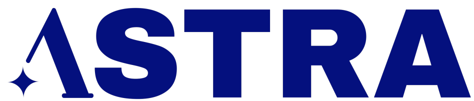
Brought to you by:



FUNDAMENTALS OF CS THEORY MIDTERM

1ST YEAR BEMACS

Written by
Giulia Pinciroli

2022-2023 Edition

Find more at:
astrabocconi.it

This handout has no intention of substituting University material for what concerns exams preparation, as this is only additional material that does not grant in any way a preparation as exhaustive as the ones proposed by the University.

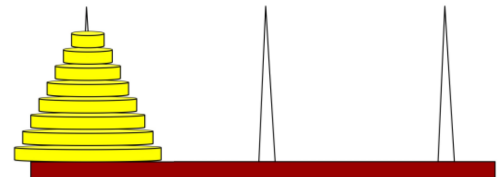
Questa dispensa non ha come scopo quello di sostituire il materiale di preparazione per gli esami fornito dall'Università, in quanto è pensato come materiale aggiuntivo che non garantisce una preparazione esaustiva tanto quanto il materiale consigliato dall'Università.

Algorithm → a procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation.
Computational Complexity of a problem → the amount of resources that you need to invest to find the solution.
In the case of computers: number of elementary operations that you have to perform and the amount of memory you use while using the best possible algorithm.

The notion of computational complexity is related to the **worst case scenario**.

TOWER OF HANOI

- transfer the disks from the first needle to the second needle, using the third as necessary
- you can only move one disk at a time, and could never put a larger disk on top of a smaller one.



Recursive solution

Definition: $S(n)$ = minimum number of moves in order to solve the problem. This is what we want to find.

Upper bound:

- we move $(n - 1)$ disks → $S(n - 1)$ moves
- move largest disk → 1 move
- move back the $(n - 1)$ disks → $S(n - 1)$ moves

Overall: $S(n - 1) + 1 + S(n - 1) \Rightarrow S(n) \leq 2S(n - 1) + 1$

Lower bound:

- to move the last disk, the others must be in order on one needle → $S(n - 1)$ moves must have been done *at least*
- moving the last disk → 1 move
- move back the $(n - 1)$ disks on top of the larger one → $S(n - 1)$

Overall: $S(n - 1) + 1 + S(n - 1) \geq 2S(n - 1) + 1$

Since the upper and the lower bound are equal: $S(n) = 2S(n - 1) + 1$

Solve the recurrence relation to find a closed form:

1. we guess a solution from small n
2. we show it is true for some $n = n_0$
3. we show that it is correct for any n assuming it has been shown to be true for values between n_0 and $n - 1$.

n	$S(n) = 2S(n - 1) + 1$
0	0
1	1
2	3
3	7
4	15
5	31
6	63
7	…

Guess:

$S(n) = 2^n - 1$

It works for $n \leq 6$.

Induction: substitute in the recurrence relation the Ansatz.

- $S(n) = 2S(n-1) + 1$
- $S(n) = 2^n - 1$

$S(n) = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 =$
$= 2^n - 1$

Other method: add 1 to LHS and RHS

- $S(0) + 1 = 1$
- $S(n) + 1 = 2S(n-1) + 2$

Def: $U(n) = S(n) + 1$

$U(n) = 2U(n-1) \Rightarrow U(n) = 2^n$

1 move in 1 µsec: $2^{64} - 1 \simeq 500\,000$ years.

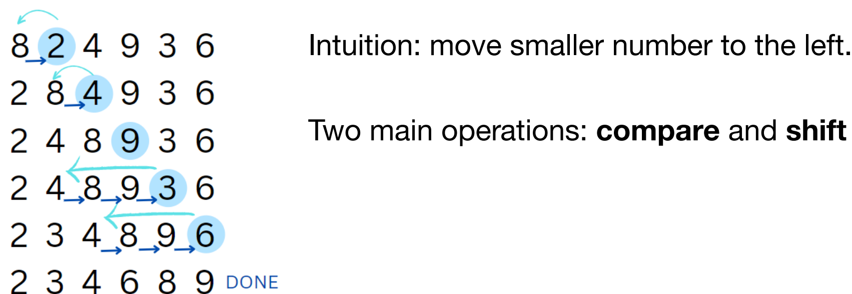
INTRODUCTION TO ALGORITHMS

Algorithm for **sorting**:

- numbers according to inequalities
- symbols if there is an order relation between them

Problem of sorting:

- **input**: sequence of numbers (string of numbers) $< a_1, a_2, a_3, \dots, a_n >$
- **output**: permutation $< a'_1, a'_2, \dots, a'_n >$ such that $a'_1 \le a'_2 \le \dots \le a'_n$ → numbers in increasing order.



Pseudocode notation: $x \leftarrow y$ = assign the value y to the variable x (usually $x = y$)
Def. of pseudocode: code that uses English but in a way that is sufficiently precise to be uniquely defined.

Insertion sort pseudocode

INSERTION-SORT (A, n) ⊳ A[1 .. n] ⟶ A = array of numbers of length n

for j 2 to n ⟶ start from 2 because the 1st elements alone would be already sorted

do key A[j] ⟶ read the jth number and put it in the auxiliary variable "key"

i⟵j-1 ⟶ starting position for scanning the list backwards

while i > 0 and A[i] > key ⟶ continue moving backwards until you find the right place for the insertion, or the list ends

do A[i+1] A[i] ⟶ while scanning backwards, shift the elements to the right.

i⟵ i-1 ⟶ move backwards

A[i+1] = key ⟶ put the selected element (key ⟵A[j]) in the correct position

Running time

We always want to choose the fastest algorithm (the one that has the least computational cost) → we need to know the running time.

- The running time depends on the input: an already sorted sequence is easier to sort
- parametrize the running time by the size of the input (the length of the list we want to sort), since short sequences are easier to sort than the long ones
- difficult to estimate the precise running time → we seek upper bounds on the running time (e.g. we want to be sure that the running time is not larger than n^2)

3 kind of analysis

1. **worst case** (usually) → $T(n)$ = maximum time of algorithm on any input of size n. It is a guarantee that you cannot do worse than predicted by the worst case analysis.
2. **average case** (sometimes) → $T(n)$ = expected time of algorithm over an input of size n. It needs an assumption of the statistical distribution of the input → know on average how long it takes to sort your string.
3. **best-case** (bogus) → cheat with a slow algorithm that works fast on some input → not very informative. E.g.: if a list is already sorted, the running time will be something of the order of n operations.

Machine-Independent time

What is insertion sort's worst-case time?

It depends on the speed of your computer → running time is machine dependent:

- relative speed (on the same machine)
- absolute speed (on different machines)

Big idea: ignore machine-dependent constants → consider a conventional unit cost for any single elementary operation. E.g.: number of operations, since it is the same for everyone.

Look at the growth of $T(n)$ (computational cost function) as $n \to \infty$ → how the number of operations grows as the size of the problem increases.

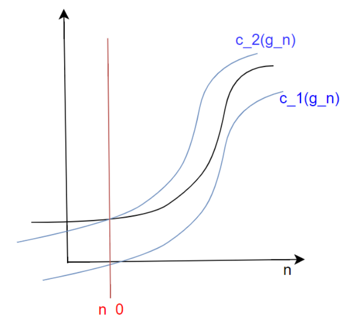
Asymptotic analysis of algorithm

The computational cost is a function of the size, not a number.

Θ-notation

Math: $f(n) = \Theta(g(n))$: there exists positive constants c_1, c_2 and n_0 such that

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \qquad \text{for all } n \ge n_0$$



Explanation: a mathematical function $f(n)$ is a Θ of another function $g(n)$ if $f(n)$ is in between $g(n)$ multiplied by a certain constant c_1 and $g(n)$ multiplied by another constant c_2, with $c_1 < c_2$, for all n that are sufficiently large.

You are interested in what happens for large problems.

Practitioners:

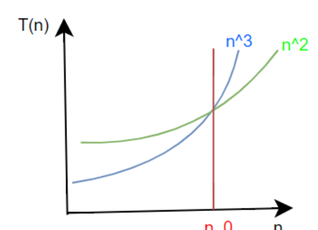
Drop the low-order terms and ignore leading constants.

Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

Asymptotic performance

When n gets large enough , a $\Theta(n^2)$ algorithm always beats a $\Theta(n^3)$ algorithm.

- However, we should not ignore asymptotically slower algorithms.
- Real-world design situations often call for a careful balancing of engineering objectives



- Asymptotic analysis is a useful tool to help to structure our thinking

Insertion sort analysis: computational cost

Worst possible case: : Input sorted in the opposite direction → at each iteration you have to shift.

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2) = c\sum_{j=2}^{n} j \quad \alpha \quad \sum_{j=1}^{n} j \text{ } (\alpha \rightarrow \text{proportional to})$$

$T(n) = \frac{n(n+1)}{2}$ $\quad n \gg 1 \quad T(n) = \Theta(n^2)$ → computational cost of the insertion sort algorithm

Average case (with respect to the fact that the list of numbers is created using an uniform probability distribution → we need to specify the probability distribution used): numbers in uniform random position → half of the numbers are already sorted, the other half are not. All permutations are equally likely.

$$T(n) = \sum_{j=2}^{n} \Theta(\frac{j}{2}) = \Theta(n^2)$$

The insertion sort in the average case is as bad as in the worst case.

Is insertion sort a fast sorting algorithm?

- moderately so, for small n
- not at all, for large n

Merge Sort Algorithm (recursive)

Algorithm that uses recursion → function that takes as a function another function (itself) → nested function.

MERGE SORT A[1 … n] → A = array of numbers

1. if $n = 1$ → done
2. recursively sort A[1 … ⌈n/2⌉] and A[⌈n/2⌉+1 … n]
    - "recursively sort" = use the same code applied to the subproblems → program that calls itself inside its instructions → recursion
    - "⌈…⌉" = the integer part of
    - Split an array of length n in 2 parts of length $\frac{n}{2}$ → if n is even, you divide by 2 and you obtain an integer; if n is odd, you divide by 2 and you round it up.
3. "merge" MERGE-SORT the two sorted lists

Divide and conquer

Very general strategy in computer science.

Merge-sort:

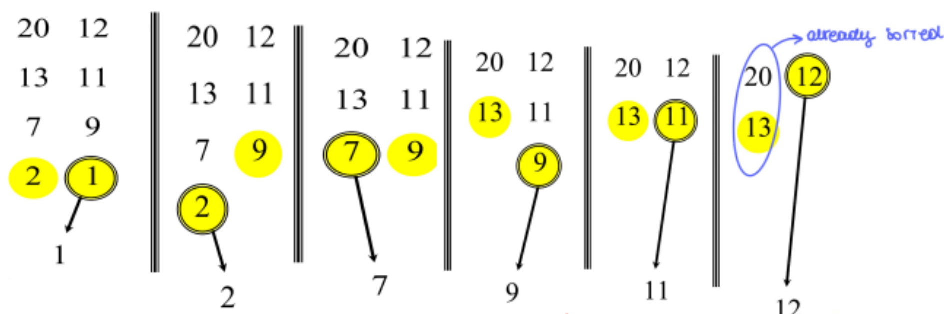
1. **divide**: divide the given n-element array A into 2 subarrays of $\frac{n}{2}$ elements each
2. **conquer** (reapply the same procedure): recursively sort the two subarrays
3. **combine**: merge 2 sorted subarrays into 1 sorted array

Recursive Pseudocode

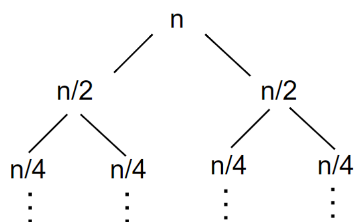
```
MERGE-SORT A[1 .. n]
{if n>1  ——→ re-execute the same code on different inputs of smaller size
      MERGE-SORT(A[1 … [n/2]])  ——→  call the same code on the right subarray
      MERGE-SORT(A[[n/2]+1 … n])
      MERGE(A[1…n])  ——→  merge the results of the two previous instructions
}
```

Merging two sorted arrays

To merge two sorted lists, you just have to compare the numbers, look for the smaller one, extract it and put it aside. You are then left with two arrays of different length, where the smallest element of the first array is still the same as before, but the first element of the second array has changed, because the smallest number has been taken away . You repeat the steps until you are left with already sorted numbers or just one number .



Time of $= \Theta(n)$ ot merge a total of n elements (linear time).
The number of comparisons is of the same order of the length of the two arrays.



You stop when you reach one: $n/2^k = 1 \rightarrow$ solve this equation to know how many steps you have to do.
$n = 2k$
$\log_2 n = k$ $\quad k = \log n \rightarrow$ number of operation will be of order of $\log n$

The computational cost is $T(n) = \Theta(n \ln n)$
Other example: $a^n = a \cdot a \cdot a \cdot a \cdot a ... a)$ n times $= \Theta(n)$
$a^n = a^{n/2} \cdot a^{n/2} = a^{n/4} \cdot a^{n/4} \cdot a^{n/4} \cdot a^{n/4} = \Theta(\ln n) \rightarrow$ all the factors are the same, so you can compute them only once.

Analyzing merge sort

Profiling = looking at where your code spends the most time (where there are the heaviest operations)

$T(n)$ — **MERGE-SORT** $A[1..n]$
$\Theta(1)$ — 1. If $n = 1$, done.
$2T(n/2)$ — 2. Recursively sort $A[1..\lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1..n]$.
Abuse
$\Theta(n)$ — 3. ***"Merge"*** the 2 sorted lists

Sloppiness: Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically.

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

We do not know for sure the computational cost, but we know that it will satisfy this equation.

We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence .

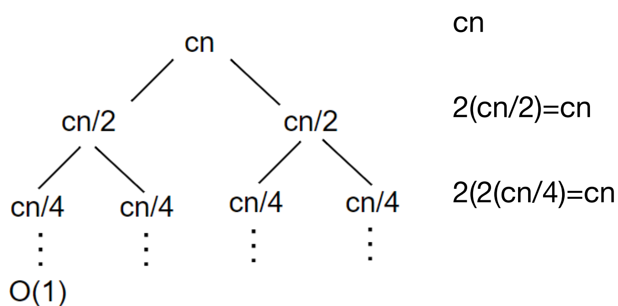
$T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \ln n) \rightarrow$ in the asymptotic form the base of the logarithm is not important, because I can always change it by multiplying by a constant.

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

- Expand the equation $\rightarrow T(n/2) = 2T(n/4) + c(n/2)$
- $T(n) = 2(2T(n/4) + c(n/2)) + cn = 2(2T(n/8) + c(n/4) + c(n/2)) + c(n/4)$

Recursively apply the algorithm substituting for each T the right-hand side of the equation that T has to satisfy.

Graphic representation of the residual terms c·n



cn

2(cn/2)=cn

2(2(cn/4)=cn

At each iteration you accumulate this computational cost, and to evaluate the total computational cost, you have to count how many times you are going to execute this.

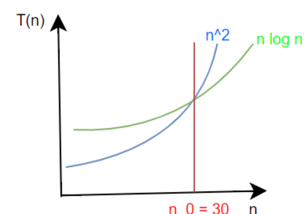
It is already decided $\rightarrow$ # of levels of the structure = $\log n$

Total cost = # levels · cn $\Rightarrow T(n) = \log n \cdot cn = \Theta(n \cdot \log n)$

Conclusions

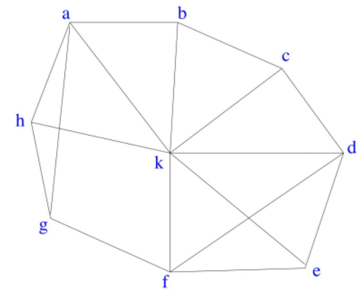
$\Theta(n \log n)$ grows more slowly than $\Theta(n^2)$

- Therefore merge-sort asymptotically beats insertion-sort in the worst case



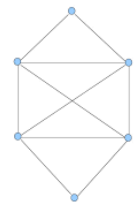
- In practice, merge-sort beats insertion sort for n>30 or so.

GRAPH THEORY



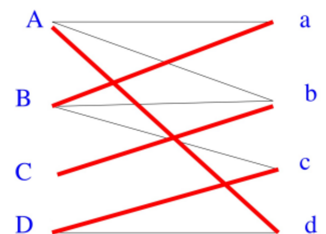
Simple Graph: $G(V, E)$ → collection of vertices and edges
V = {vertices}, E = {edges} → 2 sets
Vertices are indicated by symbols.
$V = \{a, b, c, d, e, f, g, h, k\}$
An edge, graphically, is something connected to vertices.
In terms of set theory an edge is identified by a pair of symbols.
$E = \{(a, b), (a, g), (a, h), (a, k), \ldots, (h, k)\}$
$|E| = 16$ → cardinality of the set → number of elements inside the set

Eulerian graphs



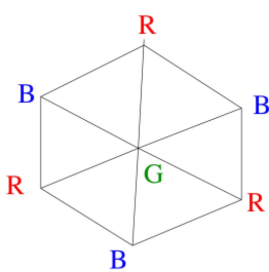
Graph that you can draw without taking your pen off the paper or going over the same line twice → you never have overlapping edges.

Bipartite graphs



A graph in which the set of vertices is given by the union of two sets X and $Y$and all the edges are of the form x and y , where $x \in X$ and $y \in Y$ → all the edges go from 1 set to the other set, but there are no edges that connect vertices inside the two subsets.
G is bipartite if $V = X \cup Y$ where X and Y are disjoint and every edge is of the form (x, y) where $x \in X$ and $y \in Y$.

Vertex coloring



You have a graph and a certain number of colors.
Colors $\{R, B, G\}$
Let C = {colors}
A vertex coloring of G is a map from the vertices to the color ($f: V \to C$)
We say that $v \in V$ gets colored with $f(v)$
The coloring is **proper** if and only if the endpoints of an edge always have different colors:

$$\text{coloring proper iff } (a, b) \in E \Rightarrow f(a) \neq f(b)$$

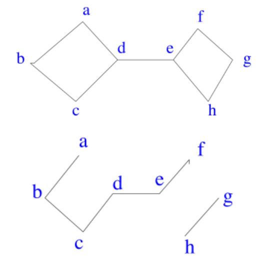
The **chromatic number** $\chi(G)$ is the minimum number of colors you need in order to color properly your graph.
Theorem: if you have a graph which can be drawn on a plane without overlapping edges (graph corresponding to a map) , the chromatic number for planar (no overlapping edges) graph $\chi(G_{planar}) = 4$

Subgraphs

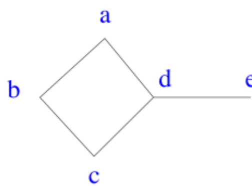
A graph obtained by taking subsets of the original graph.

$G' = (V', E')$ is a subgraph of $G(V, E)$ if $V' \subseteq V$ (subset) and $E' \subseteq E$.

G' is a **spanning subgraph** if $V' = V \rightarrow$ it contains all the vertices but a smaller number of edges.



If $V' \subseteq V$ then $G[V'] = (V', \{u, v\} \in E: u, v \in V'\})$ is the subgraph of G **induced** by $V' \rightarrow$ a subgraph can be induced by the choice of the vertices.

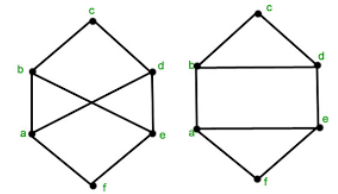


Induced subgraph: composed by the chosen vertices + all the edges that connect them.

Similarly, you can fix the edges and include all the vertices that appear as end points of those edges: if $E_1 \subseteq E$ then $G[E_1] = (V_1, E_1)$ where $V_1 = \{v \in V_1 : \exists\, e \in E_1 : v \in e\}$ is also induced (by E_1).

Isomorphism

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are **isomorphic** if there exists a bijection $f: V_1 \rightarrow V_2$ such that the two graph coincide: $(v, w) \in E_1 \leftrightarrow (f(v), f(w)) \in E_2$.



Necessary condition for the isomorphism : you need to have the same number of vertices, edges, and degree of the nodes .

Complete graphs

A graph in which all the modes are connected to all the other nodes

$K_n = ([n], \{(i, j): 1 \leq i < j \leq n\}$ is the complete graph on n vertices.

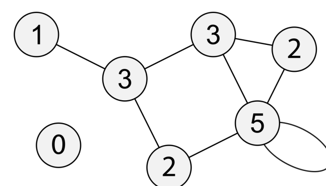
Complete bipartite graph: there are edges that connect all nodes of one side to all nodes on the other side

$K_{m,n} = ([m] \cup [n], \{(i, j): i \in [m], j \in [n]\})$ is the complete bipartite graph on the $m + n$ vertices.

Vertex Degree

Degree of a vertex $\rightarrow$ number of edges incident with the vertex

$d_G(v)$ = degree of vertex v in G = number of edges incident with v



$S(G) = \min_v d_G(v)$ = minimum degree of a graph $\rightarrow$ degree of vertex with minimum degree

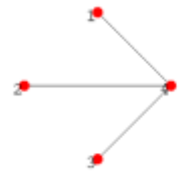
$\Delta(G) = \max_v d_G(v)$ = maximum degree among all nodes $\rightarrow$ degree of vertex with maximum degree .

MATRICES AND GRAPHS

Important because you can use linear algebra to make operations on a graph.

Incidence matrix (rectangular matrix)

M: $|V| \times |E|$ matrix



$M(v, e) =$ 1 $v \in e \rightarrow$ if vertex belongs to edge

0 $v \notin e \rightarrow$ if vertex does not belong to the edge

- n° of rows = n° of vertices
- n° of columns = n° of edges

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Properties:

- each column can only have two 1s → each edge can only have two endpoints
- each row has a number of 1s equal to the degree of the corresponding vertex

Adjacency matrix

A: $V \times V$ matrix → square matrix indexed by the vertices

$A(v, w) =$ 1 v, w adjacent

0 otherwise

"adjacent" → of vertices are directly connected by an edge → $(v, w) \in E$: the pair of vertices (v, w) belongs to the edge set.

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

On the diagonal there are always 0s because we do not allow self loops.

Properties:

- it is symmetric
- each row/column has a number of 1s equal to the degree of the corresponding vertex

Theorem 1

The sum of the degree of all the vertices for every graph is twice the number of edges

$$\sum_{v \in V} d_G(v) = 2|E|$$

Proof

Consider the incidence matrix M .

- If you count the 1s in the rows (each row corresponds to a vertex) , you find that the number of 1s corresponds to the sum of the degree of the vertices

  # 1s in matrix M is $\sum_{v \in V} d_G(v)$

- If you count the 1s in the columns (each columns corresponds to an edge) , you find that the number of 1s is twice the number of edges

  # 1s in matrix M is $2|E|$ → each column has 2 1s. # edges = # of columns

Corollary 1

In any graph, the number of vertices of odd degree is even.

If the degree of a vertex is 0 we define it as even.

Proof

Let Odd = {odd degree vertices}→ set of vertices that have odd degree

and $Even = V \setminus Odd$ → Total set of vertices - Set of odd degree vertices

$$\sum_v d_G(v) = 2|E|$$

$$\sum_{v \in Odd} d_G(v) + \sum_{v \in Even} d_G(v) = 2|E|$$ → the sum of even number is always even

$$\sum_{v \in Odd} d_G(v) = 2|E| - \sum_{v \in Even} d_G(v)$$ → the difference between two even numbers is even

If you sum odd numbers, the only way to obtain an even number is to have an even number of terms.

Path and walks over graphs

Walks

Sequence of vertices in G.

$W = (v_1, v_2, \dots, v_k)$ is a walk in G if $(v_i, v_{i+1}) \in E$ for $1 \le i \le k$ + empty walks (a vertex is only connected to itself).

Path

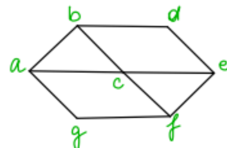
Walk in which the vertices are distinct (no vertex is visited twice → shortest walk).

Every path is a walk , but not every wave is a path .

$w_1 = a, b, c, e, d$ → path

$w_2 = a, b, a, c, e$ → walk

$w_3 = g, f, c, e, f$ → walk

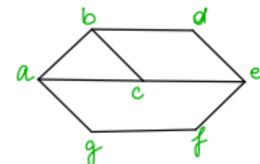


A walk is **closed** if the starting point and the end point are the same $(v_1 = v_k)$.

A **cycle** is a closed walk in which the vertices are distinct except for the first and the last one (v_1, v_k)

b, c, e, d, b → cycle → → only vertex visited twice is the starting one (= last one)

b, c, a, b, d, e, c, b → not a cycle → vertex is visited more than once but it is not a starting and needing point.



Connected components

Intuition : a connected component is a piece of graph such that you can always find a path connecting two points. We define a relation ~ on V (set of vertices).

a and b are related $(a \sim b)$ if and only if there is a walk from a to b (doesn't matter if a vertex is visited many times , the important thing is that the vertices are topologically connected).

Claim: $\sim$ is an equivalence relation → it satisfies the properties of :

- **reflexivity**: v connected to v ($v \sim v$) is a (trivial) walk from v to v → an empty walk is still a walk → a vertex is always connected to itself .
- **symmetry**: $u \sim v$ implies $v \sim u$ → you can reverse the walk , there is no direction on the graph (we are not considering oriented graphs).
  $(u = u_1, u_2, \dots, u_k = v)$ is a walk from u to v implies that $(u_k, u_{k-1}, \dots, u_1)$ is a walk from v to u.
- **transitivity**: $u \sim v$ and $v \sim w$ implies $u \sim w$
  $w_1 = (u = u_1, u_2, \dots, u_k = v)$ is a walk from u to v and
  $w_2 = (v_1 = v, v_2, v_3, \dots, v_l = w)$ is a walk from v to w implies that
  $(w_1, w_2) = (u_1, u_2, \dots, u_k, v_2, v_3, \dots, v_l)$ is a walk from u to v.
  You have to join two walks , without writing twice the connecting point.

You can decompose your vertices in equivalence classes under the relation: all the nodes that are related by the definition of connectivity are in the same class.
We can divide the set of vertices as the union of subsets which compose the classes.
Given the three properties of the equivalence relation, you can define the notion of equivalence classes.
The equivalence classes of $\sim$ are called **connected components**.

In general the set of vertices can be decomposed as the union of many connected components. ($v = c_1 \cup c_2 \cup \dots \cup c_r$ where $c_1, c_2, \dots, c_r$ are connected components).

We let $\omega(G)(= r)$ be the number of connected components of G.
If a graph is fully connected (we have just 1 connected component): $\omega(G) = 1$.
G is connected iff $\omega(G) = 1$ i.e. there is a walk between every pair of vertices in the graph.
Thus $c_1, c_2, \dots, c_r$ induce the decomposition of the graph in connected subgraphs $G[c_1], \dots, G[c_r]$ of G.
You can associate to each connected component a subgraph → if you take the union of them you can reconstruct G.

Computational important properties
Given a walk w on a graph, we call $l(w)$ the n° of edges in w (where l stands for "length").
<u>Lemma 1</u>
Suppose w is a walk from vertex a to vertex b and that w minimizes the length l over all possible walks from a to b. Then w is a path.
If you want to minimize the length, you should avoid going through the same vertex twice.
Proof
Suppose $w = (a = a_0, a_1, \dots, a_k = b)$ it is some path and at a certain point you repeat a vertex twice $a_i = a_j$ where $0 \le i < j \le k$. Then I can construct a new walk

$w' = (a_0, a_1, ..., a_i, a_{j+1}, ..., a_k)$ in which I delete everything in between the two identical vertices and join what remains.
w' is also a walk from a to b and $l(w') = l(w) - (j - i) < l(w)$ (with $(j - 1)$ being the number of elements previously deleted) → the initial walk could not have been a minimum length walk → contradiction
A minimum length walk is a path.
Corollary
If two vertices a and b are in relation, then there is a path from a to b (you can always create a path) so G is connected ↔ $\forall a, b \in V$ there is a path from a to b.

In the algorithm, if I am looking for a path, I should avoid loops (repetition of vertices).

BREADTH FIRST SEARCH - BFS
Algorithms used to discover connected components (who is connected with who) using only local information.
If you have a complicated graph you can only see the nodes that are connected to you → you have only local information.
We start with a vertex $v \in V$. We define $w \in V$ such that $d(v, w)$ is the length of the shortest path from v to w.

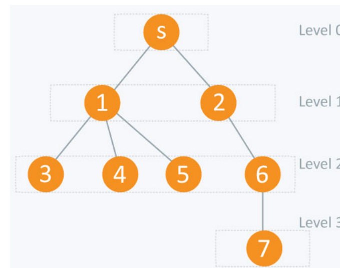
- $d(v, w)$ = distance: minimum length (always) among all possible lengths.

For $t = 0, 1, 2, ...$ let $A_t = \{w \in V: d(v, w) = t\}$

- t = stages of the algorithm.

We are constructing a sequence of layers of vertices that are at distance t from the set of vertices constructed at the previous stage .

$A_0 = \{v\}$
$A_1 = \{w \in V: d(v, w) = 1\}$
$A_2 = \{w \in V: d(v, w) = 2\}$
$A_3 = \{w \in V: d(v, w) = 3\}$



Each time I add nodes just by looking at the nearest neighborhood of tha last set I have built → local information.
$A_0 = \{v\}$ and $v \sim w \leftrightarrow d(v, w) < \infty$
Once we include a node to a subset, we put a flag (assign a binary value) to recognize if you have already seen it or not.
In BFS we construct $A_0, A_1, A_2, ...$ by

$$A_{t+1} = \{w \notin A_0 \cup A_1 \cup ... \cup A_t : \exists \text{ an edge } (u, w) \text{ such that } u \in A_t\}$$

- A_{t+1} = set of vertices in layer $t + 1$ → all the vertices that do not belong to the previous layer, such that there exists an edge (u, w) that have vertices at distance 1 from any vertex u belonging to the last subset constructed A_t.

Note: no edges (a, b) between A_k and A_l for $l - k \geq 2$, else $w \in A_{k+1} \neq A_l$

In this way we can find all the vertices in the same component C as the starting vertex v just using local information . By repeating for $v' \notin C$ we find another component etc..

Local information → something that does not divert with the size of the problem.

Characterization of bipartite graphs

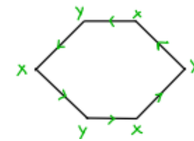
Theorem 1

G is bipartite if and only if G has no cycles of odd length.

- bipartite → can be decomposed in two subset of vertices that are not connected (there are no edges connecting vertices of the same subset)
- if and only if → you have to prove the theorem in both directions in order for it to be true.

Proof → : $G = (X \cup Y, E)$

If a bipartite graph is composed of the union of the subset of vertices X and Y such that they are connected, whenever you make a move over an edge you are going from a subset to the other.



You never have ledges connecting two vertices of the same subset

Suppose $C = (u_1, u_2,..., u_k, u_1)$ is a cycle. Suppose $u_1 \in X$. Then $u_2 \in Y$, $u_3 \in X$, ... , $u_k \in Y$ implies k is even (because in order to connect X to X you have to make 2 steps → the cycle must be a multiple of 2 steps).

This direction of the proof (if bipartite than k is even) is trivial.

Second part → if the graph has no cycles of odd length, then it is bipartite (more complicated)

Proof by contradiction

← Assume G is connected, else apply the following argument to each component

- only look at connected components of a graph → we can reapply the statement to each connected component.

Choose $v \in V$ and construct A_0, A_1, A_2,... by BFS.

- A_0, A_2, A_4,... have no direct connection because, by construction of the algorithm, we are dividing the graph in sets

$X = A_0 \cup A_2 \cup A_4 \cup$... and $Y = A_1 \cup A_3 \cup A_5 \cup$... → all the nodes are separated = they do not share an edge.

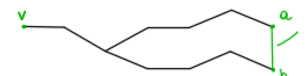
We need only to show that X raved Y contain no internal edges round then all edges must join X raved Y in order to show that it is bipartite.

Suppose that X contains an edge (a, b) where $a \in A_k$ and $b \in A_l$

There can be no edges between a and b if their distance is ≥ 2, otherwise, they would be merged in the same set.

1. if $k \neq l$ then $|k - l| \geq 2$ which contradicts (1)

- There can be no edges between a and b if their distance is ≥ 2, otherwise, they would be merged in the same set.



If the green edge existed, the length of the cycle would be odd, which is a contradiction

2. $k = l \rightarrow$ we want to show that X contains no edges inside a given layer of the algorithm.

There exists paths $(v = v_0, v_1, v_2, ..., v_k = a)$ and $(v = w_0, w_1, w_2, ..., w_k = b)$.

Let $j = \max\{t: v_t = w_t\} \rightarrow (v_j, v_{j+1}, ..., v_k, w_{k-1}, ..., w_j)$ is an odd-cycle with length $2(k - j) + 1$.

If the graph contains one edge inside the subset X, then there must exist a cycle of odd length , which is impossible because we are assuming that the graph has no odd cycle (starting assumption) → edge inside the subset cannot exist → the graph is **bipartite**.

In this proof we are using an algorithm: the properties of BFS.

Walks and powers of matrices

Theorem 2

$A^k(v, k)$ = number of walks of length k from v to w with k edges.

- $A^k(v, k)$ adjacency matrix $\rightarrow \neq [A(v, w)]^k$. $(A^k)(v, k)$ means A to the power of k and then take the elements v and w of the matrix.
- Suppose we have a graph and we want to go from v to w in k steps. How many walks are there? (we can go through the same vertex more than once.)

$A(v, w) =$ 1 if $(v, w) \in E \rightarrow$ two vertices are connected

0 else

The theorem says that in order to compute this combinatorial number we just have to have the adjacency matrix , take the k power and compute the matrix elements.

Computational complexity : complexity of multiplying matrices.

Proof by induction

By induction on k. Trivially true for $k = 1$ (initial condition)

- We have to show that our statement is true for the starting point. We have to show that the fact that it is true at the k step implies that it is true at the $k + 1$ step → we find all the possible cases → we prove the result true: if we have two vertices, the number of walks can either be 0 (not connected) or 1 (connected).

Assume true for some $k \geq 1$

- assume true for step k. You have to prove that the transition from k to $k + 1$ preserves the property.

1) Let $N_t(v, w)$ be the number of walks from v to w with t edges. (definition)

2) Let $N_t(v, w; u)$ be the number of walks from v to w with t edges whose penultimate vertex is u.

- Same as before but conditioned on the fact that on the penultimate step you have to go through a given vertex.

- We start from v and we want to reach w , but there are many possible walks. All the walks that go through u are a subset of all the path of length t between v and w. If I sum over all possible choices of u (penultimate vertex) I am going to recover the total number of walks.

$N_t(v, w) = \sum_u N_t(c, w; u)$ → sum of all possible choices of u

$N_{k+1}(v, w) = \sum_{u \in V} N_{k+1}(v, w; u)$ → we are assuming that $N_k = A^k$

- The number of walks of length $k + 1$ between v and w can be written as a sum over all possible choices of the penultimate step of the number of walks of length $k + 1$ between v and w that are conditioned to go through u as a penultimate step.

Property of the adjacency matrix:

$= \sum_{u \in V} N_k(u, v)A(u, w)$

- $A(u, w)$ → the matrix element works as a filter: if it is equal to 0 it means that there is no edge between the u we are looking at and w, so we are not going to count that as a walk. If there is an edge we have to count it, and it is going to add 1 to the total length.
- When you reach u you have done k steps . Then you have to do 1 more step (the last one).
  Total number of steps = total number of walks of length k summed over all the us that are connected to w → this would give the last step, so we add 1 to the length.

Using the assumption of the theorem $N_k(v, w) = (A^k)_{v,w} = A^k(v, w)$ and substituting:

$= \sum_{u \in V} A^k(v, u)A(u, w)$ → matrix multiplication → $(A^k \cdot A)_{v,w} = (A^{k+1})_{v,w}$

- assuming that the theorem is true for length k, we find that it is true also for length $k + 1$ → given the initial condition that it is true for $k = 1$, then it is true for all the others.

$= A^{k+1}(v, w) \rightarrow N_{k+1} = A^{k+1}$

Additional notion

A graph that has no loops is called a **tree**.

If a tree has n vertices, it will have $n - 1$ edges.

If you cut any edge of the tree, you are going to divide your graph in two subgraphs that are not connected → you can solve the subproblems and add the solutions together.

ASYMPTOTIC NOTATION

- $O - \Omega - \Theta$ - notation used to have a common language
- Recursion tree → for an intuition
- Master method → given any recursive equation that describes the behavior of any recursive algorithm allows you to compute the asymptotic behavior (computational cost).

O **− notation (upper bounds)**:

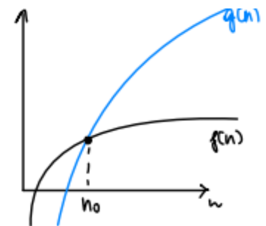
we say that a certain function f is a big $- O$ of a function g ($f(n) = O(g(n))$ if it is upper-bounded by some constant c multiplied by $g(n)$ if n (number of steps of the algorithm) is sufficiently large .

For $c > 0,\ n_0 > 0, \forall n > 0$: $\quad 0 \leq f(n) \leq cg(n)$

f is therefore upper-bounded by g: there exists a constant multiplied by g such that f is always smaller or equal to g if n is sufficiently large ($n \geq n_0$).

Set definition of O-notation

Instead of having one precise function , you can say that you have a function plus some term which is smaller than n for n sufficiently large → it is a set of functions. You can say that this set of functions is $O(g(n))$ if the same condition applies for n sufficiently large.



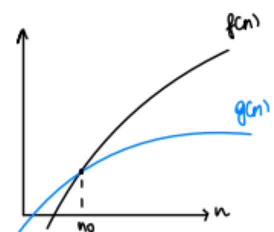
$$O(g(n)) = \{f(n)\text{: there exists constants } c > 0, n_0 > 0 \text{ such that}$$
$$0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0\}$$

Convention: a set in a formula represents an anonymous function in the set.

Example: $n^2 + O(n) = O(n^2)$ means for any $f(n) \in O(n)$: $n^2 + f(n) = n(n)$ for some $n(n) \in O(n^2)$

Ω **−notation (lower bounds)**:

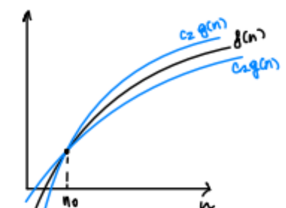
We say that f is Ω of g if f is greater or equal to some constant c times g



$$\Omega(g(n)) = \{f(n)\text{: there exists constants } c > 0, n_0 > 0 \text{ such that}$$
$$0 \leq cg(n) \leq f(n) \quad \forall n \geq n_0\}$$

Θ **−notation (tight bounds)**

If $f(n)$ is within a band , upper bounded by g and lower bounded by g multiplied by different constant ($f(n)$ both big O and big Ω of g) it means that $f(n)$ is equal to $g(n)$ up to a constant.



$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

$$\Theta(g(n)) = \{f(n)\text{: there exist positive constants } c_1, c_2 \text{ and } n \text{ such that}$$
$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0$$

Example: $\frac{1}{2}n^2 - 2n = \Theta(n^2)$

o −notation and ω −nonation:
O − and Ω − notation are like ≤ and ≥.
o − and ω − notation are like < and >.
A function is a little o of another if it is strictly dominated.

$o(g(n) = \{f(n)$: for any constant $c > 0, n_0 > 0$ such that $0 \le f(n) < cg(n)\ \forall n \ge n_0\}$

- { } set notation to indicate the family of functions
- In practice you take the limit $\lim_{n\to\infty} f(n)/g(n) = 0$ because $g(n)$ grows faster than $f(n)$.

es. $2n^2 = o(n^3)$ $(n_0 = 2/c)$

A function $f(n)$ is a little ω of $g(n)$ if $f(n)$ is strictly dominating $g(n)$

$\omega(g(n)) = \{f(n)$: for any constant $c > 0, n_0 > 0$ such that $0 \le cg(n) < f(n)\ \forall n \ge n_0\}$

- In practice you take the limit $\lim_{n\to\infty} f(n)/g(n) = \infty$ because $f(n)$ grows faster than $g(n)$.

$\exists c$:

$f(n) = O(g(n)$	$f(n) \le cg(n)$	$n > 0$
Ω	$f(n) \ge cg(n)$	$n > 0$
o	$f(n) < cg(n)$	$n > 0$
ω	$f(n) > cg(n)$	$n > 0$

Solving recurrences

For a given algorithm, we want to be able to say that the running time is upper bounded and lower bounded by some functions → we want to characterize how hard it is going to be to solve the problem, in terms of computational cost.
Particular case of recursive algorithms: the running time of the algorithm satisfies a recursive equation.

- The analysis of merge sort required us to solve a recurrence.
- Recurrences are like solving integrals, differential equations, etc. → there is not a single method to solve the equation.
- Applications of recurrences to divide-and-conquer algorithms.

Recursion-tree method

- A recursion tree models the costs (time) of a recursive execution of an algorithm →It gives us a mental picture of what happens as you iterate the recurrence.
- The recursion-tree method can be unreliable , just like any method that uses ellipses (... in a sentence).

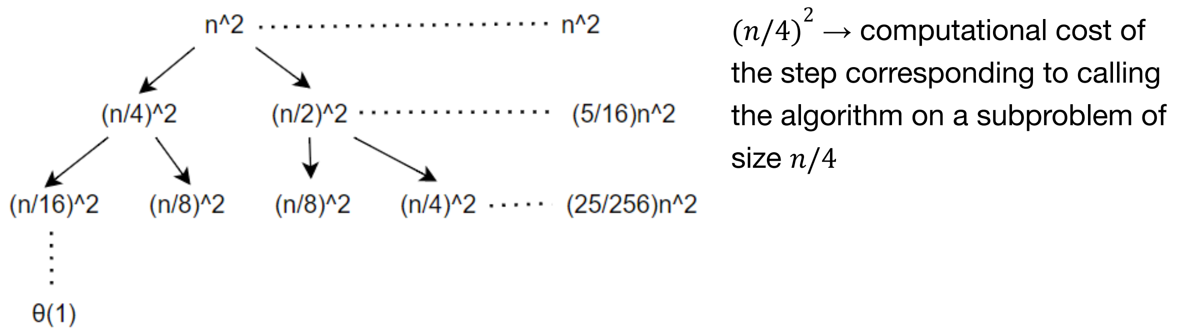
- The recursion-tree method promotes intuition, however.
- The recursion-tree method is good for generating guesses for the substitution method.

Example of recursion tree:

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

The problem takes as an input an array of size n, and it divides it in two sub-arrays of size $n/2$ and $n/4$ and when merging together the solutions of these two subproblems it is going to pay a computational cost of n^2.

We will reapply the algorithm to the subproblems with a certain computational cost → the recursion tree is a picture of this process.



$(n/4)^2$ → computational cost of the step corresponding to calling the algorithm on a subproblem of size $n/4$

Total (sum of all the partial costs of all levels)=$n^2(1 + \frac{5}{16} + (\frac{5}{16})^2 + ...) = \Theta(n^2)$

$T(n) = n^2 \sum_{l=0}^{L\,max} (\frac{5}{16})^l = n^2 \sum_{l=0}^{L\,max} = ((1 + \lambda^{Lmax+1})/(1 - \lambda)) \cdot n^2$ → it is a geometric series.

$T(n) = \Theta(n^2)$

Master method (theorem)

The master method applies to recurrences of the form:

$$T(n) = aT(n/b) + f(n)$$

- You start with a problem of size n , you divide it into a subproblems of size a/b, and while doing this you pay a computational cost of $f(n)$
- $a > 0$ → how many times you are applying this recursion on problems of size n/b
- $n/b < 1$ → reduce the size.

where $a \geq 1, b \geq 1$, and f is asymptotically positive ($f(n) > 0,\ n > n_0$).

- We can only apply this theorem when the algorithm solves the problem by dividing it into subproblems of equal size.
  When n/b is not an integer number, you either take the ceiling or the floor of the number.

Compare $f(n)$ with $n^{\log_b a}$ → compare the two functions, and depending on their relation you have different cases:

1. $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.

$f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ϵ} factor) → $f(n)$ upper-bounded by $n^{\log_b a - \epsilon}$.

Solution: $T(n) = \Theta(n^{\log_b a})$

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.

   $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$

3. $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constat $\epsilon > 0$.

   $f(n)$ grow polynomially faster than $n^{\log_b a}$ (by an n^{ϵ} factor) and $f(n)$ satisfies the **regularity condition** that $af(n/b) \leq cf(n)$ for some constant $c < 1$.

   Solution: $T(n) = \Theta(f(n))$.

If one of these 3 conditions holds, we can use this theorem, otherwise we have to use other (complicated) methods.
Examples:

1. $T(n) = 4T(n/2) + n$

   $a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n$

   Case 1: $f(n) = O(n^{2-\epsilon})$ for $\epsilon = 1$

   $\therefore T(n) = \Theta(n^2)$

   $f(n) = O(n^{\log_b a - \epsilon})$
   $\lg_2 4 = 2\lg_2 4 = 2$
   $f(n) = O(n^{2-\epsilon})$
   $f(n) = n \leq n^{2-\epsilon}$
   $\epsilon > 0,\ n > n_0$

2. $T(n) = 4T(n/2) + n^2$

   $a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2$

   Case 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$

   $\therefore T(n) = \Theta(n^2 \lg n)$

   $\lg_2 4 = 2$
   Case 1: $f(n) = O(n^{2-\epsilon})$
   n^2 → NO
   Case 2: $f(n) = \Theta(n^2 \lg^k n)$
   true for $k = 0$ → YES

3. $T(n) = 4T(n/2) + n^3$

   $a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^3$

   Case 3: $f(n) = \Omega(n^{2+\epsilon})$ for $\epsilon = 1$

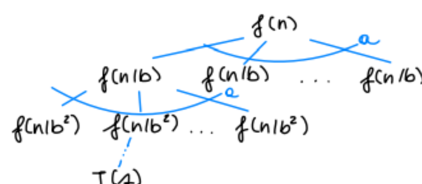
   and $4(n/2)^3 \leq cn^3$ (reg.cond.) for $c = 1/2$

   $\therefore T(n) = \Theta(n^3)$

   Case 1: $n^3 = O(n^{2-\epsilon})$ → NO
   Case 2: $n^3 = \Theta(n^2 \lg^k n)$ → NO
   Case 3: $n^3 = \Omega(n^{2+\epsilon})$ → Yes
   $\forall \epsilon: 0 < \epsilon < 1$

Idea of master theorem

$T(n) = aT(n/b) + f(n)$



$f(n)$
$af(n/b)$
$a^2 f(n/b^2)$
$\vdots$
$a^k f(n/b^k)$
$n^{\log_b a} T(1)$

Recursion tree:
We know how many levels we are going to have :
after k steps the size of the subproblem will be n/b^k. We are going to stop when
$n/b^k = \Theta(1)$
h (height of the tree) = value of k at which we are going to stop: $n/b^k = 1 \Rightarrow n = b^n$,
$\log_b n = h$.
We are going to stop after $\log_b n$ steps (very small, computationally it is the best you can do).
How many leaves do you have at the end?
Every step you have a number of leaves→ at the end # leaves = $a^h = a^{\log_b n} = n^{log_b a}$

- computations: $a^n = a^{\log_b n} = a^{\ln n/\ln b} = a^{(\ln a/\ln a)\cdot(\ln n/\ln b)} = a^{\log_a n \log_b a} = (a^{log_a n}) = n^{log_b a}$

Three cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = O(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if f satisfies the smoothness condition $af(n/b) \leq cf(n)$ for some constant $c < 1$, then $T(n) = \Theta(f(n))$

Divide and conquer (continue):

- Binary search → example of data structure. If data is stored according to some rules, it can be very easy to search for them.
- Powering a number
- Matrix multiplication

Binary search
Find an element in a sorted array: → array that contains elements that have an ordering relation

1. **divide**: check middle element
2. **conquer:** recursively search 1 subarray → reapply the procedure to half of the problem
3. **combine**: trivial

Binary search compares the target value to the middle element of the array: if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful or the remaining half is empty.
Although specialized data structures designed for fast searching (such as hash tables) can be searched more efficiently, binary search applies to a wider range of search problems.
Recurrence for binary search:
$T(n) = 1\, T(n/2) + \Theta(1)$

- # of sub problems : $a = 1$
- $n/2$ → subproblem size
- $\Theta(1)$ → work dividing and combining → just 1 comparison → computational cost of order 1.

We start with an entire array, and at the next step we reapply the same algorithm to an array whose size is half of the previous one → recursive procedure → we reapply the algorithm to sub arrays.
In this case we reapply the algorithm to only one subarray , because the other one is no longer considered.
Each time we just have to do 1 comparison (operation) → this is because the list was already sorted.
We can apply the master theorem:
$a = 1, b = 2, f(n) = \Theta(1)$
$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \Rightarrow$ Case 2 ($k = 0$)
Solution: $T(n) = \Theta(n^{\log_b a} \cdot \lg_n^{k+1}) = \Theta(\lg n)$

Powering a number
Problem: Compute a^n, where $n \in N$
Naive algorithm: $\Theta(n)$
Divide and conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} & \text{if } n \text{ is odd} \end{cases}$$

You split the problem in two identical parts, so you just need to compute one of them.
Apply this recursive algorithm → obtain the final result in a logarithmic number of steps (because you are dividing the size by 2 each time) → same recursion as the insertion sort:
$n \rightarrow n/2 \rightarrow ... \rightarrow n/2^k = 1 \Rightarrow k = \log_2 n$
$T(n) = T(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(\lg n)$ where $\Theta(1)$ is the computational cost of multiplying numbers (1,2 or 3)

Matrix multiplication
For matrices we cannot use the same methods as before.
Input: $A = [a_{ij}]$, $B = [b_{ij}]$
Output: $C = [c_{ij}] = AB$
$i, j = 1, 2, ..., n$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$ → these are n operations because you have to sum n numbers.

Standard algorithm

```
for i ← 1 to n
    do for j ← 1 to n
        do c_ij ← 0
            for k ← 1 to n
                do c_ij←c_ij + a_ik·b_kj
```

When we multiply two matrices, the computational cost is of the order of n^3, since we have to do n operations and we have n^2 elements → multiplying two matrices is a costly procedure.

Randomized algorithms (algorithms that use randomness - like flipping a coin)
Advantages:

- simplicity
- performance

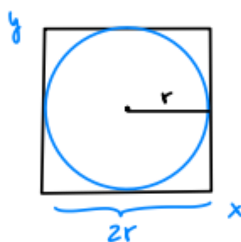
For many problems a randomized algorithm is the simplest or the fastest; or both.
Definition of randomness
We can think of it as generated by fluctuation.
The only real source of randomness in nature is quantum mechanics, because the equations that describe it are equations for probability.
In the context of computer science we are talking about pseudo-random number generators (undistinguishable for not too long sequences).

Randomized algorithms
Algorithms that make random decisions.
That is:

- can generate a random number x for some range $\{1 \dots R\}$ → if for merge-sort we choose as an element to separate arrays a random element → randomized version of merge-sort called **quicksort**.
- make decision based on the value of x.

Why would it make sense? If we give an input i and random bits r to a randomized algorithm, this will give us an $\text{output}_{i,r}$

Example: **approximating** π



$A = \pi \cdot r^2$
$T = (2r)^2$
$A/T = \pi r^2/(2r)^2 = \pi/4$ $\pi \simeq 4 \cdot A/T$ → we could compute π by estimating the ratio of the two areas.

If you have a random number generator that generates coordinates , where x and y are in between 0 and $2r$:

- $rand: (x, y)$
- $x \in [0, 2r]$
- $y \in [0, 2r]$

You see if the coordinate generated is:

- inside the circle → you call the point red

- outside the circle → you call it blue

The random numbers are distributed uniformly → the area of the circle is proportional to the number of points inside, because the probability distribution of these points is uniform over the square → if you count how many points fall inside the circle This will be proportional To the area.
Similarly, the area of the square will be proportional to the number of blue + the number of red points.
You can estimate A/T as the number of red points divided by the number of red + blue points-
If you do this for n very large you get an approximation of π.

Example of Randomized Algorithm: **Monte Carlo e Las Vegas**
There are two classes of randomized algorithms:
1. a **Monte Carlo** algorithm runs for a fixed number of steps and produces one answer that is correct with a certain probability.
2. a **Las Vegas** algorithm always produces the correct answer; its running time is a random variable whose expectation is bounded (say by a polynomial).

Notice: the probabilities are defined by the random numbers of the algorithm, not by random choices of the problem instance.

Two basics examples
1. Matrix product checker: is $AB = C$?
2. Quicksort: example of divide and conquer → fast and practical sorting algorithm.

Another example of randomized algorithm : a randomized version of merge-sort, which has some advantages compared to standard merge-sort: choosing the element for splicing at random makes it impossible for an adversary to produce a list which produces the worst case scenario for the algorithm → there is no way to initialize the problem in a way that the algorithm is going to fail badly.

Matrix product checker
Given: $n \times n$ matrices A, B, C
Goal: is $A \times B = C$? → we want to check this with a randomized algorithm.
We will see an $O(n^2)$ algorithm that:
- if answer = YES, then Pr[output = YES]=1
- if answer = NO, then Pr[output=YES] ≤ ½ → when you have a scalar product (multiply something by a vector) on both sides you have a sum. The two sums might be equal even though the single elements are different.

How long would it take by a trivial algorithm to do this?
- you compute $A \cdot B \rightarrow n^3$ operations
- you compute each matrix element with the target matrix → check n^2 elements
- computational cost $T \sim \Theta(n^3) + \Theta(n^2) = \Theta(n^3)$

<u>The algorithm</u>
- Choose a binary vector $x[1 \dots n]$ such that $Pr[x_i = 1] = 1/2,\ i = 1,\dots, n \rightarrow$
  $x = [0, 0, 1, 0, 1,\dots, 1, 0, 1, 0, 0]$
- Check if $ABx = Cx$

Does it run in $O(n^2)$ time? Yes, because $ABx = A(Bx)$ → you first compute Bx, which is a matrix times a vector: it requires $O(n^2)$ operations and you obtain a vector. You then multiply again by a matrix (n^2 operations) → overall: $O(n^2) + O(n^2) = O(n^2)$.

Correctness
Let $D = AB$, need to check if $D = C$.
If $D = C$, then $Dx = Cx$, so the output is YES (for any x)→ each product requires n operations, and at the end you have n elements you have to compute . You repeat it twice: overall $O(n^2)$.
If $D \neq C$, presumably there exists x such that $Dx \neq Cx$:

- if $AB = C$, then check $Dx = Cx$ always returns "Yes"
- if $AB \neq C$, then check $Dx = Cx$ returns "Yes" with a probability that is less than or equal to one half.

We cannot imply that $Dx \neq Cx$ because even if the matrices have different elements, the sum of all the elements can be the same.
We want to be able to generate a lot of vectors $\bar{x}$ at random that are statistically independent in such a way that for each test that we do, the probability of error gets multiplied [P(err)·P(err)]

Proof
Generic vectors

- consider vectors $d \neq c$ (rows of the matrices) (say, $d_i \neq c_i$) → they need to differ at least in one index.
- Choose a random binary x
- We have $dx = cx$ if and only if $(d - c)x = 0$
- $Pr(d - c)x = 0]$?

If $x_i = 0$, then $(c - d)x = s_1$
If $x_i = 1$, then $(c - d)x = s_2 \neq s_1$
So, ≥ 1 of the choices gives $(c - d)x \neq 0 \rightarrow Pr[cx = dx] \leq 1/2$, i.e. given that the two vectors are not equal , there exists at least one choice of the index i such that $(c - d)x \neq 0$ to with probability $1/2$.
The probability that a "Yes" check on the random vector x, corresponds in fact to an overall "No" for the matrix product equality is less or equal to $1/2$.
If we want to reduce the probability from $1/2$ to $1/4$ we run the algorithm twice, using independent random generators.

Is $A \times B = C$?

- If answer = YES, then Pr[output=YES]=1
- If answer = NO, then Pr[output=YES]≤ $1/2$

To reduce $1/2$ to $1/4$:

- run the algorithm twice, using independent random numbers.
- output YES only if both runs say YES

Analysis:

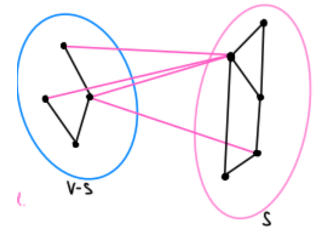
- If answer = YES, then Pr[$output_1$=YES,$output_2$=YES]=1
- If answer = NO, then Pr[$output_1$=YES]=Pr[$output_1$=YES,$output_2$=YES]= Pr[$output_1$=YES]*Pr[$output_2$=YES]≤ $1/4$

K runs, i.e. $O(kn^2)$ operations, lead to an error probability which is $\leq 1/2^k$.

$$O(kn^2) \rightarrow Pr(err) \leq 1/2^k$$

The number of repetition is going to depend on the error but not on the size, so the total cost is going to be $k \cdot n^2$, so it depends both on the error that you are ready to tolerate and on the size of the matrix.

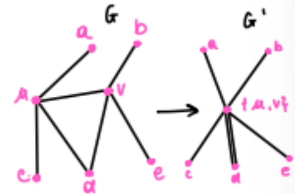
MIN-CUT for UNDIRECTED GRAPH
Find a minimum cut in a graph.
Given an **undirected graph**, a global MIN-CUT is a cut $(s, v - s)$ minimizing the number of **crossing edges**, where a crossing edge is an edge (u, v) s.t. $u \in S$ and $v \in V - S$.
You divide the vertices in 2 sets, The cut is the number of edges that connect the two sets.
You want to find a partition of the graph in 2 parts such that the number of the edges connecting the two sets is minimized.



Graph contraction (contraction of an edge)
For an undirected graph G, we can construct a new graph G' by contracting (contract only vertices that are connected) two vertices u, v in G as follows:

- u and v become one vertex $\{u, v\}$ and the edge (u, v) is removed → you fuse two vertices together.
- the other edges incident to u or v in G are now incident on the new vertex $\{u, v\}$ (new vertex with double label) in G'.

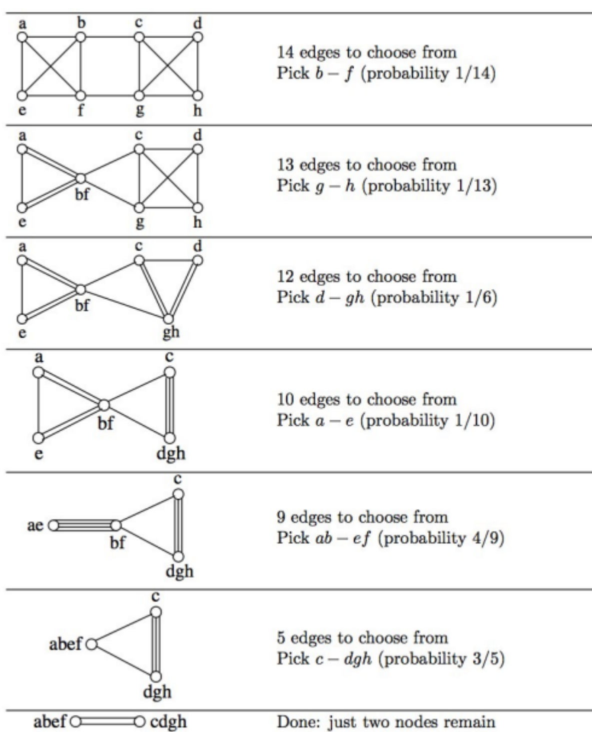
Note: there may be multi-edges between two vertices. We just keep them.



Karger's min-cut algorithm

1) Graph G
2) Contract nodes C and D
3) contract nodes A and CD
4) cut $C = \{(A, B), (B, C), (B, D)\}$

Note: C is a cut but not necessarily a min-cut.
When we contract, we pick at random one edge and we contract the two vertices connected by this edge. All the edges that connect these two nodes disappear.
Example



We assign to each edge the same probability of being chosen → we are uniformly choosing one edge at random whose vertices are going to be contracted → if you have two nodes that have a double edge they have a higher chance of being merged (twice the probability).

By contracting you find a cut

Karger's MIN-CUT

```
INPUT: A graph G=(V,E)
OUTPUT: A cut C ⊆ E
BEGIN
     REPEAT
          choose a random edge e;
          contract e and merge its endpoints into a single vertex;
     UNTIL there are only two vertices a,b left;
     Let C be the set of edges between a and b;
     RETURN C;
END
```

By following this procedure, we are going to end up with a cut.
The probability that what you find is a minimum cut is $1/n^2$ (really high probability).
If you run the same procedure an order n^2 of times, you have a probability of order 1 of finding a minimum cut.

Another example
After three contractions, a particular minimum cut C_{min} consisting of the two dashed edges survives.
Note that, in the second step, there are two edges connecting the upper two vertices, so the probability that they will be merged is $2/5$.

We repeat this process until only two vertices are left. At that point, let C be the set of surviving edges connecting these two vertices. C is a cut, since it separates a into the two pieces that contracted to form a and b respectively .
What is surprising is that, with reasonably high probability, C is as small as possible.

To see this , let C_{min} be the minimum cut, or one of the minimum cuts if it is not unique, and suppose that C_{min} has size k (optimal minimum size of the cut). Then $C = C_{min}$ if and only if these k edges survive to the end of the algorithm without being contracted.

Now consider the step of the algorithm when there are t vertices left.
Each vertex must have at least k edges, since otherwise cutting its edges would give a cut of size less than k (minimum possible size → not possible).
Thus there are at least $t\,k/2$ (divide by 2 to avoid overcounting) edges left, and the probability that none of the k edges in C_{min} are chosen for contraction on that step is:
$1 - k/(\#\, edges) \geq 1 - k/(tk/2) = (t - 2)/t$
This is the probability of not choosing one of the edges that belong to the minimum cut.

- $k/(\#edges)$ → probability that they are chosen
- $k/(tk/2) < \#edges$

If G has n vertices to start with, the probability that C_{min} survives until the end is the product of this probability over all $n - 2$ steps . This gives the lower bound.

$$p \geq \prod_{t=3}^{n} \frac{t-2}{t} = \frac{1 \cdot 2 \cdot 3 \ldots (n-2)}{3 \cdot 4 \cdot 5 \ldots n} = \frac{2}{n(n-1)} = O(\frac{1}{n^2})$$

Probability from $t = 3$ to n (when you have two vertices left you stop). For each step, you

take the probability of not selecting one of the golden edges. The probability of doing the whole process without touching the minimum cut is given by the product of the intermediate probabilities.
Thus, the probability of success is $\Omega(1/n^2)$ → lower bound

Note that if the minimum we are not unique , this lower band applies to the probability that any particular one survives.
Succeeding with probability $\Omega(1/n^2)$ may not seem like a great achievement.
However, we can increase the probability of success by trying the algorithm multiple times.
Since each attempt is independent, if we make $\frac{1}{p} = O(n^2)$ attempts, then the probability that none of them finds C_{min} is:

$$(1 - p)^{1/p} \leq \frac{1}{e}$$

- p: probability of succeeding in 1 run
- 1-p: probability of not succeeding
- probability that at least one run finds the optimal solution: $1 - (1 - p)^{1/p}$

Where we used the inequality $1 - x \leq e^{-x}$.
Therefore, the probability that at least one of these $\frac{1}{p}$ attempts succeeds is at least $1 - \frac{1}{e} \approx 0.63$ (by choosing a number of repetition that is proportional to 1/p you get a probability of success of 63%).
This gives an algorithm that succeeds with constant probability, i.e., with probability $\Omega(1)$.
If we want an algorithm that succeeds with high probability, i.e. with probability $1 - o(1)$, we can male a somewhat larger number of attempts.

- $1 - o(1)$ for $n \to \infty$ goes to 0. The probability of success is arbitrarily close to 1.

If we try the algorithm

$$(\frac{1}{p}) \ln n = O(n^2 \log n)$$

times, say, then the probability that every attempt fails is:

$$(\frac{1}{p})^{(1/p)\ln(n)} \leq e^{-\ln(n)} = \frac{1}{n} = o(1)$$

Thus, we can raise the probability of success from $\Omega(1)$ to $1 - o(1)$ with just a $\log n$ increase in the running time (you get arbitrarily close to 1).
Moreover, the total running time of this algorithm is competitive with the best known deterministic algorithms.

This “boosting” technique is a simple but important theme in randomized algorithms.
If each attempt succeeds with probability p, the average number of attempts before we succeed is $\frac{1}{p}$, and we can succeed with high probability by making slightly more than $\frac{1}{p}$ attempts.
If each attempt takes polynomial time and $p = 1/poly(n)$, then the total running time is $poly(n)$.