algorithm: precise method for a class of problems
- Describe with a template: spec $\rightarrow$ procedure $\rightarrow$ complexity $\rightarrow$ correctness
- Complexity predicts scaling; data structures matter
- Tutorial: implement, benchmark, then mini-projects