

# Algorithmics: Basic definitions and concepts

The course  
Algorithms:  
our context  
Time  
complexity  
Asymptotic  
notation  
Graphs  
Data structures  
Traversals  
Reductions  
Divide and  
conquer



# The Algorithms course

## Already known (EDA level)

- Algorithms cost and Asymptotic notation
- Sorting algorithms: Mergesort, Quicksort, ...
- Divide and conquer, recurrences, master theorem
- Complexity, P and NP, reductions
- Foundations on probability
- Basic data structures: Arrays, lists, stacks, queues, heaps, hashing ...
- Basics on graph theory, graph data structures
- Graph and digraph traversals (BFS, DFS) and applications.
- Backtracking algorithms

# The Algorithmics course

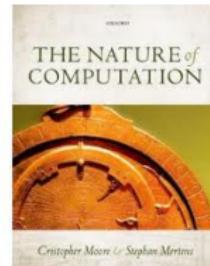
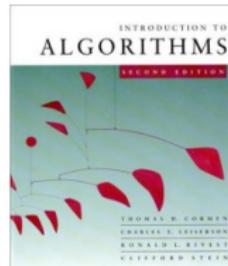
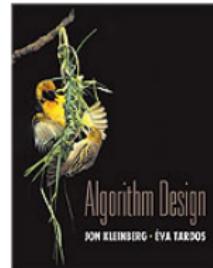
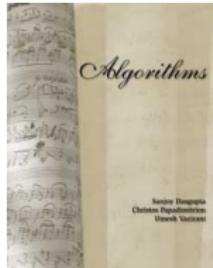
## Topics to cover:

- Divide and conquer: Linear Selection
- Sorting in linear time
- Greedy algorithms
- Dynamic programming
- Distances in graphs
- Flow networks: problems, algorithms and applications
- Linear Programming
- Approximation algorithms

Provide models to **solve** real problems

# References

Main references:



# "The algorithmic lenses: C. Papadimitriou"

- In 1936 Alan Turing demonstrated the universality of computational principles with his mathematical model of the Turing machine.
- Theoretical Computer Science views computation as a ubiquitous phenomenon, not one that it is limited to computers.
- Algorithms themselves have evolved into a complex set of techniques, for instances self-learning, Web services, concurrent, distributed or parallel, etc... Each of them with ad-hoc relevant computational limitations and social implications.

# "The algorithmic lenses: C. Papadimitriou"

- In 1936 Alan Turing demonstrated the universality of computational principles with his mathematical model of the Turing machine.
- Theoretical Computer Science views computation as a ubiquitous phenomenon, not one that it is limited to computers.
- Algorithms themselves have evolved into a complex set of techniques, for instances self-learning, Web services, concurrent, distributed or parallel, etc... Each of them with ad-hoc relevant computational limitations and social implications.
- However, this course will be a course on classical algorithms, which are the core needed to understand more advanced computational material.

# Algorithms

Algorithm: Precise recipe for a precise computational task.  
Each step of the process must be clear and unambiguous, and  
it should always yield a clear answer.

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

Data structures

Traversals

Reductions

Divide and  
conquer

# Algorithms

The course  
Algorithms:  
our context  
Time  
complexity  
Asymptotic  
notation  
Graphs  
Data structures  
Traversals  
Reductions  
Divide and  
conquer

Algorithm: Precise recipe for a precise computational task.  
Each step of the process must be clear and unambiguous, and  
it should always yield a clear answer.

**Sqrt (n)**

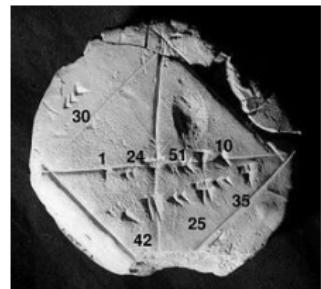
$x_0 = 1$

**for**  $i = 1$  to 6 **do**

$x_i = (x_{i-1} + n/x_{i-1})/2$

**end for**

Babilònia (XVI BC)



# Algorithms

The course  
Algorithms:  
our context  
Time  
complexity  
Asymptotic  
notation  
Graphs  
Data structures  
Traversals  
Reductions  
Divide and  
conquer

Algorithm: Precise recipe for a precise computational task.  
Each step of the process must be clear and unambiguous, and  
it should always yield a clear answer.

Sqrt ( $n$ )

$x_0 = 1$

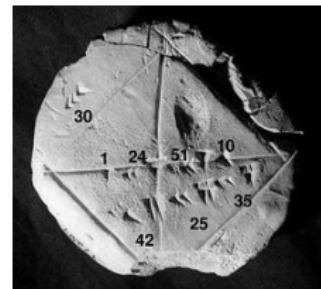
**for**  $i = 1$  to 6 **do**

$x_i = (x_{i-1} + n/x_{i-1})/2$

**end for**

Babilònia (XVI BC)

For  $n = 20$ ,  $x$ 's are 1 10.5 6.2023 4.7134 4.4783



# Once we designed an algorithm: What do we want to know?

- Correctness, it always does what it should?
- Performance,
  - computing time,
  - memory use
  - communication cost, ...

For an algorithm  $\mathcal{A}$ ,  $t_{\mathcal{A}}(x)$  is the computing time on input  $x$ .

In this course, we use a **worst case analysis**: Given a problem, for which you designed an algorithm, you assume that your meanest adversary gives you the worst possible input.

We use as measure of **time complexity** or **cost** the function

$$T(n) = \max_{|x|=n} t_{\mathcal{A}}(x)$$

# Time complexity

The time complexity must be independent of the "used" machine

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

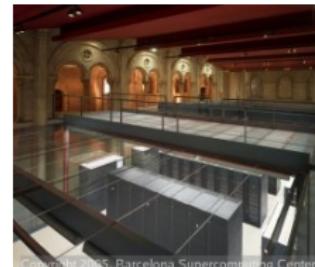
Graphs

Data structures

Traversals

Reductions

Divide and  
conquer



# Time complexity

The time complexity must be independent of the "used" machine

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

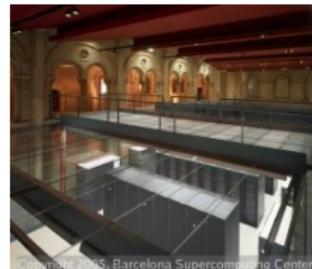
Graphs

Data structures

Traversals

Reductions

Divide and  
conquer



We must consider carefully how operations scale with respect to size.

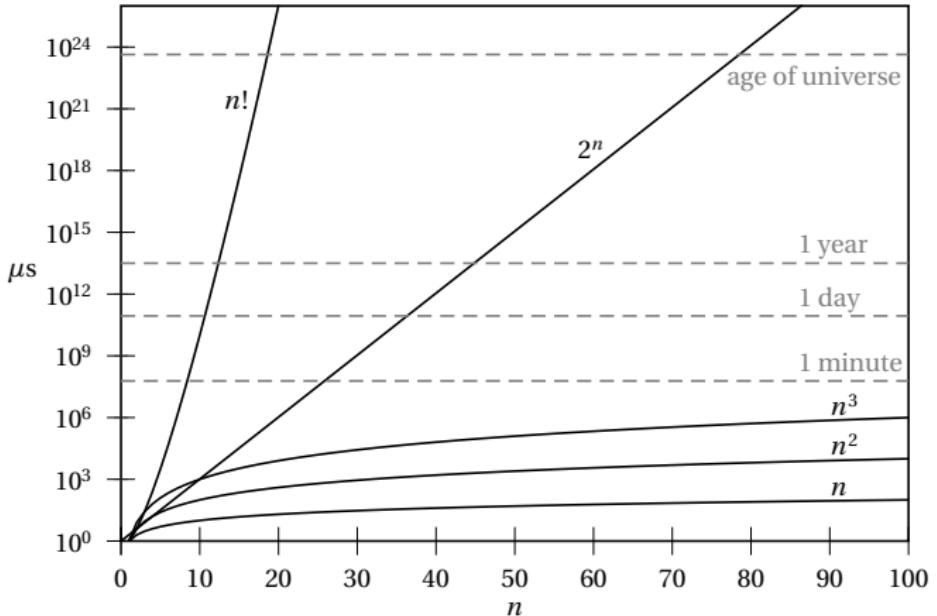
# Typical computation times

The course  
Algorithms:  
our context  
  
Time  
complexity  
  
Asymptotic  
notation  
  
Graphs  
Data structures  
Traversals  
  
Reductions  
  
Divide and  
conquer

We study the behavior of  $T(n)$  when  $n$  can take very large values (i.e.,  $n \rightarrow \infty$ )

- if  $n = 10$ ,  $n^2 = 100$  and  $2^n = 1024$ ;
- if  $n = 100$ ,  $n^2 = 10000$  and  
 $2^n = 12676506002282244014696703205376$ ;
- if  $n = 10^3$ ,  $n^2 = 10^6$  and  $2^n$  is a number with 302 digits.
- As a comparison,  $10^{64}$  is estimated to be the number of atoms in hearth ( $< 2^{213}$ ).

# Computation time assuming that an input with size $n = 1$ can be solved in 1 $\mu$ second:



From: Moore-Mertens, The Nature of Computation

# An example

- Assume that the input to an algorithm is an integer  $x$  that uses 64 bits.

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

Data structures

Traversals

Reductions

Divide and  
conquer

# An example

- Assume that the input to an algorithm is an integer  $x$  that uses 64 bits.
- The cost of the algorithm is  $O(x)$  and the time units are nanoseconds.

# An example

- Assume that the input to an algorithm is an integer  $x$  that uses 64 bits.
- The cost of the algorithm is  $O(x)$  and the time units are nanoseconds.
- Thus, processing this input takes more than

# An example

- Assume that the input to an algorithm is an integer  $x$  that uses 64 bits.
- The cost of the algorithm is  $O(x)$  and the time units are nanoseconds.
- Thus, processing this input takes more than 500 years

# An example

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

Data structures

Traversals

Reductions

Divide and  
conquer

- Assume that the input to an algorithm is an integer  $x$  that uses 64 bits.
- The cost of the algorithm is  $O(x)$  and the time units are nanoseconds.
- Thus, processing this input takes more than 500 years

Note that the cost of this algorithm is a polynomial function on the input value not on the input size.

# An example

- Assume that the input to an algorithm is an integer  $x$  that uses 64 bits.
- The cost of the algorithm is  $O(x)$  and the time units are nanoseconds.
- Thus, processing this input takes more than 500 years

Note that the cost of this algorithm is a polynomial function on the input value not on the input size.  
Such algorithms are classified as pseudopolynomial.

# Efficient algorithms and practical algorithms

- When analyzing an algorithm, we say that it is **feasible** if its cost is polynomial.  
 $n^{10^{10}}$  is a polynomial but this computing time could be prohibitive!
- In the same way, if we have  $cn^2$  for constant  $c = 10^{64}$ , then  $c$  dominates inputs up to a size of  $n > 10^{64}$ .
- In this course, we will not enter in the analysis up to constants, but keep in mind that constants matter!!!!
- In practice, even for feasible algorithms with time complexity of for example  $n^4$ , it could be too slow for  $n \geq 1000$ .

# Asymptotic notation

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

Data structures

Traversals

Reductions

Divide and  
conquer

| Symbol                | $L = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ | intuition  |
|-----------------------|---|------------|
| $f(n) = O(g(n))$      | $L < \infty$  | $f \leq g$ |
| $f(n) = \Omega(g(n))$ | $L > 0$   | $f \geq g$ |
| $f(n) = \Theta(g(n))$ | $0 < L < \infty$                                    | $f = g$    |
| $f(n) = o(g(n))$      | $L = 0$   | $f < g$    |
| $f(n) = \omega(g(n))$ | $L = \infty$  | $f > g$    |

# Names used for specific function classes

The course  
Algorithms:  
our context  
Time  
complexity  
Asymptotic  
notation  
Graphs  
Data structures  
Traversals  
Reductions  
Divide and  
conquer

| name               | definition                                     |
|--------------------|--|
| polylogarithmic    | $f = O(\log^c n)$ ( $c$ constant)              |
| polynomial         | $f = O(n^c)$ ( $c$ constant) or $n^{O(1)}$     |
| subexponential     | $f = o(2^{n^\epsilon})$ ( $0 < \epsilon < 1$ ) |
| exponential        | $f = 2^{\text{poly}(n)}$                       |
| double exponential | $f = 2^{\exp(n)}$                              |

## Notation:

$\lg \equiv \log_2$ ;  $\ln \equiv \log_e$ ;  $\log \equiv \log_{10}$ .

# Some math. you should remember

Given an integer  $n > 0$  and a real  $a > 1$  and  $a \neq 0$ :

- Arithmetic summation:  $\sum_{i=0}^n i = \frac{n(n+1)}{2}$ .
- Geometric summation:  $\sum_{i=0}^n a^i = \frac{1-a^{n+1}}{1-a}$ .

Logarithms and Exponents: For  $a, b, c \in \mathbb{R}^+$ ,

- $\log_b a = c \Leftrightarrow a = b^c \Rightarrow \log_b 1 = 0$
- $\log_b ac = \log_b a + \log_b c$ ,  $\log_b a/c = \log_b a - \log_b c$ .
- $\log_b a^c = c \log_b a \Rightarrow c^{\log_b a} = a^{\log_b c} \Rightarrow 2^{\log_2 n} = n$ .
- $\log_b a = \log_c a / \log_c b \Rightarrow \log_b a = \Theta(\log_c a)$

Stirling:  $n! = \sqrt{2\pi n}(n/e)^n + O(1/n) + \gamma \Rightarrow n! + \omega((n/2)^n)$ .

$n$ -Harmonic:  $H_n = \sum_{i=1}^n 1/i \sim \ln n$ .

# Graphs

See for ex. Chapter 3 of Dasgupta, Papadimitriou, Vazirani (DPV).

**Graph:**  $G = (V, E)$ , where  $V$  is the set of vertices,  $n = |V|$ , and  $E \subset V \times V$  is the set of edges,  $m = |E|$ ,

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

**Graphs**

Data structures

Traversals

Reductions

Divide and  
conquer

# Graphs

See for ex. Chapter 3 of Dasgupta, Papadimitriou, Vazirani (DPV).

**Graph:**  $G = (V, E)$ , where  $V$  is the set of vertices,  $n = |V|$ , and  $E \subset V \times V$  is the set of edges,  $m = |E|$ ,

- Graphs: *undirected graphs (graphs)* and *directed graphs (digraphs)*
- The **degree** of  $v$  ( $d(v)$ ) is the number of edges which are incident to  $v$ .
- A **clique** on  $n$  vertices ( $K_n$ ) is a **complete graph** (with  $m = n(n - 1)/2$ ).
- A undirected  $G$  is said to be **connected** if there is a path between any two distinct vertices.
- If  $G$  is connected, then  $m \geq n - 1$ .

# Directed graphs

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

Data structures  
Traversals

Reductions

Divide and  
conquer

- Edges are directed.
- The connectivity concept in digraphs is the so called **strong connectivity**: There is a directed path between any two vertices.
- In a digraph  $m \leq n(n - 1)$ .

# Density of a graph

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

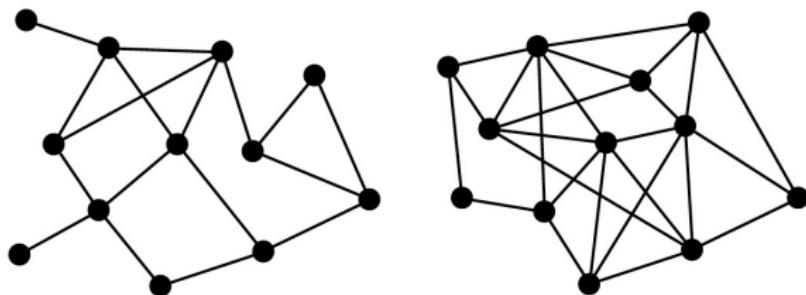
Data structures

Traversals

Reductions

Divide and  
conquer

A  $G$  with  $n$  vertices is said to be **dense** when  $m = \Theta(n^2)$ .  
When  $m = o(n^2)$ ,  $G$  is said to be **sparse**.



# Common graph's data structures

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs  
Data structures

Traversals

Reductions

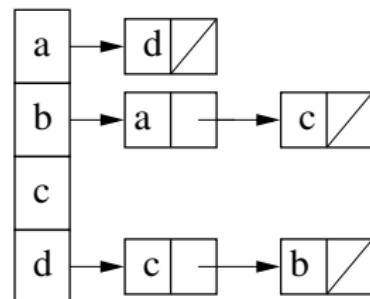
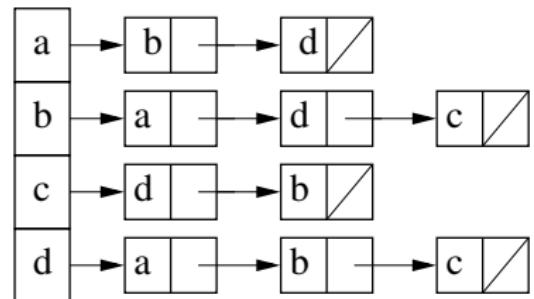
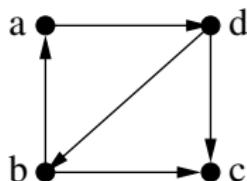
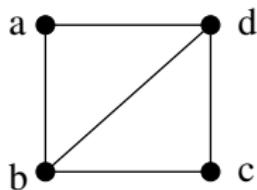
Divide and  
conquer

Let  $G$  be a graph with  $V = \{1, 2, \dots, n\}$ .

Adjacency list

Adjacency matrix

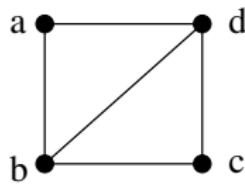
# Adjacency list



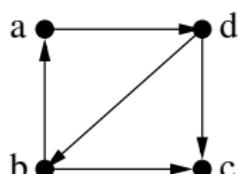
# Adjacency matrix

Given  $G$  with  $|V| = n$  define its **adjacency matrix** as the  $n \times n$  matrix:

$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{if } (i,j) \notin E. \end{cases}$$



$$\begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{matrix}$$



$$\begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} \end{matrix}$$

# Adjacency matrix

- If  $G$  is undirected, its adjacency matrix  $A(G)$  is symmetric.
- If  $A$  is the adjacency matrix of  $G$ , then  $A^2$  gives, for  $i, j \in V$ , whether there is a path between  $i$  and  $j$  in  $G$ , with length 2.  
For  $k > 0$ ,  $A^k$  indicates if there is a path with length  $k$  in  $G$ .
- If  $G$  has weights on edges, i.e.  $w_{i,j}$  for each  $(i,j) \in E$ ,  $A(G)$  has  $w_{ij}$  in  $a_{i,j}$ .
- Adjacency matrix allows the use of tools from linear algebra.

# Comparison between the matrix and the list DS

- The adjacency list uses a register per vertex and two per edge. As each register needs 64 bits, then the space to represent a graph is  $\Theta(n + m)$ .
- The use of the adjacency matrix needs  $n^2$  bits ( $\{0, 1\}$ ), so for an unweighted graph  $G$ , we need  $\Theta(n^2)$  bits.
- For weighted  $G$ , we need  $64n^2$  bits (assuming weights are reasonably “small”).
- In general, for unweighted dense graphs, the adjacency matrix is better, otherwise the adjacency list is a shorter representation.

# Complexity issues between matrix and list DS

The course  
Algorithms:  
our context  
Time  
complexity  
Asymptotic  
notation  
Graphs  
Data structures  
Traversals  
Reductions  
Divide and  
conquer

- Adding a new edge to  $G$ : In both data structures we need  $\Theta(1)$ .
- Edge query, for  $u$  and  $v$  in  $V(G)$ ,  $(u, v) \in E(V)$ ?:
  - For matrix representation:  $\Theta(1)$ .
  - For list representation:  $O(n)$ .
- Explore all neighbours of vertex  $v$ :
  - For matrix representation:  $\Theta(n)$
  - For list representation:  $\Theta(|d(v)|)$
- Erase an edge in  $G$ : The same as Edge query.
- Erase a vertex in  $G$ :
  - For matrix representation:  $\Theta(n)$ .
  - For list representation:  $O(m)$ .

# Searching a graph: Breadth First Search

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

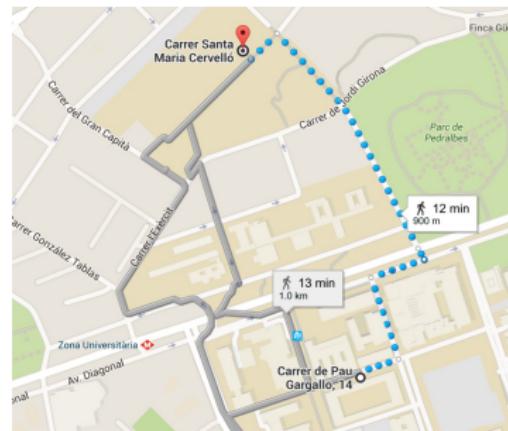
Data structures

Traversals

Reductions

Divide and  
conquer

- 1 Start with vertex  $v$ , visit  $v$  and all its neighbors.
- 2 Then, the non-visited neighbors of visited ones.
- 3 Repeat until all vertices are visited.



BFS use a QUEUE, (FIFO) to keep the neighbors of visited vertices.

Recall that vertices are labeled to avoid visiting them more than once.

# Searching a graph: Depth First Search

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

Data structures

Traversals

Reductions

Divide and  
conquer

## explore

- 1 From current vertex, move to a neighbor.
- 2 Until you get stuck.
- 3 Then backtrack till new place to explore.



DFS use a STACK, (LIFO)

# Time Complexity of DFS and BFS

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

Data structures

Traversals

Reductions

Divide and  
conquer

For graphs given by adjacency lists:

$$O(|V| + |E|)$$

For graphs given by adjacency matrix:

$$O(|V|^2)$$

Therefore, both procedures can be implemented in linear time  
with respect to the size of the input graph.

# Connected components in undirected graphs

A connected component is a maximal connected subgraph of  $G$ .

A connected graph has a unique connected component.

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

Data structures

Traversals

Reductions

Divide and  
conquer

# Connected components in undirected graphs

A connected component is a maximal connected subgraph of  $G$ .

A connected graph has a unique connected component.

## Connected Components

INPUT: undirected graph  $G$

QUESTION: Find all the connected components of  $G$ .

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

Data structures

Traversals

Reductions

Divide and  
conquer

# Connected components in undirected graphs

A connected component is a maximal connected subgraph of  $G$ .

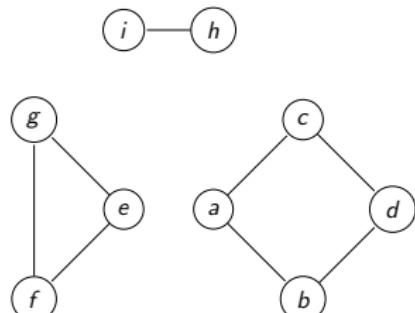
A connected graph has a unique connected component.

## Connected Components

INPUT: undirected graph  $G$

QUESTION: Find all the connected components of  $G$ .

To find connected components in  $G$  use DFS and keep track of the set of vertices visited in each **explore** call.



The problem can be solved in  $O(|V| + |E|)$ .

# Strongly connected components in a digraph

A digraph  $G = (V, E)$ , is *strongly connected*, if for all  $u, v \in V$ , there are paths  $u \rightarrow v$  and  $v \rightarrow u$ .

A strongly connected component is a maximal strongly connected graph.

## Strongly Connected Components

INPUT: digraph  $G$

QUESTION: Find the strongly connected components of  $G$ .

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

Data structures

Traversals

Reductions

Divide and  
conquer

# Strongly connected components in a digraph

A digraph  $G = (V, E)$ , is *strongly connected*, if for all  $u, v \in V$ , there are paths  $u \rightarrow v$  and  $v \rightarrow u$ .

A strongly connected component is a maximal strongly connected graph.

## Strongly Connected Components

INPUT: digraph  $G$

QUESTION: Find the strongly connected components of  $G$ .

Kosaraju-Sharir's algorithm: Uses DFS (twice). Complexity  $T(n) = O(|V| + |E|)$

# Strongly connected components in a digraph

A digraph  $G = (V, E)$ , is *strongly connected*, if for all  $u, v \in V$ , there are paths  $u \rightarrow v$  and  $v \rightarrow u$ .

A strongly connected component is a maximal strongly connected graph.

## Strongly Connected Components

INPUT: digraph  $G$

QUESTION: Find the strongly connected components of  $G$ .

Kosaraju-Sharir's algorithm: Uses DFS (twice). Complexity  $T(n) = O(|V| + |E|)$

Tarjan's algorithm: Using DFS (once). Complexity  $T(n) = O(|V| + |E|)$

Both algorithms are optimal, i.e. linear time, but in practice Tarjan's algorithm is easier to implement.

# Strongly connected components in a digraph

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

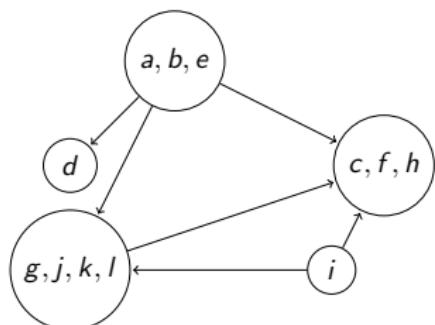
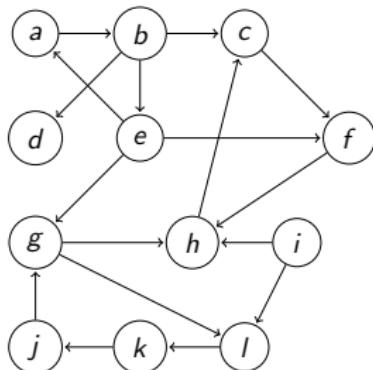
Data structures

Traversals

Reductions

Divide and  
conquer

A nice property: For every digraph, the graph on its strongly connected components is *acyclic*.



# The classes P and NP

- Recall that a problem belongs to the class P if there exists an algorithm that is polynomial in the worst-case analysis, (for the worst input given by a malicious adversary)
- A problem given in **decisional form** belongs to the class NP **non-deterministic polynomial time** if given a certificate of a solution we can verify in polynomial time that indeed the certificate is a valid solution to the problem in decisional form and those certificates have polynomial size
- It is easy to see that  $P \subseteq NP$ , but it is an open problem to prove that  $P=NP$  or that  $P \neq NP$ .
- The class NP-complete is the class of most difficult problems in decisional form that are in NP. Most difficult in the sense that if one of them is proved to be in P then  $P=NP$ .

# Beyond worst-case analysis

- Under the hypothesis that  $P \neq NP$ , if the decision version of a problem is NP-complete, then the optimization problem will require at least exponential time, for some inputs.
- The classification of a problem as NP-complete is a case of worst-case analysis, and for many problems the "expensive inputs" are few, and far from practical typical inputs. We will see some examples through the course.
- Therefore, there are alternative ways to get in practice, solutions for NP-complete problems, with the use of alternative algorithmic techniques, as approximation (we will see some examples), heuristics and self-learning algorithms, that are deferred to other courses.

# A powerful tool to solve problems: Reductions

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

Data structures

Traversals

**Reductions**

Divide and  
conquer

You have been introduced in previous courses to the concept of reduction between decision problems, to define the class NP-complete.

We have to extend the concept to function problems.

# Reductions

Given problems  $A$  and  $B$ , assume we have an algorithm  $\mathcal{A}_B$  to solve the problem  $B$  on any input  $y$ .

A polynomial time **reduction**  $A \leq B$  is a pair of polynomial time functions  $(f, g)$  such that

- $f$  maps any input  $x$  to  $A$ , in polynomial time to an input  $f(x)$  to problem  $B$  in such a way that  $x$  has a valid solution for  $A$  iff  $f(x)$  has a valid solution for  $B$ .
- $g$  maps solutions to  $f(x)$  into solutions to  $x$ .

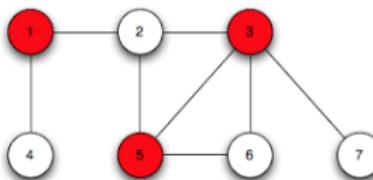
Therefore if we have that  $A \leq B$ , as there is an algorithm  $\mathcal{A}_B$  to solve problem  $B$  in polynomial time, then we have an algorithm  $\mathcal{A}_A$ , for any input  $x$  of  $A$ : Compute  $g(\mathcal{A}_B(f(x)))$ , that runs in polynomial time.

# The VERTEX COVER problem

The course  
Algorithms:  
our context  
Time  
complexity  
Asymptotic  
notation  
Graphs  
Data structures  
Traversals  
Reductions  
Divide and  
conquer

**VERTEX COVER:** Given a graph  $G = (V, E)$  with  $|V| = n, |E| = m$ , find the minimum set of vertices  $S \subseteq V$  such that it covers every edge of  $G$ .

Example:



The VERTEX COVER problem is known to be in NP-hard.

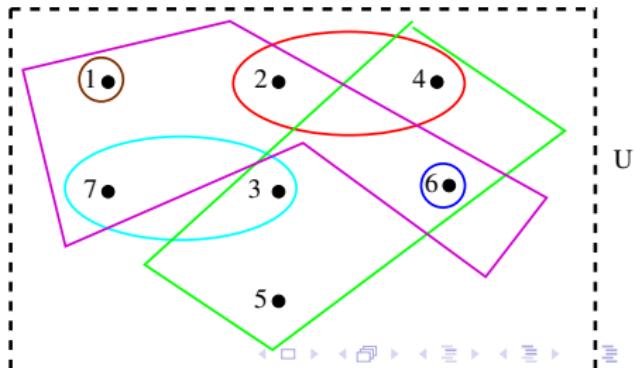
# The SET COVER problem

**SET COVER:** Given a set  $U$  of  $m$  elements, a collection  $S = \{S_1, \dots, S_n\}$  where each  $S_i \subseteq U$ , select the minimum number of subsets in such a way that their union is equal to  $U$ .

There is a weighted version of the problem, but this simpler version already is NP-hard.

**Example:** Given  $U = \{1, 2, 3, 4, 5, 6, 7\}$  ( $m = 7$ ), with  $S_a = \{3, 7\}$ ,  $S_b = \{2, 4\}$ ,  $S_c = \{3, 4, 5, 6\}$ ,  $S_d = \{6\}$ ,  $S_e = \{1\}$ ,  $S_f = \{1, 2, 6, 7\}$ .

**Solution:**  $\{S_c, S_f\}$



# SET COVER

The course

Algorithms:  
our context

Time  
complexity

Asymptotic  
notation

Graphs

Data structures

Traversals

Reductions

Divide and  
conquer

The VERTEX COVER problem is a special case of the SET COVER problem. As a model, the SET COVER has important practical applications.

To understand the computational complexity of SET COVER it is important to understand first the complexity of special cases as VERTEX COVER.

# VERTEX COVER $\leq$ SET COVER

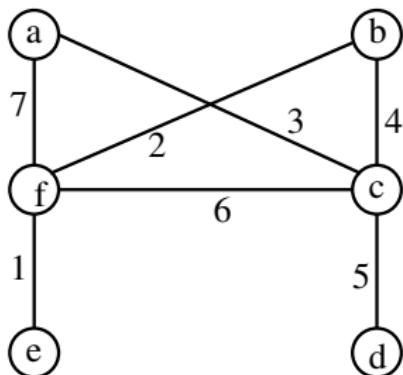
Given an input to VERTEX COVER  $G = (V, E)$ , of size  $|V| + |E| = n + m$  we want to construct in polynomial time on  $n + m$  a specific input  $f(G) = (U, S)$  to SET COVER such that if there exist a polynomial algorithm  $\mathcal{A}$  to find a min. vertex cover in  $G$ , then  $\mathcal{A}(f(G))$  is an efficient algorithm to find an optimal solution to set cover.

## REDUCTION $f$ :

- Consider  $U$  as the set  $E$  of edges.
- For each vertex  $i \in V$ ,  $S_i$  is the set of edges incident to  $i$ .  
Therefore  $|S| = n$  and for each  $S_i$ ,  $|S_i| \leq m$ .
- The cost of the reduction from  $G$  to  $(U, S)$  is  $O(n + m)$

# Example for the reduction

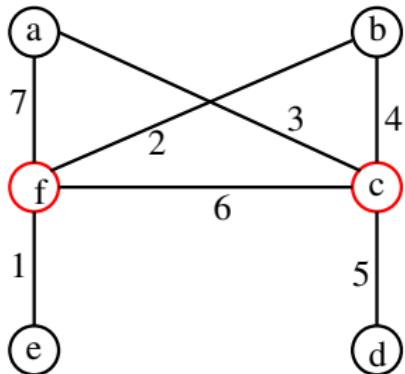
The course  
Algorithms:  
our context  
Time  
complexity  
Asymptotic  
notation  
Graphs  
Data structures  
Traversals  
Reductions  
Divide and  
conquer



$$\begin{aligned} U &= \{1, 2, 3, 4, 5, 6, 7\} \\ S &= \{S_a, S_b, S_c, S_d, S_e, S_f\} \\ \overset{f}{\overbrace{S_a}} &= \{3, 7\}, S_b = \{3, 7\}, \\ \overset{f}{\overbrace{S_c}} &= \{3, 4, 5, 6\}, S_d = \{5\}, \\ S_e &= \{1\}, S_f = \{1, 2, 6, 7\}. \end{aligned}$$

# Example for the reduction

The course  
Algorithms:  
our context  
Time  
complexity  
Asymptotic  
notation  
Graphs  
Data structures  
Traversals  
Reductions  
Divide and  
conquer



$$\begin{aligned}U &= \{1, 2, 3, 4, 5, 6, 7\} \\S &= \{S_a, S_b, S_c, S_d, S_e, S_f\} \\S_a &= \{3, 7\}, S_b = \{3, 7\}, \\&\stackrel{f}{\Rightarrow} S_c = \{3, 4, 5, 6\}, S_d = \{5\}, \\S_e &= \{1\}, S_f = \{1, 2, 6, 7\}.\end{aligned}$$

If there is an algorithm to solve the SET COVER for  $G$ , the same algorithm apply to  $(U, S) = f(G)$  will yield a solution for VERTEX COVER on input  $(U, V)$ .

But both VERTEX COVER and SET COVER are known to be NP-hard.

# The divide-and-conquer strategy.

The course  
Algorithms:  
our context  
Time  
complexity  
Asymptotic  
notation  
Graphs  
Data structures  
Traversals  
Reductions  
Divide and  
conquer

- 1 Break the problem into smaller subproblems,
- 2 recursively solve each problem,
- 3 appropriately combine their answers.



Julius Caesar (I-BC)  
*"Divide et impera"*

## Known Examples:

- Binary search
- Merge-sort
- Quicksort
- Strassen matrix multiplication

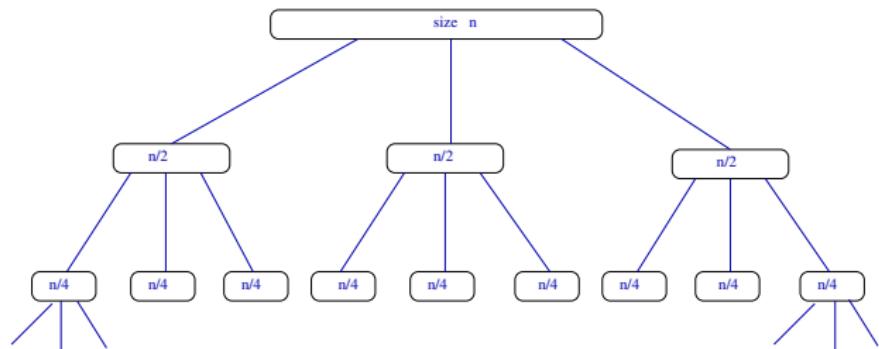


J. von Neumann  
(1903-57) Merge sort

# Recurrences Divide and Conquer

$$T(n) = 3T(n/2) + O(n)$$

The algorithm under analysis divides input of size  $n$  into 3 subproblems, each of size  $n/2$ , at a cost (of dividing and joining the solutions) of  $O(n)$

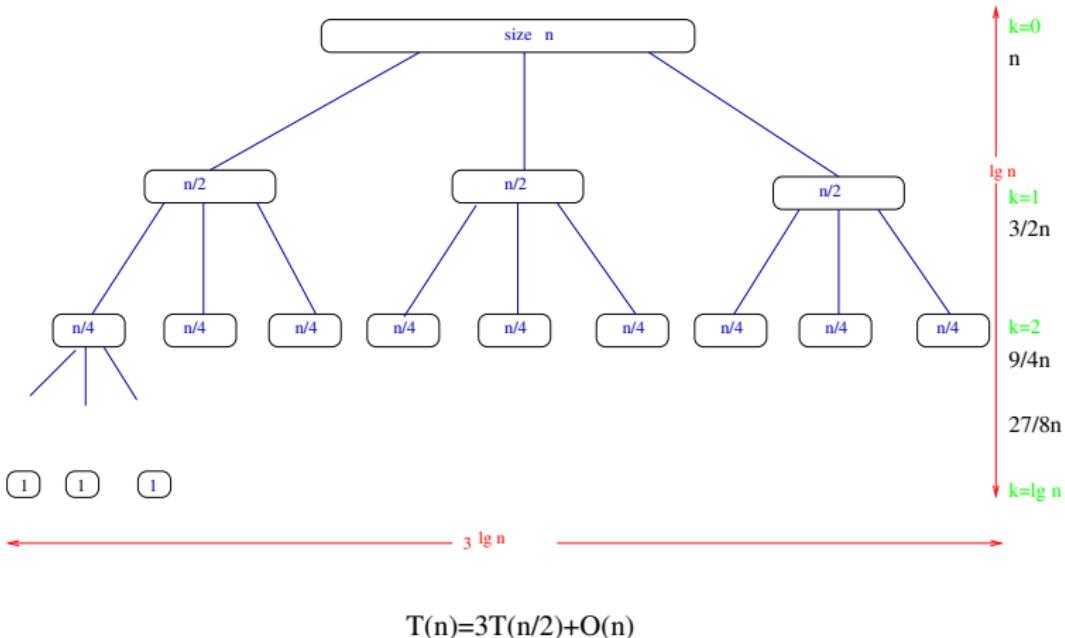


1 1 1

1 1 1

$$T(n) = 3T(n/2) + O(n).$$

The course  
 Algorithms:  
 our context  
 Time  
 complexity  
 Asymptotic  
 notation  
 Graphs  
 Data structures  
 Traversals  
 Reductions  
 Divide and  
 conquer



At depth  $k$  of the tree there are  $3^k$  subproblems, each of size  $n/2^k$ .

For each of those problems we need  $O(n/2^k)$  (splitting time + combination time).

Therefore, for some constant  $c$ , the cost at depth  $k$  is:

$$3^k \times \left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \times c n.$$

with max. depth  $k = \lg n$ , so  $T(n)$  is

$$\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \left(\frac{3}{2}\right)^3 + \cdots + \left(\frac{3}{2}\right)^{\lg n}\right) c n$$

From  $T(n) = c n \left( \sum_{k=0}^{\lg n} \left(\frac{3}{2}\right)^k \right)$ ,

We have a **geometric series** of ratio  $3/2$ , starting at 1 and ending at  $\left(\left(\frac{3}{2}\right)^{\lg n}\right) = \frac{n^{\lg 3}}{n^{\lg 2}} = \frac{n^{1.58}}{n} = n^{0.58}$ .

As the series is increasing,  $T(n)$  is dominated by the last term:

$$T(n) = c n \left( \frac{n^{\lg 3}}{n} \right) = O(n^{1.58}).$$

# A basic Master Theorem

There are several versions of the Master Theorem to solve D&C recurrences. The one presented below is taken from DPV's book.

## Theorem (DPV-2.2)

If  $T(n) = aT(\lceil n/b \rceil) + O(n^d)$  for constants  $a \geq 1, b > 1, d \geq 0$ , then has asymptotic solution:

$$T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b a, \\ O(n^d \lg n), & \text{if } d = \log_b a, \\ O(n^{\log_b a}), & \text{if } d < \log_b a. \end{cases}$$

# Master Theorems

The course  
Algorithms:  
our context  
  
Time  
complexity  
  
Asymptotic  
notation  
  
Graphs  
Data structures  
Traversals  
  
Reductions  
  
Divide and  
conquer

This basic Master Theorem does not provide always exact bounds

A different one can be found in CLRS's book providing exact bounds but leaving cases outside.

For stronger versions:

**Akra-Bazi Theorem:** <https://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf>

**Salvador Roura Theorems**  
<http://www.lsi.upc.edu/~diaz/RouraMT.pdf>