Algorithmique et Complexité

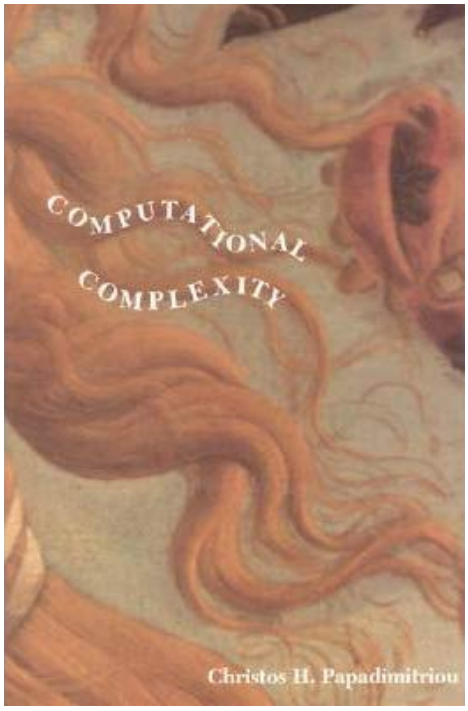
Emmanuel Hebrard et Mohamed Siala

Plan

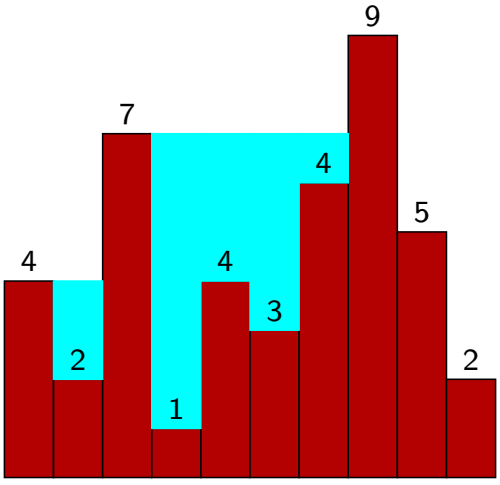
1. Introduction à la Complexité des Algorithmes
2. Analyse Asymptotique
3. Algorithmes Récursifs
4. Programmation Dynamique
5. Algorithmes gloutons
6. Représentation des Données
7. Classes de Complexité
8. Les Classes NP et coNP

Supports de cours

- Transparents sur la page du cours (ils seront distribués !)
- Support de cours d'*Olivier Bournez* pour l'Ecole Polytechnique (lien sur la page du cours)
- "**Introduction to Algorithms**"
  Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
  MIT Press.
- "**Computational Complexity**"
  Christos H. Papadimitriou
  Addison-Wesley.
- "**Computational Complexity : A Modern Approach**"
  Sanjeev Arora and Boaz Barak
  Princeton University.



Question d'un entretien d'embauche chez Google



- Soit un histogramme avec n barres *sur lequel on a versé un volume d'eau infini*.
  - Donnez un algorithme pour calculer le volume d'eau résiduel (16).
  - Donnez un algorithme pour calculer le volume d'eau résiduel en temps linéaire.

Vaincre la Combinatoire : Algo. ou Matériel ?

Problème du Voyageur de Commerce

- **donnée** : ensemble de villes
- **question** : quel est le plus court chemin passant une fois par chaque ville ?

- Méthode "Brute-force" : trois instructions par nano seconde
- Un ordinateur plus rapide : une instruction par *temps de Planck* ($5.39 \times 10^{-44} s$)
- Un ordinateur plus parallèle : remplissons l'univers de processeurs d'un mm^3

donnée	processeur 3 GHz	processeur de Planck	massivement parallèle
10 villes	1/100s		
15 villes	1 heure		
19 villes	1 an		
27 villes	8 × âge de l'univers		
35 villes	5e+23 × âge de l'univ.	5/1000s	
40 villes	4e+31 × âge de l'univ.	12 heures	
50 villes	1,5e+48 × âge de l'univ.	4000 × âge de l'univers	temps de planck
95 villes	5e+131 × âge de l'univ.	1,3e+87 × âge de l'univ.	3 × âge de l'univers

Complexité des algorithmes

- Savoir développer des algorithmes *efficaces*
- Savoir analyser l'*efficacité* d'un algorithme
- Comprendre la notion de *complexité* d'un problème

Introduction à la Complexité des Algorithmes

Problème & Donnée

Définition : Problème $\simeq$ fonction sur les entiers

- Une question Q qui associe une donnée x à une solution $Q(x)$
  - "Quel est le plus court chemin de x_1 vers x_2 par le réseau R ?"
  - "Quel est la valeur du carré de x ?"
- Q est une relation, pas toujours une fonction : plus court(s) chemin(s)
- On peut se restreindre aux fonctions

1	1
2	4
3	9
4	16
5	25
...	...

- **Problème :** "Étant donné un ensemble de villes, quel est le plus court chemin passant une fois par chaque ville ?"
- **Instance :** "Les préfectures d'Occitanie"
- **Solution :** "Auch $\rightarrow$ Montauban $\rightarrow$ Cahors $\rightarrow$ Rodez $\rightarrow$ Mende $\rightarrow$ Nimes $\rightarrow$ Montpellier $\rightarrow$ Albi $\rightarrow$ Toulouse $\rightarrow$ Carcassonne $\rightarrow$ Perpignan $\rightarrow$ Foix $\rightarrow$ Tarbes"

Algorithme

- Un algorithme est une méthode pour *calculer* la solution $Q(x)$ d'un problème, pour toute valeur de la donnée x

Algorithme pour le problème Q

- Composée d'instructions primitives : exécutable par une machine
- Déterministe : une seule exécution possible pour chaque donnée
- Correct : termine et retourne la bonne solution $Q(x)$ pour toute valeur de la donnée x

Qu'est-ce qu'une "instruction primitive" ?

Pas de définition formelle dans ce cours : langages de programmations classiques (*boucles*, *conditions*, *assignements*, *opérations arithmétiques*, etc.)

Preuve de correction

- Pour prouver qu'un algorithme est correct (terminaison + résultat attendu) on va souvent utiliser la notion d'**invariant de boucle**

Invariant de boucle

- Initialisation : L'*invariant* est vrai avant la première itération de la boucle.
- Conservation : Si l'*invariant* est vrai avant une itération de la boucle, il le reste avant l'itération suivante [a].
- Terminaison : Une fois la boucle terminée, l'*invariant* implique que la solution est correcte.

a. Avant une itération veut dire avant de faire le test de la boucle

- $\simeq$ preuve par récurrence

Exemple : TriSélection

L'algorithme suivant trie un tableua L de n éléments.

Algorithme : `TriSélection`
Données : tableau L de n éléments comparables
Résultat : le tableau trié

```
1 pour i allant de 1 à n faire
2     m ← i;
3     pour j allant de i + 1 à n faire
4         si L[j] < L[m] alors
5             m ← j;
6     échanger L[i] et L[m];
7 retourner L;
```

$i = 1$	29	30	17	9	0	24
$i = 2$	0	30	17	9	29	24
$i = 3$	0	9	17	30	29	24
$i = 4$	0	9	17	30	29	24
$i = 5$	0	9	17	24	29	30

Exemple : Prouver que `TriSélection` est correct

- `TriSélection` termine ? Oui car :
  - En dehors de la boucle principale, il y a un nombre fini d'instructions (0)
  - 2ème boucle : n est constant, j est strictement croissant et la boucle se termine pour $j > n$
  - 1ère boucle : n est constant, i est strictement croissant, la boucle se termine pour $i > n$, et la 2ème boucle termine

- `TriSélection` retourne un résultat correct ?
  Invariant de boucle Inv(i) : Au début de la $i$ème itération de la 1ère boucle "pour",

(a) $trié(i)$: Les $i - 1$ premiers éléments sont triés
(b) $mins(i)$: Les $i - 1$ premiers éléments sont les plus petits

Invariants pour TriSélection

L'algorithme suivant trie un tableau T de n éléments.

Algorithme : `TriSélection`
Données : tableau L de n éléments comparables
Résultat : le tableau trié

```
1 pour i allant de 1 à n faire
2     m ← i;
3     pour j allant de i + 1 à n faire
4         si L[j] < L[m] alors
5             m ← j;
6     échanger L[i] et L[m];
7 retourner L;
```

Invariants :

Au début de l'itération i :
(a) $i-1$ 1ers éléments triés
(b) $i-1$ 1ers éléments minimums

$i=1$	29	30	17	9	0	24
$i=2$	0 (trié)	30	17	9	29	24
$i=3$	0	9 (trié)	17	30	29	24
$i=4$	0	9	17 (trié)	30	29	24
$i=5$	0	9	17	24 (trié)	29	30

Démonstration de `TriSélection` par invariant

Au début de la $i$ème itération de la 1ère boucle "pour", 2 invariants :

(a) $trié(i)$: Les $i-1$ premiers éléments sont triés
(b) $mins(i)$: Les $i-1$ premiers éléments sont les plus petits

Preuve

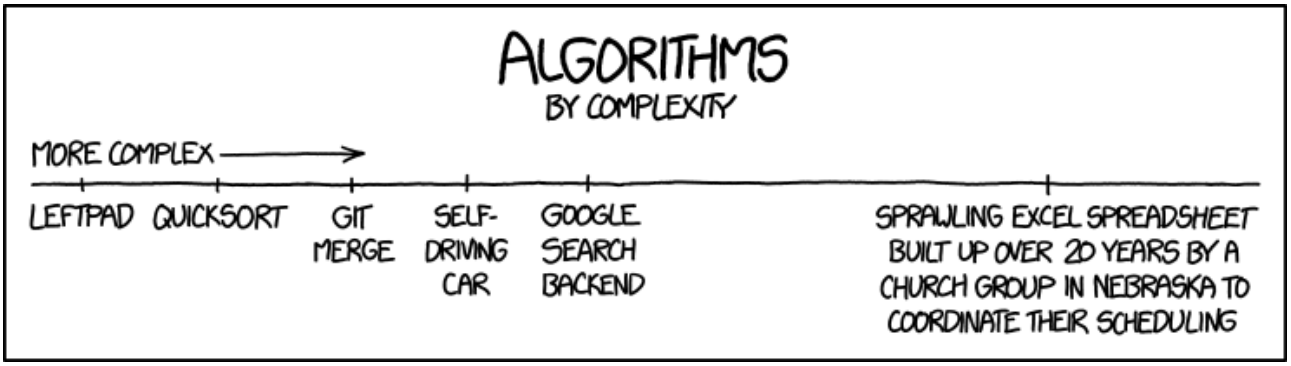
- Initialisation : $trié(i)$ et $mins(i)$ sont vrais lors de la première itération de la boucle car pour $i=1$ la liste des $i-1$ premiers éléments est vide
- Conservation : Supposons que les invariants soient vrais à l'itération i. On montre qu'ils sont vrais à l'itération $i+1$:
  - Les $i-1$ premiers éléments du tableau L ne changent pas (le seul changement est à la ligne 6 et $m \geq i$). Donc $trié(i)$ et $mins(i)$ impliquent $trié(i+1)$.
  - A la ligne 6, $L[m]$ est le plus petit élément parmi $L[i], \ldots, L[n]$ [a], et il et échangé avec $L[i]$. Donc $mins(i)$ implique $mins(i+1)$.
- Terminaison : La fin de la boucle correspond au début d'une itération $i = n+1$, Mais $trié(n+1)$ implique que L est totalement trié et donc l'algorithme est correct.

a. Il faudrait faire une autre preuve par invariant pour montrer ça ! !

Complexité Algorithmique : pourquoi ?

Pour développer des algorithmes *efficaces*, il faut pouvoir :

- Évaluer la complexité d'un algorithme ;
- Comparer deux algorithmes entre eux ;



XKCD `https://xkcd.com/`

Qu'est ce qu'un algorithme efficace ?

Critère : utilisation d'une *ressource*, e.g., le **temps** (d'exécution) ou l'espace (mémoire)

Le temps d'exécution

Le temps d'exécution

Le temps d'exécution est la durée (en secondes, minutes, etc.) nécessaire au programme pour s'éxecuter.

Mais le temps d'éxecution dépend :

- de la machine ;
- du système d'exploitation ;
- du langage ;
- de la donnée ;

On veut une méthode **indépendante de l'environnement**.

Nombre d'opérations élémentaires (I)

Opération élémentaire

Une opération élementaire est une opération qui prend un temps constant

- Même temps d'exécution quelque soit la donnée

Exemples d'opérations en temps constant

- Instructions assembleur
- Opérations arithmétiques $(+, \times, -)$, affectation, comparaisons sur les **types primitifs** (entiers, flottants, etc.)

Exemple

L'algorithme suivant calcule $n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$ (avec $0! = 1$).

Algorithme : `Factorielle`
Données : un entier n
Résultat : un entier valant $n!$

```
1 fact ← 1;
2 pour i allant de 2 à n faire
3     fact ← fact * i;
4 retourner fact;
```

	nombre	coût
initialisation :	$1\times$	1 op.
itérations :	$n\times$	1 op.
mult. + affect. :	$(n-1)\times$	2 op.
retour fonction :	$1\times$	1 op.

Nombre total d'opérations :

$$1 + n + (n-1) * 2 + 1 = 3n$$

Exemple : TriSélection

Algorithme : `TriSélection`
Données : tableau L de n éléments comparables
Résultat : le tableau trié

```
1 pour i allant de 1 à n faire
2     m ← i;
3     pour j allant de i + 1 à n faire
4         si L[j] < L[m] alors
5             m ← j;
6     échanger L[i] et L[m];
7 retourner L;
```

	nombre	coût
itérations :	$n\times$	1 op.
affectation :	$n\times$	1 op.
itérations :	$\sum_{i=1}^{n}(n-i-1)\times$	1 op.
comparaison :	$\sum_{i=1}^{n}(n-i-1)\times$	1 op.
affectation :	$n\times ?\times$	1 op.
échange :	$n\times$	3 op.

Nombre total d'opérations :

$$n(n+4) \leq n+n+2\sum_{i=1}^{n}(n-i-1)+?+3n \leq n(2n+5)$$

Nombre d'opérations élémentaires (II)

- Le nombre d'opérations dépend en général de la donnée du problème ;
  - **(a)** trier 10 entiers est plus facile que trier 1000000 entiers ?
  - **(b)** trier une liste très désordonnée est plus difficile ?
- Le nombre d'opérations est calculé en fonction de la donnée, mais comment tenir compte de toutes les valeurs possibles ?
  - Plusieurs types de complexités → pire/meilleur cas ou en moyenne.
- Quel paramètre choisir ? Est-il possible de comparer des algorithmes pour des problèmes distincts ?
  - On calcule la complexité en fonction de la **taille de la donnée** : $|x|$ est le nombre de bits de la représentation en mémoire de la donnée x
- Comment connait-on la taille $|x|$ de la donnée x ? (cf. "Représentation des Données")

Complexité en fonction de la taille de la donnée

Soit $\text{Coût}_A(x)$ la complexité de l'algorithme A sur la donnée x de taille $|x|$.

Complexité dans le meilleur des cas

$$\text{Inf}_A(|x|) = \min\{\text{Coût}_A(x) \mid x \text{ de taille } |x|\}$$

Complexité dans le pire des cas

$$\text{Sup}_A(|x|) = \max\{\text{Coût}_A(x) \mid x \text{ de taille } |x|\}$$

Complexité en moyenne

Besoin d'une probabilité $P()$ pour toutes les données de tailles n

$$\text{Moy}_A(|x|) = \sum_{x \text{ de taille } |x|} P(x) \cdot \text{Coût}_A(x)$$

Exemple (Recherche dans un tableau)

L'algo. suivant recherche l'élément e dans un tableau.

Algorithme : `RechercheElmt`
Données : un entier e et un tableau L contenant e
Résultat : l'indice i t.q. $L[i] = e$

$i \leftarrow 0$;
tant que $L[i] \neq e$ **faire**
  $i \leftarrow i + 1$;
retourner i;

Le nombre de comparaisons dépend de la donnée L :

- e est dans la case 1 $\rightarrow$ 1 comp.
- e est dans la case j $\rightarrow$ j comp.
- e est dans la case n $\rightarrow$ n comp.
  ($n = |L|$: taille de L)

meilleur : 1 comp.
pire : n comp.
moyenne : $\frac{n+1}{2}$ (voir slide suivant)

Complexité en moyenne (Recherche)

L'algo. suivant recherche l'élément e dans un tableau.

Algorithme : `RechercheElmt`
Données : un entier e et un tableau L contenant e
Résultat : l'indice i t.q. $L[i] = e$

```
i : entier;
début
    i ← 0 ;
    tant que L[i] ≠ e faire
        i ← i + 1;
    retourner i;
```

moyenne : $\frac{n+1}{2}$

Hyp. :

- distribution uniforme
- $nbOcc(e) = 1$

$\Rightarrow P(L[i] = e) = 1/n.$

On applique la formule :

$$\mathrm{Moy}_A(n) = \sum_{x \text{ de taille } n} P(x) \cdot \mathrm{Coût}_A(x)$$

$$\mathrm{Moy}_A(n) = \frac{1}{n} \times \frac{n(n+1)}{2}$$

Complexité en moyenne (TriSélection)

- $\mathrm{Inf}_{\texttt{TriSélection}}(n) = n(n+4)$, $\mathrm{Sup}_{\texttt{TriSélection}}(n) = n(2n+5)$
- Le temps de calcul $T(n)$ de `TriSélection` pour n élément est tel que :

$$c_1 \cdot (n^2 + 4n) \leq T(n) \leq c_2 \cdot (2n^2 + 5n)$$

- Les valeurs des constantes c_1 et c_2 dépendent de :
  - Le coût exact des opérations (comparaisons, affectations, etc.)
  - Le matériel (processeur, RAM, etc.)
  - Le logiciel (langage, compilateur, système d'exploitation, etc.)
- **Impossible à quantifier !**
- Les variations de c_1 et c_2 sont plus importantes que le facteur (inférieur à 2) entre $n^2 + 4n$ et $2n^2 + 5n$
- $\mathrm{Inf}_{\texttt{TriSélection}}(n) \simeq \mathrm{Moy}_{\texttt{TriSélection}}(n) \simeq \mathrm{Sup}_{\texttt{TriSélection}}(n) \simeq cn^2$

Exemple : TriRapide

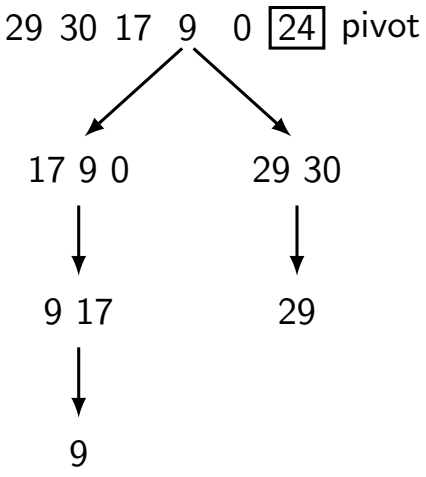
Algorithme : `TriRapide`

Données : tableau L d'elts comparables, entiers s, e
Résultat : le tableau trié entre les indices s et e

```
1  Procedure TriRapide(L, s, e)
2      si s < e alors
3          p ← Partition(L, s, e);
4          TriRapide(L, s, p − 1);
5          TriRapide(L, p + 1, e);
6  Procedure Partition(L, s, e)
7      pivot ← L[e];
8      i ← s;
9      pour j allant de s à e − 1 faire
10         si L[j] < pivot alors
11             échanger L[i] avec L[j];
12             i ← i + 1;
13     échanger L[i] avec L[e];
14     retourner i;
```

Invariants

- $L[0], \ldots, L[i-1] < pivot$
- $L[i], \ldots, L[j-1] \geq pivot$



Complexité de TriRapide

Algorithme : `TriRapide`

```
Procedure TriRapide(L, s, e)
    si s < e alors
        p ← Partition(L, s, e);
        TriRapide(L, s, p − 1);
        TriRapide(L, p + 1, e);
Procedure Partition(L, s, e)
    pivot ← L[e];
    i ← s;
    pour j allant de s à e − 1 faire
        si L[j] < pivot alors
            échanger L[i] avec L[j];
            i ← i + 1;
    échanger L[i] avec L[e];
    retourner i;
```

Opération caractéristique

Ici on compte le **nombre de comparaisons**, égal au nombre total d'opérations, **à une constante près**.

- TriRapide fait un nombre constant (disons c_1) d'opérations pour chaque comparaison
  - Au plus un échange et entre 1 et 2 incrémentation(s)

Complexité dans le pire des cas (TriRapide)

Algorithme : `TriRapide`

Procedure `TriRapide`(L, s, e)
- **si** $s < e$ **alors**
  - $p \leftarrow$ `Partition`(L, s, e);
  - `TriRapide`$(L, s, p-1)$;
  - `TriRapide`$(L, p+1, e)$;

Procedure `Partition`(L, s, e)
- $pivot \leftarrow L[e]$;
- $i \leftarrow s$;
- **pour** j *allant de* s *à* $e-1$ **faire**
  - **si** $L[j] < pivot$ **alors**
    - échanger $L[i]$ avec $L[j]$;
    - $i \leftarrow i+1$;
- échanger $L[i]$ avec $L[e]$;
- **retourner** i;

- Pire des cas : les éléments sont déja triés !
- Le pivot est comparé aux $n-1$ éléments et reste en dernière position
- `Partition` retourne toujours e
  - `Partition` $(L, 1, n)$,`Partition` $(L, 1, n-1)$,...
- Nombre total de comparaisons :

$$\sum_{i=1}^{n}(n-i) = n^2 - \sum_{i=1}^{n} i = n(n-1)/2$$

Complexité en moyenne (TriRapide)

Algorithme : `TriRapide`

Procedure `TriRapide`(L, s, e)
- **si** $s < e$ **alors**
  - $p \leftarrow$ `Partition`(L, s, e);
  - `TriRapide`$(L, s, p-1)$;
  - `TriRapide`$(L, p+1, e)$;

Procedure `Partition`(L, s, e)
- $pivot \leftarrow L[e]$;
- $i \leftarrow s$;
- **pour** j *allant de* s *à* $e-1$ **faire**
  - **si** $L[j] < pivot$ **alors**
    - échanger $L[i]$ avec $L[j]$;
    - $i \leftarrow i+1$;
- échanger $L[i]$ avec $L[e]$;
- **retourner** i;

- Deux éléments sont comparés une fois au plus
  - Si deux éléments sont comparés, un des deux est un pivot, et ils seront séparés
- On calcule l'espérance E du nombre total de comparaisons

Complexité en moyenne (TriRapide)

z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_9

- Soit la liste triée des éléments de T : $z_1 < z_2 < \ldots < z_n$
- Si on note $p(z_i, z_j)$ la probabilité que z_i et z_j soient comparés, alors l'espérance E du nombre de comparaisons est donc :

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} p(z_i, z_j)$$

- z_i et z_j sont comparés ssi un des deux est le premier pivot parmi $z_i, z_{i+1}, \ldots, z_j$
  - sinon, le pivot z_k sépare $z_i < z_k$ et $z_j > z_k$!
- Donc $p(z_i, z_j) = 2/(j - i + 1)$ (les choix de pivot sont équiprobables)

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{j=1}^{n-i} \frac{2}{j+1} \leq 2 \sum_{i=1}^{n-1} \sum_{j=1}^{n} \frac{1}{j} \simeq 2n \ln n$$

Résumons

	`TriSélection`	`TriRapide`
$\mathrm{Sup}(n)$	$c_1 n^2$	$c_2 n^2$
$\mathrm{Moy}(n)$	$c_3 n^2$	$c_4 n \ln n$

- Soient :
  - $T^s(n)$ le temps effectif de calcul pour `TriSélection` de n éléments, $\simeq \mathrm{Moy}_s(n) = c_3 n^2$
  - $T^r(n)$ le temps effectif de calcul pour `TriRapide` de n éléments, $\simeq \mathrm{Moy}_r(n) = c_4 n \ln n$
- Expérience : essayons pour $n = 100000$ et estimons $n = 300000$

$$c_3 = \frac{T^s(100000)}{100000^2} \qquad c_4 = \frac{T^r(100000)}{100000 \ln 100000}$$

et donc (pour $T^s(100000) = 1.65$ et $T^r(100000) = .006$) :

$$T^s(n) = \frac{T^s(100000)}{100000^2} n^2 \quad \text{pour } n = 300000 : \simeq 14.67$$

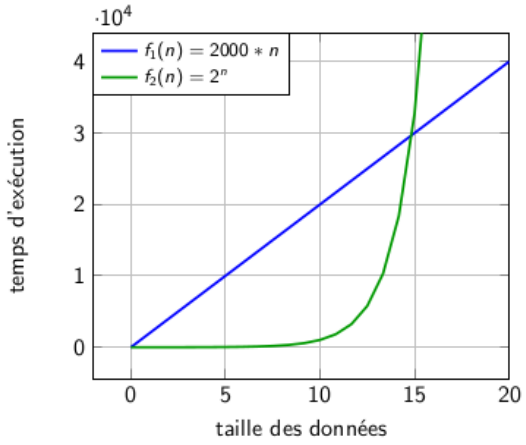
$$T^r(n) = \frac{T^r(100000)}{100000 \ln 100000} n \ln n \quad \text{pour } n = 300000 : \simeq 0.019$$

`https://www.youtube.com/watch?v=ZZuD6iUe3Pc`

Analyse Asymptotique

Complexité Algorithmique

- Vision pessimiste : la **complexité** d'un algortihme est souvent définie comme sa performance **asymptotique** dans le **pire cas**
- Que signifie **dans le pire des cas** ?
  - Parmi toutes les données x de taille n, on ne considère que celle qui maximise $\mathrm{Coût}_A(x)$
- Que signifie **asymptotique** ?
  - comportement de l'algorithme pour des données de taille *n arbitrairement grande*
  - pourquoi ?



- Soit deux algorithmes de complexités $f_1(n)$ et $f_2(n)$
- Quel algorithme préférez-vous ?
- La courbe verte semble correspondre à un algorithme plus efficace...
- ... mais seulement pour de très petites valeurs !

Ordre de grandeur : motivation

- Les calculs à effectuer pour évaluer le temps d'exécution d'un algorithme peuvent parfois être longs et pénibles ;
- De plus, le degré de précision qu'ils requièrent est souvent inutile ;
  - $n \log n + 5n \rightarrow 5n$ va devenir "négligeable" ($n >> 1000$)
  - différence entre un algorithme en $10n^3$ et $9n^3$: effacé par une accélération de $\frac{10}{9}$ de la machine
- On aura donc recours à une **approximation** de ce temps de calcul, représentée par les notations $\mathcal{O}, \Omega$ et Θ

Hypothèse simplificatrice

On ne s'intéresse qu'aux fonctions **asymptotiquement positives** (positives pour tout $n \geq n_0$)

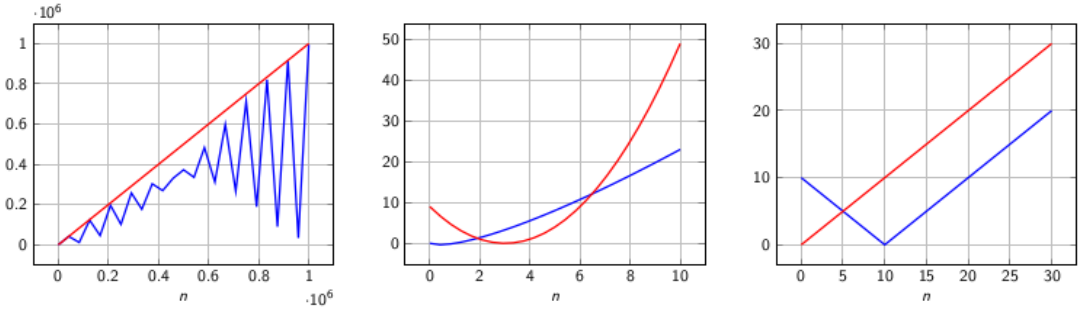
Notation $\mathcal{O}$: définition

$\mathcal{O}(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

$$\exists n_0 \in \mathbb{N},\ \exists c \in \mathbb{R},\ \forall n \geq n_0 \ : \qquad f(n) \leq c \times g(n)$$

Borne supérieure : $f(n) \in \mathcal{O}(g(n))$ **s'il existe une constante** c**, et un seuil à partir duquel** $f(n)$ est inférieure à $g(n)$, à un facteur c près ;

Exemple : $f(n) \in \mathcal{O}(g(n))$



Notation $\mathcal{O}$: preuve

$\mathcal{O}(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

$$\exists n_0 \in \mathbb{N},\ \exists c \in \mathbb{R},\ \forall n \geq n_0\ :\quad f(n) \leq c \times g(n)$$

Prouver que $f(n) \in \mathcal{O}(g(n))$: jeux contre un perfide adversaire $\forall$

Tour du joueur $\exists$ objectif : $f(n) \leq cg(n)$ choisit c et n_0

Tour du joueur $\forall$ objectif : $f(n) > cg(n)$ choisit $n \geq n_0$

Arbitre détermine le gagnant : $f(n) \leq cg(n)$

Notation $\mathcal{O}$: exemple

$\mathcal{O}(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

$$\exists n_0 \in \mathbb{N},\ \exists c \in \mathbb{R},\ \forall n \geq n_0\ :\quad f(n) \leq c \times g(n)$$

Jeux : **prouver** que la fonction $f_2(n) = 6n^2 + 2n - 8$ est en $\mathcal{O}(n^2)$:

Tour du joueur $\exists$ choisit $c = 6$ et $n_0 = 0$

Tour du joueur $\forall$ choisit $n = 5$

Arbitre $6 \times 5^2 + 2 \times 5 - 8 > 6 \times 5^2$

Notation $\mathcal{O}$: exemple

$\mathcal{O}(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

$$\exists n_0 \in \mathbb{N},\ \exists c \in \mathbb{R},\ \forall n \geq n_0\ :\quad f(n) \leq c \times g(n)$$

Jeux : **prouver** que la fonction $f_2(n) = 6n^2 + 2n - 8$ est en $\mathcal{O}(n^2)$:

Tour du joueur $\exists$ objectif : $f(n) \leq cg(n)$ choisit $c = 7$ et n_0

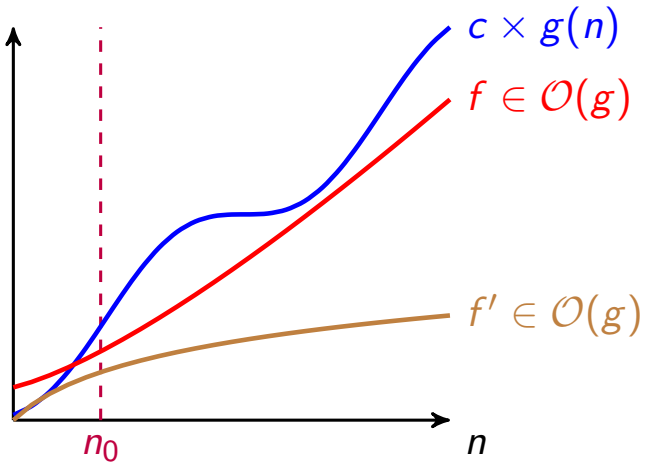
Tour du joueur $\forall$ objectif : $f(n) > cg(n)$

Arbitre $n^2 - 2n + 8 = 0$ n'a pas de solution

Focus sur $\mathcal{O}$

$\mathcal{O}(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

$$\exists n_0 \in \mathbb{N},\ \exists c \in \mathbb{R}^{+*},\ \forall n \geq n_0\ :\quad f(n) \leq c \times g(n)$$



Exercice : $2n^2$ est-il en $\mathcal{O}(n^2)$? Pareil pour $2n$.

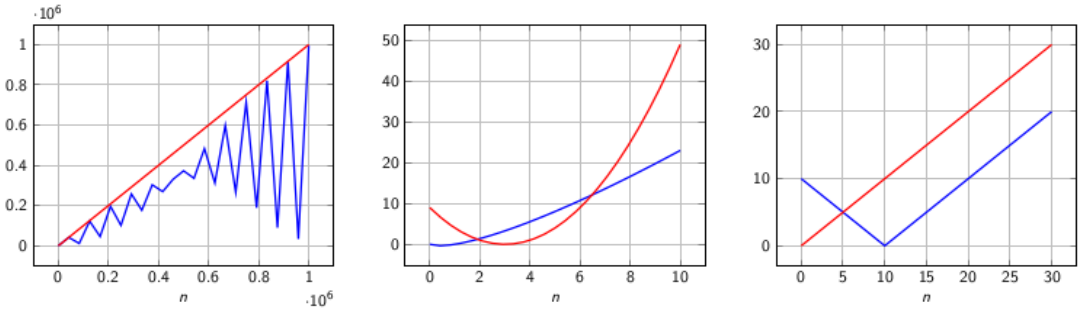
Notation Ω : définition

$\Omega(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

$$\exists n_0 \in \mathbb{N},\ \exists c \in \mathbb{R}^{+*},\ \forall n \geq n_0\ :\qquad f(n) \geq c \times g(n)$$

Borne inférieure : $f(n) \in \Omega(g(n))$ s'il existe un seuil à partir duquel $f(n)$ est supérieure à $g(n)$, à une constante multiplicative près ;

Exemple : $g(n) \in \Omega(f(n))$



Notation Θ : définition

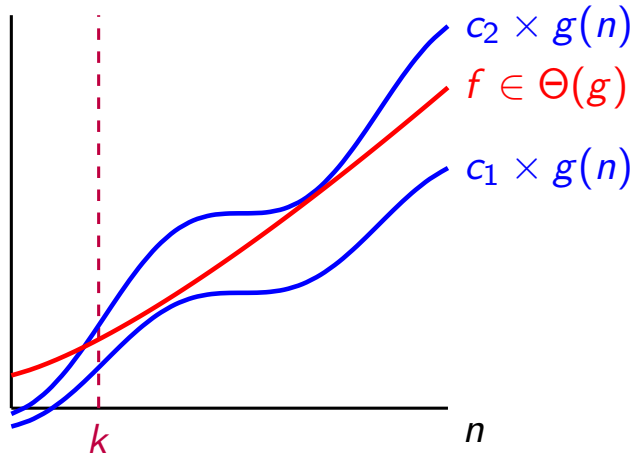
$\Theta(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

$$\exists c_1, c_2 \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}, \forall n > n_0, c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$$

Borne supérieure et inférieure : $\Theta(g(n)) = \Omega(g(n)) \cap \mathcal{O}(g(n))$; $f(n)$ est en $\Theta(g(n))$ si elle est **prise en sandwich** entre $c_1 g(n)$ et $c_2 g(n)$;

$f(n)$ est en $\Theta(g(n))$ si :

$$\exists c_1, c_2 \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}, \forall n > n_0, \qquad c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$$



Exercice : $2n^2$ est-il en $\Theta(n^2)$? Pareil pour $2n$.

Notation asymptotique d'une fonction

- Quelle est la borne asymptotique de $f(n)$?

Notation asymptotique (de l'expression fermée) d'une fonction

Les mêmes simplifications pour $\mathcal{O}, \Omega$ et Θ :

- **on ne retient que les termes dominants**
- **on supprime les constantes multiplicatives**

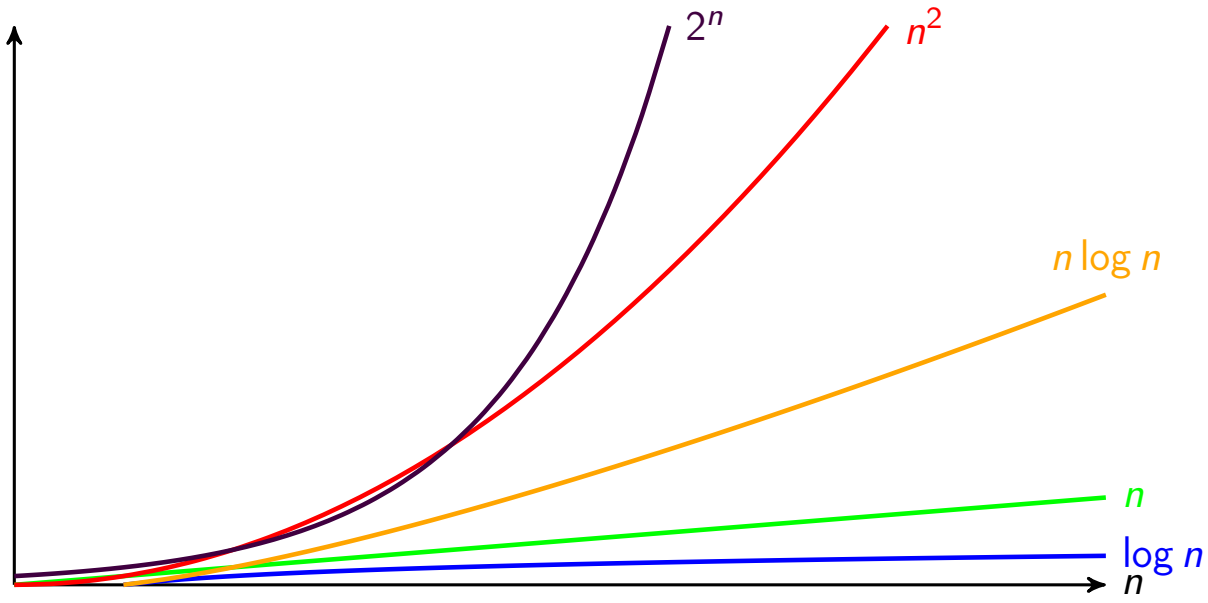
Exemple

Soit $g(n) = 4n^3 - 5n^2 + 2n + 3$;

1. on ne retient que le terme de plus haut degré : $4n^3$ (pour n assez grand le terme en n^3 "domine" les autres, en choisissant bien c_1, c_2, on peut avoir $c_1 n^3 \leq g(n) \leq c_2 n^3$)
2. on supprime les constantes multiplicatives : n^3 (on peut la choisir !)

et on a donc $g(n) \in \Theta(n^3)$

Relation des principaux ordres de grandeur



Indépendant de la taille de la donnée : $\mathcal{O}(1)/\Theta(1)$

Vocabulaire

Un algorithme dont la donnée est de taille $|x| = n$ est dit :

- **Constant** si sa complexité est en $\mathcal{O}(1)$
- **Logarithmique** si sa complexité est en $\Theta(\log n)$
- **Linéaire** si sa complexité est en $\Theta(n)$
- **Quadratique** si sa complexité est en $\Theta(n^2)$
- **Polynomial** si sa complexité est en $\mathcal{O}(n^{\mathcal{O}(1)})$
- **Exponentiel** si sa complexité est en $\Theta(c^{\Theta(n)})$ pour une constante $c > 1$

Quelques remarques (à prouver comme exercice)

- $f \in \mathcal{O}(g)$ ssi $g \in \Omega(f)$
- $f \in \Theta(g)$ ssi $g \in \Theta(f)$

- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$ (constante) alors $f \in \Theta(g)$ (et donc $g \in \Theta(f)$)
- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ alors $f \in \mathcal{O}(g)$ et $f \notin \Omega(g)$
- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ alors $f \in \Omega(g)$ et $f \notin \mathcal{O}(g)$

Règle de l'Hôpital

f et g deux fonctions dérivables t.q. $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$, alors :

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \text{ si cette limite existe.}$$

f' (respectivement g') représente la dévirée de f (respectivement g)

Règles de calculs : combinaisons des complexités

- Les instructions de base prennent un temps constant, noté $\mathcal{O}(1)$;
- On additionne les complexités d'opérations en séquence :

$$\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$$

- Branchements conditionnels : max (analyse dans le *pire des cas*)
- L'ordre de grandeur maximum est égal à la somme des ordres de grandeur :

$$\max(\Theta(f(n)), \Theta(g(n))) = \Theta(f(n)) + \Theta(g(n))$$

Exemple

si *<condition>* **alors**
  #instructions (1); — $\Theta(g(n))$, $\Theta(f_1(n))$
sinon
  #instructions (2); — $\Theta(f_2(n))$

$$= \Theta(g(n) + f_1(n) + f_2(n))$$

Règles de calculs : combinaison des complexité

- Dans les boucles, on multiplie la complexité du corps de la boucle par le nombre d'itérations ;

- Calcul de la complexité d'une boucle `while` :

Exemple

en supposant qu'on a $\Theta(h(n))$ itérations

tant que *<condition>* **faire**
  #instructions ;

$$\left.\begin{array}{l}\Theta(g(n))\\ \Theta(f(n))\end{array}\right\} = \Theta(h(n) \times (g(n) + f(n)))$$

Règles de calculs : combinaison des complexité

- Dans les boucles, on multiplie la complexité du corps de la boucle par le nombre d'itérations ;

- Calcul de la complexité d'une boucle `for` :

Exemple

pour *i allant de a à b* **faire**
  #instructions ;

$$\left.\begin{array}{l}\Theta(f(n))\end{array}\right\} = \Theta((b - a + 1) \times f(n))$$

Calcul de la complexité asymptotique d'un algorithme

- Pour calculer la complexité d'un algorithme :

1. on calcule la complexité de chaque "partie" de l'algorithme ;
2. on combine ces complexités conformément aux règles qu'on vient de voir ;
3. on simplifie le résultat grâce aux règles de simplifications qu'on a vues ;
   - ★ élimination des constantes, et
   - ★ conservation du (des) termes dominants

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- Reprenons le calcul de la factorielle, qui nécessitait $3n$ opérations :

Algorithme : Factorielle(n)
Données : un entier n
Résultat : un entier valant $n!$

```
1 fact, i : entier;
2 début
3     fact ← 2;
4     pour i allant de 3 à n faire
5         fact ← fact ∗ i;
6     retourner fact;
```

	nombre	coût
initialisation :	$\Theta(1)\times$	$\Theta(1)$
itérations :	$\Theta(n)\times$	$\Theta(1)$
mult. + affect. :	$\Theta(n)\times$	$\Theta(1)$
retour fonction :	$\Theta(1)\times$	$\Theta(1)$

Nombre total d'opérations :

$$\Theta(1) + \Theta(n) * \Theta(1) + \Theta(n) * \Theta(1) + \Theta(1) = \Theta(n)$$

Exemple : TriSélection

Algorithme : `TriSélection`
Données : tableau L de n éléments comparables
Résultat : le tableau trié

			nombre	coût
1	**pour** i *allant de* 1 *à* n **faire**	itérations :	$n\times$	1 op.
2	$m \leftarrow i$;	affectation :	$n\times$	1 op.
3	**pour** j *allant de* $i+1$ *à* n **faire**	itérations :	$\sum_{i=1}^{n}(n-i-1)\times$	1 op.
4	**si** $L[j] < L[m]$ **alors**	comparaison :	$\sum_{i=1}^{n}(n-i-1)\times$	1 op.
5	$m \leftarrow j$;	affectation :	$n\times?\times$	1 op.
6	échanger $L[i]$ et $L[m]$;	échange :	$n\times$	3 op.
7	**retourner** L;			

Séries arithmétiques

$$\sum_{i=1}^{n}(n-i-1) = n^2 - n - \sum_{i=1}^{n} i = n^2 - n - (1+2+3+\cdots+n) = n^2 - n - \frac{1}{2}n(n+1)$$

Nombre total d'opérations : $\Theta(n^2) = \Theta(|T|^2)$

Algorithmes Récursifs

Force brute

- L'approche "force brute" est une méthode de conception d'algorithmes qui se base simplement sur une énumération exhaustive de toutes les configurations possibles de la solution recherchée
- **Exemple** : Un algorithme de tri de type "force brute " parcourt toutes les permutations possibles de la liste jusqu'à ce qu'il trouve la permutation ordonnée.
- Cette méthode est souvent inefficace car elle se repose sur l'énumération complète d'un espace de recherche
- **Exemple** (algorithme de tri de type "force brute") : le nombre de permutations possible est $n!$ donc la complexité d'un tel algorithme est $\Theta(n!)$

Diviser pour régner (Divide and conquer)

- "Diviser pour régner" est une méthode de conception d'algorithmes qui se base sur une "conception par décomposition" :
  - Diviser le problème en sous problèmes
  - Résoudre les sous problèmes par des appels *récursifs*
  - Combiner les résultats des sous problèmes pour résoudre le problème initial
- Cas idéal : le problème est décomposable en sous-problèmes *indépendents*
  - Dans ce cas, combiner les résultats est trivial
  - Parfois, les sous-problèmes sont seulement plus "faiblement" liés, et combiner les résultats peut-être complexe

Exemple avec un algorithme de tri

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
  retourner L;
sinon
  $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
  $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
  $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
  retourner `Fusion`(L_l, L_r);

Algorithme : `Fusion` (L_1, L_2)

Données : deux listes triées L_1 et L_2
Résultat : une liste L triée contenant les éléments de L_1 et de L_2
L : liste vide;
$i, j, k \leftarrow 1$;
tant que $k < |L|$ **faire**
  si $i > |L_1|$ *ou* $(j \leq |L_2|$ *et* $L_1[i] > L_2[j])$ **alors**
    insérer $L_2[j]$ à la fin de L;
    $j \leftarrow j + 1$;
  sinon
    insérer $L_1[i]$ à la fin de L;
    $i \leftarrow i + 1$;
  $k \leftarrow k + 1$;
retourner L

Tri Fusion

Algorithme : `TriFusion` (L)

Données : une liste L
Résultat : la liste L triée
mil : entier;
si $|L| \leq 1$ **alors**
  retourner L;
sinon
  $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
  $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
  $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
  retourner `Fusion`(L_l, L_r);

- Déroulement de l'algorithme avec la liste $\langle 6, 2, 1, 8, 5, 4, 3, 7 \rangle$:
  - Division : $\langle 6, 2, 1, 8 \rangle$ $\langle 5, 4, 3, 7 \rangle$
  - Division : $\langle 6, 2 \rangle$ $\langle 1, 8 \rangle$
  - Division : $\langle 6 \rangle$ $\langle 2 \rangle$
  - Regroupement : $\langle 2, 6 \rangle$
  - Division : $\langle 1 \rangle \langle 8 \rangle$
  - Regroupement : $\langle 1, 8 \rangle$
  - Regroupement : $\langle 1, 2, 6, 8 \rangle$
  - Division : $\langle 5, 4 \rangle$ $\langle 3, 7 \rangle$
  - Division : $\langle 5 \rangle$ $\langle 4 \rangle$
  - Regroupement : $\langle 4, 5 \rangle$
  - Division : $\langle 3 \rangle \langle 7 \rangle$
  - Regroupement : $\langle 3, 7 \rangle$
  - Regroupement : $\langle 3, 4, 5, 7 \rangle$
  - Regroupement : $\langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$

Preuve de correction

- Terminaison :
  - `TriFusion` ne s'appelle lui même que 2 fois
  - A chaque appel récursif, la taille de la liste $|L|$ est strictement plus petite
  - Il n'y a pas d'appel récursif pour $|L| \leq 1 \implies$ **nombre total d'appels récursifs est fini**
- Correction (`TriFusion`(L) est triée, par récurrence sur $|L|$) :
  - Pour $|L| \leq 1$, la liste est déjà triée
  - `TriFusion`(L) triée si $|L| \leq n$; est-ce que `TriFusion`(L) est triée si $|L| \leq n+1$?
  - `TriFusion`(L) renvoie `Fusion`(`TriFusion`$(L[: mil])$, `TriFusion`$(L[mil :])$)
  - `TriFusion`$(L[: mil])$ et `TriFusion`$(L[mil :])$ sont triées par l'hypothèse de récurrence puisque $|L[: mil]| \leq n$ et $|L[mil :]| \leq n$
  - $\Rightarrow$ les préconditions de `Fusion` sont respectées, montront qu'il est correct, par invariants :
    - ★ L est triée
    - ★ $i = |L_1| + 1$ ou $k = 1$ ou $L[k-1] \leq L_1[i]$
    - ★ $j = |L_2| + 1$ ou $k = 1$ ou $L[k-1] \leq L_2[i]$

Analyse de la complexité d'un algorithme récursif

La structure d'un algorithme récursif `AlgoRec`(x) est :

si *condition d'arrêt* **alors**
  retourner *solution triviale*;
sinon
  retourner `Fusion(AlgoRec(`$p(x,1)$`),AlgoRec(`$p(x,2)$`),...,AlgoRec(`$p(x,a)$`))`;

- p "coupe" la donnée x (de taille $|x| = n$) en a morceaux de taille $g(n)$
- `Fusion` "recolle" les morceaux en $h(n)$

Forme récursive de la complexité $T(n) = \begin{cases} \Theta(1) & \text{si } \dots \\ aT(g(n)) + h(n) & \text{sinon} \end{cases}$

On veut trouver une *formule* de forme close

$$T(n) \in \mathcal{O}(f(n))$$

Exemple : `TriFusion`

- `TriFusion` "coupe" la donnée L (de taille $|L| = n$) en 2 morceaux de taille $\frac{n}{2}$
- `Fusion` "recolle" les morceaux en $\Theta(n)$

Forme récursive de la complexité :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ 2T(\frac{n}{2})T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n & \text{sinon} \end{cases}$$

Forme fermée de la complexité :

$$\exists c, n_0 \forall n > n_0 \quad T(n) \leq cn \log n$$
$$\text{càd} \quad T(n) \in \mathcal{O}(n \log n)$$

Méthode par substitution

- Il faut avoir une intuition sur la forme de la solution (`TriFusion` : $\mathcal{O}(n \log n)$)
- On veut montrer que $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \in \mathcal{O}(n \log n)$
- On montre par récurrence qu'il existe $f(n)$ t.q. $\forall n \; T(n) \leq f(n)$
  - On va en déduire **a posteriori** que $T(n) \in \mathcal{O}(f(n))$

Attention !

- L'hypothèse de récurrence est $T(n) \leq f(n)$, et **non** $T(n) \in \mathcal{O}(f(n))$
- L'hypothèse de récurrence "pour tout $n \leq k$, $T(n) \in \mathcal{O}(f(n))$" ne veut pas dire grand chose puisque la notation $\mathcal{O}$ est définie pour n arbitrairement grand : **on remplace tous les termes en $\mathcal{O}, \Omega, \Theta$ par une fonction élément de l'ensemble**

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- Il faut montrer que la formule est vraie pour les conditions limites de la récurrence pour des données de petite taille, i.e. $n = 1$
- **Problème :** c'est faux pour $n = 1$ car $c \times 1 \times \log 1 = 0 < T(1) = 1$;
- Mais on cherche à montrer la complexité pour des données de grande taille : $n \geq n_0$ et on a le choix pour $n_0 \implies$ vérifier pour $T(2)$ (et $T(3)$)
- On peut aussi borner par $f(n) = cn \log n + b$ puisque $cn \log n + b \in \mathcal{O}(n \log n)$
  - Ou même $f(n) = cn \log n + an + b$

Méthode par substitution (condition aux limites)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On vérifie que la formule tient pour $T(2)$ et $T(3)$

$$T(2) = 2T(2/2) + 2$$

$$T(2) = 2T(1) + 2$$

$$T(2) = 2 * 1 + 2 = 4 \leq 2c \log 2 = 2c$$

$$T(2) = 4 \leq 2c$$

$$c \geq 2$$

- On fait la même chose pour $T(3)$...
- ... et on obtient que c doit être ≥ 2.

Méthode par substitution (Récurrence)

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq cn \log n$$

- On suppose que $T(x) \leq cx \log x$ est vrai pour tout $2 \leq x \leq n-1$; En particulier :

$$T\left(\tfrac{n}{2}\right) \leq c\tfrac{n}{2} \log \tfrac{n}{2}$$

- On vérifie que c'est aussi le cas pour $x = n$ en substituant la formule pour $T(x)$ dans son expression récursive :

$$\begin{array}{rcll} T(n) & = & 2T\left(\frac{n}{2}\right) + n & \\ & \leq & 2c\frac{n}{2}\log(\frac{n}{2}) + n & \\ & \leq & cn \log \frac{n}{2} + n & \\ & = & cn \log n - cn \log 2 + n & \\ & = & cn \log n - cn + n & \\ & \leq & cn \log n & (\text{pour } c \geq 1) \end{array}$$

- On a pris $c \geq 2$, pour satisfaire les conditions initiales $T(2)$ et $T(3)$

Diviser pour régner : `TriFusion`

Exemple

Algorithme : `TriFusion` (L)
Données : une liste L
Résultat : la liste L triée
si $|L| \leq 1$ **alors**
  retourner L;
sinon
  $mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;
  $L_l \leftarrow$ `TriFusion`$(L[:mil])$;
  $L_r \leftarrow$ `TriFusion`$(L[mil:])$;
  retourner `Fusion`(L_l, L_r);

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2aT\left(\frac{n}{2b}\right) + \Theta(n^{1d}) & \text{si } n > 1 \end{cases}$$

- Trifusion : $a = 2$, $b = 2$, $d = 1$

- L'algorithme découpe la donnée en a sous-problèmes de taille $\frac{n}{b}$, les résout récursivement et rassemble les réponses en $\Theta(n^d)$

Diviser pour régner : RechBin

Exemple

Algorithme : RechBin (L)
Données : tableau trié L contenant e
Résultat : la position de e dans L

$m \leftarrow \left\lfloor \frac{|L|}{2} \right\rfloor$;
si $L[m] = e$ **alors**
  retourner m
sinon si $L[m] < e$ **alors**
  retourner RechBin($L[m+1:]$)
sinon
  retourner RechBin($L[:m]$)

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 1aT\left(\frac{n}{2b}\right) + \Theta(n^{0d}) & \text{si } n > 1 \end{cases}$$

- Recherche Binaire : $a = 1$, $b = 2$, $d = 0$

- L'algorithme découpe la donnée en a sous-problèmes de taille $\frac{n}{b}$, les résout récursivement et rassemble les réponses en $\Theta(n^d)$

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ — complexité dominée par le coût de fusion
2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ — complexité dominée par le coût du sous-problème
3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ — pas de domination

- Tri fusion :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

- $a = 2$, $b = 2$, $d = 1$, $\log_2 2 = 1 = d$

  On est donc dans le 3ème cas et la complexité en $\Theta(n \log n)$

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

  1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ — complexité dominée par le coût de fusion
  2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ — complexité dominée par le coût du sous-problème
  3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ — pas de domination

- Recherche binaire :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T(\frac{n}{2}) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $a = 1,\ b = 2,\ d = 0,\ \log_2 1 = 0 = d$

  On est donc dans le cas 3 et la complexité en $\Theta(\log n)$

Programmation Dynamique

Contexte & plan du chapitre

- Nous allons découvrir une nouvelle méthode de conception d'algorithme : la programmation dynamique
- Nous allons introduire cette méthode à travers le problème de multiplication de matrices
- Nous allons résoudre ce problème avec 3 approches différentes :

1. Approche force brute
2. Algorithme récursif
3. Programmation dynamique

Le problème de multiplication de matrices

$$\begin{array}{|c|c|c|} \hline l_{1,1} & l_{1,2} & l_{1,3} \\ \hline l_{2,1} & l_{2,2} & l_{2,3} \\ \hline \end{array} \times \begin{array}{|c|c|c|c|} \hline c_{11} & c_{21} & c_{31} & c_{41} \\ \hline c_{12} & c_{22} & c_{32} & c_{42} \\ \hline c_{13} & c_{23} & c_{33} & c_{43} \\ \hline \end{array} =$$

.	.	.	.
.	.	$l_{2,1}c_{31} + l_{2,2}c_{32} + l_{2,3}c_{33}$	.

- Cas général : A_1 de taille $t_0 \times t_1$ et A_2 de taille $t_1 \times t_2$, il y a $t_0 \times t_1 \times t_2$ multiplications à faire.

Associativité de la multiplication

- 3 matrices A_1, A_2, A_3 de dimensions (10×4), (4×100), (100×25), respectivement.
- On veut calculer $A_1 * A_2 * A_3$
- Il y a deux façons (la multiplication est associative) :
  1. $A_1 * A_2 * A_3 = ((A_1 * A_2) * A_3)$
  2. $A_1 * A_2 * A_3 = (A_1 * (A_2 * A_3))$
- Nombre de multiplications nécessaires :
  1. $((A_1 * A_2) * A_3)$:
     1. $10 * 4 * 100 = 4000$ multiplications pour calculer $M = A_1 * A_2$ de dimension 10×100
     2. $10 * 100 * 25 = 25000$ multiplications pour calculer $M * A_3$
     3. total : 29000 multiplications
  2. $(A_1 * (A_2 * A_3))$
     1. $4 * 100 * 25 = 10000$ multiplications pour calculer $N = A_2 * A_3$ de dimension 4×25
     2. $10 * 4 * 25 = 1000$ multiplications pour calculer $A_1 * N$
     3. total : 11000 multiplications
- **Choisir** $(A_1 * (A_2 * A_3))$ **!!**

Problème de multiplication de matrices

- Soit A_1, A_2, $\ldots A_n$ n matrices
- A_i de taille $t_{i-1} * t_i$
- On veut trouver un parenthésage de $A_1 \times A_2 \times \ldots A_n$ qui minimise le nombre de multiplications pour calculer le produit $A_1 \times A_2 \times \ldots A_n$

Force brute

- Énumérer tous les partenthésages possibles
- Calculer le coût de chaque parenthésage
- Choisir la solution avec la valeur mimimale

Combien de parenthésages possibles ?

- 3 matrices $A_1 * A_2 * A_3$?
  - $(A_1 * A_2) * A_3$
  - $A_1 * (A_2 * A_3)$
  - 2 possibilités
- Pour 4 matrices $\rightarrow$ 5 possibilités
- Pour 5 matrices $\rightarrow$ 14 possibilités
- Pour 10 matrices ? $\rightarrow$ 4862 possibilités
- Pour 50 matrices $\rightarrow 5 \times 10^{21}$ possibilités
- Cas général : pour n matrices, il y a $C(n-1)$ parenthésages possibles, où $C(n)$ est le nombre de Catalan $C(n) = \frac{1}{n+1} \times \binom{2n}{n}$ (à étudier en détail en TD)
- $C(n) \in \Omega(\frac{4^n}{n^{1.5}})$
- C'est exponentiel ! !
- **La complexité de l'approche force brute est $\Omega(\frac{4^n}{(n-1)^{1.5}})$**
- **Très inefficace**

Une première solution récursive (diviser pour régner)

- Pour chaque solution, il faut découper la séquence $A_1, ..A_n$ en deux sous séquences $A_1..A_k$ et $A_{k+1}..A_n$ (le calcul sera $(A_1 \times ..A_k) * (A_{k+1} \times ..A_n)$)
- Soit $m[i][j]$ le coût minimal pour la séquence $A_i \dots A_j$ (avec $i < j$)
- Solution du problème est $m[1][n]$
- Pour le parenthésage $(A_1..A_k) \times (A_{k+1}..A_n)$, le coût est

$$cout = m[1][k] + m[k+1][n] + t_0 \times t_k \times t_n$$

$$\Longrightarrow m[1][n] = \min_{k \in [1,n-1]} \{m[1][k] + m[k+1][n] + t_0 \times t_k \times t_n\} :$$

- Cas général de la récursivité :
  1. Si $i < j$, $m[i][j] = \min_{k \in [i,j-1]} \{m[i][k] + m[k+1][j] + t_{i-1} \times t_k \times t_j\}$:
  2. $m[i][i] = 0$

Algorithme récursif

Algorithme : `cout_recursif` $(A_1 \dots A_n)$

Données : Une liste de matrices $A_1, A_2, \dots A_n$ de tailles $t_0 \times t_1$, $t_1 \times t_2$, $\dots t_{n-1} \times t_n$
Résultat : nombre minimal de multiplications pour calculer $A_1 \times \dots \times A_n$

```
début
    c, tmp : entier;
    c ← ∞ ;
    si n = 1 alors
        retourner 0;
    sinon
        pour k de 1 à n − 1 faire
            tmp ← cout_recursif(A_1 ... A_k) + cout_recursif(A_{k+1}, A_n) + t_0 × t_k × t_n ;
            si tmp < c alors
                c ← tmp ;
        retourner c ;
```

Complexité de `cout_recursif`

$$T(n) = \begin{cases} c_1 & \text{si } n = 1 \\ \sum_{k=1}^{n-1}(T(k) + T(n-k) + c_2) & \text{si } n > 1 \end{cases}$$

Avec c_1 et c_2 deux constantes

- Comment calculer la complexité sous une forme non-récursive ?
- Le théorème maître ne s'applique pas !
- On va essayer d'utiliser la méthode par substitution. D'abord on simplifie la récursion
- Pour $n > 1$: $T(n) = 2\sum_{k=1}^{n-1} T(k) + c_2 n$
- Pour $n > 1$: $T(n) - T(n-1) = 2T(n-1) + c_2$
- Donc

$$T(n) = \begin{cases} c_1 & \text{si } n = 1 \\ 3T(n-1) + c_2 & \text{si } n > 1 \end{cases}$$

Complexité de `cout_recursif`

-

$$T(n) = \begin{cases} c_1 & \text{si } n = 1 \\ 3T(n-1) + c_2 & \text{si } n > 1 \end{cases}$$

- On applique la méthode par substitution
- $T(1) = c_1$
- $T(2) = 3T(1) + c_2 = 3c_1 + c_2$
- $T(3) = 3T(2) + c_2 = 9c_1 + 4c_2$
- $T(4) = 3T(4) + c_2 = 27c_1 + 13c_2$
- ..
- $T(n) = c_1 \times 3^{n-1} + c_2 \times \sum_{i=0}^{n-2} 3^i$ (On peut le prouver par récurrence)
- Par conséquent : $T(n) \in \Theta(3^n)$

Analyse de l'algorithme récursif par rapport à l'approche force brute

- La complexité de l'approche force brute est en $\Omega(\frac{4^n}{n^{1.5}})$ et la complexité de l'algorithme récursif est en $\Theta(3^n)$. Que choisir ?
- Soit $f(n) = \frac{4^n}{n^{1.5}}$ et $g(n) = 3^n$. On veut comparer asymptotiquement f et g.
- Rappel :
  - Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$ (constante) alors $f \in \Theta(g)$ (et donc $g \in \Theta(f)$)
  - Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ alors $f \in \mathcal{O}(g)$ et $f \notin \Omega(g)$
  - Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ alors $f \in \Omega(g)$ et $f \notin \mathcal{O}(g)$

Règle de l'Hôpital

f et g deux fonctions dérivables t.q. $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$, alors :

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \text{ si cette limite existe.}$$

f' (respectivement g') représente la dévirée de f (respectivement g)

Analyse de l'algorithme récursif par rapport à l'approche force brute

- Rappel : la complexité de l'approche force brute est en $\Omega(\frac{4^n}{n^{1.5}})$ et la complexité de l'algorithme récursif est en $\Theta(3^n)$. Que choisir ?
- On va comparer $\frac{4^n}{n^{1.5}}$ et $3^n \implies$ on calcule $\lim_{n\to\infty} \frac{\frac{4^n}{n^{1.5}}}{3^n}$?
  - $\frac{\frac{4^n}{n^{1.5}}}{3^n} = \frac{\frac{4}{3}^n}{n^{1.5}}$
  - On utilise la règle de l'Hôpital :
    $\lim_{n\to\infty} \frac{(\frac{4}{3}^n)'}{(n^{1.5})'} = \lim_{n\to\infty} \frac{ln(\frac{4}{3})\frac{4}{3}^n}{1.5n^{0.5}} = \lim_{n\to\infty} \frac{(ln(\frac{4}{3})\frac{4}{3}^n)'}{(1.5n^{0.5})'} = \lim_{n\to\infty} \frac{ln(\frac{4}{3})*ln(\frac{4}{3})\frac{4}{3}^n}{1.5*0.5*n^{-0.5}} = \infty$
  - Donc $\lim_{n\to\infty} \frac{\frac{4^n}{n^{1.5}}}{3^n} = \infty$ et par conséquent : $\frac{4^n}{n^{1.5}} \in \Omega(3^n)$ et $\frac{4^n}{n^{1.5}} \notin \mathcal{O}(3^n)$
  - L'algorithme récursif est meilleur que l'approche force brute

Calcul redondant

Algorithme : `cout_recursif` $(A_1 \dots A_n)$

Données : Une liste de matrices $A_1, A_2, \dots A_n$ de tailles $t_0 \times t_1$, $t_1 \times t_2$, $\dots t_{n-1} \times t_n$
Résultat : nombre minimal de multiplications pour calculer $A_1 \times \dots \times A_n$

```
début
    c, tmp : entier;
    c ← ∞ ;
    si n = 1 alors
        retourner 0;
    sinon
        pour k de 1 à n − 1 faire
            tmp ← cout_recursif(A_1 ... A_k) + cout_recursif(A_{k+1}, A_n) + t_0 × t_k × t_n ;
            si tmp < c alors
                c ← tmp ;
        retourner c ;
```

- `cout_recursif` (A_i, A_j) **est appelé plusieurs fois** (e.g. pour $n = 5$, on appelle `cout_recursif`$(3, 5)$ quand $k = 1$, $k = 2$, et $k = 3$).
- L'algorithme récursif fait beaucoup de calculs redondants ! on peut l'améliorer
- $\Longrightarrow$ **Programmation dynamique**

Programmation dynamique

- Méthode de conception de type "diviser pour régner"
- Souvent utilisée avec des problèmes d'optimisation (on cherche une solution qui minimise ou maximise un critère)
- Assure que chaque sous-problème est traité une seule fois afin d'éviter le problème de redondance.
- Idée clé :
  - mémoriser les solutions des sous-problèmes (dans un tableau/matrice par exemple)
  - approche ascendante : Soit $P(n)$ le problème à résoudre de taille n. Pour tout $k < i$, si $P(i)$ dépend de $P(k)$, alors résoudre $P(k)$ avant de résoudre $P(i)$

Programmation dynamique pour la multiplication des matrices

- Il faut s'assurer que l'algorithme calcule le coût de chaque séquence de longueur l avant de calculer le coût d'une séquence de taille $l+1$
- Donc l'algorithme doit calculer (dans l'ordre)
  1. Le coût des séquences de taille 2 : $m[1][2], m[2][3], \ldots m[n-1][n]$
  2. Puis le coût des séquences de taille 3 : $m[1][3], m[2][4], \ldots m[n-2][n]$
  3. ...
  4. Finalement le coût de la séquence de taille n (coût de la solution optimale) : $m[1][n]$

Programmation dynamique pour la multiplication des matrices

Algorithme : `CoutMultiplication_ProgDynamique`

Données : Une liste de matrices $A_1, A_2, \ldots A_n$ de tailles $t_0 \times t_1$, $t_1 \times t_2$, $\ldots t_{n-1} \times t_n$
Résultat : nombre minimal de multiplications pour calculer $A_1 \times \ldots \times A_n$
l, tmp : entier;
$m[][]$: matrice de taille $n \times n$;
% $m[i][j]$ est le coût minimal pour la séquence $A_i, \ldots A_j$ $(i < j)$;
% Initialisation ;
pour i *de* 1 *à* n **faire**
  $m[i][i] = 0$;
pour l *de* 2 *à* n **faire**
  pour i *de* 1 *à* $n - l + 1$ **faire**
    %On va calculer le coût minimal de la séquence de longeur l qui commence à A_i et le sauvegarder dans $m[i][j]$ avec $j = i + l - 1$;
    $j \leftarrow i + l - 1$;
    $m[i][j] \leftarrow \infty$;
    pour k *de* i *à* $j - 1$ **faire**
      $tmp \leftarrow m[i][k] + m[k+1][j] + t_{i-1} \times t_k \times t_j$;
      si $tmp < m[i][j]$ **alors**
        $m[i][j] \leftarrow tmp$;
retourner *m[1][n]*

Complexité de `CoutMultiplication_ProgDynamique`

- Clairement `CoutMultiplication_ProgDynamique` $\in O(n^3)$
- Meilleur que l'algorithme récursif ($\Theta(3^n)$) et la force brute ($\Omega(\frac{4^n}{(n-1)^{1.5}})$)

Programmation dynamique : résumé

- Les 5 étapes de la programmation dynamique :
  1. Caractériser la structure d'une solution optimale
  2. Définit récursivement la valeur d'une solution optimale
  3. Calculer la valeur d'une solution optimale en remontant progressivement jusqu'au problème initial
  4. Construire une solution optimale en se basant sur les informations calculées
- La dernière étape n'est pas obligatoire si on ne s'intéresse qu'au coût de la solution optimale
- Compléter `CoutMultiplication_ProgDynamique` pour retourner le parethésage optimal.

Algorithmes gloutons

Rappel

- Nous avons traité deux types de problèmes :
  - Les problèmes de décision : Trouver une solution qui satisfait des critères (i.e., problème de tri, problème de recherche d'élément, PGCD, etc)
  - Les problèmes d'optimisation : Trouver une solution qui satisfait des critères et qui minimise ou maximise un coût (e.g., parenthèsage pour la multiplication de matrices). Le coût dans ce cas s'associe à une "fonction objectif".
- Nous avons étudié différentes approches de résolutions :
  - L'approche force brute (recherche exhaustive, énumération)
  - Paradigme diviser pour régner (et les algorithmes récursifs)
  - Programmation dynamique
- On découvre aujourd'hui une nouvelle approche de résolution (l'approche gloutonne) et une jolie structure de représentation de problèmes qui s'appelle "les matroïdes"

Algorithmes gloutons (Greedy algorithms)

- Contexte : typiquement pour les problèmes d'optimisation
- Idée de base :
  - Résoudre le problème en une séquence d'étapes/choix
  - Pour chaque étape, faire un choix qui semble optimal à l'étape courante
- Avantage : Rapide en temps de calcul
- Inconvénient : Pas de garantie sur l'optimalité

Exemple : Problème du Voyageur de commerce

Voyageur de commerce (optimisation)

- **donnée** : ensemble de villes
- **question** : quel est le **plus court** cycle passant par toutes les villes une seule fois ?

Figure – Instance du problème du voyageur de commerce sous forme de graphe

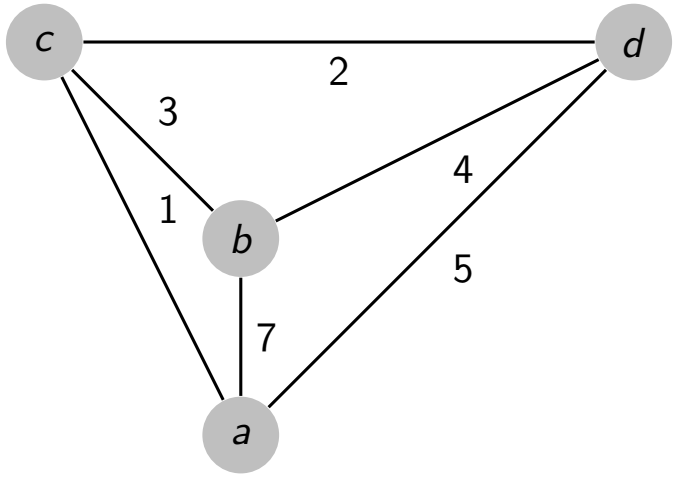
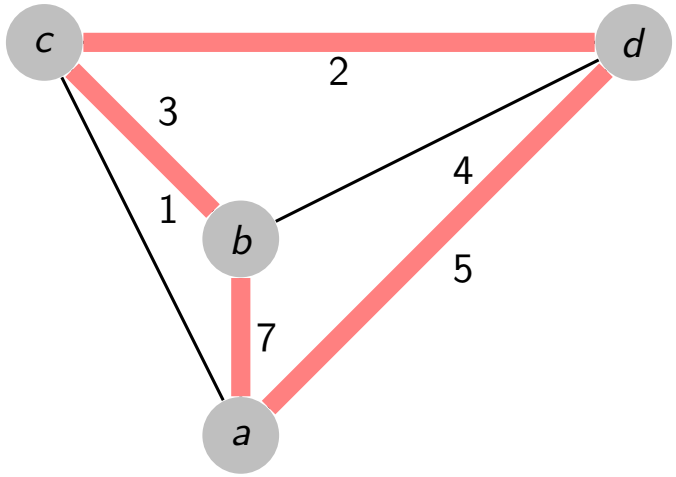
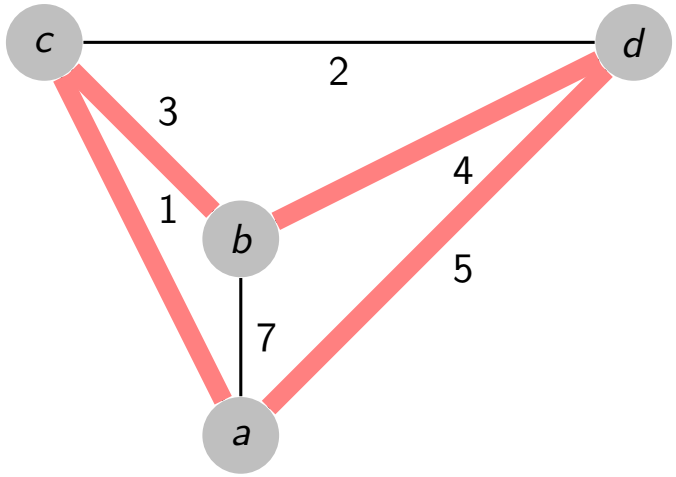


Figure – Une solution non optimale



- Cycle : a,b,c,d,a
- Coût de la solution :$7 + 3 + 2 + 5 = 17$

Figure – Une solution optimale



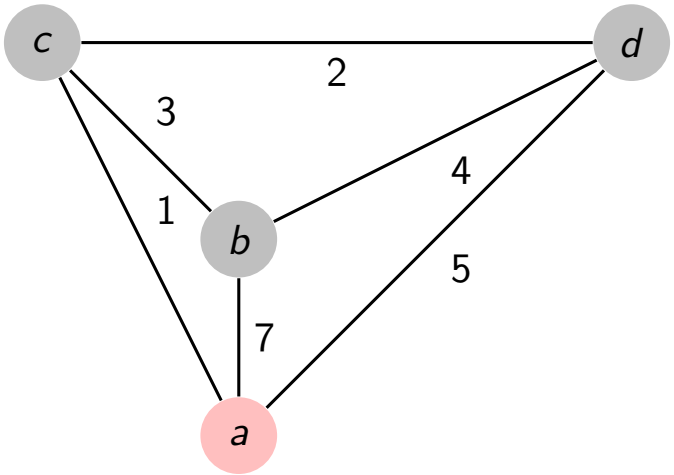
- Cycle : a,c,b,d,a
- Coût de la solution :$1 + 3 + 4 + 5 = 13$

Énumération exhaustive ?

- 2 Villes –> 1
- 3 Villes –> 1
- 4 Villes –> 3
- 5 Villes –> 12
- n Villes –> $\frac{(n-1)!}{2}$ (la moitié du nombre de permutations possible de taille $n-1$)
- 40 villes –> à peu près 10^{46} solutions à tester !
- Avec une machine moderne : 3×10^{29} années (plus que $AgeUnivers^3$) !
- La recherche exhaustive est inefficace ! !

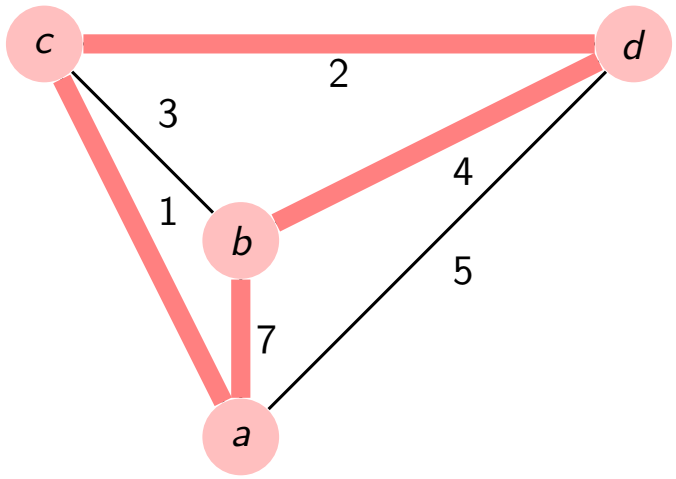
- Idée gloutonne :
  - Construction de la solution ville par ville selon l'ordre de visite
  - À partir de la dernière ville visitée, choisir la ville la plus proche qui est non visitée.
  - Arrêter quand on visite toutes les villes
  - Ajouter la première ville pour construire un cycle.

Figure – Construction de la solution gloutonne étape par étape : Initialisation à partir de "a"



- Chemin initial : a
- Coût initial :0

Figure – Construction de la solution gloutonne étape par étape



- Cycle : a,c,d, b, a
- Coût courant :$1 + 2 + 4 + 7 = 14$
- C'est la solution retournée par l'algorithme glouton
- Ce n'est pas une solution optimale mais c'est assez rapide à trouver

Algorithme Glouton pour le voyageur de commerce

Algorithme : `Glouton` (n, *distance*)

Données : $n \in \mathbb{N}^*$: nombre de villes, $distance[i][j] \in \mathbb{R}^+$: la distance entre ville i et ville j
Résultat : Permutation de $1, \ldots, n$
début

```
Ensemble ← {1, ... n} ;
element ← 1 ;
Permutation ← element ;
Ensemble ← Ensemble \ {element} ;
tant que |Permutation| < n faire
    min ← +∞ ;
    pour e ∈ Ensemble faire
        si distance[element][e] < min alors
            min ← distance[element][e] ;
            ville ← e ;
    Permutation ← Permutation, ville ;      // Ajouter ville à la fin de Permutation
    Ensemble ← Ensemble \ {ville} ;
    element ← ville ;
retourner Permutation ;
```

Complexité : $O(n^2)$

Matroïdes

- Résoudre des problèmes d'optimisation avec des algorithmes gloutons
- Un **matroïde** représente une structure particulière utilisée pour concevoir un algorithme glouton optimal

Définition

- Un matroïde est un couple $\mathcal{M} = (\mathcal{E}, \mathcal{I})$ qui satisfait les conditions suivantes :
  - $\mathcal{E}$ est un ensemble fini non vide
  - $\mathcal{I}$ est un ensemble de sous ensembles de $\mathcal{E}$ tel que :
    - ★ Si $H \in \mathcal{I}$, et $F \subset H$, alors $F \in \mathcal{I}$ (on dit que $\mathcal{I}$ est héréditaire)
    - ★ Si $F \in \mathcal{I}$, $H \in \mathcal{I}$ et $|F| < |H|$, alors $\exists x \in H \setminus F$ tel que $F \cup \{x\} \in \mathcal{I}$ (propriété d'échange)
- Si $\mathcal{M} = (\mathcal{E}, \mathcal{I})$ est un matroïde et $H \in \mathcal{I}$, alors H est appelé "**sous ensemble indépendant**"

Exemple (simple) de Matroïde

- $E = \{1, 2, 3, 4\}$
- $I = \{\{\}, \{1\}, \{2\}, \{4\}, \{1, 4\}, \{2, 4\}\}$
- Preuve
  - E est un ensemble fini non vide (évident)
  - I est héréditaire car :
    - Pour $\{2, 4\}$: $\{\} \in I, \{2\} \in I, \{4\} \in I$
    - Pour $\{1, 4\}$: $\{\} \in I, \{1\} \in I, \{4\} \in I$
    - Pour $\{1\}$: $\{\} \in I, \{1\} \in I$
    - Pour $\{2\}$: $\{\} \in I, \{2\} \in I$
    - Pour $\{3\}$: $\{\} \in I, \{4\} \in I$
    - Pour $\{\}$: $\{\} \in I$
  - Propriété d'échange :
    - Pour $H = \{1, 4\}$ et $F = \{2\}$: $F \cup \{4\} \in I$
    - Pour $H = \{2, 4\}$ et $F = \{1\}$: $F \cup \{2\} \in I$
    - Pour $H = \{4\}$ et $F = \{\}$: $F \cup \{4\} \in I$
    - ...

Matroïde pondéré

- Soit $\mathcal{M} = (\mathcal{E}, \mathcal{I})$ un matroïde
- $\mathcal{M}$ est pondéré s'il existe une fonction de poids pour les éléments de $\mathcal{E}$. Pour chaque $x \in \mathcal{E}$, $w(x) \in \mathbb{R}^{+*}$ est le poids de x.
- Si F est un sous ensemble de $\mathcal{E}$, alors le poids de F se définit avec $w(F) = \sum_{x \in F} w(x)$
- **Problème de sous ensemble indépendant de poids maximal :**
  - Donnée : $\mathcal{M} = (\mathcal{E}, \mathcal{I})$: matroïde et w : fonction de poids
  - Question : Trouver $F \in \mathcal{I}$ de poids maximal

Algorithme glouton

Algorithme : `Glouton` $(\mathcal{M}(\mathcal{E},\mathcal{I})$, $w)$

Données : $\mathcal{M}(\mathcal{E},\mathcal{I})$: matroïde, w : fonction de poids
Résultat : Sous ensemble indépendant de E (probablement de poids maximal)
début
    $F \leftarrow \{\}$;
    $n \leftarrow |\mathcal{E}|$;
    $L \leftarrow Trier(\mathcal{E})$ par poids décroissant ;
    pour $i \in [1..n]$ **faire**
        si $F \cup \{L[i]\} \in \mathcal{I}$ **alors**
            $F \leftarrow F \cup \{L[i]\}$;
    retourner F ;

Complexité : Si le test d'appartenance (ligne 7) se fait en $O(f(n))$, alors la complexité de `Glouton` $(\mathcal{M}(\mathcal{E},\mathcal{I})$, $w)$ est $O(nlog(n) + nf(n))$ avec $n = |\mathcal{E}|$ (car le tri peut se faire en $O(nlog(n))$).

L'importance des matroïdes

Theorem

`Glouton` $(\mathcal{M}(\mathcal{E},\mathcal{I})$, $w)$ *retourne un sous ensemble indépendant optimal*

Algorithme : `Glouton` $(\mathcal{M}(\mathcal{E},\mathcal{I})$, $w)$

Données : $\mathcal{M}(\mathcal{E},\mathcal{I})$: matroïde, w : fonction de poids
Résultat : Sous ensemble indépendant de E **de poids maximal**
début
    $F \leftarrow \{\}$;
    $n \leftarrow |\mathcal{E}|$;
    $L \leftarrow Trier(\mathcal{E})$ par poids décroissant ;
    pour $i \in [1..n]$ **faire**
        si $F \cup \{L[i]\} \in \mathcal{I}$ **alors**
            $F \leftarrow F \cup \{L[i]\}$;
    retourner F ;

Glouton $(\mathcal{M}(\mathcal{E},\mathcal{I})\ ,\ w)$, alors ?

- Rappel pour un problème d'optimisation :
  - **Une solution** est une sortie qui respecte les exigences du problème
  - **Une solution optimale** est une solution qui (minimise ou maximise) une fonction (objectif).
  - Le **coût** d'une solution est la valeur correspondante à la fonction objectif
- Pour exploiter Glouton $(\mathcal{M}(\mathcal{E},\mathcal{I})\ ,\ w)$ pour un problème d'optimisation $\mathcal{P}$:
  - Il faut trouver un matroïde $\mathcal{M}(\mathcal{E},\mathcal{I})$ pondéré tel qu'une solution optimale de $\mathcal{P}$ correspond à un sous ensemble indépendant (i.e., élément de $\mathcal{I}$) de poids maximal qu'on peut calculer à partir de Glouton $(\mathcal{M}(\mathcal{E},\mathcal{I})\ ,\ w)$.
  - Dans ce cas, l'algorithme glouton est garanti de retourner une solution optimale
- Cette approche ne marche pas pour tous les problèmes. En particulier, il y a souvent deux limites :
  1. Difficulté à définir l'ensemble I des sous ensembles indépendants
  2. Même si on trouve I, le test d'appartenance ($F \in I$?) est coûteux en temps (par exemple quand $f(n) = O(2^n)$.

Représentation des Données

Complexité en fonction de la taille de la donnée

- Pourquoi calculer la complexité en fonction de la *taille de la donnée* ?
- Sinon quel paramètre choisir ?
  - Multiplication de x par y : en fonction de x ? de y ? de $x + y$? de xy ?
- Sinon comment comparer des algorithmes avec des données différentes ?
  - Est-ce que `Factorielle` est plus efficace que `triSélection` ?
  - `Factorielle` : $\Theta(x)$ opérations, $|x| = \log_2 x$, donc $\Theta(2^{|x|})$ opérations
  - `triSélection` : $\Theta(n^2)$ opérations, $|T| = n$, donc $\geq \Theta(|L|^2)$ opérations
- Comment connaitre la taille de la donnée ?

Calculer la taille de la donnée

On compte le nombre de *bits* mémoire, en ordre de grandeur (Θ)

- Exemples :
  - Types `char`, `int`, `float`, etc. : $\mathcal{O}(1)$
  - Type $\mathbb{N}$: $\Theta(\log n)$ (pour un entier $\leq n$)
  - Type `liste d'int` : $\Theta(n)$ (pour une liste de longueur $\leq n$)

Borne supérieure ($\mathcal{O}$)

Trouver un encodage

Borne inférieure (Ω)

Principe des tiroirs

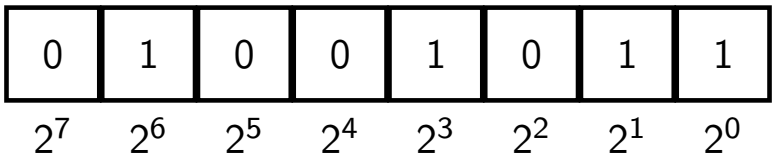
Encodage

Un encodage pour un type de donnée $\mathcal{T}$ est une fonction *injective* :

$$f : \mathcal{T} \mapsto \{0,1\}^k$$

- Tout $x \in \mathcal{T}$ a un seul code $f(x)$ (fonction)
  - Sinon on ne peut pas toujours encoder
- Pour $x, y \in \mathcal{T}$ distincts, $f(x) \neq f(y)$ (injective)
  - Sinon on ne peut pas toujours decoder

- Exemples : ASCII, Morse,...

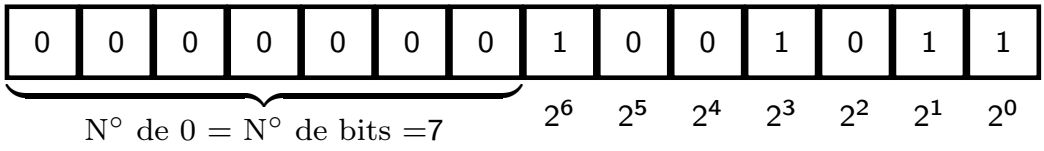


- Représenter les entiers entre 0 et 255 (`char`)
- Chaque bit repésente un terme de la somme $x = \sum_{i=0}^{7} b_i 2^i$
- Pour $x = 75$: $1 \times 2^6 + 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0$

Borne supérieure (et infereieure)

Si c est de type "`char`" alors $|c| \in \mathcal{O}(1)$, et donc $|c| \in \Theta(1)$

Encodage d'un entier $n \in \mathbb{N}$

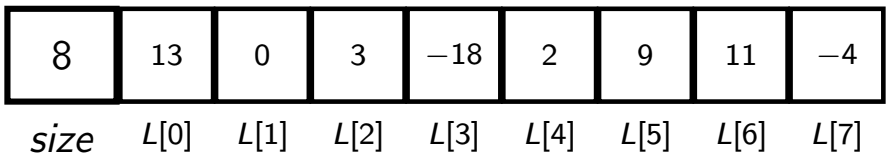


- Code en base 2 : $\mathcal{O}(\log_2 n)$ pour coder tout entier naturel $\leq n$
- Problème : combien de bits allouer pour coder *n'importe quel* entier ? ∞ ?
- Donner le nombre de bits "utiles" en préfixe, en code unaire (autant de 0 que de bits significatifs)
- Code en $\mathcal{O}(\log_2(n))$ pour le préfixe + $\mathcal{O}(\log_2(n))$ pour le suffixe

Borne supérieure

Si $n \in \mathbb{N}$ alors $|n| \in \mathcal{O}(\log n)$

Encodage d'un tableau de n `int`



- Chaque `int` requiert 4 octets : $n \times \mathcal{O}(1)$
- Même astuce que pour les entiers, le nombre d'éléments en préfixe : $\mathcal{O}(\log n)$
- $n \times \mathcal{O}(1) + \mathcal{O}(\log n) = \mathcal{O}(n)$

Borne supérieure

Si L est un tableau contenant n `int`, alors $|L| \in \mathcal{O}(n)$

Borne inférieure sur la taille de la donnée

Quelques principes de base de *dénombrement*

Principe additif

Les choix *mutuellement exclusifs* se combinent par addition.

- Ex : combien de choix possibles de plats principal si on a 3 types de viande, 2 poisson et 3 plats végétariens ? $3 + 2 + 3 = 8$ plats

Principe multiplicatif

Les choix *indépendents* se combinent par multiplication.

- Combien de menus s'il y a 3 entrées, 4 plats, et 4 desserts ? $3 \times 4 \times 4 = 48$ menus
- Combien de valeurs possibles pour un `int` sur 32 bits ? 2^{32} valeurs

Principes des tiroirs

Principe des tiroirs

Si m objets sont rangés dans n tiroirs, alors un tiroir en contient au moins $\lceil \frac{m}{n} \rceil$.

- Si $m > n$ objets sont rangés dans n tiroirs, alors un tiroir en contient au moins 2
- Il y a deux londoniens avec exactement le même nombre de cheveux
  - Il n'y a pas plus d'un million de cheveux sur un crâne, donc pas plus d'un million de nombres de cheveux distincts
  - Il y a plus d'un million de londoniens
  - m londoniens à répartir parmi n chevelures possibles $\Rightarrow$ au moins deux londoniens avec la même chevelure

Minorant pour l'espace mémoire

- un encodage est une *fonction injective* d'un type de donnée vers les mots binaires :

$$f : \mathcal{T} \mapsto \{0,1\}^k \quad \text{avec } f(x) \in \{0,1\}^{|x|}$$

- Il y a 2^k mots binaires de longueur k (Principe multiplicatif)
- Il faut au moins autant de mots binaires que de valeurs possibles pour la donnée
  - Principe des tiroirs : sinon, des données distinctes ont le même code, et f est non-injective

Minorant pour la taille $|x|$ d'une donnée x de type $\mathcal{T}$

Soit $\#(\mathcal{T})$ le nombre de valeurs possibles du type de donnée $\mathcal{T}$, la mémoire $|x|$ nécessaire pour stocker une donné $x \in \mathcal{T}$ est telle que $2^{|x|} \geq \#(\mathcal{T})$, $\Rightarrow$

$$|x| \in \Omega(\log \#(\mathcal{T}))$$

Bornes inférieures

- Calculons $\#(\mathcal{T})$, le nombre de valeurs possibles pour la donnée x de type $\mathcal{T}$:
  - Entier naturel inférieur ou égal à n : $n+1$, soit $\Theta(n)$

Pour x un entier naturel inférieur ou égal à n :

$|x| \in \Omega(\log n)$ et puisqu'il existe un encodage tel que $|x| \in \mathcal{O}(\log n)$, alors $|x| \in \Theta(\log n)$

  - Tableau d'`int` de longueur n : $(2^{32})^n$, soit $\Theta(2^{32n})$
  - Tableau d'`int` de longueur $\leq n$: $\sum_{i=0}^{n} 2^{32i}$, soit $\Theta(2^{32n})$
  - $\log_2(2^{32n}) = 32n log_2(2) = 32n$

Pour L un tableau d'`int` de longueur $\leq n$:

$|L| \in \Omega(n)$ et puisqu'il existe un encodage tel que $|L| \in \mathcal{O}(n)$, alors $|L| \in \Theta(n)$

Complexité en fonction de la taille de la donnée

- L'algorithme **A** est en $\Theta(f(x))$ pour une donnée x
- La taille $|x|$ de la donnée est en $\Theta(g(x))$
- Alors la complexité de **A** est en $\Theta(f(g^{-1}(x)))$

Exemple

Algorithme : Carré(x)
Données : un entier x
Résultat : un entier valant x^2
r : entier;
début
  $r \leftarrow 0$;
  pour *i allant de* 1 *à* x **faire**
    pour *j allant de* 1 *à* x **faire**
      $r \leftarrow r + 1$;
  retourner r;

- Complexité : $\Theta(x^2)$
- Taille de la donnée $|x| = \Theta(\log x)$
- Autrement dit, $x = \Theta(2^{|x|})$
- Donc complexité en $\Theta(2^{2|x|})$ (exponentielle !)

Structure de données

Le choix de la structure de données est très important dans la conception d'un algorithme

- Choisir la structure pour laquelle les opérations *utiles* sont les plus *efficaces*
- Un *type abstrait* est défini par les opérations qui sont **efficaces** sur ce type
  - **Insertion** (I) : ajouter un nouvel élément
  - test d'**Appartenance** (A) : vérifier si l'élément x est présent
  - **Suppression** (S) : supprimer un élément x
  - **Suppression du Dernier** (SD) : supprimer le dernier élément inséré
  - **Suppression du Premier** (SP) : supprimer le premier élément inséré
  - **Suppression du Minimum** (SM) : supprimer l'élément minimum

Structure de données

- Un *type abstrait* est défini par les opérations qui sont **efficaces** sur ce type
- Une *réalisation* correspond à du code (des algorithmes)

Type Abstrait	Réalisation	I	A	S	SP	SD	SM
Pile	Liste chainée	$\mathcal{O}(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\mathcal{O}(1)$	$\Theta(n)$
File	Liste chainée avec pointeurs *début* et *fin*	$\mathcal{O}(1)$	$\Theta(n)$	$\Theta(n)$	$\mathcal{O}(1)$	$\Theta(n)$	$\Theta(n)$
Index statique	Vecteur trié	$\Theta(n)$	$\mathcal{O}(\log n)$	$\Theta(n)$	N/A	N/A	$\Theta(n)$
File de priorité	Tas binaire	$\mathcal{O}(\log n)$	$\Theta(n)$	$\Theta(n)$	N/A	N/A	$\mathcal{O}(\log n)$
Ensemble	Table de hâchage	$\mathcal{O}(1)$	$\mathcal{O}(1)^*$ $\Theta(n)$	$\mathcal{O}(1)^*$ $\Theta(n)$	N/A	N/A	$\Theta(n)$
Ensemble trié	ABR, AVL Arbre rouge-noir	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	N/A	N/A	$\mathcal{O}(\log n)$

Exemple : Table de Hâchage

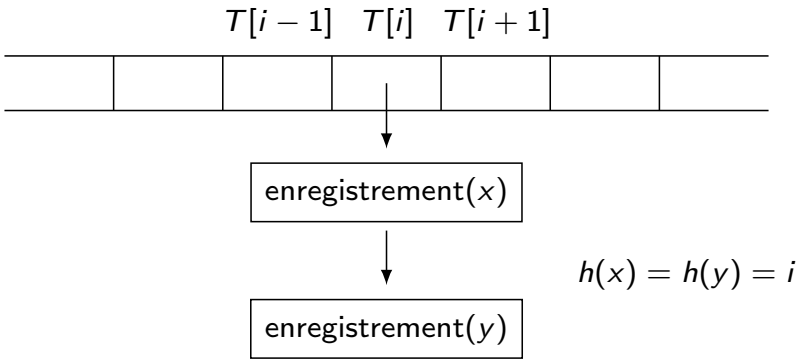
Fichier des étudiants de l'INSA, il faut une *clé unique* pour identification

- Tatouer chaque étudiant avec son numéro d'inscrit $(1, \ldots, n)$?
  - Une table T avec $T[x]$ contenant les informations pour l'étudiant x
- Utiliser le numéro de sécurité sociale ?
  - Beaucoup trop de clés possibles !

Table de Hâchage : n enregistrements / table de taille $m \in \Theta(n)$

- Soit U l'ensemble des clés possibles, avec $|U| = M, m \ll M$
- Soit $h : U \mapsto \{1, \ldots, m\}$ un fonction de hâchage : renvoie un index pour chaque clé, par ex. $h(x) = ((ax + b) \bmod p) \bmod m$ avec $m \ll p \ll M$ premier et $0 < a, b < p$
- L'enregistrement de clé x est stocké dans $T[h(x)]$. Si $h(x) = h(y)$ et $x \neq y$ on dit qu'il y a une *collision*

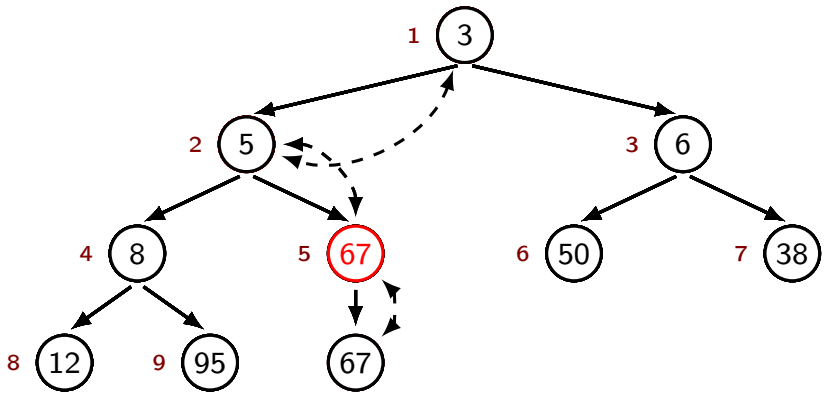
Exemple : Table de Hâchage



- Analyse de la complexité du test d'appartenance (A)
  - Pire des cas $T_{\max}(n) = \Theta(n)$ (tous les enregistrements ont la même valeur de hâchage)
  - Pour $T_{moy}(n)$ on suppose une distribution uniforme des valeurs de $h(x)$ dans $\{1, \ldots, m\}$
  - Soit L_i la longueur de la liste $T[i]$, et $E(L_i)$ l'espérance de L_i : $\sum_{i=1}^{m} E(L_i) = n$
  - Mais $E(L_i)$ ne dépend pas de i, donc $T_{moy}(n) = E(L_i) = n/m \in \mathcal{O}(1)$

Exemple : Tas Binaire

- Invariants : arbre binaire complet ; sommet parent $\leq$ fils



réalisation :

1	2	3	4	5	6	7	8	9	10
22673	533675	6	8	673567	50	38	12	95	.367

parent de i : $\lfloor i/2 \rfloor$
fils gauche de i : $2i$
fils droit de i : $2i + 1$

- Insertion : la position libre la plus à gauche possible sur le dernier niveau
  - *Percolation* échange avec le parent jusqu'à ce que l'invariant soit rétabli $\mathcal{O}(\log n)$
- Suppression du minimum : la racine, qu'on remplace par le "dernier" sommet
  - *Percolation* échange avec le fils minimum jusqu'à ce que l'invariant soit rétabli $\mathcal{O}(\log n)$

Donnée : une liste L d'éléments comparables

Pour chaque $x \in L$:

- Insérer x dans le tas binaire H

Tant que H n'est pas vide :

- Extraire le minimum de H et l'afficher

- n insertions en $\mathcal{O}(\log n)$
- n suppressions en $\mathcal{O}(\log n)$
- Complexité du tri par tas : $\mathcal{O}(n \log n)$

Tri par tas est optimal !

- $\mathcal{O}(n \log n)$ dans le pire des cas

Classes de Complexité

Borne inférieure pour les algorithmes de tri

Tri par comparaison

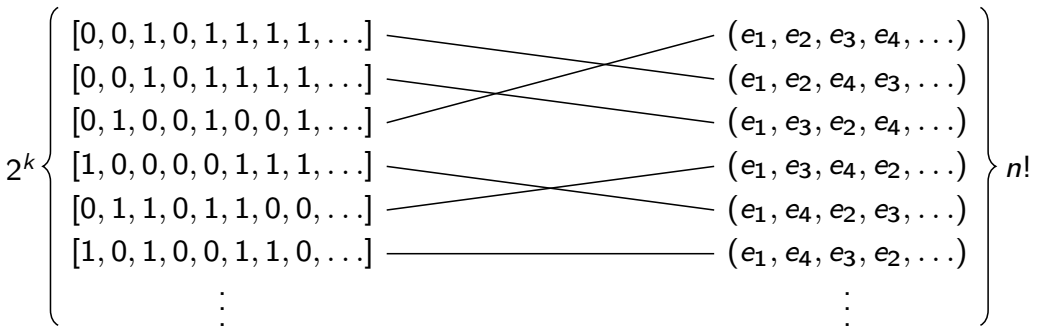
- **donnée** : une liste d'éléments comparables
- **question** : quelle est la liste triée de ces éléments ?

- Considérons les algorithmes de tri qui ne peuvent pas "lire" ces éléments, seulement les comparer (e.g. un tableau de pointeurs vers une classe d'objets comparables).
- Lors de son execution, cet algorithme va comparer k paires d'éléments (x, y), le résultat peut être 0 ($x < y$) ou 1 ($x \geq y$)
- On peut considérer que la donnée de l'algorithme est une table de longueur k avec les résultats des comparaisons :

$$x = \underbrace{[0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1]}_{k}$$

Borne inférieure pour les algorithmes de tri

- Un algorithme est deterministe, donc deux tables de comparaisons identiques donnent la même exécution, et donc la même liste triée



- Au plus k comparaisons, donc au plus 2^k données/exécutions distinctes
- Chacune des $n!$ permutations de la donnée doit correspondre à une exécution distincte

principe des tiroirs

$$2^k \geq n!$$

Borne inférieure pour les algorithmes de tri

$$
\begin{aligned}
2^k \geq n! \implies k &\geq \log(n!) \\
&= \log(n(n-1)(n-2)\dots 2) \\
&= \log n + \log(n-1) + \log(n-2) + \dots + \log(2) \\
&= \sum_{i=2}^{n} \log i \\
&= \sum_{i=2}^{n/2-1} \log i + \sum_{i=n/2}^{n} \log i \\
&\geq \sum_{i=n/2}^{n} \log \frac{n}{2} \\
&= \frac{n}{2} \log \frac{n}{2} \\
&= \Omega(n \log n)
\end{aligned}
$$

LAAS CNRS

Borne inférieure pour les algorithmes de tri

Théorème

Tout algorithme de tri *par comparaison* est en $\Omega(n \log n)$

- Attention, il existe des algorithmes de tri en $\mathcal{O}(n)$
  - Mais ces algorithmes font des hypothèses sur les éléments à trier

- Dans le cas général d'éléments comparables sans propriété particulière : impossible de les trier avec une complexité dans le pire des cas inférieure à $\Omega(n \log n)$

La complexité d'un problème, à quoi ça sert ?

- Pour pouvoir objectivement analyser un algorithme
  - Optimalité
- Parce que la difficulté du problème détermine le type de méthode
  - solution approchée pour un problème difficile ?
- Parce que la difficulté du problème est parfois une garantie
  - Cryptographie, Block Chain

Problème

Définition : Problème $\simeq$ fonction sur les entiers

- Une question Q qui associe une donnée x à une réponse y
  - "Quel est le plus court chemin de x_1 vers x_2 par le réseau R ?"
  - "Quel est la valeur du carré de x ?"

- Q_{pcc} : Réseau : R, Villes : $x_1, x_2 \mapsto$ Route : $x_1, u_1, u_2, \ldots, u_k, x_2$
- Q_{carr} : Entier : $x \mapsto$ Entier : x^2

Types de Problèmes

- Problèmes généraux (fonctions)
- Problèmes d'optimisation : la solution est le *minimum* d'un ensemble
- Problèmes de décision : la réponse est dans {oui, non}

Problème de décision

Problème de décision Q

Fonction $Q : x \mapsto \{\mathbf{oui}, \mathbf{non}\}$

- Pour un problème d'optimisation, on peut généralement définir un problème **polynomialement** équivalent dont la réponse est dans $\{\mathbf{oui}, \mathbf{non}\}$, :

Voyageur de commerce (optimisation)

- **donnée** : ensemble de villes
- **question** : quel est le **plus court** chemin passant par toutes les villes ?

Voyageur de commerce (décision)

- **donnée** : ensemble de villes, entier k
- **question** : est-ce qu'il existe un chemin de longueur inférieure à k passant par toutes les villes ?

Instance

- Instance : problème avec la donnée (“Combien vaut 567^2 ?”, “Quel est le plus court chemin entre Toulouse et Paris ?”)
- Ne pas confondre **problème** et **instance**
- En particulier, se poser la question de la difficulté d'une instance n'a pas (beaucoup ?) de sens
  - L'algorithme qui renvoie systématiquement, et sans calcul, la solution de cette instance (correct et en $\mathcal{O}(1)$)

Classes d'Algorithmes

Un algorithme est dit :

- en temps *constant* si sa complexité dans le pire des cas est bornée par une constante
- *linéaire* si sa complexité dans le pire des cas est en $\Theta(n)$
- *quadratique* si sa complexité dans le pire des cas est en $\Theta(n^2)$
- *polynomial* si sa complexité dans le pire des cas est en $\mathcal{O}(n^c)$ avec $c > 0$
- *exponentiel* si elle est en $\Theta(2^{n^c})$ pour un certain $c > 1$

Classes de problèmes

- Comment évaluer la complexité d'un problème ?

La complexité d'un problème :

La complexité du meilleur algorithme pour le résoudre

$f(n)$-TIME

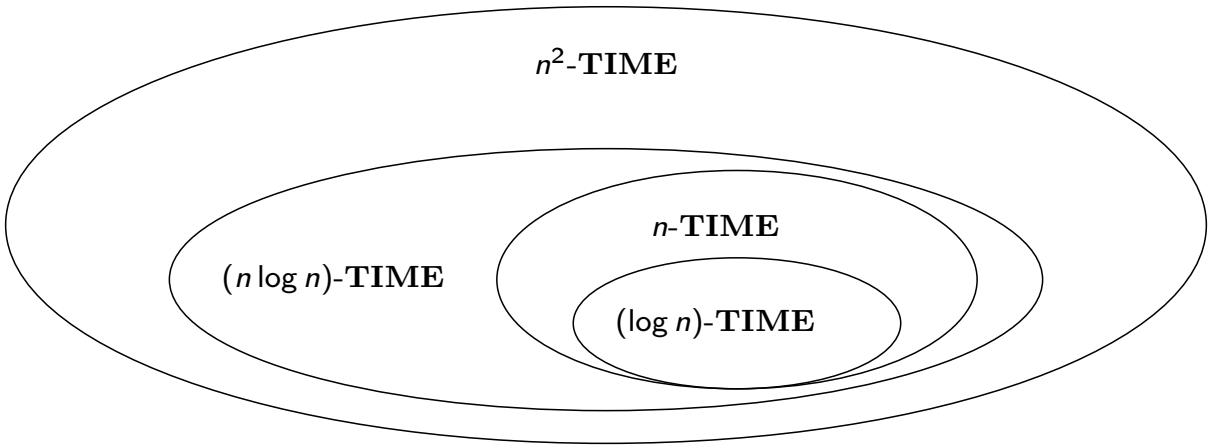
Ensemble des problèmes pour lesquels il existe un algorithme en $\mathcal{O}(f(n))$

- *Tri* $\in (n \log n)$-**TIME**
- *Recherche dans un tableau trié* $\in (\log n)$-**TIME**
- *Recherche du maximum dans un tableau* $\in (n)$-**TIME**

Relation d'Inclusion

Inclusion

Si $f(n) \in \mathcal{O}(g(n))$ alors $f(n)$-**TIME** $\subseteq g(n)$-**TIME**

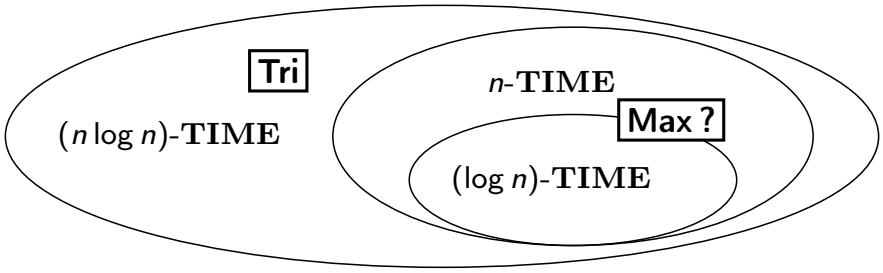


Problème de Tri

Tri

donnée: Une liste L d'objet comparables

question: La séquence triée des objets de L



- $Tri \in (n \log n)$-**TIME**
- Tout algorithme de tri est en $\Omega(n \log n) \implies Tri \notin n$-**TIME**
- La recherche du maximum est dans n-**TIME**, dans $(\log n)$-**TIME** ?

(Non-)Appartenance à une Classe

$f(n)$-TIME

Ensemble des problèmes pour lesquels il existe un algorithme en $\mathcal{O}(f(n))$

- Pour prouver qu'un problème appartient à la classe $f(n)$-**TIME**
  - il suffit d'un algorithme en $\mathcal{O}(f(n))$
- Pour prouver qu'un problème n'appartient pas à une classe inférieure
  - il faut montrer que *tout* algorithme est en $\Omega(f(n))$

Classe "Temps Polynômial"

- Rappel :

$f(n)$-TIME

Ensemble des problèmes pour lesquels il existe un algorithme en $\mathcal{O}(f(n))$

- Classe des problèmes pour lesquels il existe un algorithme polynômial :

P-TIME ou simplement : P

Ensemble des problèmes pour lesquels il existe un algorithme en $\mathcal{O}(n^c)$ pour une constante c.

$$\mathbf{P} = \bigcup_{c \geq 0} n^c\text{-TIME}$$

Classe "Temps Exponentiel"

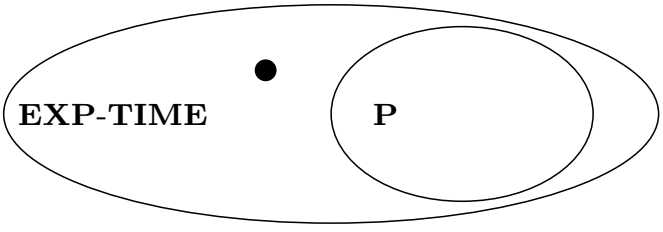
- Classe des problèmes pour lesquels il existe un algorithme exponentiel :

EXP-TIME :

Ensemble des problèmes pour lesquels il existe un algorithme en $\mathcal{O}(2^{n^c})$ pour une constante c

$$\mathbf{EXP\text{-}TIME} = \bigcup_{c \geq 1} 2^{n^c}\text{-TIME}$$

- Evidemment, $\mathbf{P} \subseteq \mathbf{EXP\text{-}TIME}$
- Est-ce que $\mathbf{P} \subset \mathbf{EXP\text{-}TIME}$? **Oui !**
- Il existe des problèmes en $\Omega(2^n)$



Classes Caractérisées par l'Espace

- On peut analyser l'espace mémoire utilisé par un algorithme de manière similaire

$f(n)$-SPACE

Ensemble des problèmes pour lesquels il existe un algorithme nécessitant $\mathcal{O}(f(n))$ octets de mémoire

- On ne compte pas la taille de la donnée, mais on compte la taille de la réponse

Exemple : Sous-Séquence Maximale

Sous-Séquence Maximale

donnée: Un tableau L avec n éléments dans $\{-1, 1\}$

question: Quelles sont les valeurs de s et e dans $[1, n]$ qui maximisent $\sum_{i=s}^{e} L[i]$

-1 -1 $\underbrace{1\ 1\ -1\ 1\ -1\ 1\ 1}_{3}$ -1 -1 1 1 -1 1

Algorithme 1

Pour chaque $1 \leq i \leq j \leq n$:
calculer $\sum_{k=i}^{j} L[k]$; garder le max.

- Temps : $\Theta(n^3)$
- Espace : $\Theta(\log n)$
  - $\log n$ pour le max, la somme et le resultat

Exemple : Sous-Séquence Maximale

$L[i]$	1	1	1	-1	-1	-1	-1	1	-1	-1	**1**	**1**	**1**	**1**	**-1**	**-1**	**1**	**1**	**1**	-1	-1	1	-1	1	-1	-1	-1
$s[i]$	1	2	3	2	1	0	-1	0	-1	-2	-1	0	1	2	1	0	1	2	3	2	1	2	1	2	1	0	-1
$\min[i]$	0	0	0	0	0	0	0	-1	-1	-1	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
$\max[i]$	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	1	0	-1
$mss[i]$	3	3	3	3	3	3	3	4	4	4	5	5	5	5	5	5	5	5	5	4	4	4	4	4	3	2	1

Algorithme 2

- $s[i] = \sum_{k=1}^{i} L[k]$
  - $\sum_{k=i}^{j} L[k] = \sum_{k=1}^{j} L[k] - \sum_{k=1}^{i-1} L[k]$
- $\min[i] = \min(s[j] \mid j < i)$
- $\max[i] = \max(s[j] \mid j \geq i)$
- $mss[i] = \max_i - \min_i$

- $\Theta(1)$ tables de taille $\Theta(n)$
- Temps : $\Theta(n)$
- Espace : $\Theta(n)$

Temps ou Espace

	Temps	Espace
Algorithme 1	$\Theta(n^3)$	$\Theta(\log n)$
Algorithme 2	$\Theta(n)$	$\Theta(n)$

Temps ou Mémoire ?

Théorème

$f(n)$-**TIME** $\subseteq$ $f(n)$-**SPACE**

- Supposons qu'il existe un problème **A** t.q. **A** $\in$ $f(n)$-**TIME** et **A** $\notin$ $f(n)$-**SPACE**.
- Alors tout algorithme pour **A** utilise $\Omega(g(n))$ mémoire avec $g(n) \notin \mathcal{O}(f(n))$.
- Si un algorithme nécessite $\Omega(g(n))$ mémoire il fait $\Omega(g(n))$ écritures.
- Autrement dit, il est en $\Omega(g(n))$ temps (contradiction).
- Donc **A** $\in$ $f(n)$-**TIME** $\implies$ **A** $\in$ $f(n)$-**SPACE**.

Classe "Espace Polynomial"

- Classe des problèmes pour lesquels il existe un algorithme polynômial **en espace** :

P-SPACE

Ensemble des problèmes pour lesquels il existe un algorithme qui utilise $\mathcal{O}(n^c)$ octets de mémoire pour une constante c.

$$= \textbf{P-SPACE} \bigcup_{c \geq 0} n^c\textbf{-SPACE}$$

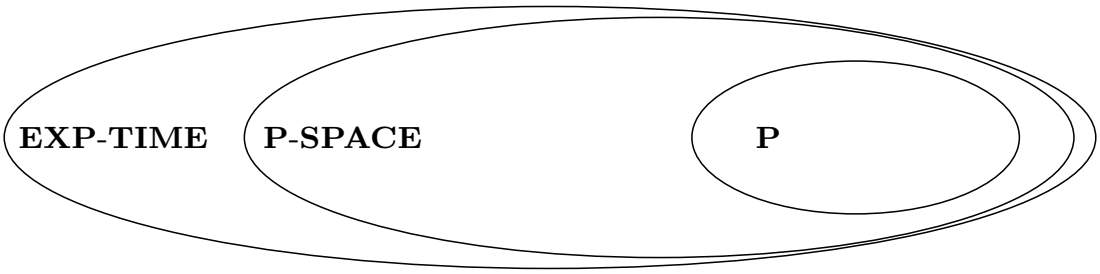
- Est-ce que **P-SPACE** est inclu dans **EXP-TIME**, **EXP-TIME** inclu dans **P-SPACE**, ou ni l'un ni l'autre ?

P-SPACE et EXP-TIME

Théorème

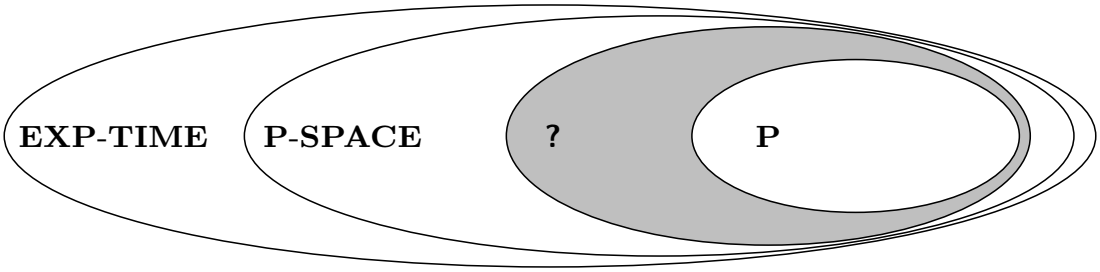
P-SPACE $\subseteq$ EXP-TIME

- Un problème **A** dans **P-SPACE** admet un algorithme qui utilise $\mathcal{O}(n^c)$ octets
- Cet algorithme est constitué de $k \in \Theta(1)$ instructions ($\simeq$ lignes de code)
- La mémoire utilisée par cet algorithme peut être dans $\mathcal{O}(2^{(8n)^c})$ configurations
- Il y a donc $\mathcal{O}(k2^{(8n)^c}) = \mathcal{O}(2^{(n)^c})$ configurations possibles
- Si l'algorithme passe deux fois par la même configuration il ne s'arrête jamais



Problèmes "intermédiaires"

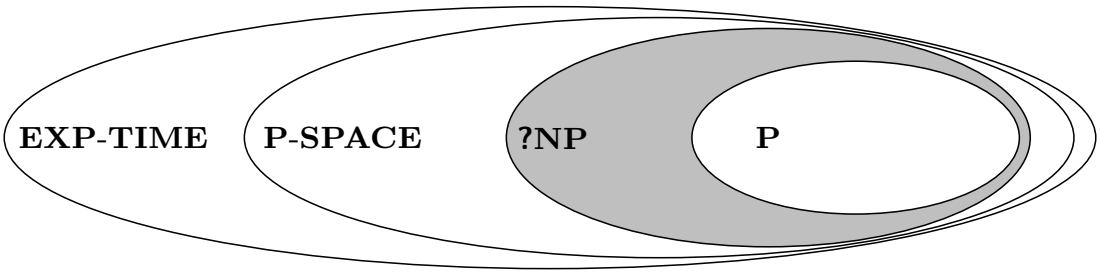
- Il y a un nombre infini de classes de complexité entre **P** et **P-SPACE** !



- Il existe des problèmes pour lesquels on ne connait pas d'algorithme en temps polynômial, sans pouvoir prouver qu'ils n'en n'ont pas
- Ces problèmes sont très nombreux...
- ... et très intéressants : *Voyageur de Commerce*, *Programmation Linéaire en Nombres Entiers*, *SAT*, *Isomorphisme de Graphes*, etc.

Les Classes NP et coNP

Classes non-déterministes



- A priori *difficiles* (pas d'algorithme polynomial connu)
- *Faciles* à **vérifier**, ex. *Voyageur de Commerce*

TSP

donnée: Un ensemble de villes S, une matrice de distance $D : S \times S \mapsto \mathbb{N}$, un entier k

question: Est-ce qu'il existe une route de longueur au plus k passant par toutes les villes ?

Classes non-déterministes

- Ces problèmes peuvent être résolus en espace polynomial
- Pas d'algorithme en temps polynomial connu, mais aucune preuve d'impossibilité
- Problèmes faciles pour des algorithmes **non-déterministes**

Algorithme non-déterministe pour le problème Q

- Composée d'instructions primitives : exécutable par une machine
- Déterministe : une seule exécution possible pour chaque donnée
- Non-déterministe : peut **deviner un certificat** (par ex. la solution !)
- **Correct** : termine et retourne la bonne solution $Q(x)$ pour toute valeur de la donnée x

Oracle

Algorithme non-déterministe

- Un algorithme non-déterministe peut avoir recours à un **oracle** qui, si la réponse est **"oui"**, devine un **certificat polynomial** en temps $\mathcal{O}(1)$
- Le certificat peut-être n'importe quoi, mais :
  - Il faut pouvoir le coder en espace **polynomial** dans la taille de la donnée
  - Il faut pouvoir **prouver** que **"oui"** est bien la réponse correcte en temps **polynomial** dltd
- Quel certificat pour le problème du voyageur de commerce ?

TSP

donnée: Un ensemble de villes S, une matrice de distance $D : S \times S \mapsto \mathbb{N}$, un entier k

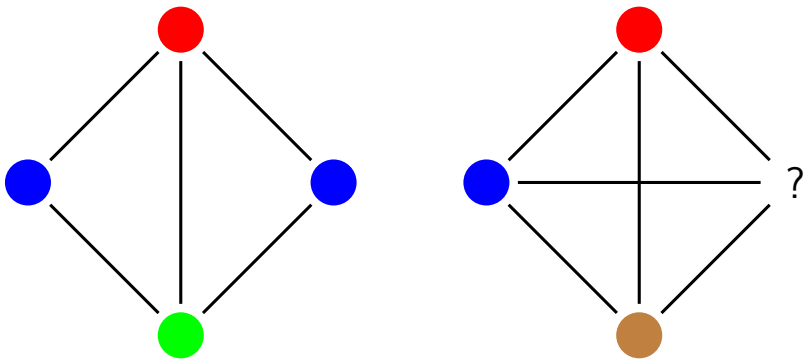
question: Est-ce qu'il existe une route de longueur au plus k passant par toutes les villes ?

- La séquence de villes : on peut vérifier la longueur de ce tour, et être convaincu que la réponse est correcte, même si on ne sait pas comment elle a été obtenue

Exemple : Le problème de *3-coloration*

Donnée : Un graphe $G = (S, A)$ (sommets S ; arêtes A)

Question : Est-ce qu'il est possible de colorier les sommets de G avec au plus 3 couleurs en évitant que deux sommets adjacents partagent la même couleur.



3-Coloration est dans NP

Certificat : La coloration (tableau – couleur du i-ème sommet dans la case i)

- De taille polynomiale (dans la TDLD)
  - quelle est la taille de la donnée du problème ?
    taille : $|G| = \Theta(|S| + |A|)$
  - quelle est la taille du certificat ?
    taille : $\Theta(|S|)$
- Vérifiable en temps polynomial (dans la TDLD) : algorithme

Algorithme de vérification

Algorithme : verification(L, G)

Données : un graphe $G = (S, A)$ et un tableau T
Résultat : `vrai` si T est une 3-coloration de G, `faux` sinon
début
  pour *chaque arrête* (i, j) *de* A **faire**
    si $T[i] = T[j]$ **alors**
      retourner $faux$;
  retourner $vrai$;

Complexité : $\mathcal{O}(|A|)$ (liste d'adjacence)

$\Rightarrow$ *3-Coloration* est bien dans **NP**

LAAS CNRS

P et NP

- **P** est la classe des problèmes "faciles à résoudre"
- **NP** est la classe des problèmes "faciles à vérifier"
- Est-ce qu'il y a une différence ? (500 000 € pour la bonne réponse !)
  - On ne connait pas d'algorithme polynomial pour *Voyageur de commerce* ou *3-Coloration*
  - Mais personne ne sait s'il en existe !

P, NP et la Cryptographie

Système d'authentification A

- comparaison entre
  - une *clé privée* y (mot de passe / pin)
  - et une information publique x (login / carte bancaire)
- autorise la transaction si et seulement si $A(x) = y$

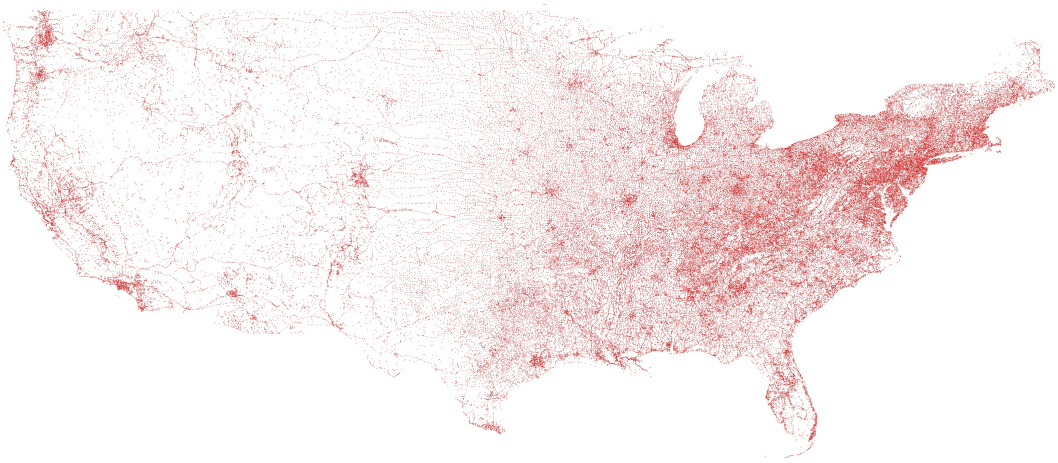
- Supposons que calculer A soit polynomial : trouver le mot de passe est facile !
- Supposons que vérifier $A(x) = y$ ne soit pas polynomial : authentifier est difficile !

La majorité des systèmes d'authentification

sont basés sur un problème A tel que $A \in \mathbf{NP}$ et $A \notin \mathbf{P}$

L'importance de la classe NP

- Ces problèmes sont partout : en intelligence artificielle, en cryptographie, dans l'industrie...
- Savoir adapter la méthode à la complexité du problème ; le problème du voyageur de commerce n'a pas d'algorithme polynomial connu, mais...
  - Domaine de recherche très actif, des algorithmes "intelligents" peuvent résoudre (optimalement) de très grandes instances



115 475 villes

L'importance de la classe NP (suite)

Conjecture $\mathbf{P} \neq \mathbf{NP}$

- Un des 7 "problèmes du millénaire" du Clay Mathematics Institute
  Mise-à-prix \$1 000 000

- Preuve de $\mathbf{P} \neq \mathbf{NP}$: un problème dans $\mathbf{NP}$ mais pas dans $\mathbf{P}$
  - Montrer qu'un problème est dans $\mathbf{NP}$ est facile : certificat polynomial
  - Montrer qu'un problème n'est pas dans $\mathbf{P}$ est difficile : tout algorithme est en $\Omega(2^n)$

- Prouver que $\mathbf{P} = \mathbf{NP}$ ne nécessite qu'un algorithme ! (cours de 4ème année)

Et si l'oracle ne devine rien ?

- La réponse "oui" est accompagnée d'un certificat grace auquel on peut vérifier que la réponse est correcte

- Si la réponse est "non"...**il faut faire confiance !**
  - Ou alors vérifier avec un algorithme qui n'est pas polynomial

- Donc le problème suivant n'est pas forcément équivalent au problème du voyageur de commerce ?

co-*TSP*

donnée: Un ensemble de villes S, une matrice de distance $D : S \times S \mapsto \mathbb{N}$, un entier k

question: Est-ce qu'il **n'**existe **pas** de route de longueur au plus k passant par toutes les villes ?

Problème complémenaire

Le Problème complémenaire co-**A** d'un problème de décision **A**, est le problème qui associe la réponse "non" à toute donnée associée à "oui" dans **A**, et vice versa.

- Si **A** est dans $\mathbf{P}$ alors co-**A** est dans $\mathbf{P}$
  - Algorithme polynomial pour co-**A** : algorithme pour **A** + inversion de la réponse
- Est-ce que c'est vrai pour $\mathbf{NP}$?

Co-problème du voyageur de commerce

- **donnée** : ensemble de villes, entier k
- **question** : est-ce qu'il **n'**existe **aucun** chemin passant par toutes les villes, et de longueur inférieure à k ?

- Quel est le certificat ?
- Un chemin qui ne passe pas par toutes les villes, ou de longueur $> k$? **pas suffisant !**
- Lister tous les chemins ? **pas polynomial !**
- Le co-problème du voyageur de commerce ne semble pas avoir de certificat polynomial
- Les complémentaires de certain problèmes dans $\mathbf{NP}$ ne semblent pas être dans $\mathbf{NP}$

La classe coNP

- Le complémentaire de certain problèmes dans **NP** ne semblent pas être dans **NP**

coNP

Ensemble des problèmes de décision dont le problème complément est dans **NP**

Conjecture

$\mathbf{NP} \neq \mathbf{coNP}$?

Inclusion dans P-SPACE

Soit un algorithme non-déterministe en $\mathcal{O}(n^c)$ temps, et donc mémoire.

- il existe un algorithme déterministe qui n'utilise pas plus d'espace :
  - Le certificat nécessite un espace fini (vérifiable en $\mathcal{O}(n^c)$ pour une donnée de taille n)
  - Il y a donc un nombre fini de certificat : $\mathcal{O}(2^{n^c})$
  - On génère et verifie *tous* les certificats (en temps $\mathcal{O}(n^c 2^{n^c})$)

Théorème

$\mathbf{NP} \subseteq \mathbf{P\text{-}SPACE}$ et $\mathbf{coNP} \subseteq \mathbf{P\text{-}SPACE}$

Résumé : les classes de complexité