# Chapter 1. Analysis of Algorithms

Data Structures and Algorithms

FIB

Enric Rodríguez
(slides by Antoni Lozano)
Q2 2017–2018

# Chapter 1. Analysis of Algorithms

1. Computation Time
   - Efficiency of algorithms
   - Input size and cost
   - Orders of magnitude

2. Asymptotic notation
   - Asymptotic notation: definitions
   - Asymptotic notation: properties
   - Growth rates

3. Cost of algorithms
   - Iterative algorithms
   - Recursive algorithms
   - Master theorems

1 Computation Time
- Efficiency of algorithms
- Input size and cost
- Orders of magnitude

2 Asymptotic notation
- Asymptotic notation: definitions
- Asymptotic notation: properties
- Growth rates

3 Cost of algorithms
- Iterative algorithms
- Recursive algorithms
- Master theorems

# Efficiency of algorithms

Goals:

- Compare different algorithmic solutions
- Predict the resources to be used by an algorithm
- Improve existing algorithms

# Efficiency of algorithms

Goals:

- Compare different algorithmic solutions
- Predict the resources to be used by an algorithm
- Improve existing algorithms

# Efficiency of algorithms

Goals:

- Compare different algorithmic solutions
- Predict the resources to be used by an algorithm
- Improve existing algorithms

# Efficiency of algorithms

Remarks about efficiency:

- It depends on the input size
- Implementation-dependent factors (machine, compiler) are linear

# Efficiency of algorithms

Remarks about efficiency:

- It depends on the input size
- Implementation-dependent factors (machine, compiler) are linear

# Example 1: selection problem

**Selection Problem**

Given a list of $n$ natural numbers, find out the $k$-th largest one.

**First solution**

Use a vector to sort the numbers in decreasing order and return the $k$-th one.

**Second solution**

Write the $k$ first numbers in a vector $V$ and sort it decreasingly.
Each of the remaining elements is treated as follows:

- if smaller than the $V[k]$, discard it;
- otherwise, place it properly in $V$ and remove the smallest one from it.

Return $V[k]$.

# Example 1: selection problem

**Selection Problem**

Given a list of $n$ natural numbers, find out the $k$-th largest one.

**First solution**

Use a vector to sort the numbers in decreasing order and return the $k$-th one.

**Second solution**

Write the $k$ first numbers in a vector $V$ and sort it decreasingly.
Each of the remaining elements is treated as follows:

- if smaller than the $V[k]$, discard it;
- otherwise, place it properly in $V$ and remove the smallest one from it.

Return $V[k]$.

# Example 1: selection problem

## Selection Problem

Given a list of $n$ natural numbers, find out the $k$-th largest one.

## First solution

Use a vector to sort the numbers in decreasing order and return the $k$-th one.

## Second solution

Write the $k$ first numbers in a vector $V$ and sort it decreasingly.
Each of the remaining elements is treated as follows:

- if smaller than the $V[k]$, discard it;
- otherwise, place it properly in $V$ and remove the smallest one from it.

Return $V[k]$.

# Example 2: infinite wall

**Infinite Wall**

We are facing a wall that is infinite in both directions. We want to find the only door that allows us to cross it, but we know neither its distance nor its direction. Even though it is dark, we have a candle that allows us to see the door once we are close to it.

# Example 2: infinite wall

## First solution

- Move 1 meter forward and go back to the start
- Move 2 meters backwards and go back to the start
- Move 3 meters forward and go back to the start
- Move 4 meters backwards and go back to the start
- ...

# Example 2: infinite wall

## Second solution

- Move 1 meter forward and go back to the start
- Move 2 meters backwards and go back to the start
- Move 4 meters forward and go back to the start
- Move 8 meters backwards and go back to the start
- ...

# Efficiency of algorithms

Given an algorithm $A$ with a set of inputs $\mathcal{A}$, the efficiency or cost of $A$ can be expressed as a function $T : \mathcal{A} \to \mathbb{R}^+$.

But finding $T$ can be extremely difficult and of little practical use. Fortunately, the cost tends to be similar for inputs of the same size.

# Efficiency of algorithms

Given an algorithm $A$ with a set of inputs $\mathcal{A}$, the efficiency or cost of $A$ can be expressed as a function $T : \mathcal{A} \to \mathbb{R}^+$.

But finding $T$ can be extremely difficult and of little practical use. Fortunately, the cost tends to be similar for inputs of the same size.

# Input size and cost

## Size

The size of an input $x$ is the number of symbols needed to encode it. We will denote it by $|x|$.

## Types of inputs

- Natural numbers $\longrightarrow$ encoding in binary / value

$$|27| = 5 \;\; \text{because} \; \langle 27 \rangle = 11011$$

- Lists $\longrightarrow$ number of components

$$|(23, 1, 7, 0, 12, 500, 2, 11)| = 8$$

# Input size and cost

## Size

The size of an input $x$ is the number of symbols needed to encode it. We will denote it by $|x|$.

## Types of inputs

- Natural numbers $\longrightarrow$ encoding in binary / value

$$|27| = 5 \text{ because } \langle 27 \rangle = 11011$$

- Lists $\longrightarrow$ number of components

$$|(23, 1, 7, 0, 12, 500, 2, 11)| = 8$$

# Input size and cost

**Exercise**

Prove that $|\langle x \rangle_2| = \lfloor \log_2 x \rfloor + 1$, where $\langle x \rangle_2$ is the binary encoding of $x$.

*Hint: express x in binary*

$$x = b_{k-1} b_{k-2} \dots b_0$$

*where $b_{k-1} \neq 0$ and compute the min and max of $x$ as a function of $k$.*

# Input size and cost

- **Worst case.** $T_{worst}(n) = \max\{T(x) \mid x \in \mathcal{A} \wedge |x| = n\}$
  Determines limits that the algorithm will not exceed.
- **Average case.** $T_{avg}(n) = \sum_{x \in A, |x|=n} Pr(x)T(x)$, where $Pr(x)$ is the probability of the occurrence of input $x$ in $\mathcal{A}$
  Difficult to compute.
- **Best case.** $T_{best}(n) = \min\{T(x) \mid x \in \mathcal{A} \wedge |x| = n\}$
  Almost useless.

# Input size and cost

- **Worst case.** $T_{worst}(n) = \max\{T(x) \mid x \in \mathcal{A} \wedge |x| = n\}$
  Determines limits that the algorithm will not exceed.
- **Average case.** $T_{avg}(n) = \sum_{x \in A, |x|=n} Pr(x)T(x)$, where $Pr(x)$ is the probability of the occurrence of input $x$ in $\mathcal{A}$
  Difficult to compute.
- **Best case.** $T_{best}(n) = \min\{T(x) \mid x \in \mathcal{A} \wedge |x| = n\}$
  Almost useless.

# Input size and cost

- **Worst case.** $T_{worst}(n) = \max\{T(x) \mid x \in \mathcal{A} \wedge |x| = n\}$
  Determines limits that the algorithm will not exceed.
- **Average case.** $T_{avg}(n) = \sum_{x \in A, |x|=n} Pr(x)T(x)$, where $Pr(x)$ is the probability of the occurrence of input $x$ in $\mathcal{A}$
  Difficult to compute.
- **Best case.** $T_{best}(n) = \min\{T(x) \mid x \in \mathcal{A} \wedge |x| = n\}$
  Almost useless.

Table 1 (Garey/Johnson, *Computers and Intractability*)

Comparison between polynomial and exponential functions.

| cost | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| $n$ | 0.00001 s | 0.00002 s | 0.00003 s | 0.00004 s | 0.00005 s |
| $n^2$ | 0.0001 s | 0.0004 s | 0.0009 s | 0.0016 s | 0.0025 s |
| $n^3$ | 0.001 s | 0.008 s | 0.027 s | 0.064 s | 0.125 s |
| $n^5$ | 0.1 s | 3.2 s | 24.3 s | 1.7 min | 5.2 min |
| $2^n$ | 0.001 s | 1.0 s | 17.9 min | 12.7 days | 35.7 years |
| $3^n$ | 0.059 s | 58 min | 6.5 years | 3855 centuries | $2 \times 10^8$ cent. |

# Orders of magnitude

**Table 2 (Garey/Johnson, *Computers and Intractability*)**

Impact of technological advances on polynomial and exponential algorithms.

| cost | current technology | technology $\times 100$ | technology $\times 1000$ |
|---|---|---|---|
| $n$ | $N_1$ | $100 N_1$ | $1000 N_1$ |
| $n^2$ | $N_2$ | $10 N_2$ | $31.6 N_2$ |
| $n^3$ | $N_3$ | $4.64 N_3$ | $10 N_3$ |
| $2^n$ | $N_4$ | $N_4 + 6.64$ | $N_4 + 9.97$ |
| $3^n$ | $N_5$ | $N_5 + 4.19$ | $N_5 + 6.29$ |

# Orders of magnitude

## Table 3 (R. Sedgewick, *Algorithms in C++*)

In several applications, the only chance to deal with huge inputs is the use of efficient algorithms. A fast algorithm will allow us to solve a problem in a slow machine, but a fast machine does not help when we use a slow algorithm.

| operations per second | problem size 1 million | | | input size $10^3$ millions | | |
|---|---|---|---|---|---|---|
| | $N$ | $N \log N$ | $N^2$ | $N$ | $N \log N$ | $N^2$ |
| $10^6$ | seconds | seconds | weeks | hours | hours | never |
| $10^9$ | instant | instant | hours | seconds | seconds | decades |
| $10^{12}$ | instant | instant | seconds | instant | instant | weeks |

# Orders of magnitude

We need a notation that:

- gives an upper-bound to

$$T_{worst}(n) = \max\{T(x) \mid x \in \mathcal{A} \wedge |x| = n\}.$$

  (we will know that the algorithm will never surpass the bound)

- is independent of constant factors

  (hence, it will not depend on the implementation)

**Big-O notation**

Given a function $g$, $\mathcal{O}(g)$ is the class of functions $f$ that "do not grow faster than $g$". Formally, $f \in \mathcal{O}(g)$ iff there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that

$$\forall n \geq n_0 \quad f(n) \leq c \cdot g(n).$$

Instead of $f \in \mathcal{O}(g)$, we usually write "$f$ is $\mathcal{O}(g)$", as well as $f = \mathcal{O}(g)$.

# Orders of magnitude

We need a notation that:

- gives an upper-bound to

$$T_{worst}(n) = \max\{T(x) \mid x \in \mathcal{A} \wedge |x| = n\}.$$

  (we will know that the algorithm will never surpass the bound)

- is independent of constant factors

  (hence, it will not depend on the implementation)

**Big-O notation**

Given a function $g$, $\mathcal{O}(g)$ is the class of functions $f$ that "do not grow faster than $g$". Formally, $f \in \mathcal{O}(g)$ iff there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that

$$\forall n \geq n_0 \quad f(n) \leq c \cdot g(n).$$

Instead of $f \in \mathcal{O}(g)$, we usually write "$f$ is $\mathcal{O}(g)$", as well as $f = \mathcal{O}(g)$.

# Orders of magnitude

We need a notation that:

- gives an upper-bound to

$$T_{worst}(n) = \max\{T(x) \mid x \in \mathcal{A} \wedge |x| = n\}.$$

  (we will know that the algorithm will never surpass the bound)

- is independent of constant factors
  (hence, it will not depend on the implementation)

**Big-O notation**

Given a function $g$, $\mathcal{O}(g)$ is the class of functions $f$ that "do not grow faster than $g$". Formally, $f \in \mathcal{O}(g)$ iff there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that

$$\forall n \geq n_0 \quad f(n) \leq c \cdot g(n).$$

Instead of $f \in \mathcal{O}(g)$, we usually write "$f$ is $\mathcal{O}(g)$", as well as $f = \mathcal{O}(g)$.

# Orders of magnitude

We need a notation that:

- gives an upper-bound to

$$T_{worst}(n) = \max\{T(x) \mid x \in \mathcal{A} \land |x| = n\}.$$

  (we will know that the algorithm will never surpass the bound)

- is independent of constant factors
  (hence, it will not depend on the implementation)

**Big-O notation**

Given a function $g$, $\mathcal{O}(g)$ is the class of functions $f$ that "do not grow faster than $g$". Formally, $f \in \mathcal{O}(g)$ iff there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that

$$\forall n \geq n_0 \quad f(n) \leq c \cdot g(n).$$

Instead of $f \in \mathcal{O}(g)$, we usually write "$f$ is $\mathcal{O}(g)$", as well as $f = \mathcal{O}(g)$.

# Orders of magnitude

## Example

Let $f(n) = 3n^3 + 5n^2 - 7n + 41$. It is not difficult to prove that $f \in \mathcal{O}(n^3)$.

We only need to find constants $c$ i $n_0$ such that:

$$\forall n \geq n_0 \;\; f(n) \leq cn^3.$$

But $3n^3 + 5n^2 - 7n + 41 \leq 8n^3 + 41$. We choose $c = 9$. Then,

$$8n^3 + 41 \leq 9n^3 \iff 41 \leq n^3,$$

which is true from $n_0 = 4$ onwards. Thus,

$$\forall n \geq 4 \quad f(n) \leq 9n^3,$$

and we conclude that $f(n) = \mathcal{O}(n^3)$ with $c = 9$ and $n_0 = 4$.

(It is indeed easy to find smaller $c$ and $n_0$.)

# Example 1: selection problem

**Selection Problem**

Given a list of $n$ natural numbers, find out the $k$-th largest one.

**First solution**

Use a vector to sort the numbers in decreasing order and return the $k$-th one.

**Second solution**

Write the $k$ first numbers in a vector $V$ and sort it decreasingly.
Each of the remaining elements is treated as follows:

- if smaller than the $V[k]$, discard it;
- otherwise, place it properly in $V$ and remove the smallest one from it.

Return $V[k]$.

# Example 1: selection problem

**Selection Problem**

Given a list of $n$ natural numbers, find out the $k$-th largest one.

**First solution**

Use a vector to sort the numbers in decreasing order and return the $k$-th one.

**Second solution**

Write the $k$ first numbers in a vector $V$ and sort it decreasingly.
Each of the remaining elements is treated as follows:

- if smaller than the $V[k]$, discard it;
- otherwise, place it properly in $V$ and remove the smallest one from it.

Return $V[k]$.

# Example 1: selection problem

## Selection Problem

Given a list of $n$ natural numbers, find out the $k$-th largest one.

## First solution

Use a vector to sort the numbers in decreasing order and return the $k$-th one.

## Second solution

Write the $k$ first numbers in a vector $V$ and sort it decreasingly.
Each of the remaining elements is treated as follows:

- if smaller than the $V[k]$, discard it;
- otherwise, place it properly in $V$ and remove the smallest one from it.

Return $V[k]$.

# Example 1: selection problem

**Selection Problem**

Given a list of $n$ natural numbers, find out the $k$-th largest one.

**First Solution**

Use a vector to sort the numbers in decreasing order and return the $k$-th one.

- with a basic sorting algorithm (bubble, insertion): $\mathcal{O}(n^2)$
- with an efficient sorting algorithm: $\mathcal{O}(n \log n)$

# Example 1: selection problem

## Selection Problem

Given a list of $n$ natural numbers, find out the $k$-th largest one.

## Second solution

Write the $k$ first numbers in a vector $V$ and sort it decreasingly.
Each of the remaining elements is treated as follows:

- if smaller than the $V[k]$, discard it;
- otherwise, place it properly in $V$ and remove the smallest one from it.

Return $V[k]$.

$\mathcal{O}((k \log k) + (n - k) \cdot k)$

- If $k$ is constant, we have $\mathcal{O}(k \cdot n) = \mathcal{O}(n)$
- If $k = \lceil n/2 \rceil$, we have $\mathcal{O}(\frac{n}{2} \cdot \frac{n}{2}) = \mathcal{O}(n^2)$

# Example 2: infinite wall

## Infinite Wall

We are facing a wall that is infinite in both directions. We want to find the only door that allows us to cross it, but we know neither its distance nor its direction. Even though it is dark, we have a candle that allows us to see the door once we are close to it.

# Example 2: infinite wall

## First solution

- Move 1 meter forward and go back to the start
- Move 2 meters backwards and go back to the start
- Move 3 meters forward and go back to the start
- Move 4 meters backwards and go back to the start
- ...

Time needed when the door is at distance $n$:

$$T(n) = 2\sum_{i=1}^{n-1} i + n = 2\frac{(n-1)n}{2} + n = n^2 \in \mathcal{O}(n^2).$$

(remember that $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$)

# Example 2: infinite wall

## Second solution

- Move 1 meter forward and go back to the origin
- Move 2 meters backwards and go back to the origin
- Move 4 meters forward and go back to the origin
- Move 8 meters backwards and go back to the origin
- ...

If the door is at distance $n = 2^k$, then

$$T(n) = 2\sum_{i=0}^{k-1} 2^i + 2^k = 2(2^k - 1) + 2^k = 3n - 2 \in \mathcal{O}(n).$$

$$\left(\text{remember that } \sum_{i=0}^{k-1} 2^i = 2^k - 1\right)$$

1 Computation Time
- Efficiency of algorithms
- Input size and cost
- Orders of magnitude

2 Asymptotic notation
- Asymptotic notation: definitions
- Asymptotic notation: properties
- Growth rates

3 Cost of algorithms
- Iterative algorithms
- Recursive algorithms
- Master theorems

# Asymptotic notation: definitions

- Asymptotic notation allows one to classify functions according to their relative growth rate.
- It takes into account the behavior of functions for large inputs.
  For example, $10^6 n > n^2$ up to a certain value $n$ that we can find:
  $$n^2 \geq 10^6 n \iff n \geq 10^6.$$
  Hence, for $n \geq 10^6$, we have that $n^2$ grows faster than $10^6 n$. In this case, we will say that the function $g(n) = 10^6 n$ is asymptotically upper bounded by $f(n) = n^2$.
- The notation $\mathcal{O}(g)$, (big-O) is the set of functions asymptotically upper-bounded by $g$.

# Asymptotic notation: definitions

**$\Theta$ notation ((a): asymptotic exact bound)**

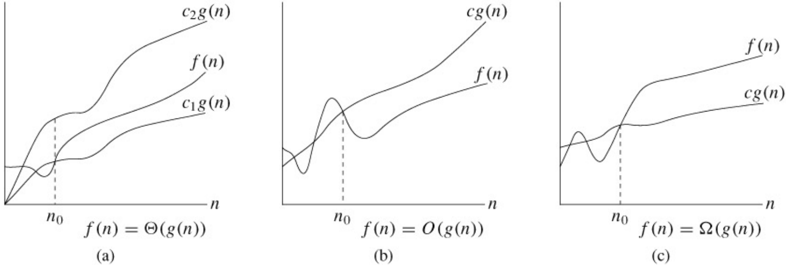
$\Theta(g) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 \ c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

**Big-O notation((b): asymptotic upper bound)**

$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \ \exists n_0 \in \mathbb{N} \ \ \forall n \geq n_0 \quad f(n) \leq c \cdot g(n)\}$

**$\Omega$ notation ((c): asymptotic lower bound)**

$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \ \exists n_0 \in \mathbb{N} \ \ \forall n \geq n_0 \quad f(n) \geq c \cdot g(n)\}$



# $\Theta$ notation

## $\Theta$ notation (asymptotic exact bound)

$\Theta(g) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0 \;\; c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

## Examples

- $75n \in \Theta(n)$
- $1023n^2 \notin \Theta(n)$
- $n^2 \notin \Theta(n)$
- $2^n \notin \Theta(2^{n^2})$
- $\Theta(n) \neq \Theta(n^2)$

# Big-O notation

## Big-O notation (asymptotic upper bound)

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \ \exists n_0 \in \mathbb{N} \ \ \forall n \geq n_0 \quad f(n) \leq c \cdot g(n)\}$$

## Examples

- $3n^2 + 5n - 7 \in \mathcal{O}(n^2)$
- $n + 15 \in \mathcal{O}(n)$
- $\mathcal{O}(n^5) \subseteq \mathcal{O}(n^6)$
- $n^3 \notin \mathcal{O}(n^2)$
- $n^3 \in \mathcal{O}(2^n)$

## Exercise

Prove that $p(n) = 7n^2 + 4n - 2$ is $\mathcal{O}(n^2)$.

# Big-O notation

## Big-O notation (asymptotic upper bound)

$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \ \exists n_0 \in \mathbb{N} \ \ \forall n \geq n_0 \quad f(n) \leq c \cdot g(n)\}$

## Examples

- $3n^2 + 5n - 7 \in \mathcal{O}(n^2)$
- $n + 15 \in \mathcal{O}(n)$
- $\mathcal{O}(n^5) \subseteq \mathcal{O}(n^6)$
- $n^3 \notin \mathcal{O}(n^2)$
- $n^3 \in \mathcal{O}(2^n)$

## Exercise

Prove that $p(n) = 7n^2 + 4n - 2$ is $\mathcal{O}(n^2)$.

# $\Omega$ notation

## $\Omega$ notation ((c): asymptotic lower bound)

$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \ \exists n_0 \in \mathbb{N} \ \ \forall n \geq n_0 \quad f(n) \geq c \cdot g(n)\}$

## Examples

- $2^n \in \Omega(n)$
- $n^2 \in \Omega(n)$
- $n \in \Omega(n)$
- $n \notin \Omega(n^2)$
- $\Omega(n^6) \subseteq \Omega(n^5)$

# Asymptotic notation: properties

## Relationship among $\mathcal{O}$, $\Omega$ and $\Theta$

Given two functions $f$ and $g$:

- $f \in \Omega(g) \iff g \in \mathcal{O}(f)$
- $\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$
- $\mathcal{O}(f) = \mathcal{O}(g) \iff \Omega(f) = \Omega(g) \iff \Theta(f) = \Theta(g)$

# Asymptotic notation: properties

## Limit criteria

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \;\Rightarrow\; f \in \mathcal{O}(g)$ but $g \notin \mathcal{O}(f)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \;\Rightarrow\; g \in \mathcal{O}(f)$ but $f \notin \mathcal{O}(g)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, where $0 < c < \infty \;\Rightarrow\; \mathcal{O}(f) = \mathcal{O}(g)$

## Exercises

1. Let $p$ be a polynomial and $c > 1$. Prove that $p \in \mathcal{O}(c^n)$ but $p \notin \Omega(c^n)$.
2. Let $k \geq 1$. Prove that $\log^k n \in \mathcal{O}(n)$.

# Asymptotic notation: properties

## Limit criteria

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \;\Rightarrow\; f \in \mathcal{O}(g)$ but $g \notin \mathcal{O}(f)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \;\Rightarrow\; g \in \mathcal{O}(f)$ but $f \notin \mathcal{O}(g)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, where $0 < c < \infty \;\Rightarrow\; \mathcal{O}(f) = \mathcal{O}(g)$

## Exercises

1. Let $p$ be a polynomial and $c > 1$. Prove that $p \in \mathcal{O}(c^n)$ but $p \notin \Omega(c^n)$.
2. Let $k \geq 1$. Prove that $\log^k n \in \mathcal{O}(n)$.

# Asymptotic notation: properties

## Big-O properties

Given functions $f$, $f_1$, $f_2$, $g$, $g_1$, $g_2$ and $h$:

- *Reflexivity*. $f \in \mathcal{O}(f)$
- *Transitivity*. $h \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f) \Rightarrow h \in \mathcal{O}(f)$
- *Characterization*. $g \in \mathcal{O}(f) \iff \mathcal{O}(g) \subseteq \mathcal{O}(f)$
- *Sum*. $g_1 \in \mathcal{O}(f_1) \wedge g_2 \in \mathcal{O}(f_2) \Rightarrow g_1 + g_2 \in \mathcal{O}(\max(f_1, f_2))$
- *Product*. $g_1 \in \mathcal{O}(f_1) \wedge g_2 \in \mathcal{O}(f_2) \Rightarrow g_1 \cdot g_2 \in \mathcal{O}(f_1 \cdot f_2)$
- *Multiplicative invariance*. For all constant $c \in \mathbb{R}^+$, $\mathcal{O}(f) = \mathcal{O}(c \cdot f)$

# Asymptotic notation: properties

## Big-O properties

Given functions $f$, $f_1$, $f_2$, $g$, $g_1$, $g_2$ and $h$:

- *Reflexivity*. $f \in \mathcal{O}(f)$
- *Transitivity*. $h \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f) \Rightarrow h \in \mathcal{O}(f)$
- *Characterization*. $g \in \mathcal{O}(f) \iff \mathcal{O}(g) \subseteq \mathcal{O}(f)$
- *Sum*. $g_1 \in \mathcal{O}(f_1) \wedge g_2 \in \mathcal{O}(f_2) \Rightarrow g_1 + g_2 \in \mathcal{O}(\max(f_1, f_2))$
- *Product*. $g_1 \in \mathcal{O}(f_1) \wedge g_2 \in \mathcal{O}(f_2) \Rightarrow g_1 \cdot g_2 \in \mathcal{O}(f_1 \cdot f_2)$
- *Multiplicative invariance*. For all constant $c \in \mathbb{R}^+$, $\mathcal{O}(f) = \mathcal{O}(c \cdot f)$

# Asymptotic notation: properties

## Big-O properties

Given functions $f$, $f_1$, $f_2$, $g$, $g_1$, $g_2$ and $h$:

- *Reflexivity*. $f \in \mathcal{O}(f)$
- *Transitivity*. $h \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f) \Rightarrow h \in \mathcal{O}(f)$
- *Characterization*. $g \in \mathcal{O}(f) \iff \mathcal{O}(g) \subseteq \mathcal{O}(f)$
- *Sum*. $g_1 \in \mathcal{O}(f_1) \wedge g_2 \in \mathcal{O}(f_2) \Rightarrow g_1 + g_2 \in \mathcal{O}(\max(f_1, f_2))$
- *Product*. $g_1 \in \mathcal{O}(f_1) \wedge g_2 \in \mathcal{O}(f_2) \Rightarrow g_1 \cdot g_2 \in \mathcal{O}(f_1 \cdot f_2)$
- *Multiplicative invariance*. For all constant $c \in \mathbb{R}^+$, $\mathcal{O}(f) = \mathcal{O}(c \cdot f)$

# Asymptotic notation: properties

## Big-O properties

Given functions $f$, $f_1$, $f_2$, $g$, $g_1$, $g_2$ and $h$:

- *Reflexivity*. $f \in \mathcal{O}(f)$
- *Transitivity*. $h \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f) \Rightarrow h \in \mathcal{O}(f)$
- *Characterization*. $g \in \mathcal{O}(f) \iff \mathcal{O}(g) \subseteq \mathcal{O}(f)$
- *Sum*. $g_1 \in \mathcal{O}(f_1) \wedge g_2 \in \mathcal{O}(f_2) \Rightarrow g_1 + g_2 \in \mathcal{O}(\max(f_1, f_2))$
- *Product*. $g_1 \in \mathcal{O}(f_1) \wedge g_2 \in \mathcal{O}(f_2) \Rightarrow g_1 \cdot g_2 \in \mathcal{O}(f_1 \cdot f_2)$
- *Multiplicative invariance*. For all constant $c \in \mathbb{R}^+$, $\mathcal{O}(f) = \mathcal{O}(c \cdot f)$

# Asymptotic notation: properties

**Exercise**

Explain why $f \in \mathcal{O}(g)$ is equivalent to

$$\exists c \in \mathbb{R}^+ \ \overset{\infty}{\forall} n \quad f(n) \leq c \cdot g(n).$$

**Note**

Notation $\overset{\infty}{\forall}\, n\, P(n)$ means that $P(n)$ holds for all $n$ except a finite number.

# Asymptotic notation: properties

## $\Theta$ properties

Given functions $f$, $f_1$, $f_2$, $g$, $g_1$, $g_2$ i $h$:

- *Reflexivity*. $f \in \Theta(f)$
- *Transitivity*. $h \in \Theta(g) \wedge g \in \Theta(f) \Rightarrow h \in \Theta(f)$
- *Symmetry*. $g \in \Theta(f) \iff f \in \Theta(g) \iff \Theta(g) = \Theta(f)$
- *Sum*. $g_1 \in \Theta(f_1) \wedge g_2 \in \Theta(f_2) \Rightarrow g_1 + g_2 \in \Theta(\max(f_1, f_2))$
- *Product*. $g_1 \in \Theta(f_1) \wedge g_2 \in \Theta(f_2) \Rightarrow g_1 \cdot g_2 \in \Theta(f_1 \cdot f_2)$
- *Multiplicative Invariance*. For all constant $c \in \mathbb{R}^+$, $\Theta(f) = \Theta(c \cdot f)$

# Asymptotic notation: properties

## Classes notation

If $\mathcal{F}_1$ i $\mathcal{F}_2$ are function classes (such as $\mathcal{O}(f)$ or $\Omega(f)$), we define:

- $\mathcal{F}_1 + \mathcal{F}_2 = \{f + g \mid f \in \mathcal{F}_1 \wedge g \in \mathcal{F}_2\}$
  (where $f + g$ is the function defined as $(f + g)(n) = f(n) + g(n)$)
- $\mathcal{F}_1 \cdot \mathcal{F}_2 = \{f \cdot g \mid f \in \mathcal{F}_1 \wedge g \in \mathcal{F}_2\}$
  (where $f \cdot g$ is the function defined as $(f \cdot g)(n) = f(n) \cdot g(n)$)

## Rules for sum and product (second version)

Given two functions $f$ and $g$:

- $\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f + g) = \mathcal{O}(\max\{f, g\})$
- $\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$
- $\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\})$
- $\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g)$

# Asymptotic notation: properties

## Classes notation

If $\mathcal{F}_1$ i $\mathcal{F}_2$ are function classes (such as $\mathcal{O}(f)$ or $\Omega(f)$), we define:

- $\mathcal{F}_1 + \mathcal{F}_2 = \{f + g \mid f \in \mathcal{F}_1 \wedge g \in \mathcal{F}_2\}$
  (where $f + g$ is the function defined as $(f + g)(n) = f(n) + g(n)$)
- $\mathcal{F}_1 \cdot \mathcal{F}_2 = \{f \cdot g \mid f \in \mathcal{F}_1 \wedge g \in \mathcal{F}_2\}$
  (where $f \cdot g$ is the function defined as $(f \cdot g)(n) = f(n) \cdot g(n)$)

## Rules for sum and product (second version)

Given two functions $f$ and $g$:

- $\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f + g) = \mathcal{O}(\max\{f, g\})$
- $\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$
- $\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\})$
- $\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g)$

# Growth rates

## Frequent costs

- Constant $\Theta(1)$. Deciding parity.
- Logarithmic $\Theta(\log n)$. Binary search.
- Linear $\Theta(n)$. Sequential search in a vector.
- Quasilinear $\Theta(n \log n)$. Sorting.
- Quadratic $\Theta(n^2)$. Sum of square matrices.
- Cubic $\Theta(n^3)$. Product of square matrices.
- Polynomial $\Theta(n^k)$, for some constant $k \geq 1$. Enumerating combinations ($n$ elements taken in groups of $k$).
- Exponential $\Theta(k^n)$, for some constant $k > 1$. Search in a configuration space (width $k$ and depth $n$).
- Other $\Theta(\sqrt{n})$, $\Theta(n!)$, $\Theta(n^n)$. (ex.: place them in the previous ordered list)

# Growth rates

## Frequent costs

- Constant $\Theta(1)$. Deciding parity.
- Logarithmic $\Theta(\log n)$. Binary search.
- Linear $\Theta(n)$. Sequential search in a vector.
- Quasilinear $\Theta(n \log n)$. Sorting.
- Quadratic $\Theta(n^2)$. Sum of square matrices.
- Cubic $\Theta(n^3)$. Product of square matrices.
- Polynomial $\Theta(n^k)$, for some constant $k \geq 1$. Enumerating combinations ($n$ elements taken in groups of $k$).
- Exponential $\Theta(k^n)$, for some constant $k > 1$. Search in a configuration space (width $k$ and depth $n$).
- Other $\Theta(\sqrt{n})$, $\Theta(n!)$, $\Theta(n^n)$. (ex.: place them in the previous ordered list)

# Growth rates

## Frequent costs

- Constant $\Theta(1)$. Deciding parity.
- Logarithmic $\Theta(\log n)$. Binary search.
- Linear $\Theta(n)$. Sequential search in a vector.
- Quasilinear $\Theta(n \log n)$. Sorting.
- Quadratic $\Theta(n^2)$. Sum of square matrices.
- Cubic $\Theta(n^3)$. Product of square matrices.
- Polynomial $\Theta(n^k)$, for some constant $k \geq 1$. Enumerating combinations ($n$ elements taken in groups of $k$).
- Exponential $\Theta(k^n)$, for some constant $k > 1$. Search in a configuration space (width $k$ and depth $n$).
- Other $\Theta(\sqrt{n})$, $\Theta(n!)$, $\Theta(n^n)$. (ex.: place them in the previous ordered list)

# Growth rates

## Frequent costs

- Constant $\Theta(1)$. Deciding parity.
- Logarithmic $\Theta(\log n)$. Binary search.
- Linear $\Theta(n)$. Sequential search in a vector.
- Quasilinear $\Theta(n \log n)$. Sorting.
- Quadratic $\Theta(n^2)$. Sum of square matrices.
- Cubic $\Theta(n^3)$. Product of square matrices.
- Polynomial $\Theta(n^k)$, for some constant $k \geq 1$. Enumerating combinations ($n$ elements taken in groups of $k$).
- Exponential $\Theta(k^n)$, for some constant $k > 1$. Search in a configuration space (width $k$ and depth $n$).
- Other $\Theta(\sqrt{n})$, $\Theta(n!)$, $\Theta(n^n)$. (ex.: place them in the previous ordered list)

# Growth rates

## Frequent costs

- Constant $\Theta(1)$. Deciding parity.
- Logarithmic $\Theta(\log n)$. Binary search.
- Linear $\Theta(n)$. Sequential search in a vector.
- Quasilinear $\Theta(n \log n)$. Sorting.
- Quadratic $\Theta(n^2)$. Sum of square matrices.
- Cubic $\Theta(n^3)$. Product of square matrices.
- Polynomial $\Theta(n^k)$, for some constant $k \geq 1$. Enumerating combinations ($n$ elements taken in groups of $k$).
- Exponential $\Theta(k^n)$, for some constant $k > 1$. Search in a configuration space (width $k$ and depth $n$).
- Other $\Theta(\sqrt{n})$, $\Theta(n!)$, $\Theta(n^n)$. (ex.: place them in the previous ordered list)

# Growth rates

## Frequent costs

- Constant $\Theta(1)$. Deciding parity.
- Logarithmic $\Theta(\log n)$. Binary search.
- Linear $\Theta(n)$. Sequential search in a vector.
- Quasilinear $\Theta(n \log n)$. Sorting.
- Quadratic $\Theta(n^2)$. Sum of square matrices.
- Cubic $\Theta(n^3)$. Product of square matrices.
- Polynomial $\Theta(n^k)$, for some constant $k \geq 1$. Enumerating combinations ($n$ elements taken in groups of $k$).
- Exponential $\Theta(k^n)$, for some constant $k > 1$. Search in a configuration space (width $k$ and depth $n$).
- Other $\Theta(\sqrt{n})$, $\Theta(n!)$, $\Theta(n^n)$. (ex.: place them in the previous ordered list)

# Growth rates

## Frequent costs

- Constant $\Theta(1)$. Deciding parity.
- Logarithmic $\Theta(\log n)$. Binary search.
- Linear $\Theta(n)$. Sequential search in a vector.
- Quasilinear $\Theta(n \log n)$. Sorting.
- Quadratic $\Theta(n^2)$. Sum of square matrices.
- Cubic $\Theta(n^3)$. Product of square matrices.
- Polynomial $\Theta(n^k)$, for some constant $k \geq 1$. Enumerating combinations ($n$ elements taken in groups of $k$).
- Exponential $\Theta(k^n)$, for some constant $k > 1$. Search in a configuration space (width $k$ and depth $n$).
- Other $\Theta(\sqrt{n})$, $\Theta(n!)$, $\Theta(n^n)$. (ex.: place them in the previous ordered list)

1 Computation Time
- Efficiency of algorithms
- Input size and cost
- Orders of magnitude

2 Asymptotic notation
- Asymptotic notation: definitions
- Asymptotic notation: properties
- Growth rates

3 Cost of algorithms
- Iterative algorithms
- Recursive algorithms
- Master theorems

# Iterative algorithms

Cost computation:

- The cost of a basic operation is $\Theta(1)$. This includes:
  - an assignment between basic types
  - a comparison
  - the evaluation of a simple expression
  - an arithmetic operation
  - the access to an element of table
- If the cost of a fragment $F_1$ is $\Theta(f_1)$ and the cost of another fragment $F_2$ is $\Theta(f_2)$, then the cost of the sequential composition

$$F_1; F_2$$

is $\Theta(f_1) + \Theta(f_2) = \Theta(\max(f_1, f_2))$. Hence, if $k$ is constant and fragment $F_k$ has cost $\Theta(f_k)$, the cost of the sequential composition

$$F_1; F_2; \ldots; F_k$$

is $\Theta(f_1) + \Theta(f_2) + \cdots + \Theta(f_k) = \Theta(\max(f_1, f_2, \ldots, f_k))$.

Cost computation:

- The cost of a basic operation is $\Theta(1)$. This includes:
  - an assignment between basic types
  - a comparison
  - the evaluation of a simple expression
  - an arithmetic operation
  - the access to an element of table
- If the cost of a fragment $F_1$ is $\Theta(f_1)$ and the cost of another fragment $F_2$ is $\Theta(f_2)$, then the cost of the sequential composition

$$F_1; F_2$$

is $\Theta(f_1) + \Theta(f_2) = \Theta(\max(f_1, f_2))$. Hence, if $k$ is constant and fragment $F_k$ has cost $\Theta(f_k)$, the cost of the sequential composition

$$F_1; F_2 \;; \ldots ; F_k$$

is $\Theta(f_1) + \Theta(f_2) + \cdots + \Theta(f_k) = \Theta(\max(f_1, f_2, \ldots, f_k))$.

# Iterative algorithms

Cost computation:

- If the cost of a fragment $F_1$ is $\Theta(f_1)$, the cost of fragment $F_2$ is $\Theta(f_2)$ and the cost of evaluating $B$ is $\Theta(g)$, then the cost of the alternative composition

  `if` $(B)$ $F_1$; `else` $F_2$

  is $\Theta(g) + \Theta(\max(f_1, f_2)) = \Theta(\max(g, f_1, f_2))$. As expected, the cost of

  `if` $(B)$ $F_1$;

  is $\Theta(g) + \Theta(f_1) = \Theta(\max(g, f_1))$.

- If the cost of $F$ during the $i$-th iteration is $\Theta(f_i)$, the cost of evaluating $B$ is $\Theta(g_i)$ and the number of iterations is $h(n)$, then the cost of the iterative composition

  `while` $(B)$ $F$;

  is $\left(\sum_{i=1}^{h(n)} \Theta(f_i(n) + g_i(n))\right) + \Theta(g_{h(n)+1}(n)))$.

  If $f = \max_{i=0\dots h(n)}\{f_i(n), g_i(n), g_{h(n)+1}(n)\}$, then the cost is $\mathcal{O}(hf)$.

# Iterative algorithms

Cost computation:

- If the cost of a fragment $F_1$ is $\Theta(f_1)$, the cost of fragment $F_2$ is $\Theta(f_2)$ and the cost of evaluating $B$ is $\Theta(g)$, then the cost of the alternative composition

  $$\texttt{if } (B)\ F_1;\ \texttt{else } F_2$$

  is $\Theta(g) + \Theta(\max(f_1, f_2)) = \Theta(\max(g, f_1, f_2))$. As expected, the cost of

  $$\texttt{if } (B)\ F_1;$$

  is $\Theta(g) + \Theta(f_1) = \Theta(\max(g, f_1))$.

- If the cost of $F$ during the $i$-th iteration is $\Theta(f_i)$, the cost of evaluating $B$ is $\Theta(g_i)$ and the number of iterations is $h(n)$, then the cost of the iterative composition

  $$\texttt{while } (B)\ F;$$

  is $\left(\sum_{i=1}^{h(n)} \Theta(f_i(n) + g_i(n))\right) + \Theta(g_{h(n)+1}(n)))$.

  If $f = \max_{i=0\ldots h(n)}\{f_i(n), g_i(n), g_{h(n)+1}(n)\}$, then the cost is $\mathcal{O}(hf)$.

# Iterative algorithms

## Example: selection sort

Steps to sort the sequence 5, 6, 1, 2, 0, 7, 4, 3 with the selection sort algorithm. In red, the already sorted elements. In yellow, the elements swapped with the maximum.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 6 | 1 | 2 | 0 | 7 | 4 | 3 |
| 5 | 6 | 1 | 2 | 0 | 3 | 4 | 7 |
| 5 | 4 | 1 | 2 | 0 | 3 | 6 | 7 |
| 3 | 4 | 1 | 2 | 0 | 5 | 6 | 7 |
| 3 | 0 | 1 | 2 | 4 | 5 | 6 | 7 |
| 2 | 0 | 1 | 3 | 4 | 5 | 6 | 7 |
| 1 | 0 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Iterative algorithms

## Selection sort

```
0 int position_max (const vector<int>& v, int m) {
1   int k = 0;
2   for (int i = 1; i <= m; ++i)
3      if (v[i] > v[k]) k = i;
4   return k; }

5 void selection_sort (vector<int>& v, int n) {
6     for (int i = n; i > 0; --i) {
7         int k = position_max(v,i);
8         swap(v[k],v[i]); }}}
```

2, 6 Loop iterations: $m = m - 1 + 1$, $n = n - 1 + 1$.

7 Cost $\Theta(i)$.

other Constant-cost instructions: $\Theta(1)$.

$$t_{sel}(n) = \Theta(1) + \sum_{i=1}^{n}(\Theta(i) + \Theta(1)) = \Theta(\sum_{i=1}^{n} i) = \Theta(\frac{n(n+1)}{2}) = \Theta(n^2)$$

# Iterative algorithms

## Example: insertion sort

Steps to sort the sequence 5, 6, 1, 2, 0,7,4,3 with the insertion sort algorithm. In red, the already sorted elements. In parenthesis, the number of positions that the inserted elements has been moved.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 1 | 2 | 0 | 7 | 4 | 3 | (0) |
| 5 | 6 | 1 | 2 | 0 | 7 | 4 | 3 | (0) |
| 1 | 5 | 6 | 2 | 0 | 7 | 4 | 3 | (2) |
| 1 | 2 | 5 | 6 | 0 | 7 | 4 | 3 | (2) |
| 0 | 1 | 2 | 5 | 6 | 7 | 4 | 3 | (4) |
| 0 | 1 | 2 | 5 | 6 | 7 | 4 | 3 | (0) |
| 0 | 1 | 2 | 4 | 5 | 6 | 7 | 3 | (3) |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | (4) |

# Iterative algorithms

## Insertion sort

```
0 void insertion_sort (vector<int>& v, int i, int j) {
1     for (int k=i+1; k<=j; ++k) {
2         int t=k-1;
3         while (t>=i and v[t+1]<v[t]) {
4             swap(v[t], v[t+1]);
5             --t;
  }   }   }
```

0 Parameter passing: $\Theta(1)$.

1 Loop iterations: $n = j - (i+1) + 1 = j - i$.

1,2 Iteration condition and line 2: $\Theta(1)$.

3 Loop iterations: between 0 and $k - 1 - i + 1 = k - i \leq n$.

4,5 Assignments with cost $\Theta(1)$.

$$\Theta(1) + (n \times \Theta(1)) \leq t_{ins}(n) \leq \Theta(1) + \sum_{k=i+1}^{j} (k - i)$$

# Iterative algorithms

We just proved that the cost of sorting $n$ elements via insertion sort is $t_{ins}(n)$, with:

$$\Theta(1) + (n \times \Theta(1)) \leq t_{ins}(n) \leq \Theta(1) + \sum_{k=i+1}^{j} (k - i)$$

But

$$\begin{aligned} \textstyle\sum_{k=i+1}^{j}(k-i) &= 1 + 2 + \cdots + (j - i) \\ &= 1 + 2 + \cdots + n \\ &= \tfrac{n(n+1)}{2} \in \Theta(n^2). \end{aligned}$$

Hence,

$$\Theta(n) \leq t_{ins}(n) \leq \Theta(n^2).$$

# Recursive algorithms

The cost of a recursive algorithm is often expressed as a recurrence.

**Definition**

A recurrence is an equation or inequality that describes the function in terms of its value for smaller inputs.

**Example**

$$C(n) = \begin{cases} 1, & \text{if } n = 1 \\ C(n-1) + n, & \text{if } n \geq 2 \end{cases}$$

Solving the recurrence consist in giving a close form for it. In the example, $C(1) = 1$, $C(2) = 3$ and $C(3) = 6$, but we would like a non-recurrent form to compute its value.

# Recursive algorithms

The cost of a recursive algorithm is often expressed as a recurrence.

**Definition**

A recurrence is an equation or inequality that describes the function in terms of its value for smaller inputs.

**Example**

$$C(n) = \begin{cases} 1, & \text{if } n = 1 \\ C(n-1) + n, & \text{if } n \geq 2 \end{cases}$$

Solving the recurrence consist in giving a close form for it. In the example, $C(1) = 1$, $C(2) = 3$ and $C(3) = 6$, but we would like a non-recurrent form to compute its value.

# Recursive algorithms

$$C(n) = \begin{cases} 1, & \text{if } n = 1 \\ C(n-1) + n, & \text{if } n \geq 2 \end{cases}$$

## Solution

$$\begin{aligned}
C(n) &= C(n-1) + n \\
&= C(n-2) + (n-1) + n \\
&= C(n-3) + (n-2) + (n-1) + n \\
&\vdots \\
&= C(1) + 2 + \cdots + (n-2) + (n-1) + n \\
&= 1 + 2 + \cdots + n \\
&= \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2).
\end{aligned}$$

# Recursive algorithms

For describing a recurrence that expresses the cost of a recursive algorithm, it is enough to determine:

- the parameter of recursion $n$,
- the cost of the base case ($n = 0$, $n = 1$,...)
- the cost of the inductive case
    - number of recursive calls
    - value of the parameter in the calls
    - cost of the additional non-recursive computations

# Recursive algorithms

## Recursive linear search

Check whether a number $v$ appears in a vector $a$ between the positions 0 and $n-1$ comparing it with $a[0], a[1], \ldots, a[n-1]$. If $v$ is found, it returns the index of the position containing it. Otherwise, it returns -1.

```
int linear_search(const vector<int>& a, int n, int v) {
    if (n==0) return -1;
    else if (a[n-1]==v) return n-1;
    else return linear_search(a, n-1, v);
}
```

The parameter of recursion is $n$, the vector size. We define the recurrence $T(n)$ that represents the wost-case cost of the algorithm as follows:

$$T(n) = T(n-1) + \Theta(1)$$

# Recursive algorithms

$T(n) = T(n-1) + \Theta(1)$ for $n \geq 1$ i $T(0) = \Theta(1)$.

## Solution

$$\begin{aligned} T(n) &= T(n-1) + \Theta(1) \\ &= T(n-2) + 2 \cdot \Theta(1) \\ &= T(n-3) + 3 \cdot \Theta(1) \\ &\vdots \\ &= T(1) + n \cdot \Theta(1) \\ &= (n+1) \cdot \Theta(1) \\ &= \Theta(n+1) = \Theta(n). \end{aligned}$$

## Recursive binary search

Check whether a number *v* appears in a vector *a* between the positions *i* and *j* using binary search. If *v* is found, it returns the index of the position containing it. Otherwise, it returns $-1$.

```
int binary_search(const vector<int>& a,int i,int j,int v)
{  if (i <= j) {
       int k = (i + j) / 2;
       if (v == a[k])
           return k;
       else if (v < a[k])
           return binary_search(a, i, k-1, v);
       else
           return binary_search(a, k+1, j, v);
   } return -1;
}
```

# Recursive algorithms

```
int binary_search(const vector<int>& a,int i,int j,int v)
{  if (i <= j) {
      int k = (i + j) / 2;
      if (v == a[k])
          return k;
      else if (v < a[k])
          return binary_search(a, i, k-1, v);
      else
          return binary_search(a, k+1, j, v);
   } return -1;
}
```

The parameter of recursion is $n = j - i$, the size of the interval to explore. We define the recurrence $T(n)$ that represents the worst-case cost of the algorithm as follows:

$$T(n) = T(n/2) + \Theta(1)$$

# Recursive algorithms

$T(n) = T(n/2) + \Theta(1)$ for $n \geq 1$ and $T(0) = \Theta(1)$.

## Solution

$$
\begin{aligned}
T(n) &= T(n/2) + \Theta(1) \\
&= T(n/4) + 2 \cdot \Theta(1) \\
&= T(n/8) + 3 \cdot \Theta(1) \\
&\vdots \\
&= T(n/2^{\log_2 n}) + \log_2 n \cdot \Theta(1) \\
&= T(1) + \log_2 n \cdot \Theta(1) \\
&= T(0) + (\log_2 n + 1) \cdot \Theta(1) \\
&= (\log_2 n + 2) \cdot \Theta(1) = \Theta(\log n + 2) = \Theta(\log n).
\end{aligned}
$$

# Master theorems

To automate the cost analysis of recursive algorithms, we classify them into two groups, depending on how they divide the input problem into subproblems in the recursive calls.

Let $A$ be an algorithm that, given an input problem of size $n$, performs $a$ recursive calls and an additional non-recursive work of cost $g(n)$. If the recursive calls have size

- $n - c$, the cost of $A$ is given by the recurrence

$$T(n) = a \cdot T(n - c) + g(n)$$

- $n/b$, the cost of $A$ is given by the recurrence

$$T(n) = a \cdot T(n/b) + g(n)$$

# Master theorems

The two previous families of recurrences:

- subtraction: $T(n) = a \cdot T(n-c) + g(n)$
- division: $T(n) = a \cdot T(n/b) + g(n)$

can be solved with the master theorems that will be explained in the following.

# Master theorems

## Master theorem I

Let $T(n)$ be the recurrence

$$T(n) = \begin{cases} f(n), & \text{if } 0 \leq n < n_0 \\ a \cdot T(n-c) + g(n), & \text{if } n \geq n_0 \end{cases}$$

where $n_0 \in \mathbb{N}$, $c \geq 1$, $f$ is an arbitrary function and $g \in \Theta(n^k)$ for some $k \geq 0$.

Then,

$$T(n) \in \begin{cases} \Theta(n^k), & \text{if } a < 1 \\ \Theta(n^{k+1}), & \text{if } a = 1 \\ \Theta(a^{n/c}), & \text{if } a > 1 \end{cases}$$

# Master theorems

## Master theorem I

Let $T(n) = \begin{cases} f(n), & \text{if } 0 \leq n < n_0 \\ a \cdot T(n-c) + g(n), & \text{if } n \geq n_0 \end{cases}$

where $n_0 \in \mathbb{N}$, $c \geq 1$, $f$ is an arbitrary function and $g \in \Theta(n^k)$ for some $k \geq 0$.

Then,

$$T(n) \in \begin{cases} \Theta(n^k), & \text{if } a < 1 \\ \Theta(n^{k+1}), & \text{if } a = 1 \\ \Theta(a^{n/c}), & \text{if } a > 1 \end{cases}$$

## Example 1

We have seen that the cost of the recursive algorithm for linear search can be described with the recurrence $T(n) = T(n-1) + \Theta(1)$ for $n \geq 1$ and $T(0) = \Theta(1)$.

Hence, $n_0 = 1$, $a = 1$, $c = 1$, $k = 0$. Then, $T(n)$ belongs to the second case:

$$T(n) \in \Theta(n^{k+1}) = \Theta(n).$$

# Master theorems

## Master theorem I

Let $T(n) = \begin{cases} f(n), & \text{if } 0 \le n < n_0 \\ a \cdot T(n-c) + g(n), & \text{if } n \ge n_0 \end{cases}$

where $n_0 \in \mathbb{N}$, $c \ge 1$, $f$ is an arbitrary function and $g \in \Theta(n^k)$ for some $k \ge 0$.

Then,

$$T(n) \in \begin{cases} \Theta(n^k), & \text{if } a < 1 \\ \Theta(n^{k+1}), & \text{if } a = 1 \\ \Theta(a^{n/c}), & \text{if } a > 1 \end{cases}$$

## Example 2

In the recurrence $T(n) = T(n-1) + \Theta(n)$, we identify the values

$$a = 1,\ c = 1,\ k = 1.$$

Then, $T(n)$ belongs to the second case:

$$T(n) \in \Theta(n^{k+1}) = \Theta(n^2).$$

# Master theorems

## Master theorem I

Let $T(n) = \begin{cases} f(n), & \text{if } 0 \leq n < n_0 \\ a \cdot T(n-c) + g(n), & \text{if } n \geq n_0 \end{cases}$

where $n_0 \in \mathbb{N}$, $c \geq 1$, $f$ is an arbitrary function and $g \in \Theta(n^k)$ for some $k \geq 0$.

Then,

$$T(n) \in \begin{cases} \Theta(n^k), & \text{if } a < 1 \\ \Theta(n^{k+1}), & \text{if } a = 1 \\ \Theta(a^{n/c}), & \text{if } a > 1 \end{cases}$$

## Example 3

In the recurrence $T(n) = 2 \cdot T(n-1) + \Theta(n)$, we identify

$$a = 2,\ c = 1,\ k = 1.$$

Hence, $T(n)$ belongs to the third case:

$$T(n) \in \Theta(2^n).$$

# Master theorems

## Master theorem II

Let $T(n) = \begin{cases} f(n), & \text{if } 0 \leq n < n_0 \\ a \cdot T(n/b) + g(n), & \text{if } n \geq n_0 \end{cases}$

where $n_0 \in \mathbb{N}$, $b > 1$, $f$ is an arbitrary function and $g \in \Theta(n^k)$ for some $k \geq 0$.

Let $\alpha = \log_b(a)$. Then,

$$T(n) \in \begin{cases} \Theta(n^k), & \text{if } \alpha < k \\ \Theta(n^k \log n), & \text{if } \alpha = k \\ \Theta(n^\alpha), & \text{if } \alpha > k \end{cases}$$

# Master theorems

## Master theorem II

Let $T(n) = \begin{cases} f(n), & \text{if } 0 \le n < n_0 \\ a \cdot T(n/b) + g(n), & \text{if } n \ge n_0 \end{cases}$

where $n_0 \in \mathbb{N}$, $b > 1$, $f$ is an arbitrary function and $g \in \Theta(n^k)$ for some $k \ge 0$.

Let $\alpha = \log_b(a)$. Then,

$$T(n) \in \begin{cases} \Theta(n^k), & \text{if } \alpha < k \\ \Theta(n^k \log n), & \text{if } \alpha = k \\ \Theta(n^\alpha), & \text{if } \alpha > k \end{cases}$$

## Example 1

We have seen that the cost of the recursive algorithm for binary search can be described by the recurrence $T(n) = T(n/2) + \Theta(1)$ for $n \ge 1$ and $T(0) = \Theta(1)$.

Hence, $n_0 = 1$, $a = 1$, $b = 2$, $k = 0$, $\alpha = 0$. $T(n)$ belongs to the 2nd case:

$$T(n) \in \Theta(n^k \log n) = \Theta(\log n).$$

## Example 2

Main function of merge sort.

```
template <typename elem>
void merge_sort (vector<elem>& a, int e, int d) {
    if (e<d) {
        int m = (e + d) / 2;
        merge_sort(a, e, m);
        merge_sort(a, m + 1, d);
        merge(a, e, m, d);
}   }
```

Taking into account that the call `merge(T, e, m, d)` is $\Theta(n)$ (where $n = d - e + 1$), the total cost can be expressed with the recurrence:

$$T(n) = 2T(n/2) + \Theta(n) \text{ for } n \geq 2, \text{ and } T(1) = \Theta(1).$$

# Master theorems

## Master theorem II

Let $T(n) = \begin{cases} f(n), & \text{if } 0 \leq n < n_0 \\ a \cdot T(n/b) + g(n), & \text{if } n \geq n_0 \end{cases}$

where $n_0 \in \mathbb{N}$, $b > 1$, $f$ is an arbitrary function and $g \in \Theta(n^k)$ for some $k \geq 0$.

Let $\alpha = \log_b(a)$. Then,

$$T(n) \in \begin{cases} \Theta(n^k), & \text{if } \alpha < k \\ \Theta(n^k \log n), & \text{if } \alpha = k \\ \Theta(n^\alpha), & \text{if } \alpha > k \end{cases}$$

## Example 2

We have seen that the cost of merge sort can be described with the recurrence $T(n) = 2T(n/2) + \Theta(n)$ for $n \geq 2$ and $T(1) = \Theta(1)$.

Hence, $n_0 = 2$, $a = 2$, $b = 2$, $k = 1$, $\alpha = 1$ and $T(n)$ belongs to the 2nd case:

$$T(n) \in \Theta(n^k \log n) = \Theta(n \log n).$$

**Exercise 1**

Solve the recurrence $T(n) = T(\sqrt{n}) + 1$.

**Hint**

Consider the variable change $m = \log n$.

# Master theorems

## Exercise 1

Solve the recurrence $T(n) = T(\sqrt{n}) + 1$.

## Solution

Consider the variable change $m = \log n$. Then,

$$T(n) = T(2^m) = T(2^{m/2}) + 1.$$

Let us define $S(m) = T(2^m)$, that fulfills

$$S(m) = S(m/2) + 1.$$

Using master theorem II, we know that $S(m) \in \Theta(\log m)$ and hence:

$$T(n) = T(2^m) = S(m) \in \Theta(\log m) = \Theta(\log\log n).$$

**Exercise 2**

Solve the recurrence $T(n) = 2T(\sqrt{n}) + \log n$.

# Lower bound of sorting algorithms

In terms of asymptotic cost, merge sort is optimal:

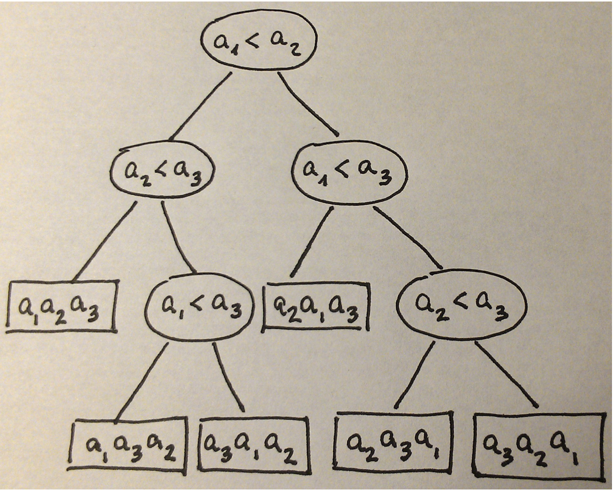
**Proposition**

Any sorting algorithm based on comparisons has cost $\Omega(n \log n)$.

This can be proved using trees to represent sorting algorithms based on comparisons.

# Lower bound of sorting algorithms

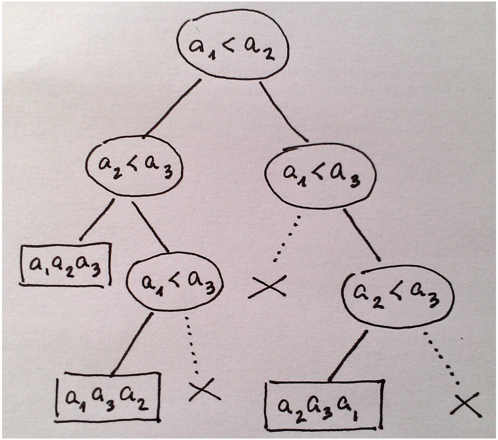
Let us assume that we want to sort $a_1$, $a_2$ i $a_3$. If $a_1 < a_2$, we follow the left branch; otherwise, the right one. Rectangles represent the orderings found. The depth of the tree is the worst-case cost.



# Lower bound of sorting algorithms

Let us consider a tree that sorts $n$ elements:

- each leaf belongs to a permutation of $\{1, 2, \ldots, n\}$
- each permutation of $\{1, 2, \ldots, n\}$ has to appear in some leaf
  (if one of the does not appear, what would happen if that was the input?)



# Lower bound of sorting algorithms

- since there are $n!$ permutations of $n$ elements, the tree has $\geq n!$ leafs
- a binary tree with depth $d$ has $\leq 2^d$ leafs
- hence, the depth of our tree is at least $\log_2 n!$

The cost of the algorithm represented by the tree is, hence, $\Omega(\log n!)$. Since

$$n! \geq n \cdot (n-1) \cdot (n-2) \cdot \ \ldots \ \cdot \lfloor n/2 \rfloor \geq (n/2)^{(n/2)}$$

we have that

$$\log_2 n! \geq \log_2 (n/2)^{(n/2)} = \frac{n}{2} \log_2 (n/2) \in \Omega(n \log n).$$

**Proposition**

Any sorting-based algorithm has cost $\Omega(n \log n)$.

# Lower bound of sorting algorithms

- since there are $n!$ permutations of $n$ elements, the tree has $\geq n!$ leafs
- a binary tree with depth $d$ has $\leq 2^d$ leafs
- hence, the depth of our tree is at least $\log_2 n!$

The cost of the algorithm represented by the tree is, hence, $\Omega(\log n!)$. Since

$$n! \geq n \cdot (n-1) \cdot (n-2) \cdot \ \ldots \ \cdot \lfloor n/2 \rfloor \geq (n/2)^{(n/2)}$$

we have that

$$\log_2 n! \geq \log_2 (n/2)^{(n/2)} = \frac{n}{2} \log_2 (n/2) \in \Omega(n \log n).$$

**Proposition**

Any sorting-based algorithm has cost $\Omega(n \log n)$.

# Lower bound of sorting algorithms

- since there are $n!$ permutations of $n$ elements, the tree has $\geq n!$ leafs
- a binary tree with depth $d$ has $\leq 2^d$ leafs
- hence, the depth of our tree is at least $\log_2 n!$

The cost of the algorithm represented by the tree is, hence, $\Omega(\log n!)$. Since

$$n! \geq n \cdot (n-1) \cdot (n-2) \cdot \ \ldots \ \cdot \lfloor n/2 \rfloor \geq (n/2)^{(n/2)}$$

we have that

$$\log_2 n! \geq \log_2 (n/2)^{(n/2)} = \frac{n}{2} \log_2 (n/2) \in \Omega(n \log n).$$

**Proposition**

Any sorting-based algorithm has cost $\Omega(n \log n)$.

# Lower bound of sorting algorithms

- since there are $n!$ permutations of $n$ elements, the tree has $\geq n!$ leafs
- a binary tree with depth $d$ has $\leq 2^d$ leafs
- hence, the depth of our tree is at least $\log_2 n!$

The cost of the algorithm represented by the tree is, hence, $\Omega(\log n!)$. Since

$$n! \geq n \cdot (n-1) \cdot (n-2) \cdot \ \ldots \ \cdot \lfloor n/2 \rfloor \geq (n/2)^{(n/2)}$$

we have that

$$\log_2 n! \geq \log_2 (n/2)^{(n/2)} = \frac{n}{2} \log_2 (n/2) \in \Omega(n \log n).$$

**Proposition**

Any sorting-based algorithm has cost $\Omega(n \log n)$.

# Lower bound of sorting algorithms

- since there are $n!$ permutations of $n$ elements, the tree has $\geq n!$ leafs
- a binary tree with depth $d$ has $\leq 2^d$ leafs
- hence, the depth of our tree is at least $\log_2 n!$

The cost of the algorithm represented by the tree is, hence, $\Omega(\log n!)$. Since

$$n! \geq n \cdot (n-1) \cdot (n-2) \cdot \ \ldots \ \cdot \lfloor n/2 \rfloor \geq (n/2)^{(n/2)}$$

we have that

$$\log_2 n! \geq \log_2 (n/2)^{(n/2)} = \frac{n}{2} \log_2 (n/2) \in \Omega(n \log n).$$

**Proposition**

Any sorting-based algorithm has cost $\Omega(n \log n)$.