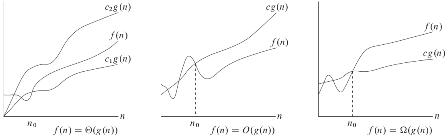
Complexity and Asymptotic Analysis

L.EIC

Algorithms and Data Structures

2025/2026



P Ribeiro, AP Tomás

The Joy of Algorithms

"For me, **great algorithms are the poetry of computation**. Just like verse, they can be terse, allusive, dense and even mysterious. But **once unlocked, they cast a brilliant new light on some aspect of computing**."

Francis Sullivan, The Joy of Algorithms, 2000

Algorithms + Data Structures = Program

A textbook by Niklaus Wirth, 1976

Algorithm: a well-defined computational procedure for solving a problem. It must **terminate after a finite number of steps**.

Correctness

It has to solve correctly **all instances** of the problem

Efficiency

The performance (**time** and **memory**) has to be adequate.

Efficient Algorithms

From textbook "Algorithms", by Jeff Erickson, chapter 12.
https://jeffe.cs.illinois.edu/teaching/algorithms/

- A minimal requirement for an **algorithm** to be considered "efficient" is that its running time is bounded by a **polynomial function** of the input size: $\mathcal{O}(n^c)$ for some constant c, where n is the size of the input.

  (this kind of notation will be the focus of this class)
- Researchers recognized early on that **not all problems can be solved this quickly**, but had a hard time figuring out exactly which ones could and which ones couldn't.
- There are several so-called **NP-hard problems**, which most people believe cannot be solved in polynomial time, even though nobody can prove a super-polynomial lower bound.

Some NP-hard problems

To be addressed in Design of Algorithms (2nd semester)

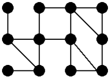
- **SAT**: Given a CNF formula $\Phi(x_1, \dots, x_n) = C_1 \wedge \dots \wedge C_m$, is Φ satisfiable, i.e., is there a truth assignment that satisfies all clauses?

  e.g., is $\Phi(p, q, r, s) = (\neg p \vee q \vee r) \wedge (\neg q \vee r \vee \neg s) \wedge (s \vee p) \wedge (\neg r \vee \neg q \vee p)$ satisfiable?

- **Partition**: Given a set $S = \{a_1, a_2, \dots, a_n\}$ of n positive integers, is there a set $A \subset S$ such that $\sum_{x \in A} x = \sum_{y \in S \setminus A} x$?

  e.g., can we split $S = \{1, 4, 7, 15, 23, 42\}$ into two sets with the same sum?

- **Hamiltonian Path**: Given an undirected graph $G = (V, E)$, does G contain a path that visits all nodes exactly once?

  
  e.g., can we find a path using the lines that visits all black circles once?

- **TSP** (*travelling salesman problem*): Given a **complete** weighted graph $G = (V, E, d)$, with $d(e) \in \mathbb{Z}^+$, for all $e \in E$, and $k \in \mathbb{Z}^+$, is there a hamiltonian cycle γ with $d(\gamma) \leq k$? Optimization version asks for shortest hamiltonian cycle.

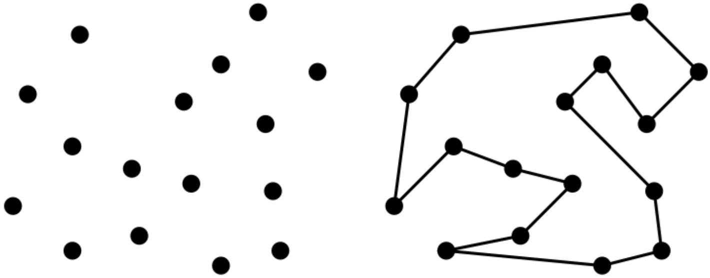
Motivational Example - TSP

- Let's have a look at a restricted version of this last problem:

Travelling Salesman Problem (Euclidean TSP version)

Input: a set S of n points in the plane
Output: the shortest possible path that starts on a point, visits all other points of S and then returns to the starting point.

An example:



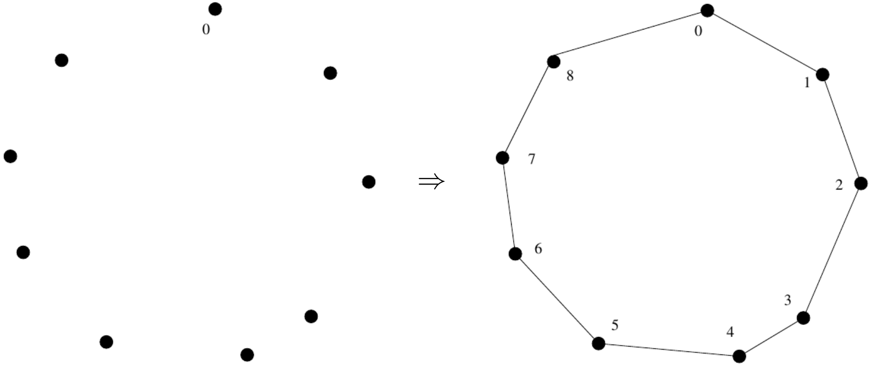
Motivational Example - TSP

A possible (greedy) algorithm - nearest neighbour

$p_1 \leftarrow$ random point
$i \leftarrow 1$
While (there are still points to visit) **do**
    $i \leftarrow i + 1$
    $p_i \leftarrow$ non visited point closest to p_{i-1}
return path $p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n \rightarrow p_1$

Motivational Example - TSP

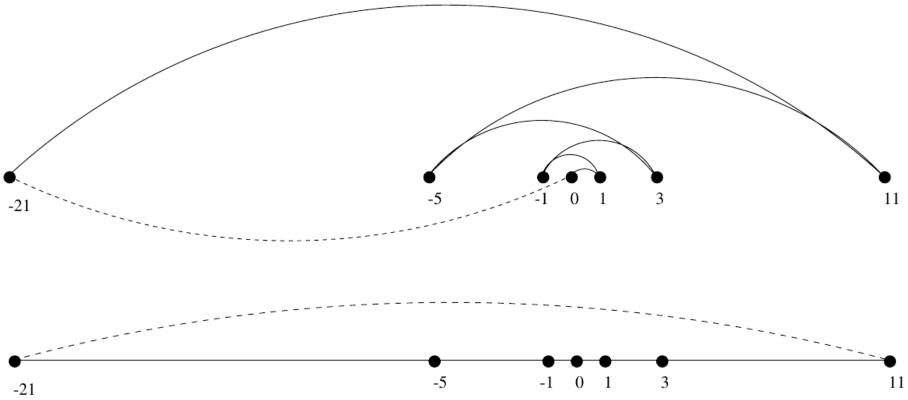
Seems to work...



Motivational Example - TSP

But it is does not produce an optimal solution for all instances!

(Note: starting with the leftmost point would not solve the problem)



Motivational Example - TSP

Another possible (greedy) algorithm

For $i \leftarrow 1$ **to** $(n-1)$ **do**

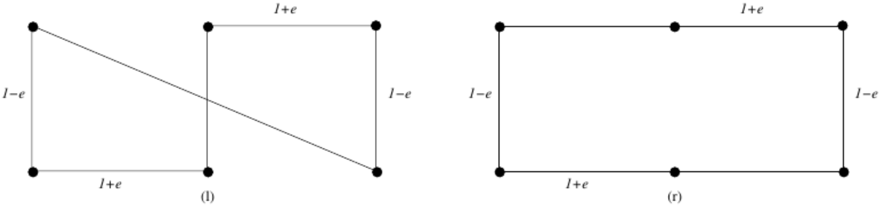
  Add connection between closest pair of points such that they are in different connected components

Add connection between the two "extremes" of the created path

return the cyclic path created

Motivational Example - TSP

It also does not produce an optimal solution for all cases!



Motivational Example - TSP

How to solve the problem then?

A possible algorithm (exhaustive search a.k.a. "brute force")

$P_{min} \leftarrow$ any permutation of the points in S
For $P_i \leftarrow$ each of the permutations of points in S
    If $(cost(P_i) < cost(P_{min}))$ **then**
        $P_{min} \leftarrow P_i$
return Path formed by P_{min}

A correct algorithm, producing an optimal solution, but extremely slow!

- $P(n) = n! = n \times (n-1) \times \ldots \times 1$
- For instance, $P(20) = 2,432,902,008,176,640,000$
- For a set of 20 points, even the fastest computer in the world would not solve it! (how long would it take?)

Motivational Example - TSP

- The present problem is a restricted version (euclidean) of one of the most well known "classic" hard problems, the **Travelling Salesman Problem** (**TSP**)
- This problem has **many possible applications**
  Ex: genomic analysis, industrial production, vehicle routing, ...
- The presented solution has $\mathcal{O}(n!)$ **temporal complexity**
  (remember, this kind of notation will be the focus of this class)
- The are other approaches with better temporal behavior: the Held-Karp algorithm has $\mathcal{O}(2^n n^2)$ temporal complexity, but requires **more memory** ($\mathcal{O}(n2^n)$ vs $\mathcal{O}(n^2)$ of the previous solution)
  (it uses dynamic programming, a technique you will hear about on another course)
- Still, there is no known **efficient solution**, that is, **polynomial** on time, for this problem
  (with optimal results, not just approximated)

The Brute Force way

Brute force: For many non-trivial problems, there is a natural **brute force search algorithm** that checks every possible solution.

- Typically takes exponential time: 2^n or worse for inputs of size n.
- Unacceptable in practice.

Brute-force for SAT:

Given a CNF formula Φ in n (boolean) variables, enumerate all truth assignments to check whether any of them satisfies all clauses. In the worst case, there are 2^n **truth assignments** to check.

Brute-force for Hamiltonian path:

Given a graph $G = (V, E)$, with $|V| = n$, check whether any permutation of V defines a cycle in G. In the worst case, there are $n! = n \times (n-1) \times \cdots \times 2 \times 1$ **permutations** to check.

$\lim_{n\to\infty} \frac{n!}{2^n} = \infty$, that is $n! \gg 2^n$

An experience: - Permutations

- Let's go back to the idea of **permutations**

Example: the 6 permutations of $\{1, 2, 3\}$

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

- Recall that the number of permutations can be computed as: $P(n) = n! = n \times (n-1) \times \ldots \times 1$
  (do you understand the intuition on the formula?)

An experience: - Permutations

- What is the execution time of a program that goes through all permutations?
  (the following times are approximated, on my notebook)
  (what I want to show is **order of growth**)

n $\leq$ 7: $< 0.001s$
n $=$ 8: $0.001s$
n $=$ 9: $0.016s$
n $=$ 10: $0.185s$
n $=$ 11: $2.204s$
n $=$ 12: $28.460s$
...
n $=$ 20: 5000 years !

How many permutations per second?
About 10^7

On computer speed

- Will a **faster computer** be of any help? **No!**
  If $n = 20 \rightarrow 5000$ years, hypothetically:
  - 10x faster would still take 500 years
  - 5,000x would still take 1 year
  - 1,000,000x faster would still take two days, but
    $n = 21$ would take more than a month
    $n = 22$ would take more than a year!
- The **growth rate** of the execution time is what matters!

Algorithmic performance vs Computer speed

A better algorithm on a slower computer **will always win** against a worst algorithm on a faster computer, for sufficiently large instances

Why worry?

- What can we do with execution time/memory analysis?

Prediction

How much time/space does an algorithm need to solve a problem? How does it scale? Can we provide guarantees on its running time/memory?

Comparison

Is an algorithm A better than an algorithm B? Fundamentally, what is the best we can possibly do on a certain problem?

- We will study a **methodology** to answer these questions
- We will focus mainly on execution time analysis

Random Access Machine (RAM)

- We need a **model** that is **generic** and **independent** from the language and the machine.
- We will consider a Random Access Machine (RAM)
  - Each **simple operation** (ex: $+$, $-$, $\leftarrow$, **If**) takes **1 step**
  - Loops and procedures, for example, are not simple instructions!
  - Each **access to memory** takes also **1 step**
- We can measure execution time by... counting the number of steps as a function of the input size n: $T(n)$.
- Operations are **simplified**, but this is useful

  Ex: summing two integers does not cost the same as dividing two reals, but we will see that on a global vision, these specific values are not important

Random Access Machine (RAM)

A counting example

```
// a simple program

int count = 0;
for (int i=0; i<n; i++)
    if (v[i] == 0) count++;
```

Let's count the number of simple operations:

Variable declarations	2
Assignments:	2
"Less than" comparisons	$n + 1$
"Equality" comparisons:	n
Array access	n
Increment	between n and $2n$

Random Access Machine (RAM)

A counting example

```
// a simple program

int count = 0;
for (int i=0; i<n; i++)
    if (v[i] == 0) count++;
```

Total number of steps on the **worst** case:
$T(n) = 2 + 2 + (n + 1) + n + n + 2n = 5 + 5n$

Total number of steps on the **best** case:
$T(n) = 2 + 2 + (n + 1) + n + n + n = 5 + 4n$

Types of algorithm analysis

Worst Case analysis: **(the most common)**

- $T(n) =$ maximum amount of time for any input of size n

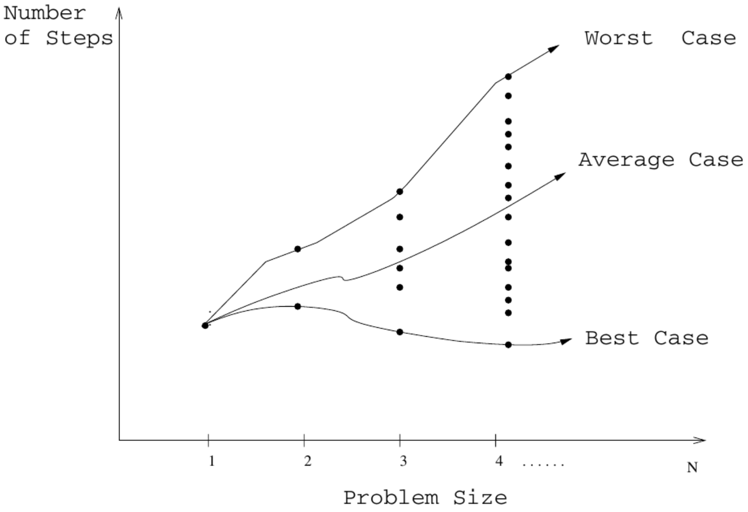
Average Case analysis: **(sometimes)**

- $T(n) =$ average time on all inputs of size n
- Implies knowing the **statistical distribution** of the inputs

Best Case analysis: **("deceiving")**

- It's almost like "cheating" with an algorithm that is fast just for **some** of the inputs

Types of algorithm analysis



Asymptotic Notation

We need a mathematical tool to **compare functions**

On algorithm analysis we use **Asymptotic Analysis**:

- "Mathematically": studying the behaviour of **limits** (as $n \to \infty$)
- Computer Science: studying the behaviour for arbitrary large input or
  "describing" **growth rate** (for the **worst case**)

- A very specific **notation** is used: $\mathcal{O}, \Omega, \Theta$ (and also o, ω)
- It allows to simplify expressions like the one before and to focus on **orders of growth**

Asymptotic Notation

Definitions

$f(n) \in \mathcal{O}(g(n))$

It means that $c \times g(n)$ is an **upper bound** of $f(n)$ *(from a certain n)*

$f(n) \in \Omega(g(n))$

It means that $c \times g(n)$ is a **lower bound** of $f(n)$ *(from a certain n)*

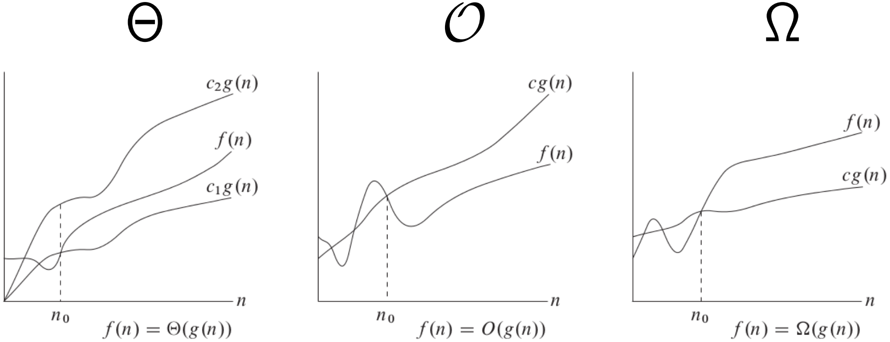
$f(n) \in \Theta(g(n))$

It means that $c_1 \times g(n)$ is a **lower bound** of $f(n)$ and $c_2 \times g(n)$ is an **upper bound** of $f(n)$ *(from a certain n)*

Where c, c_1 and c_2 are constants

Asymptotic Notation

A graphical depiction



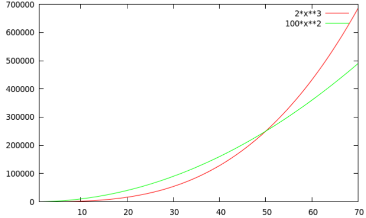
The definitions imply an n from which the function is bounded. The small values of n do not "matter".

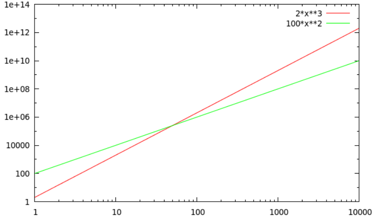
Note: Some literature uses $=$ instead of $\in$
Example: $f(n) = \mathcal{O}(g(n))$ is the same as $f(n) \in \mathcal{O}(g(n))$

An useful program to draw function plots is gnuplot.

(comparing $2n^3$ with $100n^2$)

```
gnuplot> plot [1:70] 2*x**3, 100*x**2
gnuplot> set logscale xy 10
gnuplot> plot [1:10000] 2*x**3, 100*x**2
```





Asymptotic Notation

Formalization

- $f(n) \in \mathcal{O}(g(n))$ if there exist positive constants $n_0 \in \mathbb{Z}^+$ and $c \in \mathbb{R}^+$ such that $f(n) \leq c \times g(n)$ for all $n \geq n_0$
  (g is an **upper bound**, f is *"at least as good"* as g)

- $f(n) \in \Omega(g(n))$ if there exist positive constants $n_0 \in \mathbb{Z}^+$ and $c \in \mathbb{R}^+$ such that $f(n) \geq c \times g(n)$ for all $n \geq n_0$
  (g is a **lower bound**, f is *"at least as bad"* as g)

- $f(n) \in \Theta(g(n))$ if there exist positive constants $n_0 \in \mathbb{Z}^+$, $c_1 \in \mathbb{R}^+$ and $c_2 \in \mathbb{R}^+$ such that $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ for all $n \geq n_0$
  (g is a **tight bound**, f is *"as good"* as g)

O(g(n)), Ω(g(n)) and Θ(g(n)) denote **sets of functions in natural numbers** that consist of all functions f : $\mathbb{N} \rightarrow \mathbb{R}_0^+$ related to the function g : $\mathbb{N} \rightarrow \mathbb{R}_0^+$ by the corresponding condition.

- $f(n) \in \mathcal{O}(g(n))$ if there exist positive constants $n_0 \in \mathbb{Z}^+$ and $c \in \mathbb{R}^+$ such that $f(n) \leq c \times g(n)$ for all $n \geq n_0$
- $f(n) \in \Omega(g(n))$ if there exist positive constants $n_0 \in \mathbb{Z}^+$ and $c \in \mathbb{R}^+$ such that $f(n) \geq c \times g(n)$ for all $n \geq n_0$
- $f(n) \in \Theta(g(n))$ if there exist positive constants $n_0 \in \mathbb{Z}^+$, $c_1 \in \mathbb{R}^+$ and $c_2 \in \mathbb{R}^+$ such that $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ for all $n \geq n_0$

A few consequences:

- $f(n) \in \Theta(g(n)) \longleftrightarrow f(n) \in \mathrm{O}(g(n))$ and $f(n) \in \Omega(g(n))$
- $f(n) \in \Theta(g(n)) \longleftrightarrow g(n) \in \Theta(f(n))$
- $f(n) \in \mathrm{O}(g(n)) \longleftrightarrow g(n) \in \Omega(f(n))$

Asymptotic Notation

A few practical rules

- **Multiplying by a constant** does not affect the behavior:
  $\Theta(c \times f(n)) \in \Theta(f(n))$
  $99 \times n^2 \in \Theta(n^2)$

- On a polynomial of the form $a_x n^x + a_{x-1} n^{x-1} + \ldots + a_2 n^2 + a_1 n + a_0$ we can focus on the term with the **largest exponent**:
  $3\mathrm{n}^3 - 5n^2 + 100 \in \Theta(n^3)$
  $6\mathrm{n}^4 - 20^2 \in \Theta(n^4)$
  $0.8\mathrm{n} + 224 \in \Theta(n)$

- On a sum/subtraction we can focus on the **dominant** term:
  $2^{\mathrm{n}} + 6n^3 \in \Theta(2^n)$
  $\mathrm{n}! - 3n^2 \in \Theta(n!)$
  $n \log n + 3\mathrm{n}^2 \in \Theta(n^2)$

- $99 \times n^2 \in \Theta(n^2)$
  - $n^2 \leq 99n^2 \leq 99n^2$, for all $n \geq 1$.
  - Therefore, there exist $c_1, c_2 \in \mathbb{R}^+$ and $n_0 \in \mathbb{Z}^+$ such that $c_1 n^2 \leq 99n^2 \leq c_2 n^2$, for all $n \geq n_0$.
  - We can take $c_1 = 1$, $c_2 = 99$ and $n_0 = 1$.

- $3n^3 - 5n^2 + 100 \in \Theta(n^3)$ because
  - $3n^3 - 5n^2 + 100 \geq 2n^3$, for all $n \geq 5$, since $n^3 - 5n^2 \geq 0$ for $n \geq 5$
  - $3n^3 - 5n^2 + 100 \leq 3n^3 + 5n^2 + 100 \leq 3n^3 + 5n^3 + 100n^3 = 108n^3$, for all $n \geq 1$.
  - Therefore, there exist $c_1, c_2 \in \mathbb{R}^+$ and $n_0 \in \mathbb{Z}^+$ such that $c_1 n^3 \leq 3n^3 - 5n^2 + 100 \leq c_2 n^3$, for all $n \geq n_0$.
  - We can take $c_1 = 2$, $c_2 = 108$ and $n_0 = 5$.

Note: there are many other choices of c_1, c_2 and n_0 that would work.

Some exercises - Yes or No?

- $\log_2(n) \in \mathcal{O}(n)$? Yes
- $\log_2(n) \in \Omega(n)$? No
- $\mathcal{O}(n) \subset \mathcal{O}(n^2)$? Yes
- $\Omega(n \log_2 n) \subset \Omega(n)$? Yes
- $\sqrt{n} \in \mathcal{O}(\log_2 n)$? No
  ($\sqrt{}$ grows "faster" than $\log_2$)
- $\Theta(\log_a n) = \Theta(\log_b n)$, for $a, b \in \mathbb{R}^+$, $a \neq b$, $a, b > 1$? Yes
  (that is why sometimes we omit the base of the logarithm)
- $\mathcal{O}(2^n) = \mathcal{O}(3^n)$? No
- $\mathcal{O}(2^n) \subset \mathcal{O}(3^n)$? Yes
- $\Theta(2n) = \Theta(3n)$? Yes
- $f(n) \in \Omega(1)$, for all $f : \mathbb{N} \rightarrow \mathbb{R}^+$? (Therefore, $\mathcal{O}(1) = \Theta(1)$) Yes

Asymptotic Notation

Dominance

When is a function **better** than another?

- If we want to minimize time, **"smaller" functions are better**
- A function dominates another one if as n grows it keeps getting infinitely larger
- Mathematically: $f(n) \gg g(n)$ if $\lim_{n\to\infty} g(n)/f(n) = 0$

Dominance Relations

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n\log n \gg n \gg \log n \gg 1$$

$$\mathcal{O}(1) \subset \mathcal{O}(\log n) \subset \mathcal{O}(n) \subset \mathcal{O}(n\log n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^3) \subset \mathcal{O}(2^n) \subset \mathcal{O}(n!)$$

$$\Omega(1) \supset \Omega(\log n) \supset \Omega(n) \supset \Omega(n\log n) \supset \Omega(n^2) \supset \Omega(n^3) \supset \Omega(2^n) \supset \Omega(n!)$$

Asymptotic Notation

Common Functions

Function	Name	Examples
1	constant	summing two numbers
$\log n$	logarithmic	binary search, inserting in a heap
n	linear	1 loop to find maximum value
$n \log n$	linearithmic	sorting (ex: mergesort, heapsort)
n^2	quadratic	2 loops (ex: verifying, bubblesort)
n^3	cubic	3 loops (ex: Floyd-Warshall)
2^n	exponential	exhaustive search (ex: subsets)
$n!$	factorial	all permutations

n on the base $\rightarrow$ **polynomial** function

n on the exponent $\rightarrow$ **exponencial** function

Asymptotic Growth

A practical view

If an operation takes 10^{-9} seconds...

(estimate on my laptop, but as we saw this value is not important)

	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$
20	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	77 years
30	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$1.07s$	
40	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	18.3 min	
50	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	13 days	
100	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	10^{13} *years*	
10^3	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$1s$		
10^4	$< 0.01s$	$< 0.01s$	$< 0.01s$	$0.1s$	16.7 min		
10^5	$< 0.01s$	$< 0.01s$	$< 0.01s$	$10s$	11 days		
10^6	$< 0.01s$	$< 0.01s$	$0.02s$	16.7 min	31 years		
10^7	$< 0.01s$	$0.01s$	$0.23s$	1.16 days			
10^8	$< 0.01s$	$0.1s$	$2.66s$	115 days			
10^9	$< 0.01s$	$1s$	$29.9s$	31 years			

Predicting the execution time

Pre-requirements:

- An implementation with complexity $f(n)$
- A (small) test case with input of size n_1
- The execution time of the program on that input: $time(n_1)$

We want to **estimate** the execution time for a (similar) input of size n_2.
How to do it?

Estimating the execution time

$f(n_2)/f(n_1)$ is the growth rate of the function (from n_1 to n_2)

$\mathrm{time}(\mathrm{n}_2) = \mathrm{f}(\mathrm{n}_2)/\mathrm{f}(\mathrm{n}_1) \times \mathrm{time}(\mathrm{n}_1)$

Predicting the execution time

An example

- Imagine a program with time complexity $\Theta(n^2)$ that takes **1 second** for an input of size **5 000**. What is my **estimation** for the execution time for an input of size **10 000?**

$$f(n) = n^2$$
$$n_1 = 5\,000$$
$$time(n_1) = 1 \text{ second}$$
$$n_2 = 10\,000$$

$$\begin{aligned} time(n_2) &= f(n_2)/f(n_1) \times time(n_1) = \\ &= 10\,000^2/5\,000^2 \times 1 = 4 \text{ seconds} \end{aligned}$$

Predicting the execution time

About the growth rate

Let's see what happens when we **double the input** for some of the more common functions (independently of the machine used!):

$time(2n) = f(2n)/f(n) \times time(n)$

- n : $2n/n = 2$. Time increases 2x
- n^2 : $(2n)^2/n^2 = 4n^2/n^2 = 4$. Time increases 4x
- n^3 : $(2n)^3/n^3 = 8n^3/n^3 = 8$. Time increases 8x

On polynomial functions the growth ratio is **constant**!

- 2^n : $2^{2n}/2^n = 2^{2n-n} = 2^n$. Time grows 2^n times
  Example: If $n = 5$, the time for $n = 10$ will be **32x** more!
  Example: If $n = 10$, the time for $n = 20$ will be **1024x** more!
- $\log_2(n)$: $\log_2(2n)/\log_2(n)$. Time grows $\frac{\log_2(2n)}{\log_2(n)}$ vezes
  Example: If $n = 5$, the time for $n = 10$ will be **1.43x** more!
  Example: If $n = 10$, the time for $n = 20$ will be **1.3x** more!

Asymptotic Analysis

A few more examples

- A program has two pieces of code A and B, executed one after the other, with A running in $\Theta(n \log n)$ and B in $\Theta(n^2)$.
  The program runs in $\Theta(n^2)$, because $n^2 \gg n \log n$
- A program calls n times a function $\Theta(\log n)$, and then it calls again n times another function $\Theta(\log n)$
  The program runs in $\Theta(n \log n)$
- A program has 5 loops, all called sequentially, each one of them running in $\Theta(n)$
  The program runs in $\Theta(n)$
- A program P_1 has execution time proportional to $100 \times n \log n$.
  Another program P_2 runs in $2 \times n^2$.
  Which one is more efficient?
  P_1 is more efficient because $n^2 \gg n \log n$. However, for a small n, P_2 is quicker and it might make sense to have a program that calls P_1 or P_2 depending on n.

Analyzing the complexity of programs

Let's see more concrete examples:

- **Case 1: Loops** (and summations)
- **Case 2: Recursive Functions** (and recurrences)

  this case 2 will be covered later (in the classes about sorting algorithms)

Loops and Summations

```
int count = 0;
for (int i=0; i<1000; i++)
  for (int j=i; j<1000; j++)
    count++;
cout << count << endl;
```

(the temporal complexity is proportional to the value of *count* at the end)

What does this program write?

$1000 + 999 + 998 + 997 + \ldots.. + 2 + 1$

Loops and Summations

Arithmetic progression: a sequence of numbers such that the difference d between the consecutive terms is constant. We will call a_1 to the first term.

- $1, 2, 3, 4, 5, \ldots \ldots$ $(d = 1, a_1 = 1)$
- $3, 5, 7, 9, 11, \ldots \ldots$ $(d = 2, a_1 = 3)$

How to calculate the summation of an arithmetic progression?

$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = (1 + 8) + (2 + 7) + (3 + 6) + (4 + 5) = 4 \times 9$

Summation from a_p to a_q

$$S(p, q) = \sum_{i=p}^{q} a_i = \frac{(q-p+1)\times(a_p+a_q)}{2}$$

Summation of the first n terms

$$S_n = \sum_{i=1}^{n} a_i = \frac{n\times(a_1+a_n)}{2}$$

Loops and Summations

```
int count = 0;
for (int i=0; i<1000; i++)
  for (int j=i; j<1000; j++)
    count++;
cout << count << endl;
```

What does this program write?

$1000 + 999 + 998 + 997 + \ldots\ldots + 2 + 1$

It writes $S_{1000} = \frac{1000 \times (1000+1)}{2} = 500500$

Loops and Summations

```
int count = 0;
for (int i=0; i<n; i++)
  for (int j=i; j<n; j++)
    count++;
cout << count << endl;
```

What is the execution time?

It is going to execute S_n increments:

$S_n = \sum_{i=1}^{n} a_i = \frac{n \times (1+n)}{2} = \frac{n+n^2}{2} = \frac{1}{2}n^2 + \frac{1}{2}n.$

It executes $\Theta(n^2)$ steps

Loops and Summations

If you want to know more about interesting summations on this context, take a look at *Appendix A* of the *Introduction to Algorithms* book.

Note that c cycles do not imply $\Theta(n^c)$!

```
for (int i=0; i<n; i++)
  for (int j=1; j<5; j++)
```

$\Theta(n)$

```
for (int i=1; i<=n; i++)
  for (int j=1; j<=i*i; j++)
```

$\Theta(n^3)$ $\quad (1^2 + 2^2 + 3^2 + \ldots + n^2 = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

```
i = n;
while (i>0) i = i/2;
```

$\Theta(\log n)$ (each time i becomes reduced to a half)

Divide and Conquer

This topic will be covered later, when we talk about sorting algorithms
We leave it (also) here for ease of access and coherence of material.

We are often interested in algorithms that are expressed in a recursive way

Many of these algorithms follow the divide and conquer strategy:

Divide and Conquer

Divide the problem in a set of subproblems which are smaller instances of the same problem

Conquer the subproblems solving them recursively. If the problem is small enough, solve it directly.

Combine the solutions of the smaller subproblems on a solution for the original problem

We now describe the MergeSort algorithm for sorting an array of size n

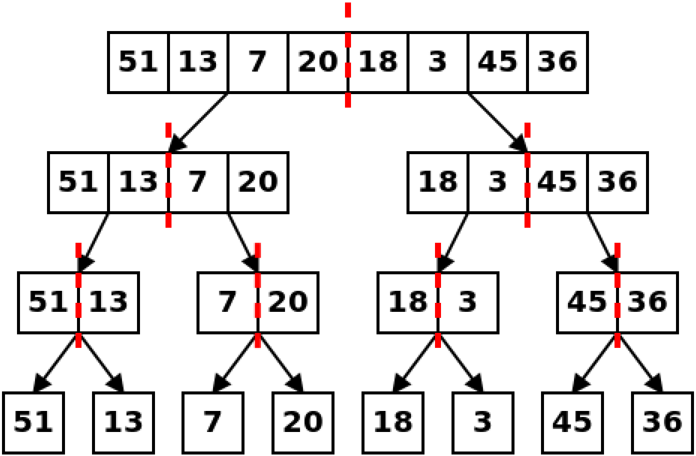
MergeSort

Divide: partition the initial array in two halves

Conquer: recursively sort each half. If we only have one number, it is sorted.

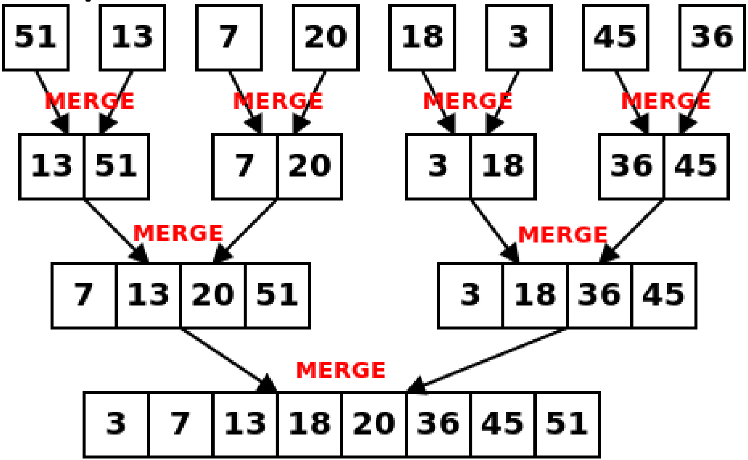
Combine: merge the two sorted halves in a final sorted array

Divide:



51 13 7 20 18 3 45 36
51 13 7 20
18 3 45 36
51 13
7 20
18 3
45 36
51
13
7
20
18
3
45
36

Conquer:


51
13
7
20
18
3
45
36
MERGE
MERGE
MERGE
MERGE
13 51
7 20
3 18
36 45
MERGE
MERGE
7 13 20 51
3 18 36 45
MERGE
3 7 13 18 20 36 45 51

What is the **execution time** of this algorithm?

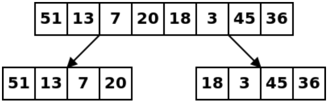
- D(n) - Time to partition an array of size n in two halves
- M(n) - Time to merge two sorted arrays of size n
- T(n) - Time for a MergeSort on an array of size n

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ D(n) + 2T(n/2) + M(n) & \text{if } n > 1 \end{cases}$$

In practice, we are ignoring certain details, but it suffices
(ex: when n is odd, the size of subproblem is not exactly $n/2$)

Divide and Conquer
MergeSort

D(n) - Time to partition an array of size *n* in two halves



I don't need to create a copy of the array

Let's use a function with two arguments:
`mergesort(a,b)`: (sort from position *a* to position *b*)

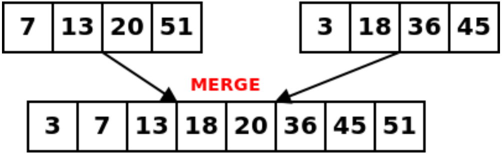
Initially, `mergesort(0, n-1)` (with arrays starting at position 0)

Let $m = \lfloor (a+b)/2 \rfloor$ be the middle position
Calls to `mergesort(a,m)` and `mergesort(m+1,b)`

I only need to make a math operation (sum + division)
I can partition the array in $\Theta(1)$ (constant time!)

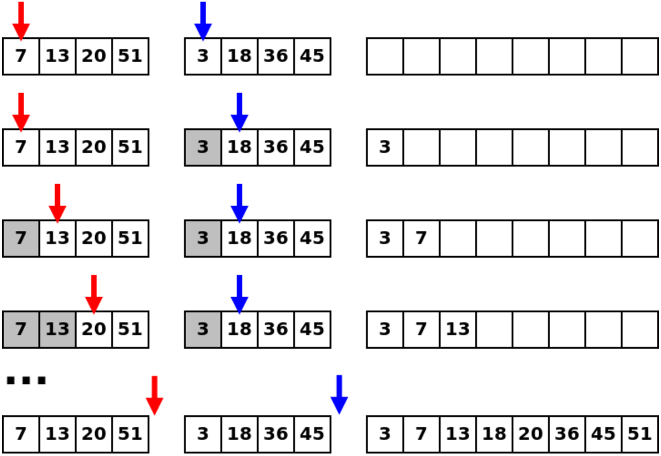
Divide and Conquer
MergeSort

M(n) - Time to merge two sorted arrays of size $n/2$



In constant time it is not possible. What about in **linear** time?

M(n) - Time to merge two sorted arrays of size $n/2$



At the end I made n comparisons + n copies, spending $\Theta(\text{n})$ (linear time)

Back to the mergesort recurrence:

- D(n) - Time to partition an array of size n in two halves
- M(n) - Time to merge two sorted arrays of size n
- T(n) - Time for a MergeSort on an array of size n

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ D(n) + 2T(n/2) + M(n) & \text{if } n > 1 \end{cases}$$

becomes

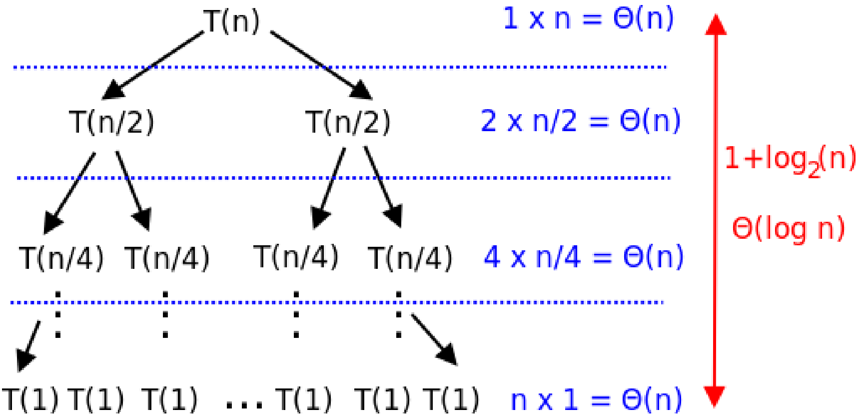
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

How to solve this recurrence?

(for a cleaner explanation we will assume $n = 2^k$,
but the results holds for any n)

Divide and Conquer
MergeSort

Let's draw the **recursion tree**:



Summing everything we get that **MergeSort** is $\Theta(n \log_2 n)$

Divide and Conquer

MaxD&C

A recursive algorithm is not always **linearithmic**!

Let's see another example. Imagine that you want to compute the **maximum** of an array of size n.

A simple **linear search** would be enough, but let's design a divide and conquer algorithm.

Computing the maximum

Divide: partition the initial array in two halves

Conquer: recursively compute the maximum in each half. If we only have one number, it is the maximum

Combine: compare the maximum of each half and keep the largest one

What is the **execution time** of this algorithm?

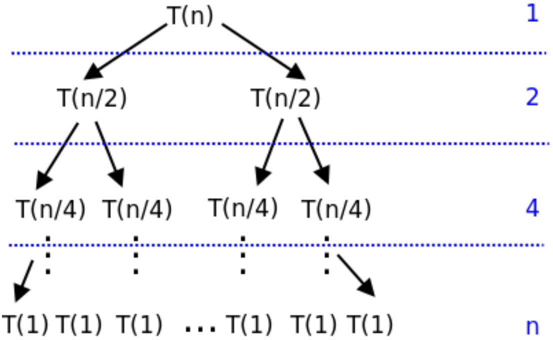
To simplify, let's again admit that n is a power of 2.
(the results are similar in their essence for other cases)

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(n/2) + \Theta(1) & \text{se } n > 1 \end{cases}$$

How does this differ from the MergeSort recurrence?
How to **solve** it?

Divide and Conquer

MaxD&C



In total we spend $1 + 2 + 4 + \ldots + n = \sum_{i=0}^{\log_2(n)} 2^i = 2n - 1$

What dominates the sum? Note that $2^k = 1 + \sum_{i=0}^{k-1} 2^i$.

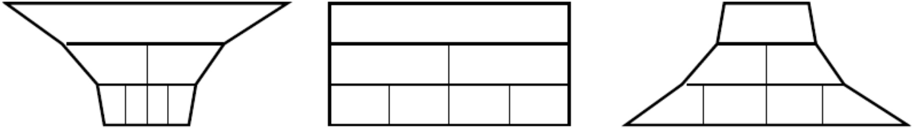
The last level dominates the weight and thus the algorithm is $\Theta(\text{n})$

Recursion

complexity

Solving general recurrences is out of the scope of this course, but the most common recursive algrithms fall on **one of three cases**:

- The time is (uniformly) **distributed** along the recursion tree (e.g. mergesort)
- The time is dominated by the **last level** of the recursion (e.g. maxD&C)
- The time is dominated by the **top level** of the recursion (e.g. naive matrix multiplication)



(to know more take a look at the Master Theorem)

Recurrences

Notation

It is common to assume that $T(1) = \Theta(1)$. In these cases we can simply write $T(n)$ to describe a recurrence.

- **MergeSort:** $T(n) = 2T(n/2) + \Theta(n)$
- **MaxD&C:** $T(n) = 2T(n/2) + \Theta(1)$

Sometimes we have an algorithm that reduces the problem to a single subproblem.

In this case we can say we use **decrease and conquer**

- **Binary Search:**
  On a sorted array of size n, compare with the middle element and continue the search on one half
  $T(n) = T(n/2) + \Theta(1)$ [$\Theta(\log n)$]

- **Max with "tail recursion":** On an array of size n, recursively find the maximum of the entire array except the first element and then compare with that first element
  $T(n) = T(n-1) + \Theta(1)$ [$\Theta(n)$]