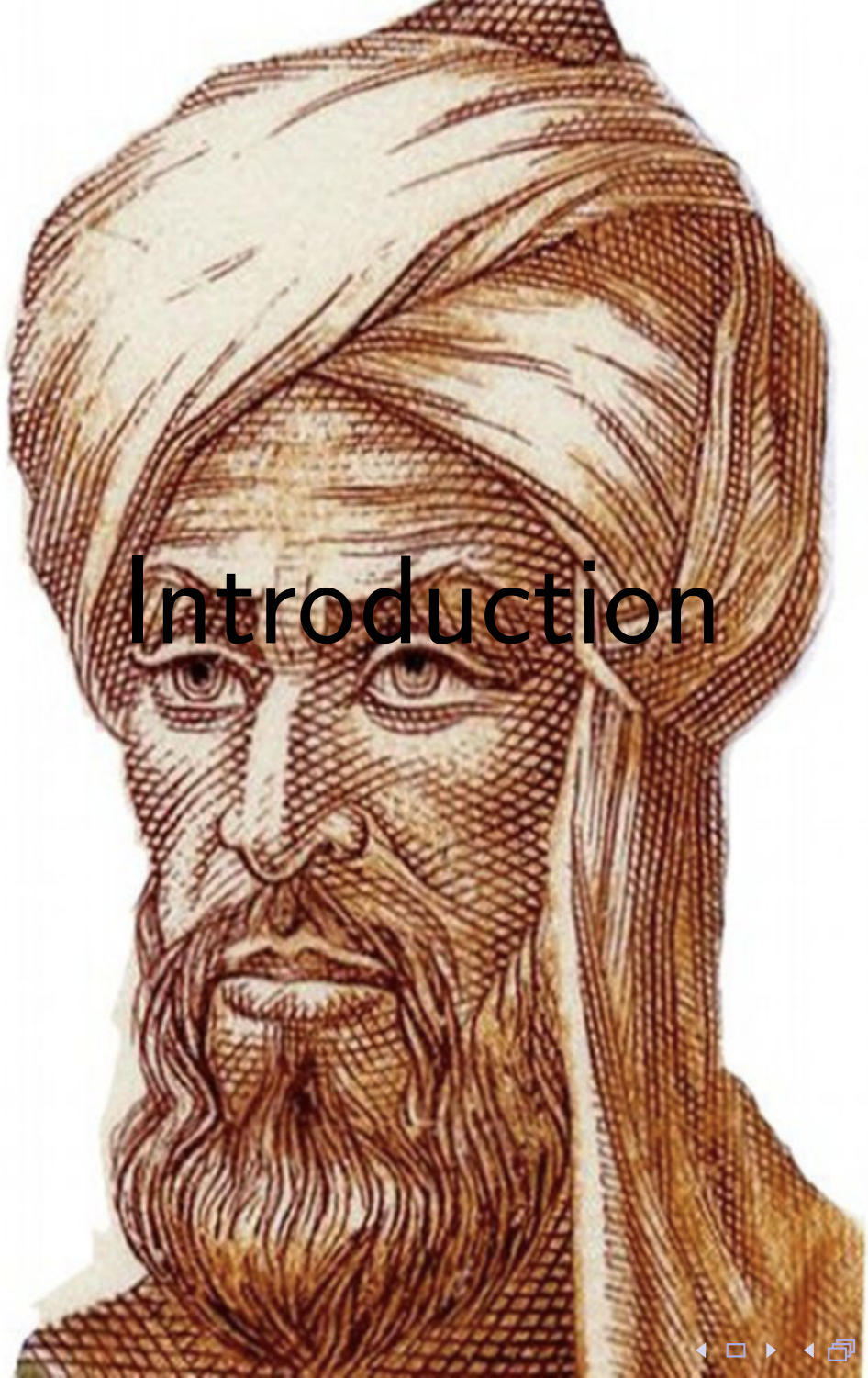


Introduction

Curs d'algorísmia:

Què hauríeu de conèixer? (nivell EDA) Eines bàsiques matemàtiques

Que heu vist abans:

- Notació asimptòtica, recurrències dividir i vèncer: Teorema mestre.
- Reduccions entre problemes, P i NP
- Fonaments de probabilitat
- Fonaments i nomenclatura de la teoria de grafs
- Algorismes d'ordenació: Mergesort, Quicksort, etc..
- Algorismes per a explorar grafs i digrafs: BFS, DFS.
- Estructures de dades bàsiques: Taules, llistes d'adjacència, piles, cues, monticles, hashing,..
- Backtraking

Metodologia per a dissenyar algorismes

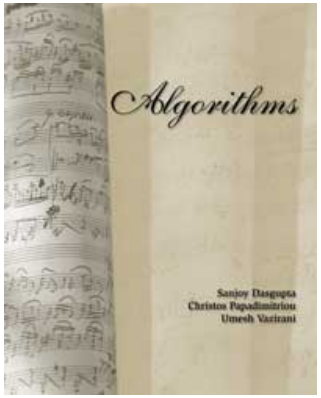
Que veurem?

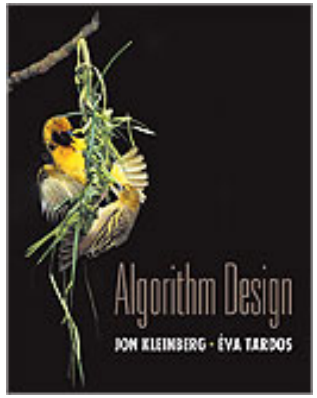
- Dividir i vèncer: Selecció lineal
- Ordenació lineal
- Voraços i compressió de dades
- Programació dinàmica
- Distancies en grafs
- Algorismes per a fluxos en xarxes: Aplicacions
- Programació Lineal
- Algorismes d'aproximació

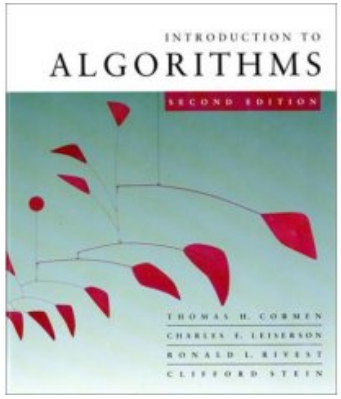
Resoldre models de problemes reals

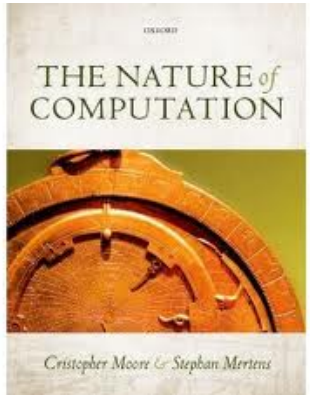
Bibliografia:

Referències principals:









"The algorithmic lenses: C. Papadimitriou"

Theoretical computer Science views computation as a ubiquitous phenomenon, not one that it is limited to computers.

In 1936 Alan Turing demonstrated the universality of computational principles with his mathematical model of the Turing machine.

Today's algorithms represent a new way to study problems and events in different individual and collective developments in humanity.

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.

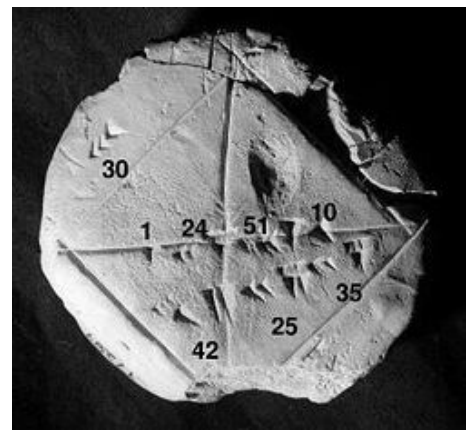
Algorithms.

Great algorithms are the poetry of computation. As verse, they can be terse, elusive and mysterious. But once unlocked, they cast a brilliant new light on some aspects of computing. *Francis Sullivan*

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

Sqrt (n)
$x_0 = 1$ $y_0 = n$
for $i = 1$ to 6 **do**
  $y_i = (x_{i-1} + y_{i-1})/2$
  $x_i = n/y_i$
end for

Babilònia (XVI BC)



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

- Its correctness, if it always does what it should
- Its efficiency:
  - computing time,
  - memory use
  - communication cost

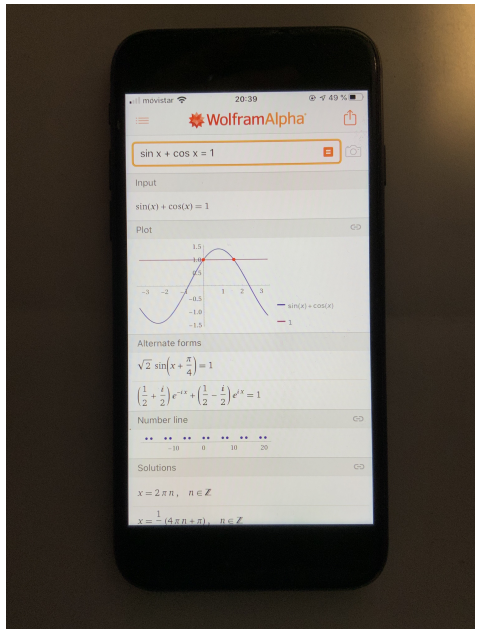
    ⋮

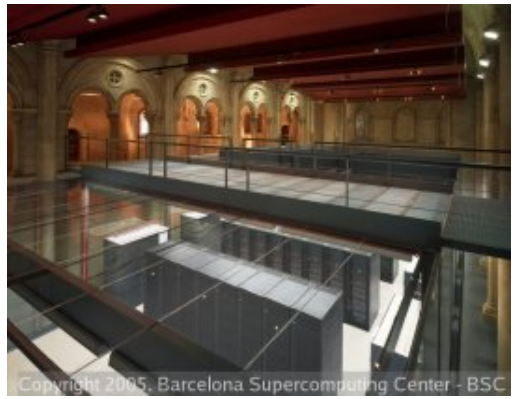
For an algorithm its time complexity is given as the computing time $T(n)$, as function of the input size.

For most of this course we will use a worst case analysis for $T(n)$: Given a problem, for which you designed an algorithm, you assume that your meanest adversary gives you the worst possible input.

Important

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





Typical computation times

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

if $n = 10$, $n^2 = 100$ i 2^n: 1024;

if $n = 100$, $n^2 = 10000$ i
$2^n = 1267650600228224401469670320537 6$;

if $n = 10^3$ $n^2 = 10^6$ 2^n is a number with 302 digits.

As a comparison, 10^{64} is estimated to be the number of atoms in hearth ($< 2^{213}$).

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

Typical computation times

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

if $n = 10$, $n^2 = 100$ i 2^n: 1024;

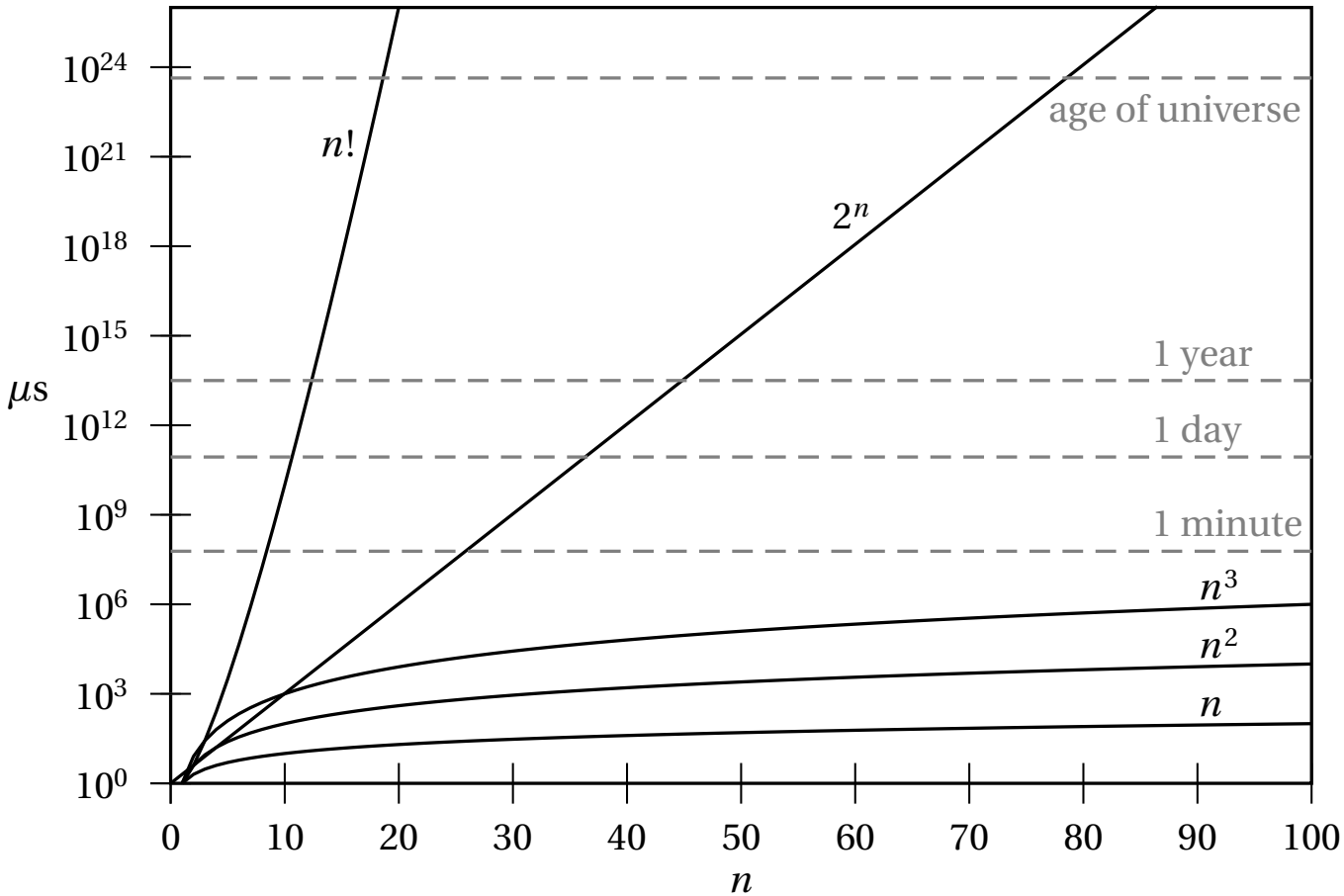
if $n = 100$, $n^2 = 10000$ i
$2^n = 1267650600228224401469670320537 6$;

if $n = 10^3$ $n^2 = 10^6$ 2^n is a number with 302 digits.

As a comparison, 10^{64} is estimated to be the number of atoms in hearth ($< 2^{213}$).

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

Computation time assuming that an input with size $n = 1$ can be solved in 1 μsecond:



From: Moore-Mertens, The Nature of Computation

Computation times as a function of input size n

	n	$n \lg n$	n^2	1.5^n	2^n
10	< 1s	< 1s	< 1s	< 1s	< 1s
50	< 1s	< 1s	< 1s	11m	36y
100	< 1s	< 1s	< 1s	12000y	10^{17}y
1000	< 1s	< 1s	< 1s	$> 10^{25}y$	$> 10^{25}y$
10^4	< 1s	< 1s	< 1s	$> 10^{25}y$	$> 10^{25}y$
10^5	< 1s	< 1s	< 1s	$> 10^{25}y$	$> 10^{25}y$
10^6	< 1s	20s	12d	$> 10^{25}y$	$> 10^{25}y$

From: Moore-Mertens, The Nature of Computation

Question: You have an algorithm that solves a problem of size $n = 10^6$ in $T(n) = 2^n$. If you use a supercomputer with 165,888 processors, how long would it takes to carry out your computation? (Assume that each processor can solve the problem with an input with size $n = 1$, in 1 μsecond)

Efficient algorithms and practical algorithms

Notice $n^{10^{10}}$ is a polynomial, but their 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!!!!

When analyzing an algorithm, we say it is feasible if it is poly time, otherwise it is say to be unfeasible.

In practice, even for feasible algorithms with time complexity of for example n^4, it could be slow for "real" values of $n \geq 100$.

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	$f = O(\log^c n)$ for cte. c
polynomial	$f = O(n^c)$ for cte. c or $n^{O(1)}$
subexponential	$f = o(2^{n^\epsilon})\ \forall 1 > \epsilon > 0$
exponential	$f = 2^{\mathrm{poly}(n)}$
double exponential	$f = 2^{\exp(n)}$

Quick review of basic concepts: Graphs

See for ex. Chapter 3 of Dasgupta, Papadimitriou, Vazirani.

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

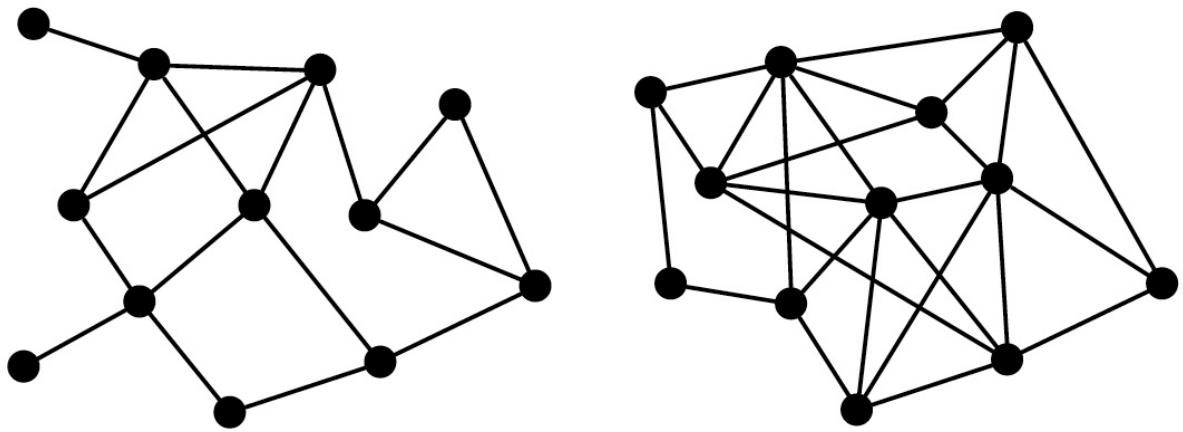
- Graphs: *undirected graphs (graphs)* and *directed graphs (digraphs)*
- The degree of v, $d(v)$ is the number of edges which are incident to v.
- A clique on n vertices K_n is a complete graph (with $m = n(n-1)/2$).
- A undirected G is said to be connected if there is an path between any two vertices.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

Directed graphs

- Edges are directed.
- The connectivity concept in digraphs is the strongly connected graph: 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 $|V| = n$ vertices is said to be dense if $m = \Theta(n^2)$; If $m = o(n^2)$ then G is said to be sparse.



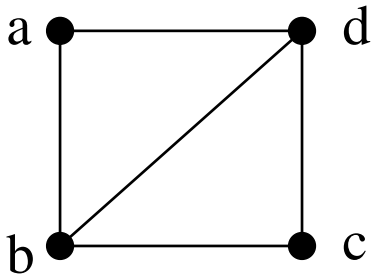
Data structure for store graphs that you should know.

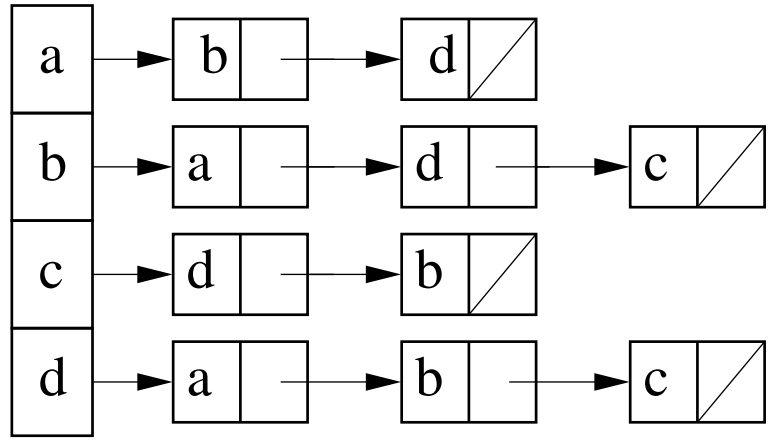
Let G be a graph with $V = \{1, 2, \ldots, n\}$. The two ways of representing G that you should know are:

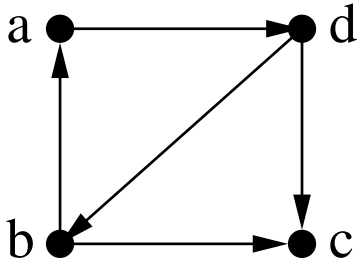
Adjacency list

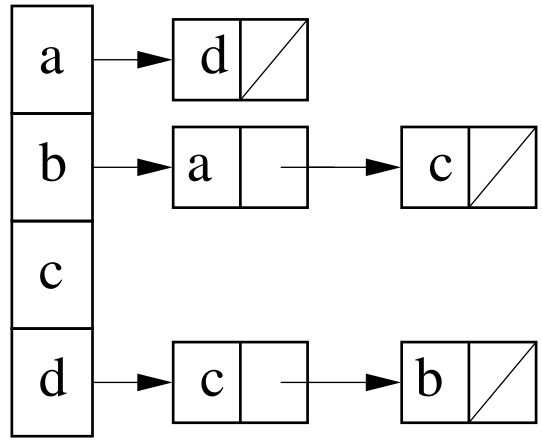
Adjacency matrix

Adjacency list





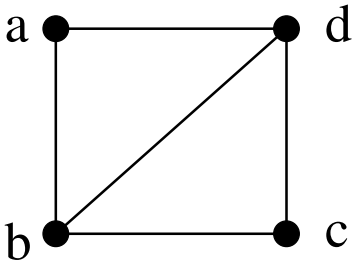




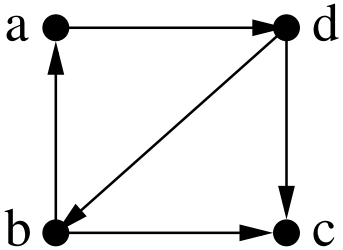
Adjacency matrix

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

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



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



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

Adjacency matrix

- If G is undirected, its adjacency matrix $A(G)$ is symmetric.
- If A is the adjacency matrix of G, then A^2 the for $i, j \in V$ $a_{i,j}$ gives if there is a path between i and j in G, with length 2 . For any $k > 0$, $a_{i,j}$ in 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 $a_{i,j}$.
- The use of the adjacency matrix allows the use of the powerful tools from the matrix algebra.

Comparison between the use of the matrix representation and the list representation of G

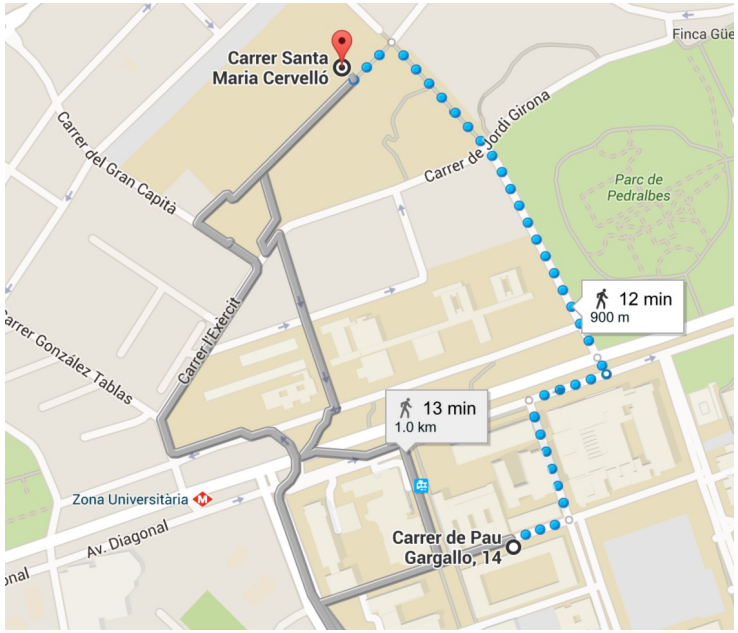
- The use of adjacency list uses a single register per vertex and a single register per edge. As each register needs 2×64 bits, then the space to represent a graph is $\Theta(n + m)$.
- The use of the adjacency matrix needs n^2 bits ($\{0, 1\}$), so for unweighted graph G, we need $\Theta(n^2)$ bits. 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 representations

- Adding a new edge to G: In both data structures we need $\Theta(1)$.
- Query if for u and v in $V(G)$ there is an edge $(u, v) \in E(V)$: 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 In both data structures we need to make a 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 and all their neighbors at distance=1.
2. Then the non-visited neighbors (at distance 2 from v).
3. Repeat until all vertices visited.



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

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:

- DFS and - BFS: $O(|V|^2)$

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

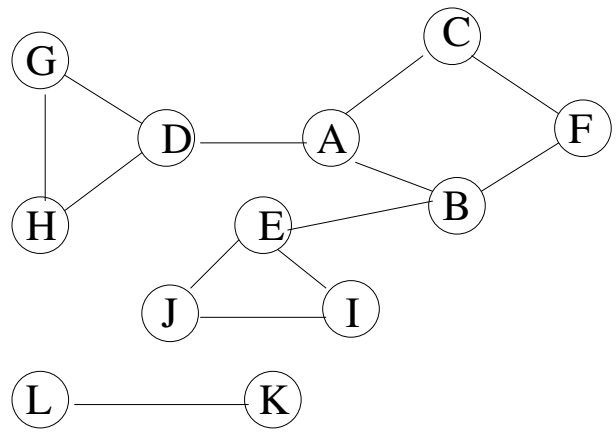
Connected components in undirected graphs.

A connected component is a maximal connected subgraph of G.

A connected graph has a unique connected component.

Connected Components
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 $O(|V| + |E|)$.

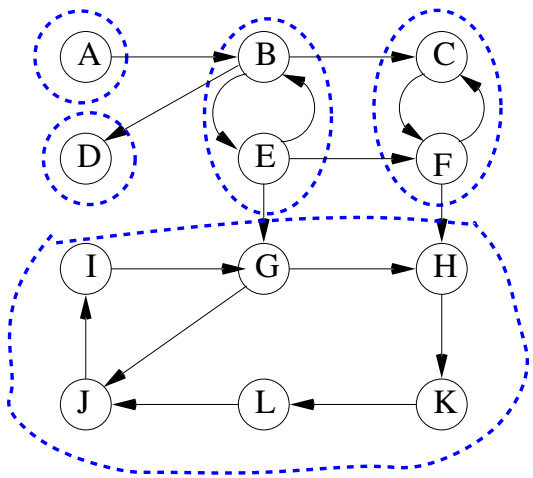
Strongly connected components in a digraph

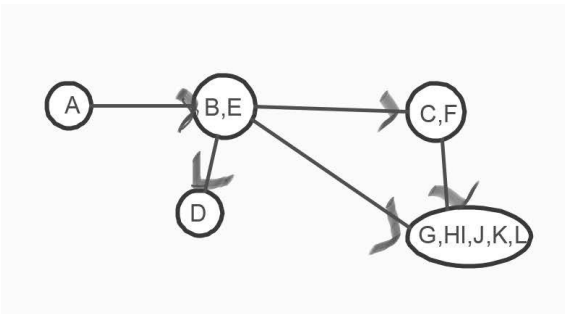
Kosharaju-Sharir's algorithm: Uses BFS (twice). Complexity $T(n) = O(|V| + |E|)$

Tarjan's algorithm: Based in using DFS. Complexity $T(n) = O(|V| + |E|)$

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

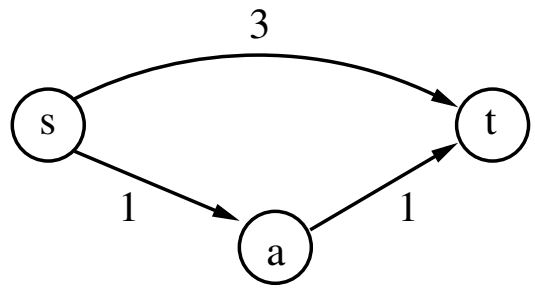
A nice property: Every digraph has a *directed acyclic graph (dag)* on its strongly connected components.





Caveat: BFS, DFS distances and weighted graphs

- BFS and DFS are algorithms to traverse a graph, if it is a weighted graph, it does not matter.
- However, using BFS or DFS to solve **optimization problems** (for example min or max distances) and doing it in the most efficient way, you need to use clever variations, to use the basic BFS (DFS) traverse.
- For instance applying BFS to the graph below, starting at vertex s, gives 3 as the min. distance $s \to t$:



- Implementing directly DFS or BFS on weighted graphs, **it does not yield necessarily a correct distance computation**.

The classes P and NP

- Recall 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 in decisional form.
- 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 are 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 in 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, as the knapsack.
- 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: Bounded Reductions

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

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 a reduction $A \leq B$ is a polynomial time computable function f that for any input x for A, in polynomial time transform it into a specific input $f(x)$ for problem B such that x has a valid solution for A iff $f(x)$ has a valid solution for B.

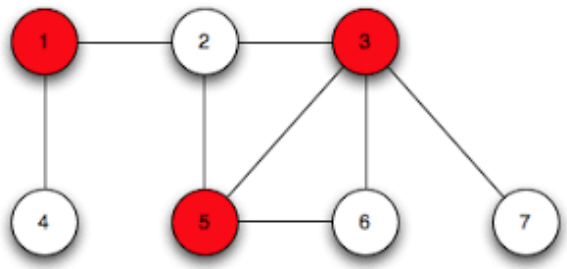
Therefore if we have that $A \leq B$, as there is an algorithm $\mathcal{A}_B$ to solve problem B, then we have an algorithm $\mathcal{A}_A$ for any input x of A: Compute $\mathcal{A}_B(f(x))$.

If B is in P, i.e. for every input y of B, $\mathcal{A}_B(y)$ yields an answer in polynomial time, then $\mathcal{A}_A(x) = \mathcal{A}_B(f(x))$ also is a polynomial time algorithm for A, so $A \in P$.

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.

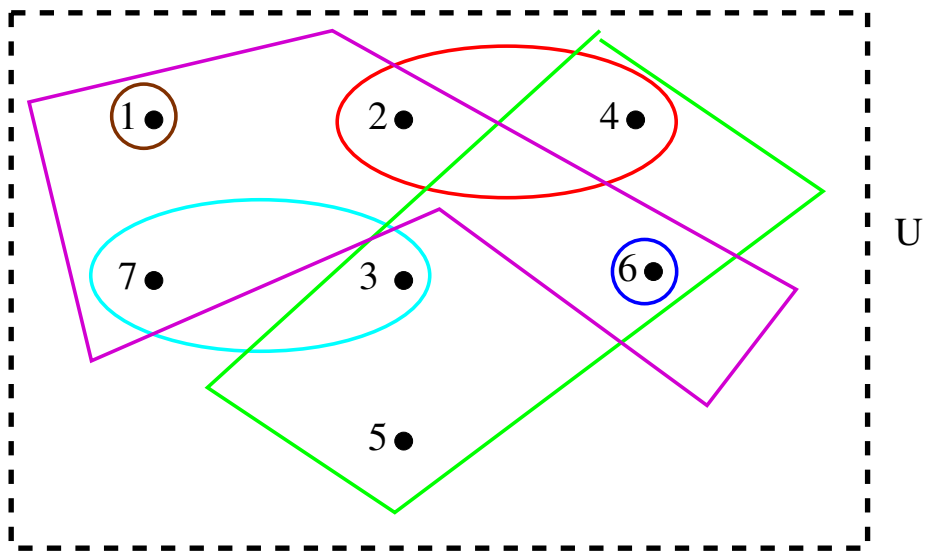
The Set Cover problem

Set Cover: Given a set U of m elements, a collection $S = \{S_1, \ldots, S_n\}$ where each $S_i \subseteq U$, find the minimal number of subsets whose union is equal to U.

There also is a weighted version, but the simpler version already is NP-hard.

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

Solution: $\{S_c, S_f\}$



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.

An example of Set Cover as a model: A software company has a list of 5000 computer viruses (U), consider the 9000 sets formed by substrings of 20 or more consecutive bytes from viruses. We want to check for the existence of those bytes of viruses in our developed code. If we can find a small set cover, say 180 substrings, then it suffices to search for these 180 substrings to verify the existence of known computer viruses.

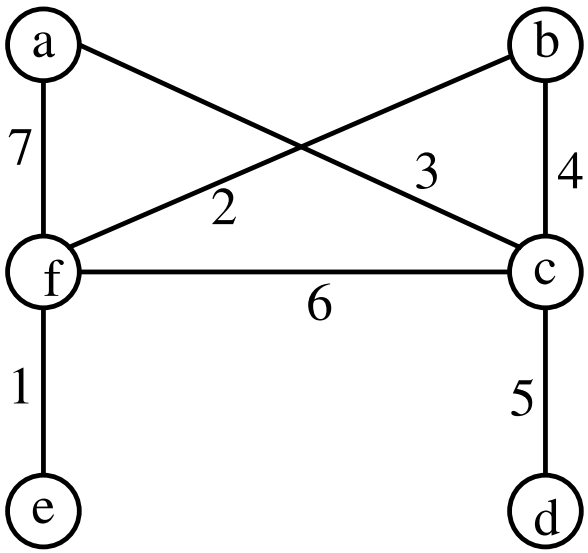
Vertex Cover $\leq$ Set Cover

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. 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 $O(m + nm)$

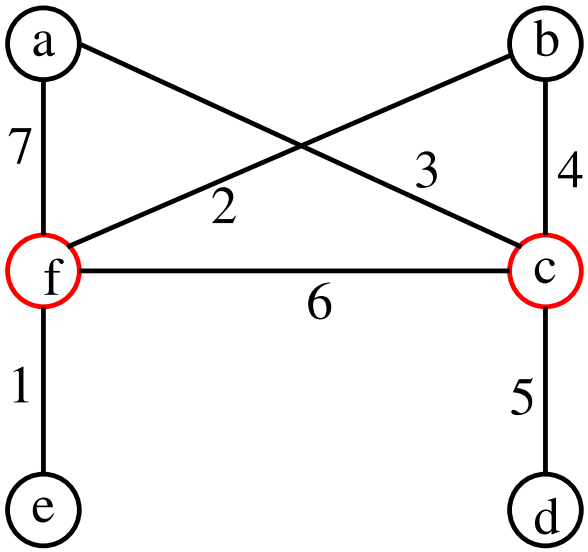
Example for the reduction



$\overset{f}{\Rightarrow}$

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

Example for the reduction



$\overset{f}{\Rightarrow}$

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

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.

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 $a, b, c \in \mathbb{R}^+$,

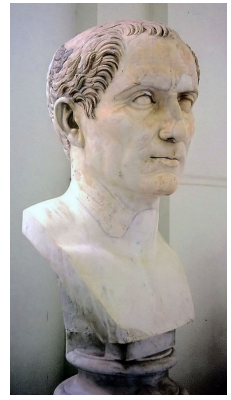
- $\log_b a = c \Leftrightarrow a = b^c \Rightarrow \log_b 1 = 0$
- $\log_b ac = \log_b a + \log_b c$, $\log_b a/c = \log_b a - \log_b c$.
- $\log_b a^c = c \log_b a \Rightarrow c^{\log_b a} = a^{\log_b c} \Rightarrow 2^{\log_2 n} = n$.
- $\log_b a = \log_c a / \log_c b \Rightarrow \log_b a = \Theta(\log_c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

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

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

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:

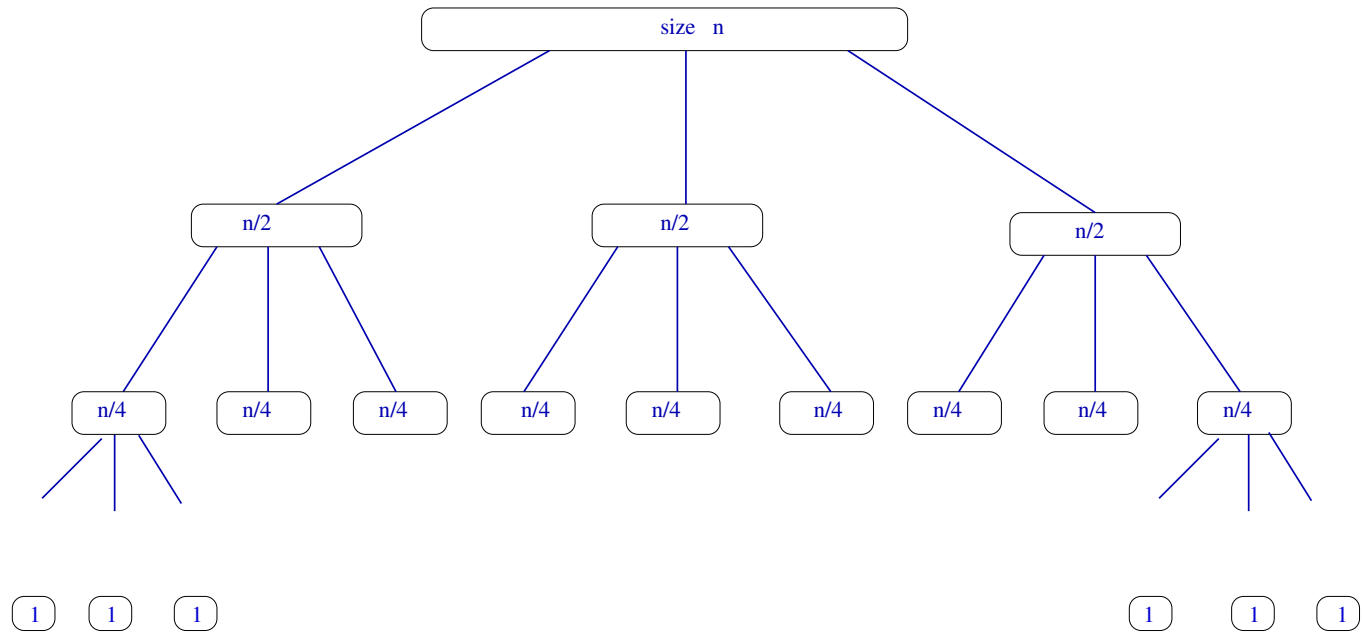
- Binary search
- Merge-sort
- Quicksort
- Strassen matrix multiplication

J. von Neumann
(1903-57)
Merge sort

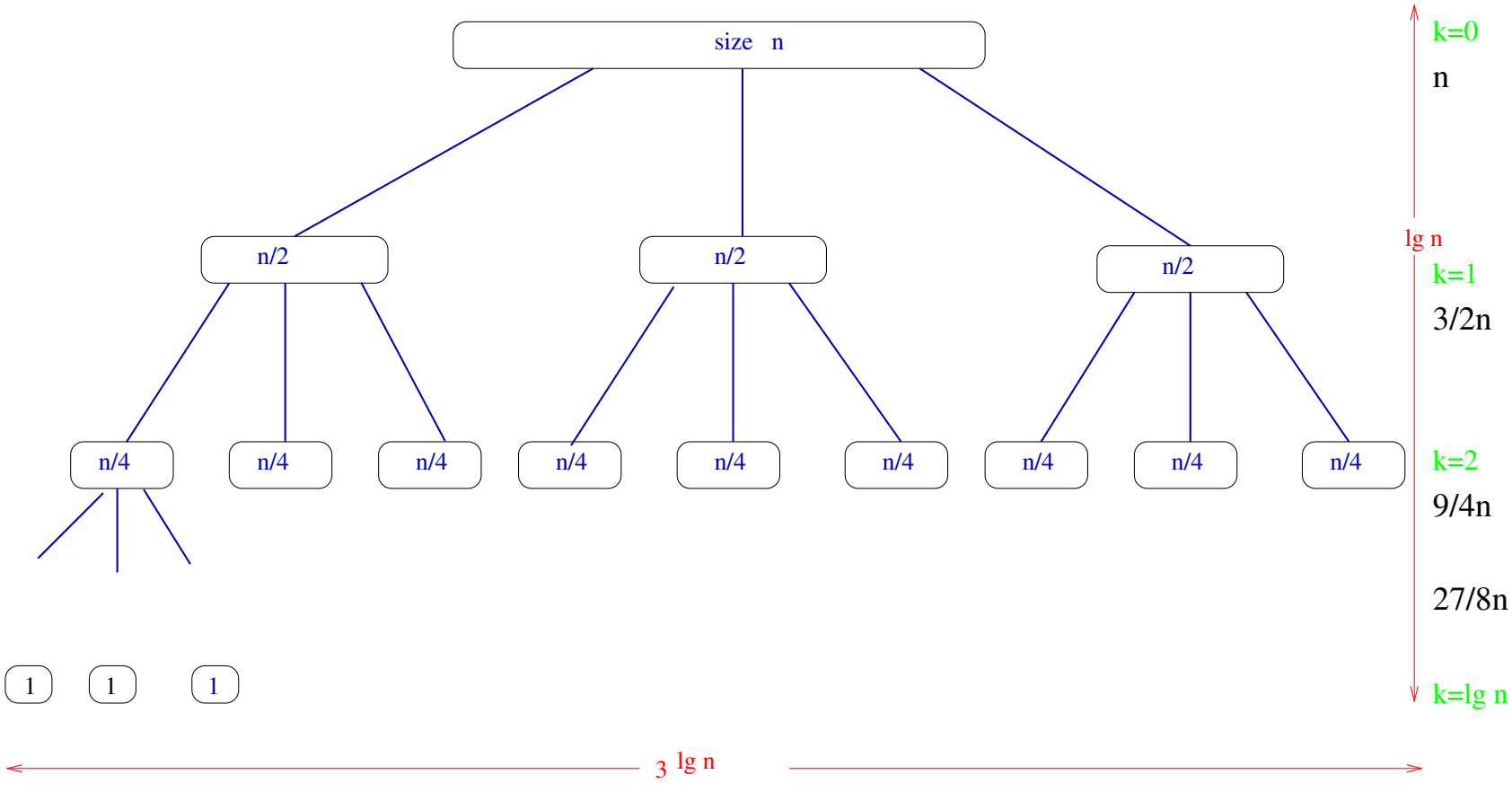
Recurrences Divide and Conquer

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

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 $O(n)$



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



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 the cost at depth k is:

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

with max. depth $k = \lg n$.

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

Therefore $T(n) = \sum_{k=0}^{\lg n} O(n)(\frac{3}{2})^k$.

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

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

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

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

The 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. A different one can be found in CLRS's book Theorem 4.1

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:*

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

The basic M.T. leave many cases outside. For stronger MT:

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`