# Algorithmics: Basic definitions and concepts



# The Algorithmics course



## Already known (EDA level)

- Algorithms cost and Asymptotic notation

# The Algorithmics course



## Already known (EDA level)

- Algorithms cost and Asymptotic notation
- Sorting algorithms: Mergesort, Quicksort, ...

# The Algorithmics course

## Already known (EDA level)

- Algorithms cost and Asymptotic notation
- Sorting algorithms: Mergesort, Quicksort, ...
- Divide and conquer, recurrences, master theorem

# The Algorithmics 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

# The Algorithmics 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

# The Algorithmics 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 ...

# The Algorithmics 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

# The Algorithmics 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.

# The Algorithmics 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

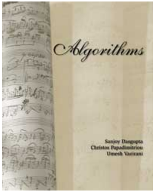
Topics to cover:

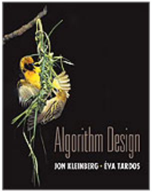
- Divide and conquer: Linear Selection
- Sorting in linear time (when? how?)
- Greedy algorithms
- Dynamic programming
- Distances in graphs
- Flow networks: problems, algorithms and applications
- Linear Programming
- Approximation algorithms
- Streaming algorithms

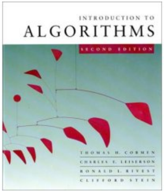
Provide models to solve real problems

# 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.

# "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.

# "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.

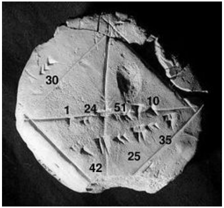
# Algorithms

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.

# 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.

**Sqrt** $(n)$
$x_0 = 1$
**for** $i = 1$ to 6 **do**
  $x_i = (x_{i-1} + n/x_{i-1})/2$
**end for**

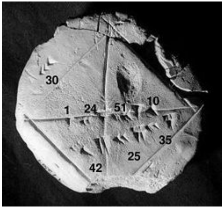



Babilònia (XVI BC)

# Algorithms

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?

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

- Correctness, it always does what it should?

# 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, . . .

# 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.

# 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 $x$.

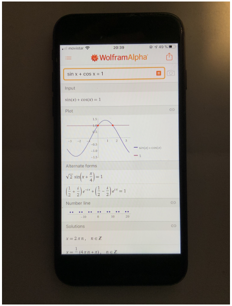
In this course, we use a worst case analysis: Given a problem, for which you designed an algorithm, you assume that your meanest adversary gives you the worst possible input.
We use as measure of time complexity or cost the function

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

.

# Time complexity

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



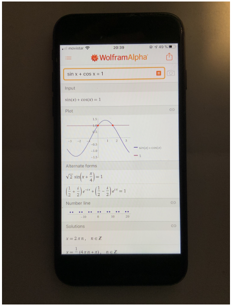


The course
Algorithms: our context
Time complexity
Asymptotic notation
Graphs
Data structures
Traversals
Reductions
Divide and conquer

# Time complexity



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





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

## Typical computation times

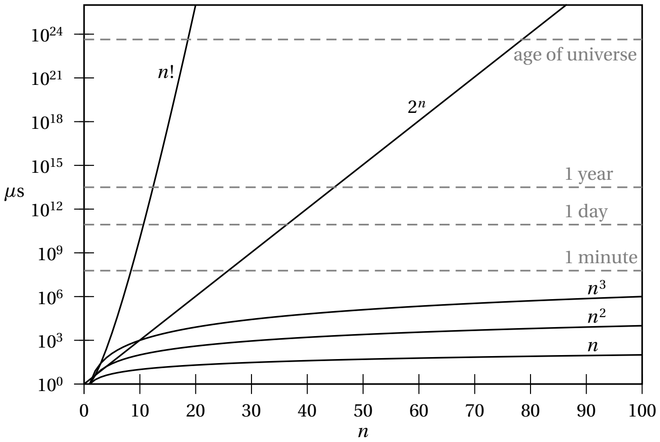


We study the behavior of $T(n)$ when $n$ can take very large values (i.e., $n \to \infty$)

- if $n = 10$, $n^2 = 100$ and $2^n = 1024$;
- if $n = 100$, $n^2 = 10000$ and
  $$2^n = 1267650600228224401469670320 5376;$$
- 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.

# 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



- 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



- We say that an algorithm is feasible if its cost is polynomial.

# Efficient algorithms and practical algorithms

- We say that an algorithm is feasible if its cost is polynomial.
- However $n^{10^{10}}$ is a polynomial but this computing time could be prohibitive!

# Efficient algorithms and practical algorithms



- We say that an algorithm is feasible if its cost is polynomial.
- However $n^{10^{10}}$ 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 $n > 10^{64}$.

# Efficient algorithms and practical algorithms



- We say that an algorithm is feasible if its cost is polynomial.
- However $n^{10^{10}}$ 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!!!!

## Efficient algorithms and practical algorithms



- We say that an algorithm is feasible if its cost is polynomial.
- However $n^{10^{10}}$ 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 a feasible algorithms with time complexity of for example $n^4$ could be too slow for $n \geq 1000$.

# Asymptotic notation

| Symbol | $L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$ | 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

| 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 | $f = 2^{\mathrm{poly}(n)}$ |
| 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: $n! = \sqrt{2\pi n}(n/e)^n + 0(1/n) + \gamma \Rightarrow n! + \omega((n/2)^n)$.

$n$-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, $m = |E|$,

# 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

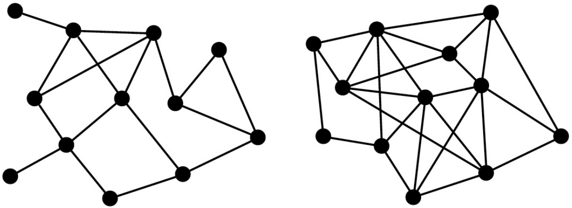


- Edges are directed.
- The connectivity concept in digraphs is the so called strong connectivity: There is is a directed path between any two vertices.
- In a digraph $m \leq n(n-1)$.

# Density of a graph



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

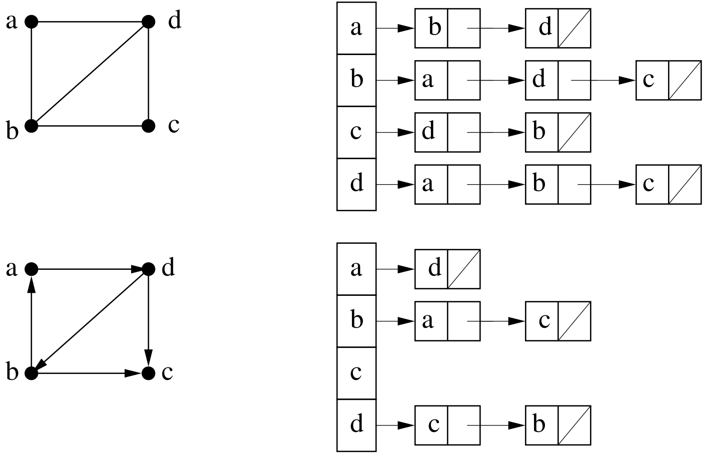


Let $G$ be a graph with $V = \{1, 2, \ldots, n\}$.

Adjacency list

Adjacency matrix

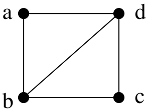
# Adjacency list



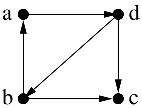
# Adjacency matrix

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

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




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




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

# 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 $k$ from $i$ to $j$.
- If $G$ has weights on edges, i.e. $w_{i,j}$ for each $(i, j) \in E$, $A(G)$ keeps $w_{ij}$ in position $(i, j)$.
- Adjacency matrices allow 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 $\Theta(n+m)$.

# 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").

# 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

# Complexity issues between matrix and list DS

- Adding a new edge to $G$:

# Complexity issues between matrix and list DS



- Adding a new edge to $G$: In both data structures, we need $\Theta(1)$.

# Complexity issues between matrix and list DS



- 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:

# Complexity issues between matrix and list DS



- 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: $\Theta(1)$.

# Complexity issues between matrix and list DS



- 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:

# Complexity issues between matrix and list DS



- 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: $O(n)$.

# Complexity issues between matrix and list DS



- 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:

# Complexity issues between matrix and list DS



- 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:

# Complexity issues between matrix and list DS



- 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: $\Theta(|d(v)|)$

# Complexity issues between matrix and list DS



- 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 $G$:

# Complexity issues between matrix and list DS



- 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.

# Complexity issues between matrix and list DS

- 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:

# Complexity issues between matrix and list DS

- 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:

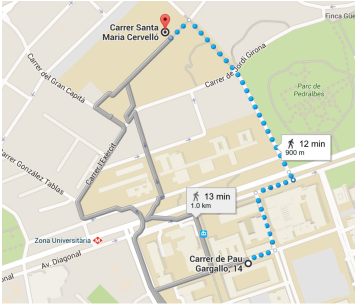
# Complexity issues between matrix and list DS



- 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

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

**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



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.

# 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 Problem
INPUT: undirected graph $G$
GOAL: Find all the connected components of $G$.

# Connected components in undirected graphs

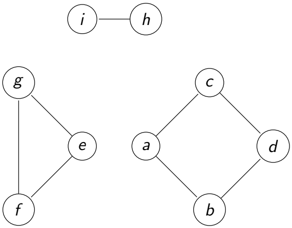


A connected component is a maximal connected subgraph of $G$.

A connected graph has a unique connected component.

Connected Components Problem
INPUT: undirected graph $G$
GOAL: Find all the connected components of $G$.

To find connected components in $G$ use DFS/BFS 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 $u \to v$ and $v \to u$.

A strongly connected component is a maximal strongly connected graph.

Strongly Connected Components Problem
INPUT: digraph $G$
GOAL: Find the strongly connected components of $G$.

# Strongly connected components in a digraph



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

A strongly connected component is a maximal strongly connected graph.

Strongly Connected Components Problem
INPUT: digraph $G$
GOAL: Find the strongly connected components of $G$.

Kosharaju-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 $u \to v$ and $v \to u$.

A strongly connected component is a maximal strongly connected graph.

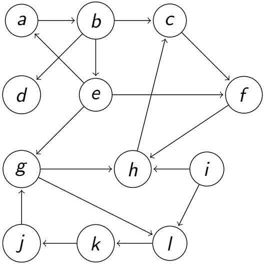
Strongly Connected Components Problem
INPUT: digraph $G$
GOAL: Find the strongly connected components of $G$.

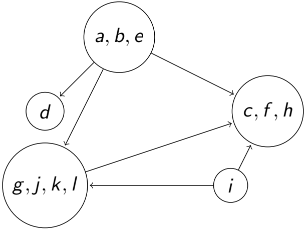
Kosharaju-Sharir's algorithm: Uses DFS (twice). Complexity $T(n) = O(|V| + |E|)$

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

# Strongly connected components in a digraph

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



# The classes P and NP



- Recall that a problem belong 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)

## The classes P and NP



- Recall that a problem belong 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 belong 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 and those certificates have polynomial size

## The classes P and NP

- Recall that a problem belong 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 belong 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 and those certificates have polynomial size
- It is easy to se that P$\subseteq$ NP, but it is an open problem to prove that P$=$NP or that P$\neq$NP.

# The classes P and NP

- Recall that a problem belong 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 belong 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 and those certificates have polynomial size
- It is easy to se 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



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 $y$.

# 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 $x$.

# 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

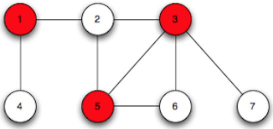
- $f$ maps any input $x$ to $A$, in polynomial time, to an input $f(x)$ to problem $B$ in such a way that $x$ has a valid solution for $A$ iff $f(x)$ has a valid solution for $B$.
- $g$ maps solutions to $f(x)$ into solutions to $x$.

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

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:



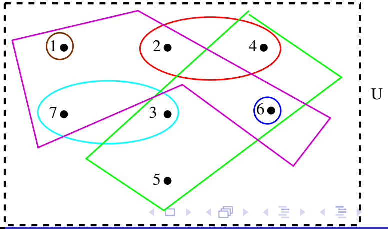

The VERTEX COVER problem is known to be in NP-hard.

SET COVER: Given a set $U$ of $m$ elements, a collection $S = \{S_1, \ldots, S_n\}$ where each $S_i \subseteq U$, select de 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 $S_1 = \{3, 7\}$, $S_2 = \{2, 4\}$, $S_3 = \{3, 4, 5, 6\}$, $S_4 = \{6\}$, $S_5 = \{1\}$, $S_6 = \{1, 2, 6, 7\}$ $(n = 6)$.

Solution: $\{S_3, S_6\}$



# Set Cover



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

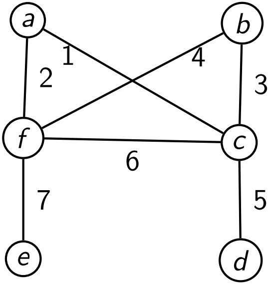
Given a input to VERTEX COVER, $G = (V, E)$ of size $|V| + |E| = n + m$, we want to construct in polynomial time on $n + m$ a specific input $f(G) = (U, S)$ to SET COVER such that if there exist a polynomial algorithm $\mathcal{A}$ to find a min set cover in $G$, then $\mathcal{A}(f(G))$ is an efficient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

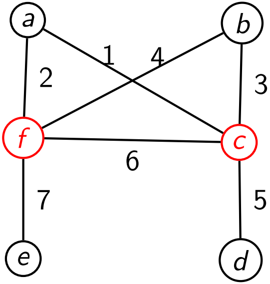





$\overbrace{\Rightarrow}^{f}$

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

# Example for the reduction




$\overset{f}{\overbrace{\Rightarrow}}$

$$U = \{1, 2, 3, 4, 5, 6, 7\}$$
$$S = \{S_a, S_b, S_c, S_d, S_e, S_f\}$$
$$S_a = \{1, 2\},\ S_b = \{3, 4\},$$
$$S_c = \{1, 3, 5, 6\},\ S_d = \{5\},$$
$$S_e = \{7\},\ S_f = \{2, 4, 6, 7\}.$$

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

As VERTEX COVER is known to be NP-hard, this shows that SET COVER is also NP-hard.

# The divide-and-conquer strategy.

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

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

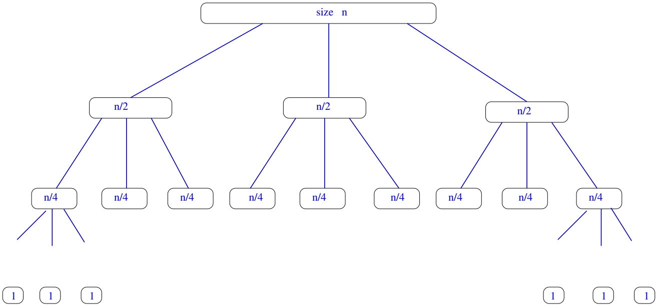
Known Examples:

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

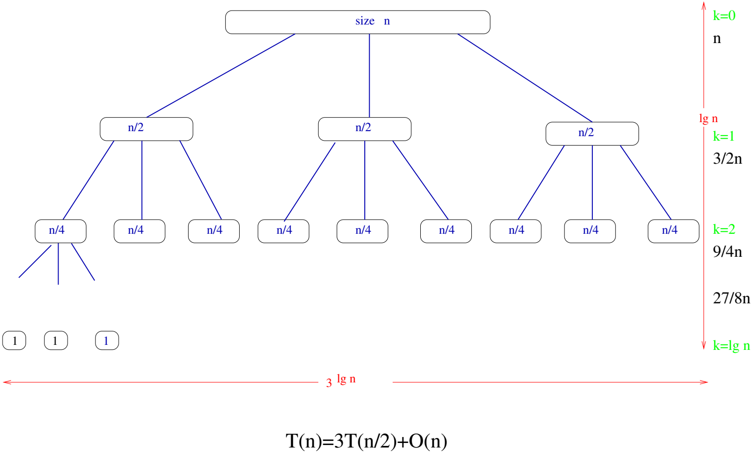
J. von Neumann (1903-57) Merge sort

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

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 $O(n)$



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


size n
n/2
n/2
n/2
n/4
n/4
n/4
n/4
n/4
n/4
n/4
n/4
n/4
1
1
1
k=0
n
lg n
k=1
3/2n
k=2
9/4n
27/8n
k=lg n
3 lg n
T(n)=3T(n/2)+O(n)

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



- 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 look at Akra-Bazi Theorem or Salvador Roura Theorems