université PARIS DIDEROT PARIS 7 UFR d'Informatique

# Algorithmique

2021-2022 M1 Linguistique

F. Laroussinie

# Fonctionnement du cours

Équipe enseignante: François Laroussinie, Riccardo Vicedomini

Contact: francoisl@irif.fr

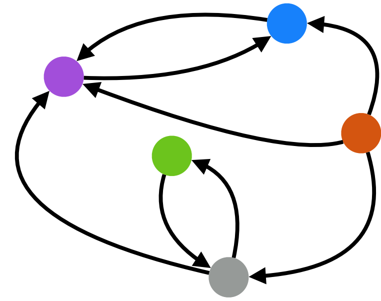
web: https://www.irif.fr/~francoisl/

Envoyer le nb d'étudiant-e-s !

Il faut programmer
les algorithmes !

# Contents

→ Les algorithmes dans les graphes.

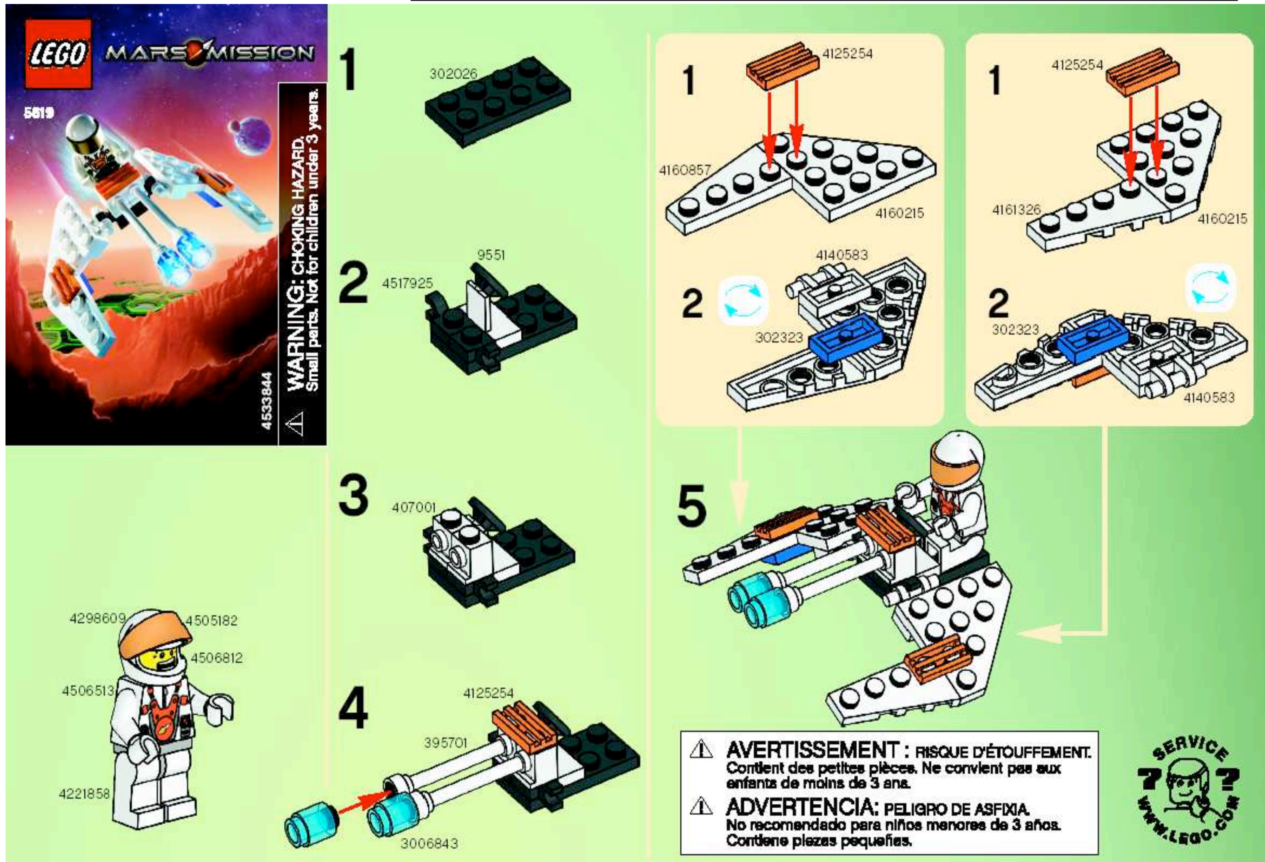


- Parcours
- Composantes (fortement) connexes
- Plus courts chemins
- Arbres couvrants minimaux
- Flots
- …

- Analysis
- Strongly connected components
- Shortest paths
- Min. Spanning Tree
- Flow
- …

# Let's go !

# Exemple d'algorithme




# Exemple d'algorithme

Résultat

les données

**484** **Émincé de rouget au pistou**

**24 heures à l'avance**
**Préparation : 40 mn - Cuisson : 30 à 35 mn**

1 kg 500 rougets.
400 g pâtes.
500 g fenouil.
2 dl huile olive.
0 dl 5 vinaigre xérès.
Coulis de tomates.
1 citron.
Basilic.
Herbes de Provence.
Thym.
Vinaigrette.
Sel.
Poivre.

Écailler les rougets, les laver, puis soulever délicatement les filets et enlever les arêtes qui restent avec une pince à épiler. Mettre les filets dans un plat creux ; arroser avec de l'huile d'olive ; saupoudrer d'herbes de Provence et laisser mariner au frais, pendant 24 heures, le tout couvert par un torchon.
Cuire le fenouil à l'eau bouillante salée, citronnée et parfumée à l'huile d'olive et avec une brindille de thym (15 à 20 mn). Égoutter.
Cuire les pâtes à l'eau bouillante salée, huilée, pendant 12 mn. Égoutter. Rafraîchir. Tenir au chaud.
Pendant ces cuissons préparer le coulis de tomates (41). Assaisonner le fenouil coupé en tranches avec la vinaigrette. Tenir au chaud. Assaisonner les pâtes avec le basilic (à volonté) haché et un mélange d'huile d'olive et de vinaigre de xérès. Disposer les pâtes sur le plat, recouvrir avec la fondue de fenouil.
Rapidement, cuire à la poêle Téfal, à feu très vif, les filets marinés pour qu'ils deviennent dorés et croustillants. Saler. Poivrer et disposer en étoile les filets de rougets, sur le plat de légumes. Servir avec le coulis de tomates en saucière.

An algorithm is a recipe to solve get a result in a finite number of instructions.

An algorithm solves a problem.

## Examples of algorithmic problems

- Input: two numbers *a* and *b*
  Output: *a+3b+ab*
- Input: an integer *a*
  Output: the sum *1+2+3+…+a*
- Input: a set of coins, a value *S*
  Output: a subset of coins whose total value is *S*
- Input: a list of words
  Output: the sorted list
- Input: two texts
  Output: the list of words occurring in the 2 texts.
- Input: a map, and two towns *A* and *B*
  Output: a shorst path from *A* to *B*
- Données: a program *P*, an input *a*
  Output: the answer to the question "does P(a) equal to 8a+5 ? »

# problem / instance

A problem

- Input: a list of words
  Output: the sorted list

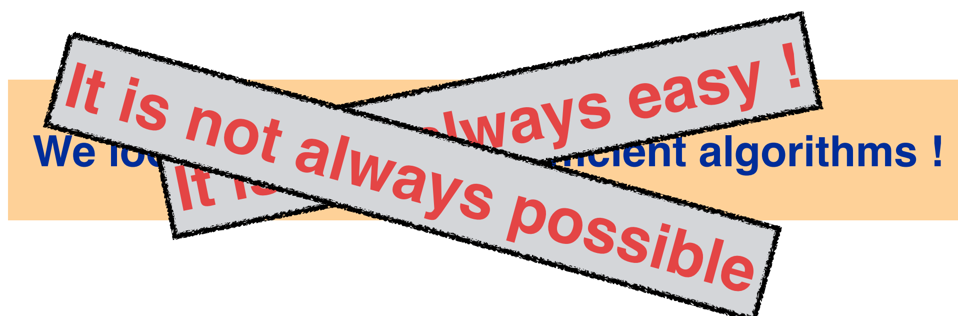
**An algorithm has to solve all instances of the problem.**

Examples of instances:

- table, vélo, chaise, pain, chocolat, tram, ciel, pomme
- Noémie, Pierre,  Alain, Célia, Juan, Leila, John
- pomme, oiseau, vélo, fgsdhgf, hjdshq, sqdkjhd
- 
- bip, bop, bip, tip

# Problems and algorithms

A given problem may admit several different algorithms (or none).



# The efficiency of algorithms

*Time complexity*

$C_A(x)$ : number of elementary instructions required for the execution of algorithm $A$ over the input $x$.

Worst case analysis: « worst case running time »

$$C_A(n) = \max_{x.|x|=n} C_A(x)$$

For every input of size n, the execution of A takes at most $C_A(n)$ instructions.

Average analysis:

$$C_A^{moy}(n) = \sum_{x.|x|=n} p(x).C_A(x)$$

probability distrib over the input of size n.

Amortized analysis:
Evaluate the cumulative cost of n operations (in worst case setting).

# Complexité des algorithmes

Obj: an order-of-magnitude

Notations: $O(\ )$, $\Omega(\ )$ et $\Theta(\ )$:

$O(g(n)) = \{f(n) \mid \exists c>0, n_0 \geq 0 \text{ s.t. } 0 \leq f(n) \leq c.g(n)\ \forall\ n \geq n_0\}$

→ contains the functions for which $c.g(n)$ is an asymptotic **upper** bound (some some constant c).

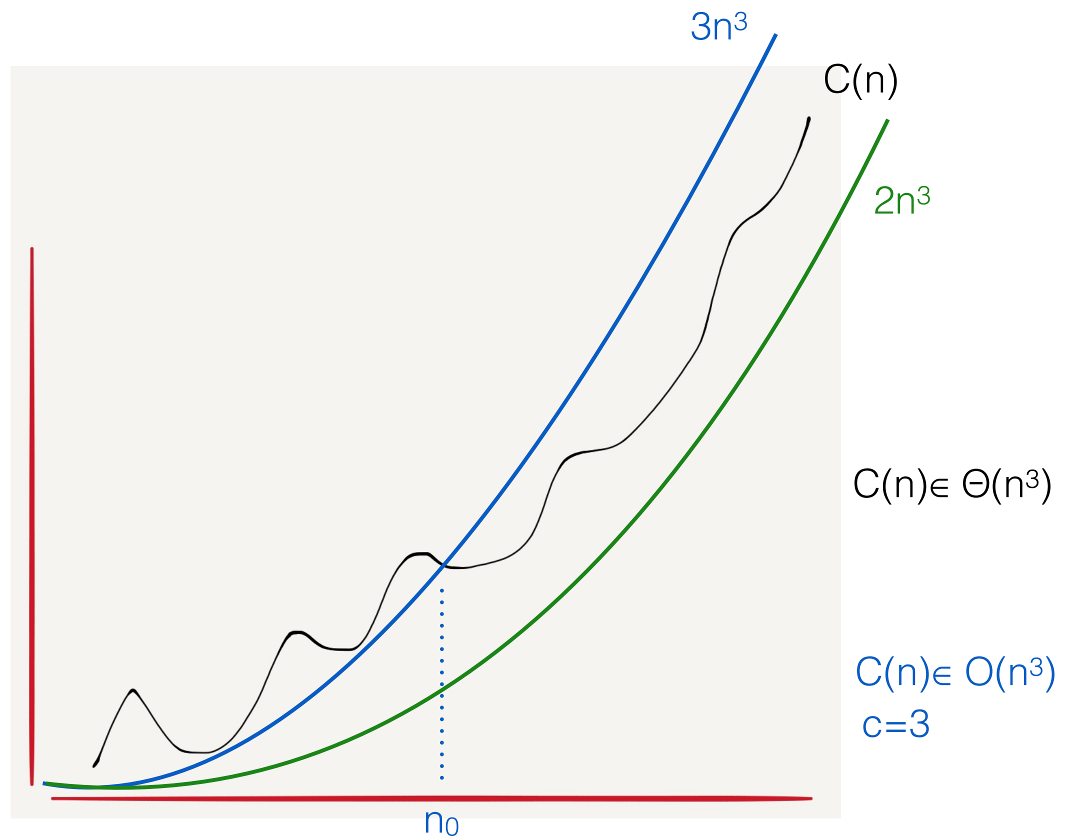
$\Omega(g(n)) = \{f(n) \mid \exists c>0, n_0 \geq 0 \text{ s.t. } 0 \leq c.g(n) \leq f(n)\ \forall\ n \geq n_0\}$

→ contains the functions for which $c.g(n)$ is an asymptotic **lower** bound.

$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2>0, n_0 \geq 0 \text{ tq } 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)\ \forall\ n \geq n_0\}$

→ $g(n)$ is an asymptotic tight bound for $f(n)$

# Notations: O,Ω,Θ




# Notations: O,Ω,Θ

- $3n^2+4n+5 \in O(n^2)$ and $3n^2+4n+5 \in O(n^5)$ …
- $3n^2+4n+5 \in \Theta(n^2)$ but $3n^2+4n+5 \notin \Theta(n^5)$
- $8n^3-3n \in \Theta(n^3)$
- $2^n+2n^3 \in \Theta(2^n)$
- $|\sin(n)| \in \Theta(1)$
- $2n+1 \in \Theta(n)$
- $3^n \notin O(2^n)$
- $(n+1)! \notin O(n!)$
- $n! \in O(n^n)$

# What is an efficient algorithms ?

We distinguish several classes of algorithms:

- Sublinear algorithms.
  Par ex. en O(log n)
- Linear algorithms: O(n)
  [or O(n.log n) ]
- Polynomial algorithms: $O(n^k)$

---

- Exponential algorithms: $O(2^n)$
- …

# What is an efficient algorithms ?

| fct $\backslash n$ | 10 | 50 | 100 | 300 | 1000 |
|---|---|---|---|---|---|
| $5n$ | 50 | 250 | 500 | 1500 | 5000 |
| $n \cdot \log_2(n)$ | 33 | 282 | 665 | 2469 | 9966 |
| $n^2$ | 100 | 2500 | 10000 | 90000 | $10^6(7c)$ |
| $n^3$ | 1000 | 125000 | $10^6(7c)$ | $27 \cdot 10^6(8c)$ | $10^9(10c)$ |
| $2^n$ | 1024 | $\ldots(16c)$ | $\ldots(31c)$ | $\ldots(91c)$ | $\ldots(302c)$ |
| $n!$ | $3.6 \cdot 10^6(7c)$ | $\ldots(65c)$ | $\ldots(161c)$ | $\ldots(623c)$ | …!!! |
| $n^n$ | $10 \cdot 10^9(11c)$ | $\ldots(85c)$ | $\ldots(201c)$ | $\ldots(744c)$ | …!!!! |

notation: (Xc) -> “a X-digit number”

(see « Algorithmics, the spirit of computing », D. Harel)

# What is an efficient algorithms ?

With a computer running $10^9$ instructions per seconde…

| Fonc.\ n | 20 | 40 | 60 | 100 | 300 |
|---|---|---|---|---|---|
| $n^2$ | 1/2500 milliseconde | 1/625 milliseconde | 1/278 milliseconde | 1/100 milliseconde | 1/11 milliseconde |
| $n^5$ | 1/300 second | 1/10 second | 78/100 second | 10 seconds | 40,5 minutes |
| $2^n$ | 1/1000 second | 18,3 minutes | 36,5 years | $400.10^9$ centuries | (72c) centuries |
| $n^n$ | $3,3.10^9$ years | (46c) centuries | (89c) centuries | (182c) centuries | (725c) centuries |

The big-bang was $13,8.10^9$ years ago.

(see « Algorithmics, the spirit of computing », D. Harel)

# Les algorithmes efficaces… et les autres.

En 10 ans, la vitesse des ordinateurs a été multipliée par 50...

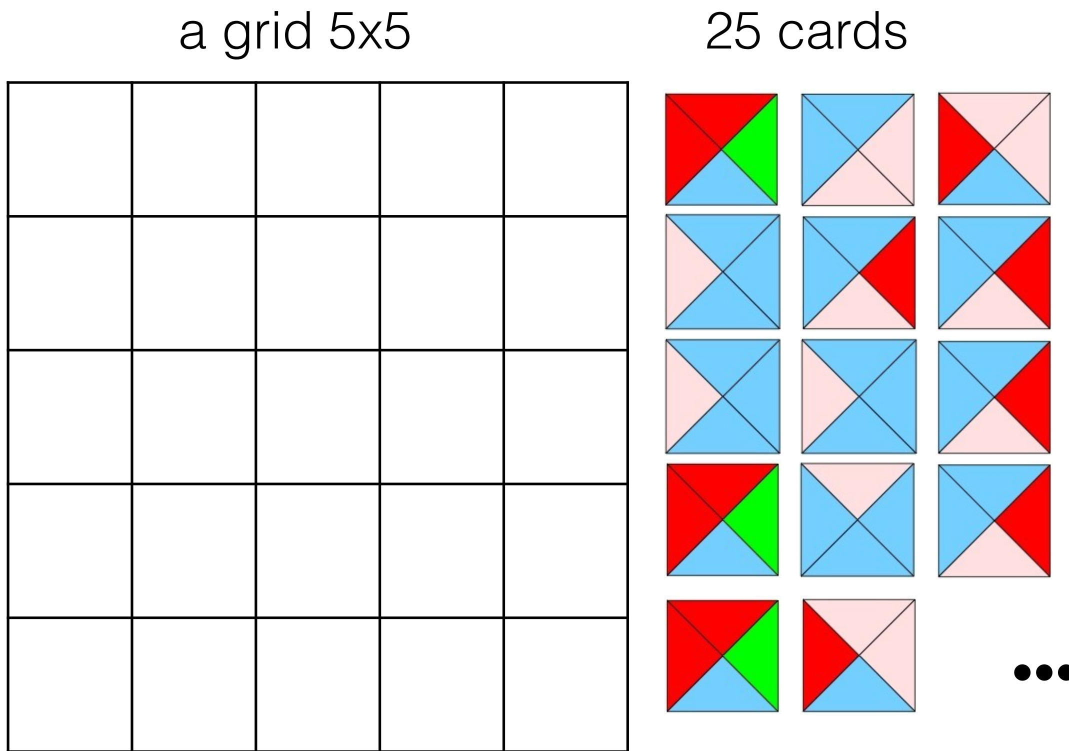
Assume that your computer solves a problem of K in one hour….

If the complexity of the algorithm is n, then...
- A computer 100 times faster would solve problem of size 100xK.
- A computer 1000 times faster would solve problem of size 1000xK.

If the complexity of the algorithm is $n^2$, then...
- A computer 100 times faster would solve problem of size 10xK.
- A computer 1000 times faster would solve problem of size 32xK.

If the complexity of the algorithm is $2^n$, then...
- A computer 100 times faster would solve problem of size **K+7**.
- A computer 1000 times faster would solve problem of size **K+10**.

# A puzzle...




# Puzzle

## We try every possible arrangement ?

25 cards to place over 25 locations:

- 25 possibilities for the first one,
- 24 for the second one,
- 23 for the third one,
- ...

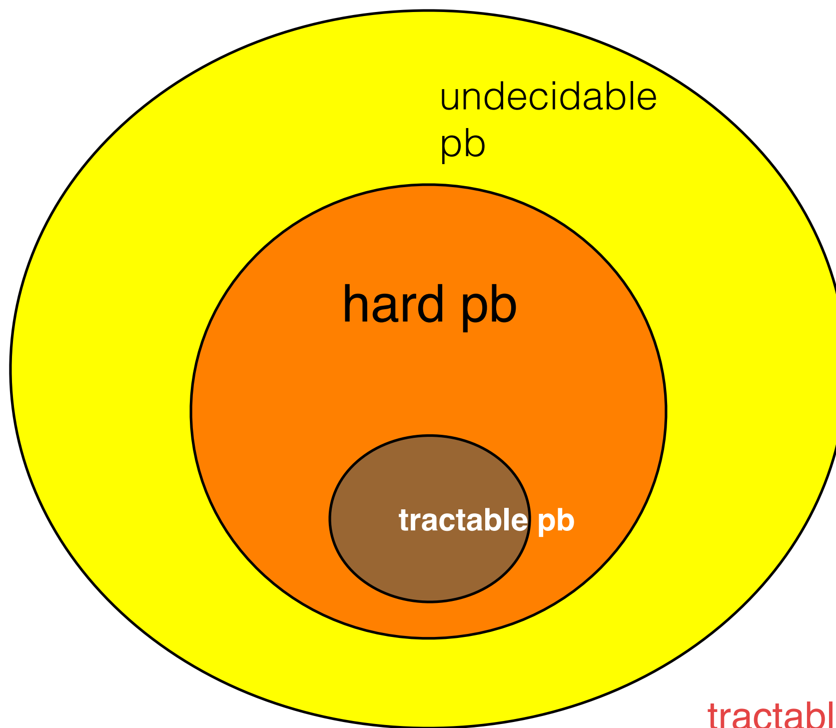
25x24x23x...x2 possibilities !

With a computer examining one billion arrangements per second, it takes…

490 millions years !

(25! is a 26-digit number…)

# Overview




tractable pb = « with a polynomial algo. »

# Efficient algorithms

$n^2 \neq 2^n$ OK !

But $n^3$, $n^2$, $n.\mathit{log}\ n$ and $n$ are significantly different !

## Comparison of « efficient » algo.
### Example

See J. Bentley
«Pearls of programming»

A problem:
Given an array of n numbers, compute the max. subsum (ie max sum of some a subarray).

| 8 | -10 | 10 | 4 | -19 | 40 | 0 | 5 | -9 | 14 | 2 | 3 | 78 | 7 | -24 | 6 | 9 | -18 | 7 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

mx sum:= 140

(≠ sum of all elements = 115)

## Comparison of « efficient » algo.

Example: find the max. subsum.

Input: an array T[0..n-1]

Output: $\text{Max} \{ \text{sum}[i,j] \mid 0 \leq i,j \leq n-1 \}$

$$\text{sum}[i,j] = \sum_{k \in [i;j]} T[k]:$$

interval: i,i+1,...,j

# Comparison of « efficient » algo.

Example: find the max. subsum.

We can easily find several algorithms for this problems:

- a naive algorithm (3 nested loops « for ») in $O(n^3)$
- an improved version in $O(n^2)$
- a « divide and conquer » solution in $O(n.\textit{log}\ n)$
- a dynamic prog. solution in $O(n)$

All solutions are «efficient algorithms »… but <u>significantly</u> different !

J. Bentley «Pearls of programming», Addison-Wesley (1986)

Example: find the max. subsum.

**TABLE I. Summary of the Algorithms**

| Algorithm | | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Lines of C Code | | 8 | 7 | 14 | 7 |
| Run time in microseconds | | $3.4N^3$ | $13N^2$ | $46N \log N$ | $33N$ |
| Time to solve problem of size | $10^2$ | 3.4 secs | 130 msecs | 30 msecs | 3.3 msecs |
| | $10^3$ | .94 hrs | 13 secs | .45 secs | 33 msecs |
| | $10^4$ | 39 days | 22 mins | 6.1 secs | .33 secs |
| | $10^5$ | 108 yrs | 1.5 days | 1.3 min | 3.3 secs |
| | $10^6$ | 108 mill | 5 mos | 15 min | 33 secs |
| Max problem solved in one | sec | 67 | 280 | 2000 | 30,000 |
| | min | 260 | 2200 | 82,000 | 2,000,000 |
| | hr | 1000 | 17,000 | 3,500,000 | 120,000,000 |
| | day | 3000 | 81,000 | 73,000,000 | 2,800,000,000 |
| If $N$ multiplies by 10, time multiplies by | | 1000 | 100 | 10+ | 10 |
| If time multiplies by 10, $N$ multiplies by | | 2.15 | 3.16 | 10– | 10 |

1984... machine: VAX-11/750

# CRAY-1 vs TRS 80 ?

**TABLE II. The Tyranny of Asymptotics**

| *N* | Cray-1, FORTRAN, Cubic Algorithm | TRS-80, BASIC, Linear Algorithm |
|---|---|---|
| 10 | 3.0 microsecs | 200 millisecs |
| 100 | 3.0 millisecs | 2.0 secs |
| 1000 | 3.0 secs | 20 secs |
| 10,000 | 49 mins | 3.2 mins |
| 100,000 | 35 days | 32 mins |
| 1,000,000 | 95 yrs | 5.4 hrs |

J. Bentley «Pearls of programming», Addison-Wesley (1986)

# And today ?

**2021**: Macbook Pro, i5 (16Gb).

(en secondes)

| | Algo 1 | Algo 2 | Algo 3 | Algo 4 | Algo 5 |
|---|---|---|---|---|---|
| 100 | 0,0145 | 0,0015 | 0,0016 | 0,0006 | 0,0000 |
| 500 | 1,6278 | 0,0345 | 0,0359 | 0,0018 | 0,0002 |
| 1 000 | 14,4006 | 0,1341 | 0,1532 | 0,0042 | 0,0005 |
| 2 000 | 111.9258 | 0.5274 | 0,6124 | 0,0096 | 0,0010 |
| 10 000 | | 13.718 | 16.3547 | 0.052 | 0.0047 |
| 30 000 | | 127.7064 | 146.9065 | 0.1731 | 0.0151 |
| 100 000 | | | | 0.6253 | 0.0525 |
| 1 000 000 | | | | 7.0648 | 0.5043 |

**1984:** 1h 22min 15min 33sec

# Remark 1

The notion of complexity is robust.

# Remark 2

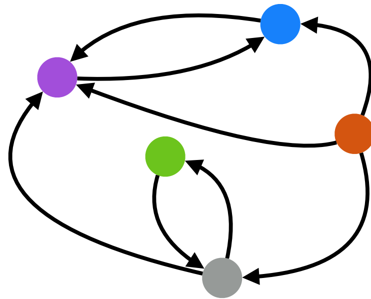
There are very different in practice

Even if all are considered as efficient.

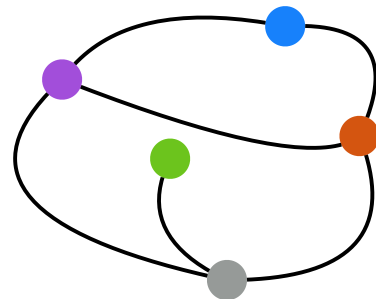
Don’t forget: there are many problems for which:

- there is no efficient algorithm… or
- no algorithm at all !

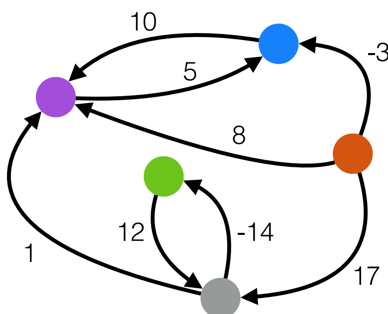
graphs



graphe orienté
directed graph or digraph

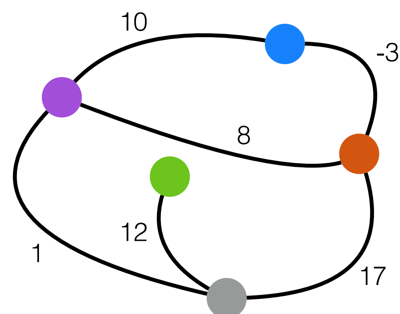


graphe non orienté
(undirected) graph




graphe orienté valué

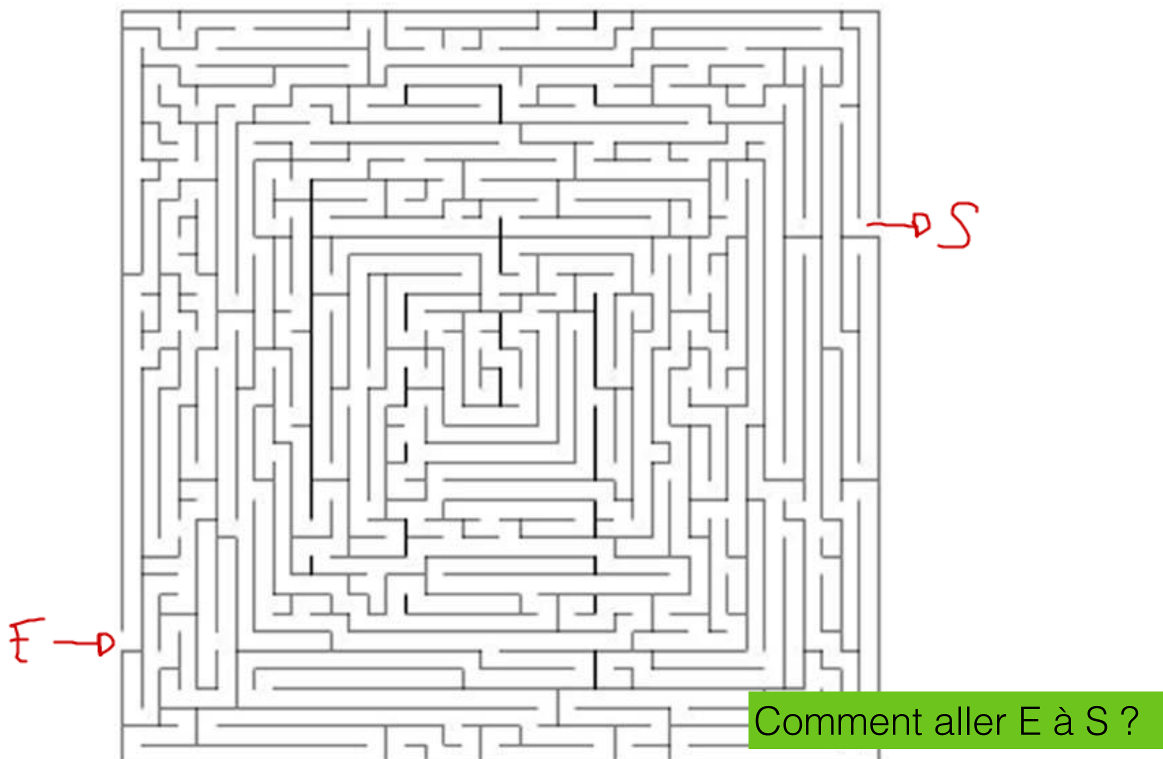
weighted directed graph



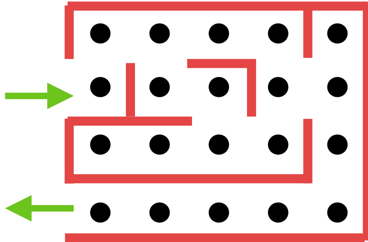

graphe non orienté valué
weighted graph

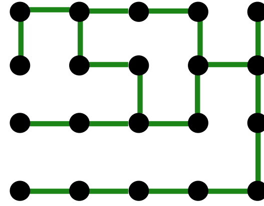
## why is it useful ?

## Un labyrinthe…



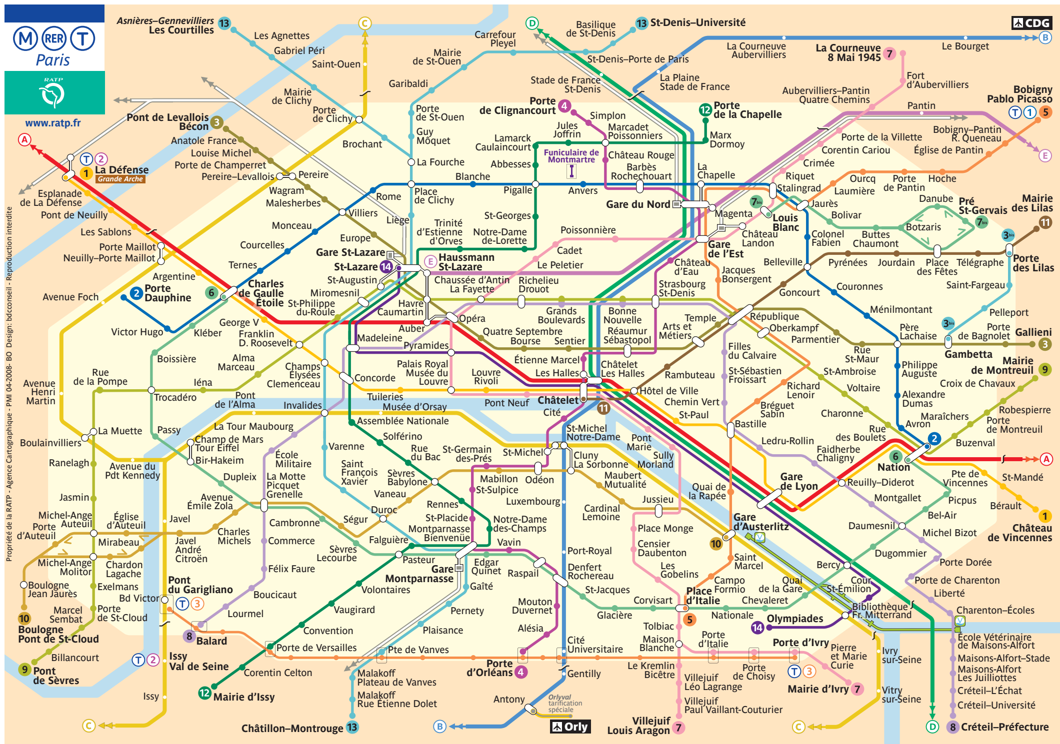
# Un labyrinthe…





= un graphe non orienté !

# Le réseau de métro…




= un graphe (non) orienté

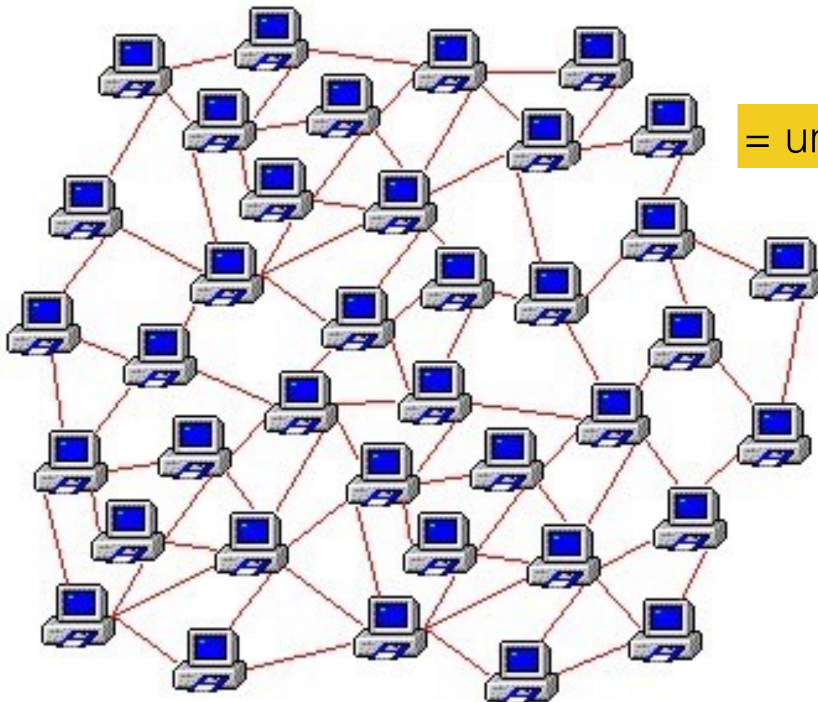
Comment aller de Glacière à Gare du Nord ? (le plus vite possible)

# Une carte routière

Comment aller de Alès à Caen ?
(le plus vite possible)

= un graphe non orienté valué !

# Un réseau d’ordinateurs



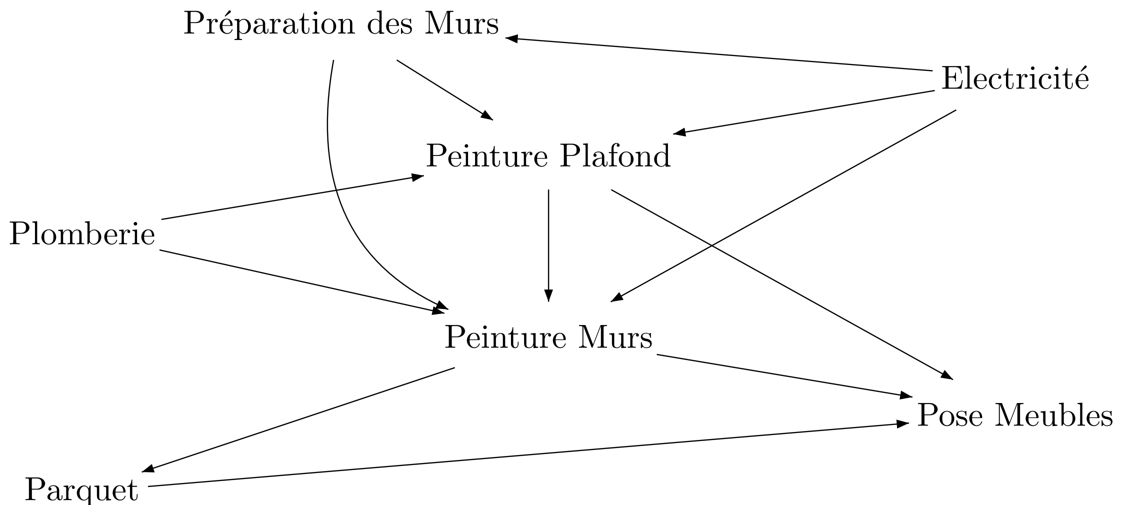
= un graphe non orienté

Le réseau est-il robuste ?
Est-ce qu’il existe au moins k chemins distincts entre deux sites ?

# Des relations binaires

La peinture se fait après l'électricité,
la peinture du plafond se fait avant celle des murs, …

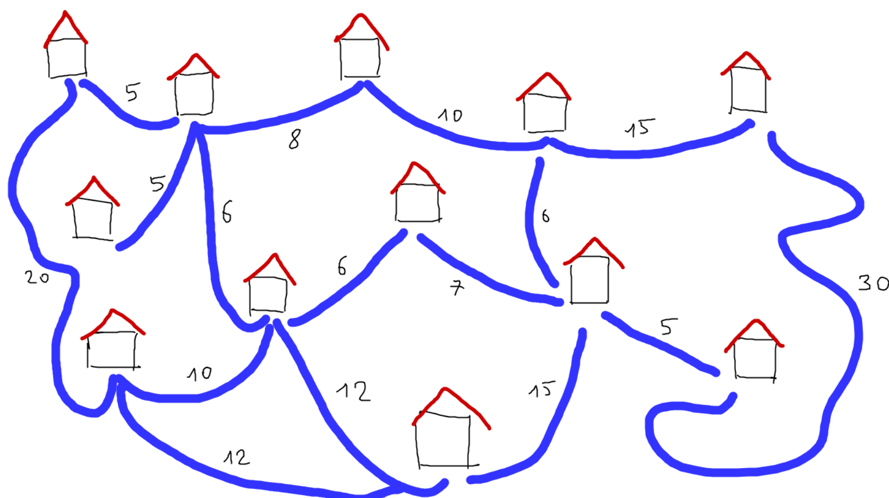
= un graphe orienté !




Comment ordonner les tâches ?

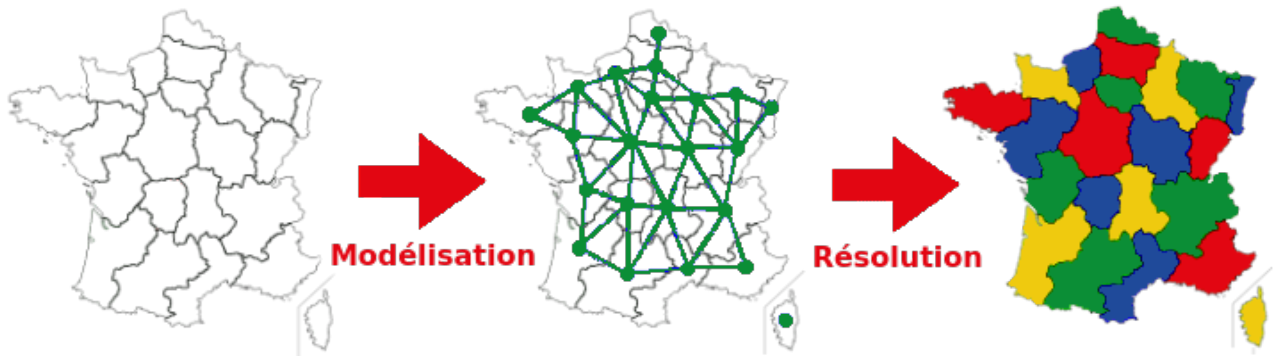
# Le problème de la ville embourbée

**Problème :** Paver suffisamment de rues pour que tous les habitants puissent se rendre n'importe où les pieds au sec. . . mais en utilisant le moins de pavés possible.




(recherche d'un arbre couvrant minimal)

# Colorier une carte



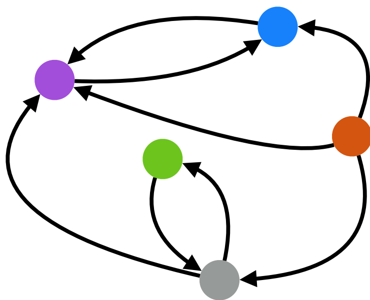

(image de paco-solver.fr)

There are many many examples where graphs are
very natural to model the considered problem.

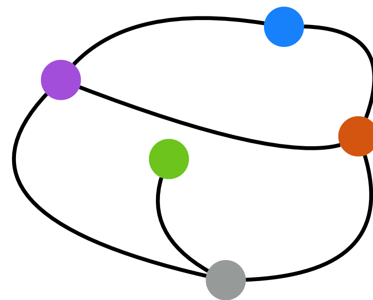
# Graphes
# Définitions

poly pages 8-15

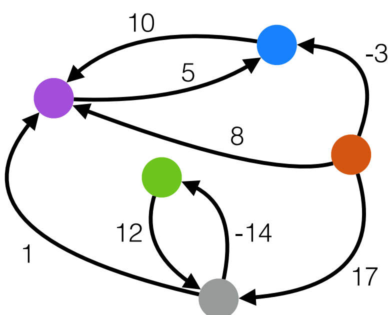
# Les graphes



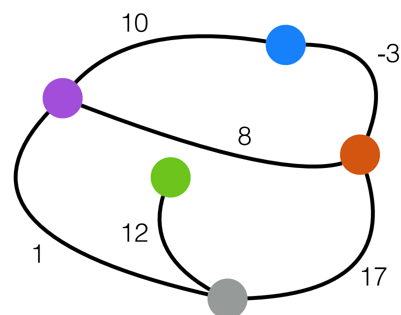
graphe orienté



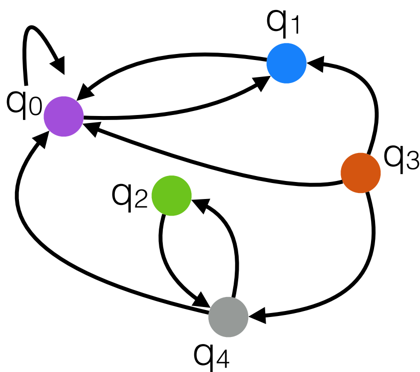
graphe non orienté




graphe orienté valué



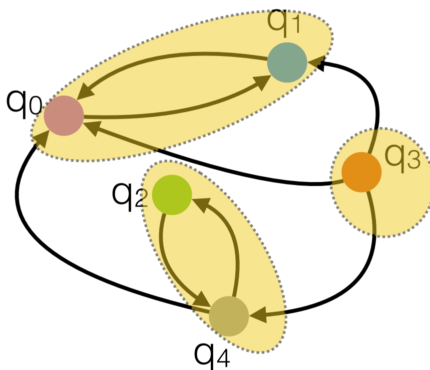

graphe non orienté valué




# Directed graphs

G=(S,A)
S = a finite set of vertices (or nodes)
A ⊆ SxS = a set of ordered pairs of vertices, called **edges** (or transitions, links, arcs…)

**Path:** for ex. q3 q4 q2 q4 q0
**Circuit:** for ex. q4 q2 q4
**(Self)loop:** for ex. q0 q0



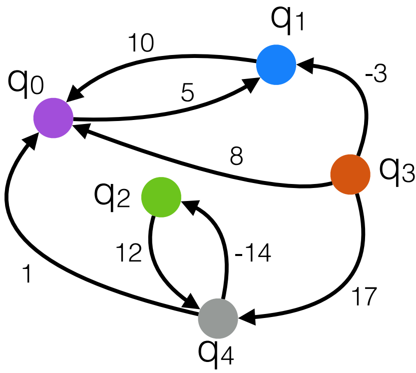

# Directed graphs

notation:
q →* q' if there exists some path from q to q'.

A strongly connected component C is a **maximal** set of vertices s.t.
if u,v ∈ C, then u →* v and v →* u.

A graph G is **strongly connected** iff it contains a unique SCC.

# Weighted directed graphs

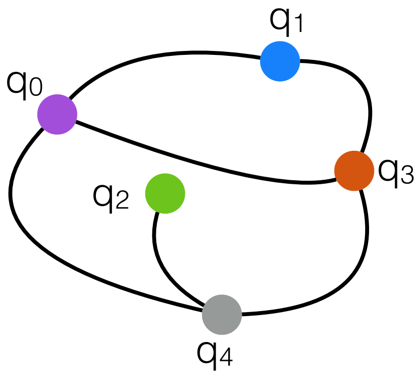



G=(S,A,w) with
S = a finite set of vertices (or nodes)
A ⊆ SxS = a set of (directed) **edges**, and:
w : A → **R:** w assigning a weight to each edge.

Paths, circuits, loops…
The weight is extended to paths:

w(q3 q4 q2 q4 q0) = 17 + -14 + 12 + 1 = 16

# (undirected) graphs

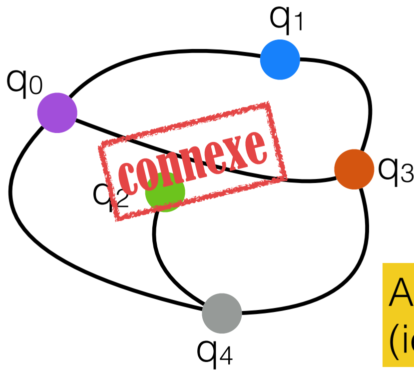



G=(S,A)
S = a finite set of vertices (or nodes)
A ⊆ SxS = a set of pairs of vertices, also called **edges.**

**Cycle:** a cycle is a path $(v_0,v_1,..,v_k)$ with $v_0=v_k$ and $k\geq 3$ and $v_1,v_2,\ldots,v_k$ are distinct.

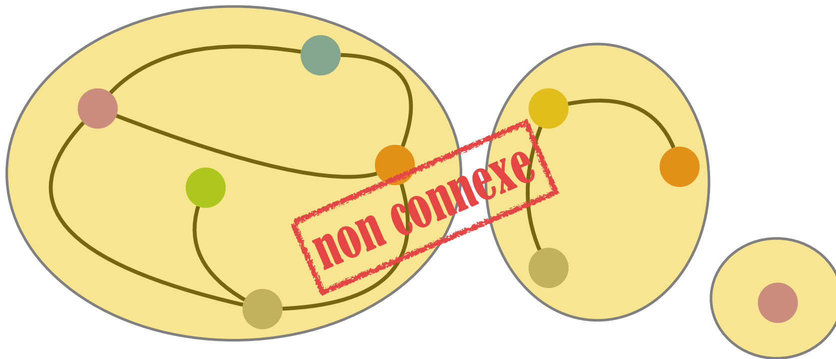
If G contains no cycle, it is said acyclic.

# (undirected) graphs

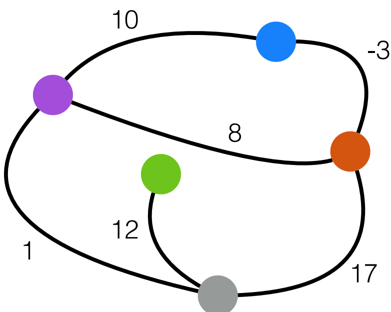



The connected components (CC) are the equivalence classes of the relation $\longleftrightarrow^*$.

A graph G is **connected** iff it has a unique CC.
(ie for any nodes x and y, the is a path from x to y).



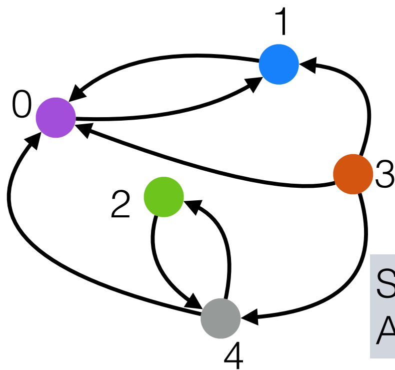

# Weighted graphs




G=(S,A,w) with
$A \subseteq \mathscr{P}_2(S)$ and $w : A \rightarrow \mathbf{R}$
w assigns a weight to every edge.

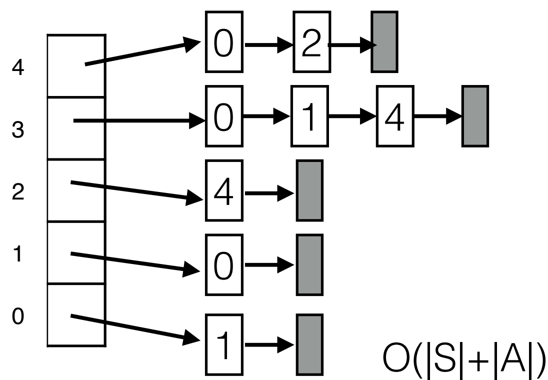
# Représentation des graphes

## How to represent directed graphs ?



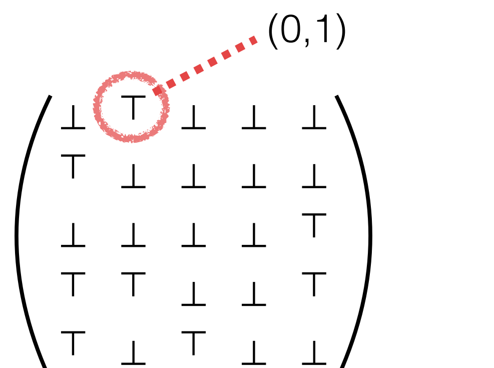

G=(S,A)
S = a finite set of vertices (or nodes)
A ⊆ SxS = a set of ordered pairs of vertices, called **edges** (or transitions, links, arcs…)

S={0, 1, …, 4}
A = { (0,1), (1,0), (3,1), (3,0), (2,4), (4,2), (3,4), (4,0) }




$O(|S|+|A|)$

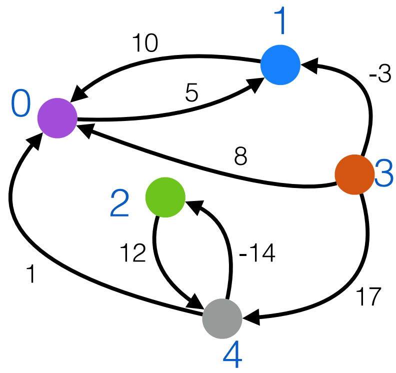
adjacency list




matrix

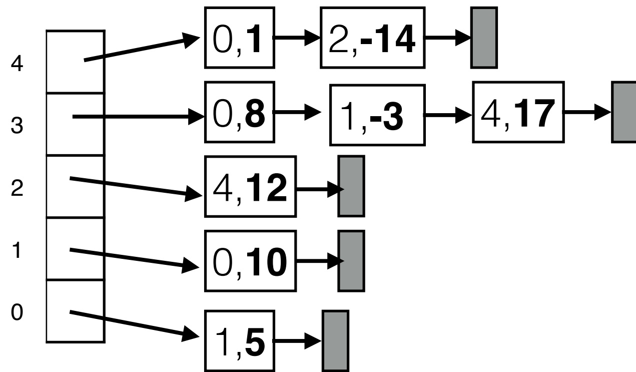
$O(|S|^2)$

# How to represent directed weighted graphs ?

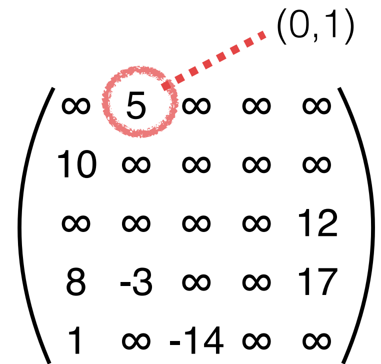



G=(S,A,w), w : A → **R**

S={0, 1, …, 4}
A = { (0,1), (1,0), (3,1), (3,0), (2,4), (4,2), (3,4), (4,0) }
w(0,1) = 5, w(1,0)=10, …

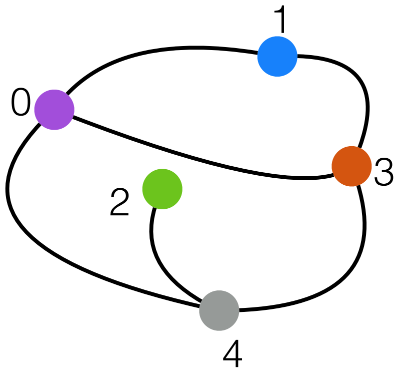



adjacency list

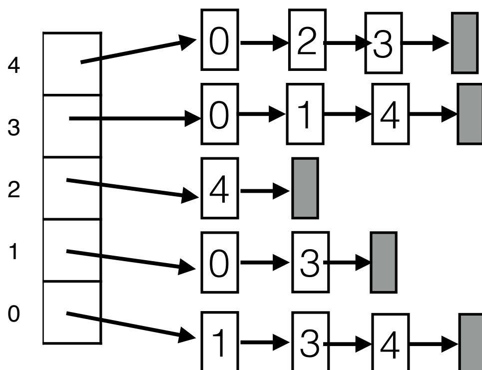



matrix

# How to represent undirected graphs ?

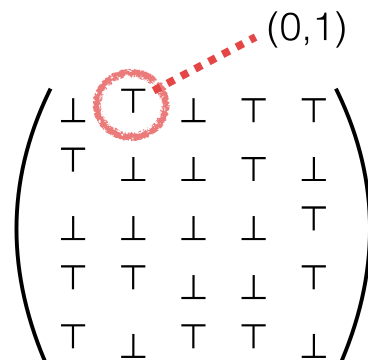



G=(S,A)
S = a finite set of vertices (or nodes)
A ⊆ SxS = a set of ordered pairs of vertices,
called **edges** (or transitions, links, arcs…)




adjacency list

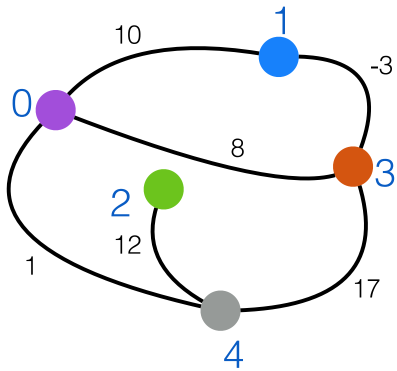
O(|S|+|A|)



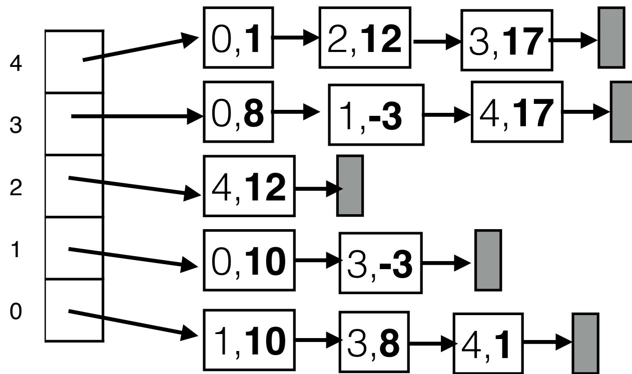

$O(|S|^2)$

symmetric matrix

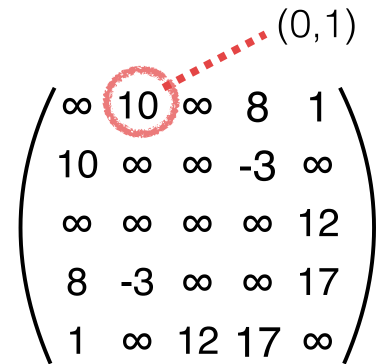
# Représentation des graphes non orientés valués




G=(S,A,w)
S = ensemble fini de sommets
A ⊆ SxS
w : A → **R**




par liste d'adjacence.




par matrice

## representation and complexity

| | Matrice d'adjacence | Liste d'ajacence |
|---|---|---|
| Espace mémoire | $O(\|S\|^2)$ | $O(\|S\|+\|A\|)$ |
| Tester si $(u,v) \in A$ | $O(1)$ | $O(\mathrm{deg}^-(u)) = O(\|S\|)$ |
| Enumérer les voisins de $u \in S$ | $O(\|S\|)$ | $O(\mathrm{deg}^-(u)) = O(\|S\|)$ |
| Enumérer tous les arcs de $A$ | $O(\|S\|^2)$ | $O(\|S\|+\|A\|)$ |
| Ajouter un arc $(u,v)$ dans $A$ | $O(1)$ | $O(1)$ |

→ the complexity of an algorithm depends on the representation.