

Algorithmique et Complexité

Emmanuel Hebrard et Mohamed Siala

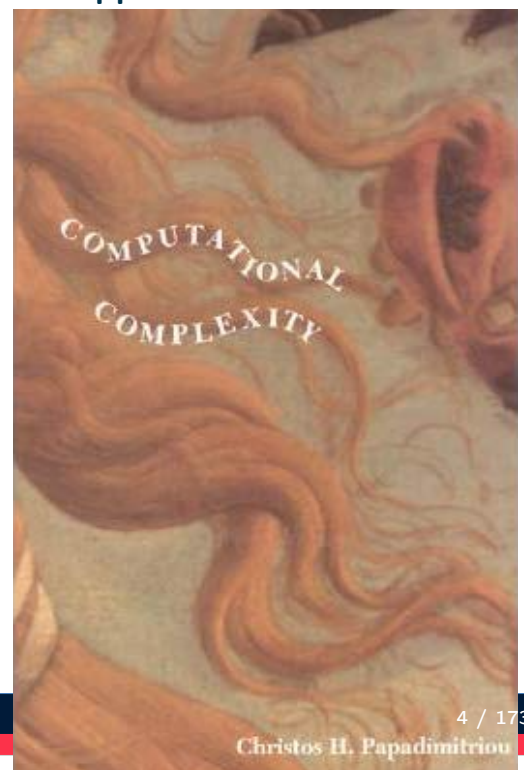
Plan

- 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 La Classe NP
- 9 Algorithmes de recherche

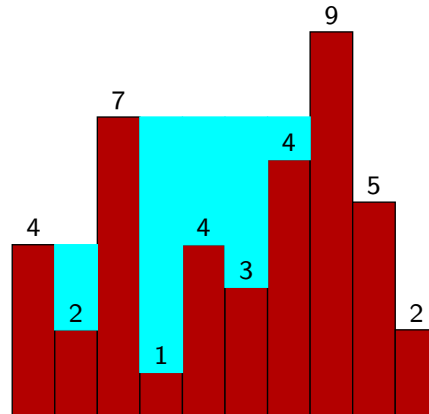
- Nos coordonnées :
 - ▶ Emmanuel Hebrard, Mohamed Siala
 - ▶ Mail : hebrard@laas.fr, siala@laas.fr
 - ▶ Page du cours : <https://moodle.insa-toulouse.fr/enrol/index.php?id=251>
 - ▶ Volume (prévu) :
 - ★ CM : 8 séance d'1h15
 - ★ TD : 9 séance d'1h15 (3 groupes : Emmanuel Hebrard, Marie-José Huguet, Mohamed Siala)
- Évaluation :
 - ▶ 1 examen à la fin du cours
 - ★ Les documents sont autorisés

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	$8 \times$ âge de l'univers		
35 villes	$5e+23 \times$ âge de l'univ.	5/1000s	
40 villes	$4e+31 \times$ âge de l'univ.	12 heures	
50 villes	$1,5e+48 \times$ âge de l'univ.	$4000 \times$ âge de l'univers	temps de planck
95 villes	$5e+131 \times$ âge de l'univ.	$1,3e+87 \times$ âge de l'univ.	$3 \times$ âge de l'univers

- 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

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

$$\begin{pmatrix} 1 & 1 \\ 2 & 4 \\ 3 & 9 \\ 4 & 16 \\ 5 & 25 \\ \dots & \dots \end{pmatrix}$$

- **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”

- 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, assignments, opérations arithmétiques, etc.*)

- Pour prouver qu'un algorithme est correct (terminaison + résultat attendu) on va souvent utiliser la notion d'**invariant de boucle**
- Invariant de boucle = Propriété maintenue par une boucle qui permet de prouver que l'algorithme est correct

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 q'une certaine propriété est maintenue (qui sert à montrer que la solution est correcte ou que l'algorithme termine).

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 tableau 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    $m \leftarrow i$ ;
3   pour  $j$  allant de  $i + 1$  à  $n$  faire
4     si  $L[j] < L[m]$  alors
5        $m \leftarrow j$ ;
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

- 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    $m \leftarrow i$ ;
3   pour  $j$  allant de  $i + 1$  à  $n$  faire
4     si  $L[j] < L[m]$  alors
5        $m \leftarrow j$ ;
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
$i = 2$	$\underbrace{0}_{\text{trié}}$ 30 17 9 29 24
$i = 3$	$\underbrace{0 \ 9}_{\text{trié}}$ 17 30 29 24
$i = 4$	$\underbrace{0 \ 9 \ 17}_{\text{trié}}$ 30 29 24
$i = 5$	$\underbrace{0 \ 9 \ 17 \ 24}_{\text{trié}}$ 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 i et $m \geq i$). Donc $trié(i)$ et $mins(i)$ impliquent $trié(i + 1)$.
 - ▶ A la ligne i , $L[m]$ est le plus petit élément parmi $L[i], \dots, L[n]$ ^a, et il est 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 ;

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 est la durée (en secondes, minutes, etc.) nécessaire au programme pour s'exécuter.

Mais le temps d'exécution 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 élémentaire 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 (+, ×, −), affectation, comparaisons sur les **types primitifs** (entiers, flottants, etc.)

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

Algorithme : Factorielle

Données : un entier n

Résultat : un entier valant $n!$

	nombre	coût
1 $fact \leftarrow 1;$	initialisation :	$1 \times$ 1 op.
2 pour i allant de 2 à n faire	itérations :	$n \times$ 1 op.
3 $fact \leftarrow fact * i;$	mult. + affect. :	$(n - 1) \times$ 2 op.
4 retourner $fact;$	retour fonction :	$1 \times$ 1 op.

Nombre total d'opérations :

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

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 :	$\sum_{i=1}^n (n - i) \times$ 1 op.
4 si $L[j] < L[m]$ alors	comparaison :	$\sum_{i=1}^n (n - i) \times$ 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 $L;$		

Nombre total d'opérations :

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

- 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 données distinctes ?
 - ▶ 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 connaît-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

$i \leftarrow 0$;

tant que $L[i] \neq e$ **faire**

$i \leftarrow i + 1$;

retourner i ;

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

Hyp. :

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

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

On applique la formule :

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

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

- $\text{Inf}_{\text{TriSélection}}(n) = n(n + 4)$, $\text{Sup}_{\text{TriSélection}}(n) = n(2n + 5)$
- 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 $2n^2 + 5n$
- $\text{Inf}_{\text{TriSélection}}(n) \simeq \text{Moy}_{\text{TriSélection}}(n) \simeq \text{Sup}_{\text{TriSélection}}(n) \simeq cn^2$

Exemple : TriRapide

Algorithme : TriRapide

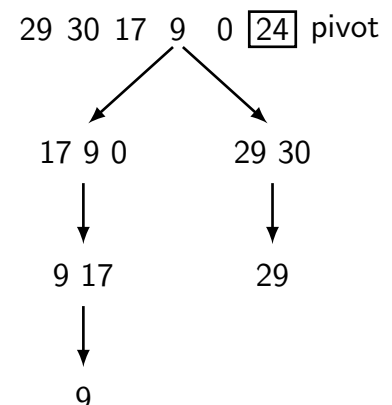
Données : tableau L d'elts comparables, entiers s, e
Résultat : le tableau trié entre les indices s et e

```

1 Procédure TriRapide( $L, s, e$ )
2   si  $s < e$  alors
3      $p \leftarrow \text{Partition}(L, s, e)$ ;
4     TriRapide( $L, s, p - 1$ );
5     TriRapide( $L, p + 1, e$ );
6 Fonction Partition( $L, s, e$ )
7    $pivot \leftarrow L[e]$ ;
8    $i \leftarrow s$ ;
9   pour  $j$  allant de  $s$  à  $e - 1$  faire
10    si  $L[j] < pivot$  alors
11      échanger  $L[i]$  avec  $L[j]$ ;
12       $i \leftarrow i + 1$ ;
13   échanger  $L[i]$  avec  $L[e]$ ;
14   retourner  $i$ ;
    
```

Invariants

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



Algorithme : TriRapide

Procédure TriRapide(L, s, e)

```

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

```

Procédure 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  $i$ ;

```

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

Procédure TriRapide(L, s, e)

```

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

```

Procédure 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 déjà 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 :

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

Algorithme : TriRapide

Procédure TriRapide(L, s, e)

```

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

```

Procédure 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

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 : $z_1 < z_2 < \dots < z_n$
- 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}, \dots, 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)

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

	TriSélection	TriRapide
Sup(n)	$c_1 n^2$	$c_2 n^2$
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, $\simeq \text{Moy}_s(n) = c_3 n^2$
 - ▶ $T^r(n)$ le temps effectif de calcul pour TriRapide de n éléments, $\simeq \text{Moy}_r(n) = c_4 n \ln n$
- Expérience : essayons pour $n = 100000$ et estimons $n = 300000$

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

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

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

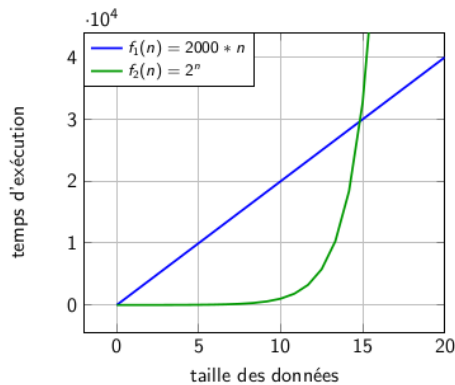
$$T^r(n) = \frac{T^r(100000)}{100000 \ln 100000} n \ln n \quad \text{pour } n = 300000 : \simeq 0.019$$

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

Analyse Asymptotique

Complexité Algorithmique

- Vision pessimiste : la **complexité** d'un algorithme 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 $\text{Coût}_A(x)$
- 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" ($n \gg 1000$)
 - ▶ 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} , Ω et Θ

Hypothèse simplificatrice

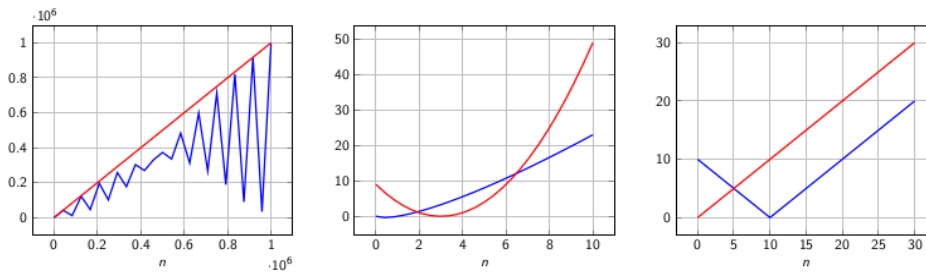
On ne s'intéresse qu'aux fonctions **asymptotiquement positives** (positives pour tout $n \geq n_0$)

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

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

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



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

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

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 : $f(n) \leq cg(n)$

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

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

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

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

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

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

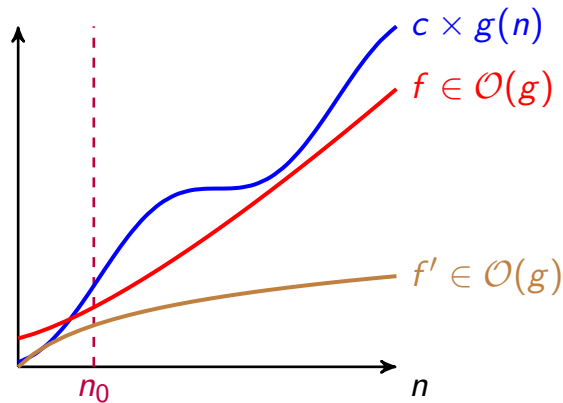
Tour du joueur \exists objectif : $f(n) \leq cg(n)$ $c = 7$ et $n_0 = 0$

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

Arbitre $6n^2 + 2n - 8 \leq 7n^2$ est toujours vrai

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

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



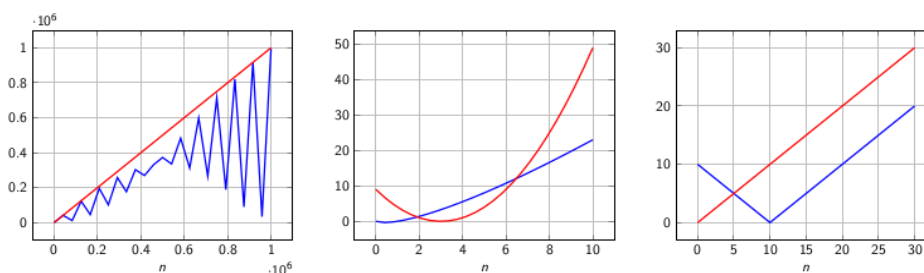
Exercice : $2n^2$ est-il en $\mathcal{O}(n^2)$? Pareil pour $2n$.

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

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

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



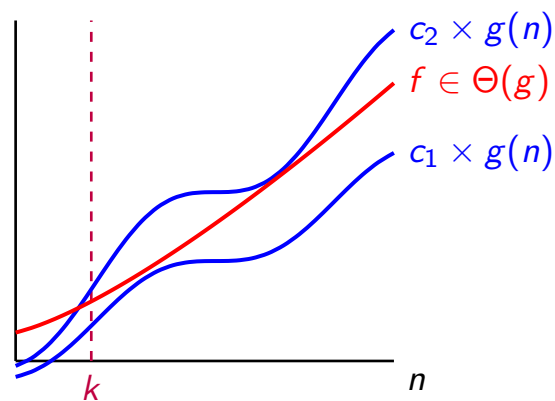
$\Theta(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

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

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 $c_1g(n)$ et $c_2g(n)$;

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

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



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

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

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

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

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

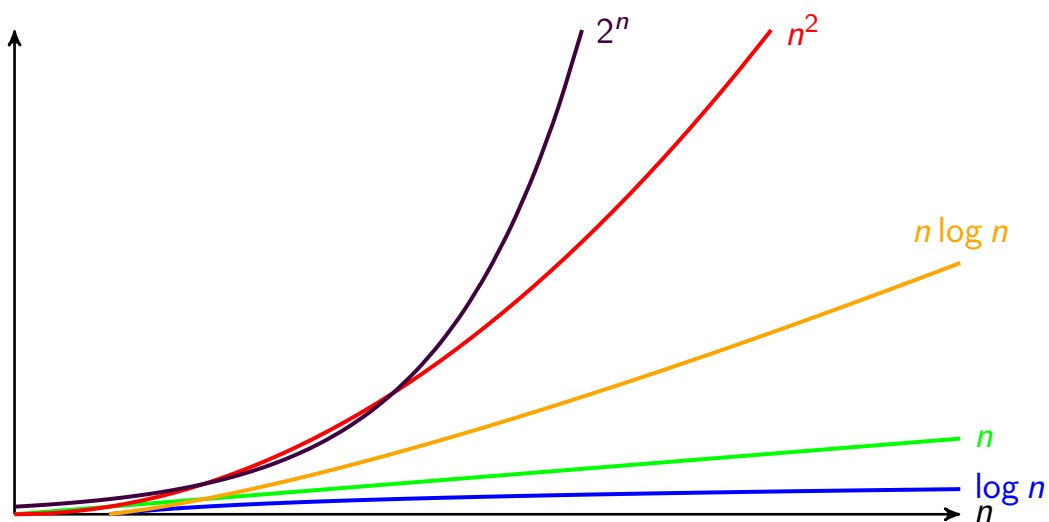
Exemple

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 $c_1n^3 \leq g(n) \leq c_2n^3$)
- 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 : $\mathcal{O}(1)/\Theta(1)$

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

Règle de l'Hôpital

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

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

f' (respectivement g') représente la dérivée 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 :

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

Exemple

$$\left. \begin{array}{l} \text{si } \langle \text{condition} \rangle \text{ alors} \\ \quad \# \text{instructions (1);} \\ \text{sinon} \\ \quad \# \text{instructions (2);} \end{array} \right\} = \Theta(g(n) + f_1(n) + f_2(n))$$

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

$$\left. \begin{array}{l} \text{tant que } \langle \text{condition} \rangle \text{ faire} \\ \quad \# \text{instructions ;} \end{array} \right\} = \Theta(h(n) \times (g(n) + f(n)))$$

- 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 ;

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

- 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 vu ;
 - ★ 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 $n!$

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

Séries arithmétiques

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

Nombre total d'opérations : $\Theta(n^2) = \Theta(|L|^2)$

Algorithmes Récursifs



Force brute

- L'approche "force brute" : énumération exhaustive de toutes les configurations possibles de la solution recherchée
- **Exemple** : Un algorithme de tri de type "force brute" génère toutes les permutations possibles de la liste jusqu'à trouver la permutation ordonnée.
- **Inefficace!**
- **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” est une méthode de conception d’algorithmes qui se base sur une “conception par décomposition” :
 - ▶ Diviser le problème en sous problèmes plus facile à résoudre
 - ▶ Combiner les résultats des sous problèmes pour résoudre le problème initial
- Cas idéal : le problème est décomposable en sous-problèmes *indépendants*
 - ▶ Dans ce cas, combiner les résultats est trivial
 - ▶ Parfois, les sous-problèmes sont seulement plus “faiblement” liés, et combiner les résultats peut-être complexe
- Algorithmes typiquement *récur­sifs*

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

$mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;

$L_l \leftarrow \text{TriFusion}(L[:mil])$;

$L_r \leftarrow \text{TriFusion}(L[mil:])$;

 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

Algorithme : TriFusion (L)

Données : une liste L

Résultat : la liste L triée

mil : entier;

si $|L| \leq 1$ alors

 retourner L ;

sinon

$mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;

$L_l \leftarrow \text{TriFusion}(L[:mil])$;

$L_r \leftarrow \text{TriFusion}(L[mil:])$;

 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 : $\langle 6 \rangle \langle 2 \rangle$
- ▶ Fusion : $\langle 2, 6 \rangle$
- ▶ Division : $\langle 1 \rangle \langle 8 \rangle$
- ▶ Fusion : $\langle 1, 8 \rangle$
- ▶ Fusion : $\langle 1, 2, 6, 8 \rangle$
- ▶ Division : $\langle 5, 4 \rangle \langle 3, 7 \rangle$
- ▶ Division : $\langle 5 \rangle \langle 4 \rangle$
- ▶ Fusion : $\langle 4, 5 \rangle$
- ▶ Division : $\langle 3 \rangle \langle 7 \rangle$
- ▶ Fusion : $\langle 3, 7 \rangle$
- ▶ Fusion : $\langle 3, 4, 5, 7 \rangle$
- ▶ Fusion : $\langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$

● 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 $|L[mil:]| \leq n$
- ▶ \implies les préconditions de Fusion sont respectées, montrent 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 $L[k-1] \leq L_2[j]$

La structure d'un algorithme récursif $\text{AlgoRec}(x)$ est :

si *condition d'arrêt* alors

 retourner *solution triviale*;

sinon

 retourner $\text{Fusion}(\text{AlgoRec}(p(x,1)), \text{AlgoRec}(p(x,2)), \dots, \text{AlgoRec}(p(x,a)))$;

- 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 *expression fermée (ou explicite)* de la complexité

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

- 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 de TriFusion :

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

Forme fermée de la complexité de TriFusion :

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

- 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 $n_0 \Rightarrow$ 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 :

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

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

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

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

Exemple

Algorithme : TriFusion (L)

Données : une liste L

Résultat : la liste L triée

si $|L| \leq 1$ alors

 retourner L ;

sinon

$mil \leftarrow \lfloor \frac{|L|+1}{2} \rfloor$;

$L_l \leftarrow \text{TriFusion}(L[:mil])$;

$L_r \leftarrow \text{TriFusion}(L[mil:])$;

 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 $\Theta(n^d)$

Exemple

Algorithme : RechBin (L)

Données : tableau trié L contenant e

Résultat : la position de e dans L

$m \leftarrow \lfloor \frac{|L|}{2} \rfloor$;

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 $d > \log_b a, T(n) = \Theta(n^d)$
- 2 Si $d < \log_b a, T(n) = \Theta(n^{\log_b a})$
- 3 Si $d = \log_b a, T(n) = \Theta(n^d \log n)$

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

complexité dominée par le coût de fusion
complexité dominée par le coût du sous-problème
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

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

- 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. La matrice résultante est de taille $t_0 \times t_2$.

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**

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

Force brute

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

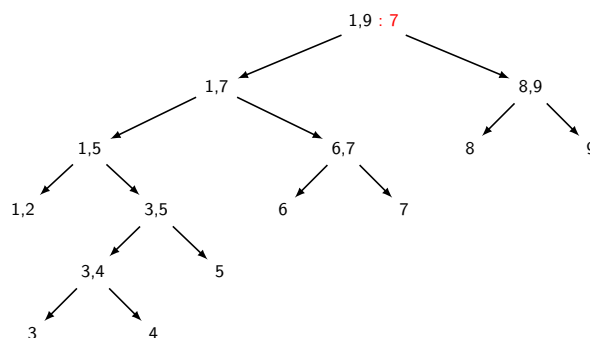
Combien de Parenthésages Possibles ?

- 3 matrices $A_1 * A_2 * A_3 \rightarrow 2$ possibilités : $(A_1 * A_2) * A_3$ et $A_1 * (A_2 * A_3)$
- Pour 4 matrices $\rightarrow 5$ possibilités
 - ▶ $((A_1 * A_2) * A_3) * A_4, (A_1 * (A_2 * A_3)) * A_4, (A_1 * A_2) * (A_3 * A_4),$
 - ▶ $A_1 * ((A_2 * A_3) * A_4)$ et $A_1 * (A_2 * (A_3 * A_4))$
- 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)
- $C(n) \in \Omega\left(\frac{4^n}{n^{1.5}}\right)$
- **La complexité de l'approche force brute est $\Omega\left(\frac{4^n}{n^{1.5}}\right)$**

Solution Récursive (Diviser pour Régner)

- Un parenthésage peut se voir comme un arbre :

$$(((A_1 * A_2) * ((A_3 * A_4) * A_5)) * (A_6 * A_7)) * (A_8 * A_9)$$



- Il y a $n-1$ possibilités pour la racine de l'arbre : *on peut résoudre récursivement les $n-1$ sous-problèmes, et choisir le meilleur*

- Pour chaque solution, il faut découper la séquence A_1, \dots, A_n en deux sous séquences $A_1 \dots A_k$ et $A_{k+1} \dots A_n$ (le calcul sera $(A_1 \times \dots \times A_k) * (A_{k+1} \times \dots \times A_n)$)
- 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 $(A_1 \dots A_k) \times (A_{k+1} \dots A_n)$, le coût en fonction de k est :

$$\text{cout}(k) = m[1][k] + m[k+1][n] + t_0 \times t_k \times t_n$$

$$\implies m[1][n] = \min\{m[1][k] + m[k+1][n] + t_0 \times t_k \times t_n \mid 1 \leq k < n\}$$

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

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_recuratif(A1 ... Ak) + cout_recuratif(Ak+1, An) + t0 × tk × tn ;
            si tmp < c alors
                c ← tmp;
        retourner c ;

```

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

Avec c_1 et c_2 deux constantes

- Comment calculer la complexité sous une forme non-réursive ?
- Le théorème maître ne s'applique pas
- On va utiliser la méthode par substitution. D'abord on simplifie la récursion

$$T(n) = \sum_{k=1}^{n-1} (T(k) + T(n-k) + c_2) = c_2 n + \sum_{k=1}^{n-1} (T(k) + T(n-k))$$

$$T(n) = c_2 n + \begin{array}{r} T(1) + T(n-1)T(1) + \\ T(2) + T(n-2)T(2) + \\ \dots + \dots + \\ T(n-1) + T(1)T(n-1) \end{array}$$

$$T(n) = c_2 n + 2 \sum_{k=1}^{n-1} T(k)$$

$$\begin{aligned} T(n) - T(n-1) &= c_2 n - c_2(n-1) + 2 \sum_{k=1}^{n-1} T(k) - 2 \sum_{k=1}^{n-2} T(k) \\ &= c_2 + 2T(n-1) \end{aligned}$$

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

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

- Arbre de profondeur n et facteur de branchement 3 : $T(n) \in \mathcal{O}(3^n)$
- Hypothèse de récurrence : $T(n) \leq c3^n$
- $T(1) = c3^{1-1} = c$, on choisit $c = c_1$
- On suppose que $T(x) \leq c_1 3^x$ est vrai pour tout $1 \leq x \leq n-1$;
- On vérifie que c'est aussi le cas pour $x = n$:

$$\begin{aligned} T(n) &= 3T(n-1) \\ &\leq 3c_1 3^{n-1} \\ &\leq c_1 3^n \end{aligned}$$

- Même preuve pour $T(n) \in \Omega(n^3)$, donc $T(n) \in \Theta(n^3)$

Comparaison : Force Brute / Solution Récursive

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

Règle de l'Hôpital

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

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

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

Comparaison : Force Brute / Solution Récursive

- 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 $\lim_{n \rightarrow \infty} \frac{\frac{4^n}{n^{1.5}}}{3^n}$?
 - ▶ $\frac{\frac{4^n}{n^{1.5}}}{3^n} = \frac{4^n}{3^n n^{1.5}}$
 - ▶ On utilise la règle de l'Hôpital :

$$\lim_{n \rightarrow \infty} \frac{(\frac{4}{3})^n}{(n^{1.5})^3} = \lim_{n \rightarrow \infty} \frac{\ln(\frac{4}{3}) (\frac{4}{3})^n}{1.5 n^{0.5}} = \lim_{n \rightarrow \infty} \frac{(\ln(\frac{4}{3}) (\frac{4}{3})^n)'}{(1.5 n^{0.5})'} = \lim_{n \rightarrow \infty} \frac{\ln(\frac{4}{3}) * \ln(\frac{4}{3}) (\frac{4}{3})^n}{1.5 * 0.5 * n^{-0.5}} = \infty$$
 - ▶ Donc $\lim_{n \rightarrow \infty} \frac{\frac{4^n}{n^{1.5}}}{3^n} = \infty$ 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

Algorithme : $\text{cout_recursif}(A_1 \dots A_n)$

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(A1 ... Ak) + cout_recursif(Ak+1, An) + t0 × tk × tn;
    | | si tmp < c alors
    | | | c ← tmp;
    | retourner c ;
  
```

- $\text{cout_recursif}(A_i, A_j)$ est appelé plusieurs fois (e.g. on appelle $\text{cout_recursif}(A_4, A_n)$ quand $k = 1, k = 2,$ et $k = 3$).
- L'algorithme récursif fait beaucoup de calculs redondants! on peut l'améliorer
- \Rightarrow **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 $P(i)$

$$m[i][j] = \min\{m[i][k] + m[k+1][j] + t_{i-1} \times t_k \times t_j \mid i \leq k < j\}$$

- Si on connaît $m[i][k]$ et $m[k+1][j]$ (pour tout $k \in [i, j-1]$), alors on peut calculer $m[i][j]$ efficacement
- Il n'y a que $n(n-1)/2$ séquences en tout
- 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 : $m[1][2], m[2][3], \dots, m[n-1][n]$
 - 2 Puis le coût des séquences de taille 3 : $m[1][3], m[2][4], \dots, m[n-2][n]$
 - 3 ...
 - 4 Finalement le coût de la séquence de taille n (coût de la solution optimale) : $m[1][n]$

Algorithme : CoutMultiplication_ProgDynamique

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 $A_1 \times \dots \times A_n$

l, tmp : entier;

$m[][]$: matrice de taille $n \times n$;

/ $m[i][j]$: coût min pour A_i, \dots, A_j ($i < j$) */*

pour i de 1 à n **faire**

$m[i][i] = 0$;

/ Initialisation */*

pour l de 2 à n **faire**

pour i de 1 à $n-l+1$ **faire**

$j \leftarrow i+l-1$;

/ séquence de longueur l qui commence en A_i */*

$m[i][j] \leftarrow \infty$;

pour k de i à $j-1$ **faire**

$tmp \leftarrow m[i][k] + m[k+1][j] + t_{i-1} \times t_k \times t_j$;

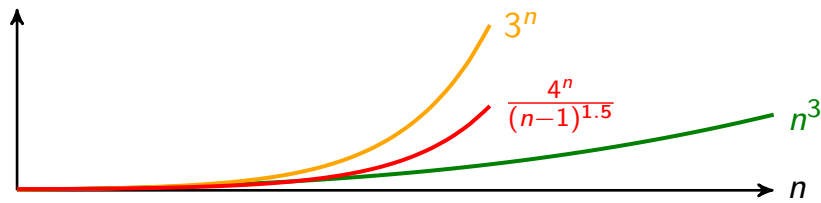
si $tmp < m[i][j]$ **alors**

$m[i][j] \leftarrow tmp$;

retourner $m[1][n]$

Complexité de CoutMultiplication_ProgDynamique

- $\Theta(n^2)$ séquences, $\Theta(n)$ pour calculer chaque séquence
- $\text{CoutMultiplication_ProgDynamique} \in \Theta(n^3)$
- Meilleur que l'algorithme récursif ($\Theta(3^n)$) et la force brute ($\Omega(\frac{4^n}{(n-1)^{1.5}})$)



Programmation Dynamique : Résumé

- Diviser le problème en sous-problèmes
- **Si le nombre de sous-problèmes n'est pas trop grand** : réserver la mémoire pour stocker la solution de chaque sous-problème
- Trouver un ordre permettant de résoudre chaque sous-problème qu'une seule fois, en utilisant les solutions stockées

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., parenthésage 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 la structure de représentation "matroïdes" qui permet de concevoir des approches gloutonnes *optimales*

- Contexte : typiquement pour les problèmes d'optimisation
- Principe :
 - ▶ Résoudre le problème en une séquence d'étapes/choix : **arbre de décision**
 - ★ Explorer toutes les branches (Algorithmes récursifs)
 - ★ N'explorer qu'une fois les branches isomorphes (Programmation Dynamique)
 - ★ **N'explorer qu'une branche!** (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 ?

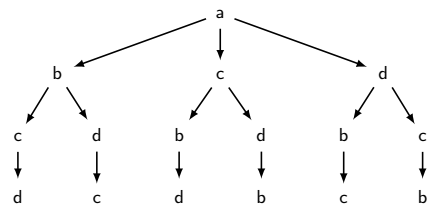
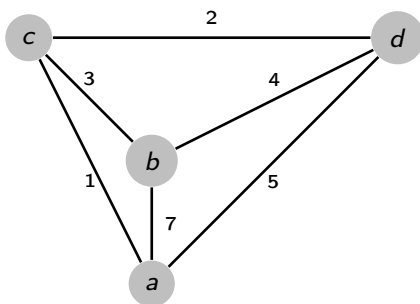
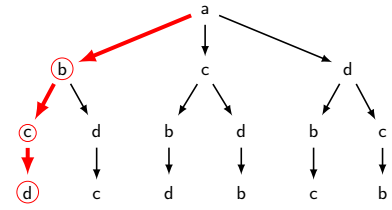
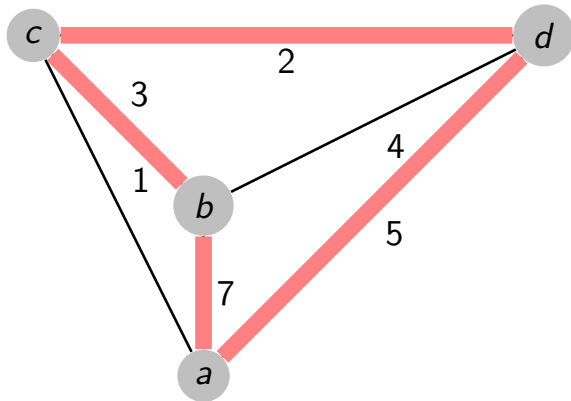
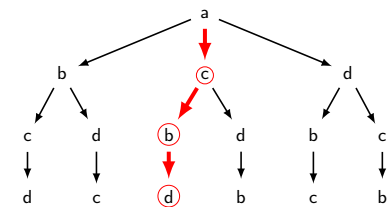
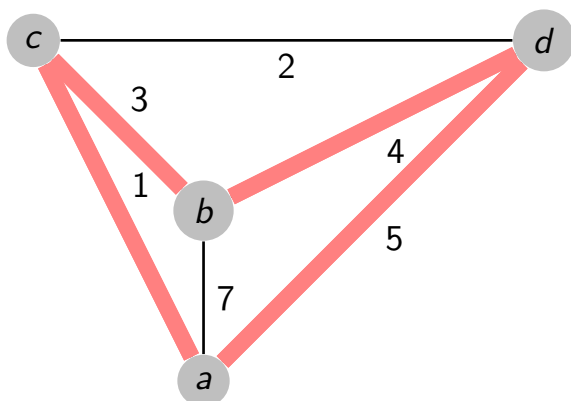


Figure – Une solution non optimale



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

Figure – Une solution optimale

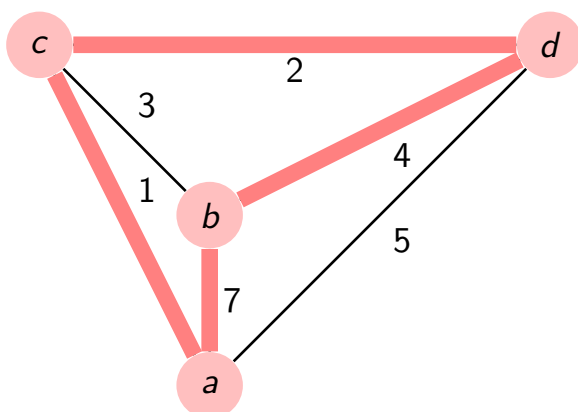


- Cycle : a,c,b,d,a
- Coût de la solution : $1 + 3 + 4 + 5 = 13$

- 2 Villes \implies 1
- 3 Villes \implies 1
- 4 Villes \implies 3
- 5 Villes \implies 12
- n Villes $\implies \frac{(n-1)!}{2}$ (la moitié du nombre de permutations possible de taille $n - 1$)
- 40 villes \implies 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 !!

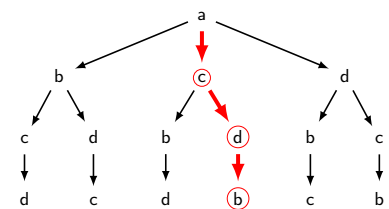
- Idée gloutonne : Choisir la ville non visitée la plus proche

Figure – Construction de la solution gloutonne



- Chemin actuel : a, c, d, b
- Coût actuel : $0 + 1 + 2 + 4 + 7 = 14$

Algorithme Glouton



Solution suboptimale

Mais algorithme efficace : $\mathcal{O}(n^2)$ contre $\Omega(n!)$ pour l'exploration complète

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 $1, \dots, n$

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 : $O(n^2)$

Cas favorable

- Dans certains cas, on peut montrer que la solution gloutonne est optimale !
- Lorsque l'espace de recherche (l'ensemble des sommets de l'arbre de recherche) possède une structure de **matroïde**
- Dans ce cas, il est possible de résoudre en temps polynomial un problème dont l'espace de recherche est (sur)exponentiel

- 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)
- Soit $\mathcal{M} = (\mathcal{E}, \mathcal{I})$ un matroïde et $H \in \mathcal{I}$, alors H est appelé "sous ensemble indépendant"
- Une **base** de $\mathcal{M} = (\mathcal{E}, \mathcal{I})$ est un sous ensemble indépendant maximal par inclusion
 - ▶ $F \in \mathcal{I}$ est une base de \mathcal{M} si et seulement si $\forall x \in \mathcal{E} \setminus F, F \cup \{x\} \notin \mathcal{I}$

Exemple (simple) de Matroïde

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

- \mathcal{E} : les arêtes E d'un graphe $G = (V, E)$
- \mathcal{I} : les sous ensembles de E qui sont des *forêts* (graphes sans cycle)
- Preuve
 - ▶ \mathcal{E} est un ensemble fini non vide (l'ensemble vide est une forêt)
 - ▶ \mathcal{I} est héréditaire car une forêt dont on retire une arête reste une forêt
 - ▶ Propriété d'échange :
 - ★ Soit I et J deux forêts telles que $|I| < |J|$, et considérons le graphe formé par les arêtes $I \cup J$
 - ★ Chaque cycle de ce graphe comporte au moins une arête dans $I \setminus J$ et une arête dans $J \setminus I$
 - ★ Pour chaque cycle, retirons la deuxième arête (celle de $J \setminus I$)
 - ★ Le graphe résultant contient I et n'a pas de cycle (et donc appartient à \mathcal{I})
 - ★ Il possède au moins une arête de $J \setminus I$ puisque qu'on en a retiré au plus $|I \setminus J| < |J \setminus I|$
- Les bases de $\mathcal{M} = (\mathcal{E}, \mathcal{I})$ sont des *arbres couvrants*

- 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 la base optimale :**
 - ▶ Donnée : $\mathcal{M} = (\mathcal{E}, \mathcal{I})$: matroïde et w : fonction de poids
 - ▶ Question : Trouver la base $F \in \mathcal{I}$ de poids minimal ou maximal

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 ← {};
    n ← |E| ;
    L ← Trier(E) par poids décroissant ;
    pour i ∈ [1..n] faire
        si F ∪ {L[i]} ∈ I alors
            F ← F ∪ {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 $O(n \log(n) + nf(n))$ avec $n = |\mathcal{E}|$ (car le tri peut se faire en $O(n \log(n))$).

Théorème

Glouton ($\mathcal{M}(\mathcal{E}, \mathcal{I})$, w) retourne une base optimale de \mathcal{M}

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 maxminimal**

début

```

    F ← {};
    n ← |E| ;
    L ← Trier(E) par poids décroissant ;
    pour i ∈ [1..n] faire
        si F ∪ {L[i]} ∈ I alors
            F ← F ∪ {L[i]} ;
    retourner F ;

```

- Supposons qu'on veuille mettre toutes les machines de l'INSA en réseau en minimisant la longueur de câble nécessaire
- Pour n machines, il y a n^{n-2} câblages tels qu'il existe un chemin unique entre toute paire de machines
 - ▶ Pour 50 machines : $3,5 \cdot 10^{81}$ câblages possibles
- **Arbre couvrant de poids minimum !**

Algorithme : Kruskal ($G = (V, E)$, w) = **Glouton sur les forêts de G**

Données : $G = (V, E)$: graphe, w : fonction de longueur des arêtes

Résultat : Arbre couvrant de poids minimum

début

```

  F ← {};
  n ← |E| ;
  L ← Trier(E) par poids croissant ;
  pour i ∈ [1..n] faire
    si F ∪ {L[i]} ne contient pas de cycle alors
      F ← F ∪ {L[i]} ;
  retourner F ;

```

- Rappel pour un problème d'optimisation :
 - ▶ **Une solution** est une sortie qui respecte les exigences du problème
 - ▶ Le **coût** d'une solution est la valeur correspondante à la fonction objectif
 - ▶ **Une solution optimale** est une solution qui optimise l'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 une base (i.e., élément maximal de \mathcal{I}) de poids optimal que l'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 s'applique pas à tous les problèmes. En particulier, il y a souvent deux limites :
 - 1 Le problème n'a pas une structure de matroïde
 - 2 Le test d'appartenance ($F \in \mathcal{I}$?) peut être coûteux en temps (e.g. en $\mathcal{O}(2^n)$)

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 ?**

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,...

0	1	0	0	1	0	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

- Représenter les entiers entre 0 et 255 (char)
- Chaque bit représente 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 inférieure)

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

0	0	0	0	0	0	0	1	0	0	1	0	1	1
$\underbrace{\hspace{10em}}_{\text{N}^\circ \text{ de } 0 = \text{N}^\circ \text{ de bits} = 7}$							2^6	2^5	2^4	2^3	2^2	2^1	2^0

- 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 $|n| \in \mathcal{O}(\log n)$

8	13	0	3	-18	2	9	11	-4
<i>size</i>	$L[0]$	$L[1]$	$L[2]$	$L[3]$	$L[4]$	$L[5]$	$L[6]$	$L[7]$

- 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 *indépendants* 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

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 donnée $x \in \mathcal{T}$ est telle que $2^{|x|} \geq \#(\mathcal{T})$, \Rightarrow

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

- 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 $\Theta(g(x))$, et donc $x \in \Theta(g^{-1}(|x|))$
- 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 ← 0;
  pour i allant de 1 à x faire
    pour j allant de 1 à x faire
      r ← 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 !)

La manière dont les données (et les variables auxiliaires) sont représentées en mémoire a souvent un impact direct sur la complexité d'un algorithme

- Que choisir entre un `std::vector`, `std::list`, `std::map`,... ?
- Le meilleur choix dépend du contexte !
- Choisir la structure pour laquelle les opérations *utiles* sont les plus *efficaces*

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

- 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 chaînée	$\mathcal{O}(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\mathcal{O}(1)$	$\Theta(n)$
File	Liste chaînée avec pointeurs <i>début</i> et <i>fin</i>	$\mathcal{O}(1)$	$\Theta(n)$	$\Theta(n)$	$\mathcal{O}(1)$	$\Theta(n)$	$\Theta(n)$
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

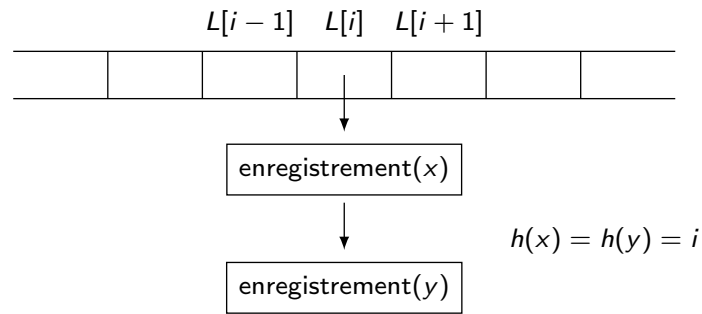
Fichier des étudiants de l'INSA, accéder rapidement aux fichier d'un étudiant qui en fait la demande : **II faut une clé unique pour identification**

- Tatouer chaque étudiant avec son numéro d'inscrit $(1, \dots, n)$?
 - ▶ Une table L avec $L[x]$ 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 $h : U \mapsto \{1, \dots, m\}$ 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 $L[h(x)]$. Si $h(x) = h(y)$ et $x \neq y$ on dit qu'il y a une *collision*

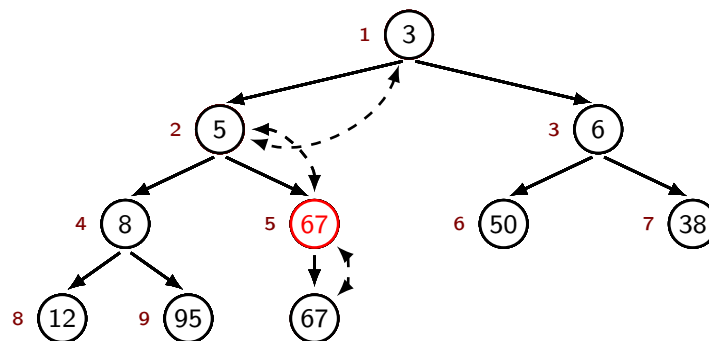
Exemple : Table de Hâchage



- 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 $T_{\text{moy}}(n)$ on suppose une distribution uniforme des valeurs de $h(x)$ dans $\{1, \dots, m\}$
 - ▶ Soit $|L[i]|$ la longueur de la liste $L[i]$, 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 $T_{\text{moy}}(n) = E(|L[i]|) = n/m \in \mathcal{O}(1)$

Exemple : Tas Binaire

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



réalisation :

1	2	3	4	5	6	7	8	9	10
2	2	6	8	6	5	3	1	9	3

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 : $\mathcal{O}(n \log n)$ (égale à celle de TriFusion)

Classes de Complexité

- Pour pouvoir analyser objectivement un algorithme
 - ▶ Optimalité

- Parce que la difficulté du problème détermine le type de méthode
 - ▶ *solutions adaptées aux problèmes difficiles*

- Parce que la difficulté du problème est parfois une garantie
 - ▶ Cryptographie, Block Chain

Définition : Problème \simeq fonction

- 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 : $x_1, u_1, u_2, \dots, u_k, x_2$

- Q_{carr} : Entier : $x \mapsto$ Entier : x^2

- 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 Q

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

- Pour un problème d'optimisation, on peut généralement définir un problème **polynomialement** équivalent dont la réponse est dans {oui, 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 ?

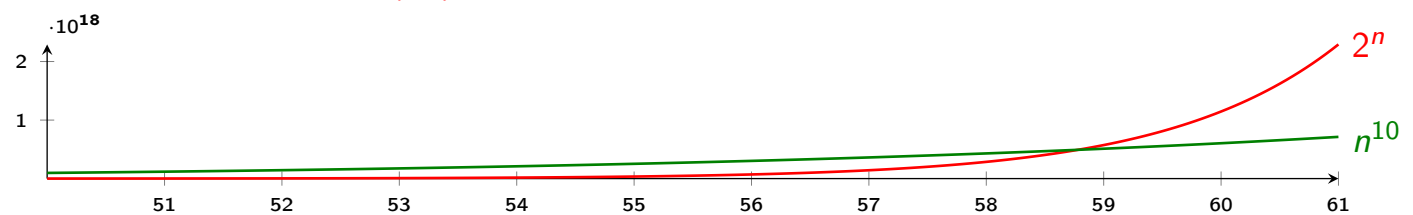
- Comment évaluer la complexité d'un problème ?

La complexité d'un problème :

La complexité du meilleur algorithme pour le résoudre

Un algorithme est 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 $c > 1$



Borne inférieure pour les algorithmes de tri

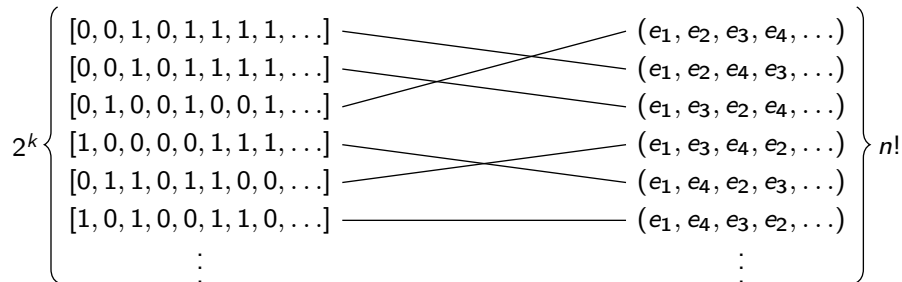
Tri par comparaison

- **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 exécution, 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 déterministe, 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

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

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 $\Omega(n \log n)$
 - ▶ **TriParTas et TriFusion sont optimaux !**

Classes "Temps Polynômial" et "Temps Exponentiel"

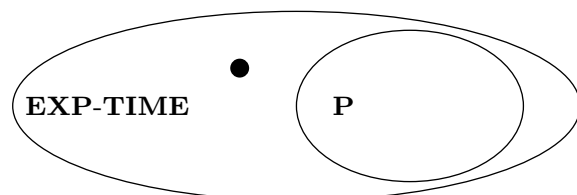
P-TIME ou simplement : P

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

EXP-TIME :

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

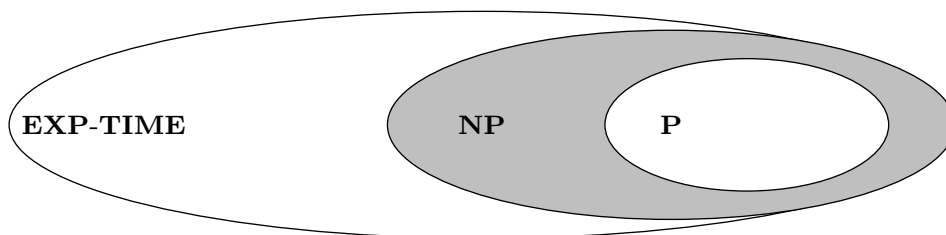
- Evidemment, $\mathbf{P} \subseteq \mathbf{EXP-TIME}$
- Est-ce que $\mathbf{P} \subset \mathbf{EXP-TIME}$? **Oui!**
- Il existe des problèmes en $\Omega(2^n)$



La Classe NP



Problèmes entre P et P-SPACE



- *Faciles à vérifier*
- Certains sont *difficiles* (pas d'algorithme polynomial connu)

Classe NP "Non-deterministe Polynomial"

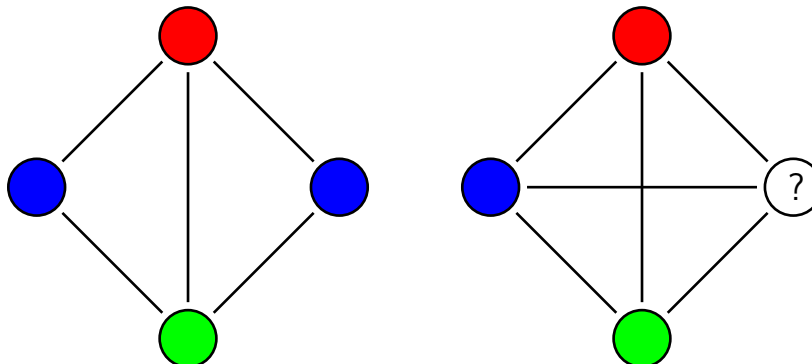
Ensemble des problèmes de décision tels que lorsque la réponse est "oui", il existe un *certificat* que l'on peut utiliser pour vérifier cette réponse en temps polynomial.

- Le certificat doit permettre de **prouver** que "oui" est bien la réponse correcte
 - ▶ Il faut pouvoir coder le certificat en espace **polynomial** dans la taille de la donnée
 - ▶ Il faut pouvoir faire la démonstration en temps **polynomial** dans la taille de la donnée

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.



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

Exemple : Le problème de *Factorisation*

Factorisation

donnée: Un nombre entier x

question: Donner la décomposition en facteurs premier de x

\approx

FacteurMinimum

donnée: Un nombre entier x

question: x admet-il un facteur premier inférieur à k

- On peut factoriser en temps polynomial s'il existe un algorithme polynomial pour le problème de décision :
 - ▶ Recherche binaire, $O(\log x)$ résolutions de *FacteurMinimum* pour trouver le facteur premier minimum p
 - ▶ Méthode récursive : résoudre *Factorisation*(x/p) tant que $x > 1$. La décomposition a $O(\log x)$ facteurs, donc au plus $\log^2 x = |\log x|^2$ appels à un algorithme polynomial en $|x|$

Autrement dit $FacteurMinimum \in \mathbf{P} \iff Factorisation \in \mathbf{P}$

Quels sont les facteurs premiers de 218633 ? $218633 = 19 \times 37 \times 311$

- Certificat : la décomposition en facteurs premiers
 - ▶ Tester si un nombre n est premier est polynomial (*Algorithme AKS* en $(\log^{12} n)$)
 - ▶ Tester si le produit des facteurs est bien égal à x est polynomial (multiplication en $O(\log^2 x)$)
 - ▶ Tester si un des facteurs est inférieur à k est polynomial

Factorisation est difficile ?

Ce problème n'est pas (à l'heure actuelle) dans **P** : on ne connaît pas d'algorithme polynomial pour le résoudre

Système d'authentification A

- comparaison entre une *clé privée* y (mot de passe / pin) et une *clé publique* x (login / carte bancaire)
- autorise la transaction si et seulement si $A(x) = y$
- Supposons que calculer A soit polynomial : **pirater le système serait facile !**
- Supposons que vérifier $A(x) = y$ ne soit pas polynomial : **authentifier serait difficile !**
- A doit être un problème "difficile" dans NP

Système d'authentification RSA (Rivest–Shamir–Adleman) : $A = \text{Factorisation}$

La clé publique x est le produit de deux grand nombres premiers y_1, y_2 qui sont gardés privés.

- Authentification : vérifier $\text{Factorisation}(x) = y_1, y_2$ est facile
- Sûreté : trouver y_1, y_2 étant donné x est difficile

- **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 ?
 - ▶ On ne connaît pas d’algorithme polynomial pour *3-coloration*, *Factorisation* ou *Voyageur de commerce*
 - ▶ Mais personne ne sait s’il en existe !

Conjecture $P \neq NP$

- Un des 7 “problèmes du millénaire” du Clay Mathematics Institute, prix de \$1 000 000
 - ▶ Conjecture de Poincaré (résolue par G. Perelman)
 - ▶ Conjecture de Birch and Swinnerton-Dyer
 - ▶ Conjecture de Hodge
 - ▶ Solutions des équations de Navier-Stokes
 - ▶ **P** versus **NP**
 - ▶ Hypothèse de Riemann
 - ▶ Existence de Yang-Mills et de l’écart de masse

- Preuve de $P \neq NP$: un problème dans **NP** mais pas dans **P**
 - ▶ Montrer qu’un problème est dans **NP** est facile : certificat polynomial
 - ▶ Montrer qu’un problème n’est pas dans **P** est difficile : tout algorithme est en $\Omega(2^n)$
- Preuve de $P = NP$: un algorithme polynomial pour un problème **NP-complet**

Problème NP-complet

Un problème est **NP-complet** si et seulement si :

- Il appartient à **NP**
- *Il est au moins aussi difficile que tout les autres problèmes dans NP (si ce problème était facile, alors tous les problèmes dans NP seraient faciles)*
- Comment peut-on dire qu’un problème *A* est *au moins aussi difficile* qu’un problème *B* ?
- Il existe des problèmes **NP-complets**
 - ▶ Le problème *SAT* est **NP-complet**, Le problème *3-coloration* est **NP-complet**
 - ▶ *Factorisation* n’est pas connu pour être **NP-complet**
 - ▶ On essaie de montrer que s’il existe un algorithme polynomial pour *A*, alors il existe un algorithme polynomial pour *SAT* ($A \in P \implies SAT \in P \implies P = NP$)

- Ces problèmes sont partout : en intelligence artificielle, en cryptographie, dans l'industrie...
- Il n'existe pas de méthode efficace *dans le pire des cas*, mais il existe des méthodes efficaces *en pratique*
 - ▶ Des algorithmes "intelligents" peuvent résoudre (optimalement) de très grandes instances

Algorithmes de recherche

Comment résoudre un problème NP-complet ?

- Si $P \neq NP$ (et tant que le contraire n'est pas prouvé) il n'existe pas d'algorithme efficace sur *toutes* les instances
- Il faut donc *chercher* parmi toutes les solutions possibles : *l'espace de recherche*
- ... ou accepter de ne pas résoudre toutes les instances de façon exacte (approximation)
- Mais beaucoup de cas sont favorables, on peut faire beaucoup mieux que "Generate and test" !
- *Recherche arborescente*

Rappel : Coloration de graphes

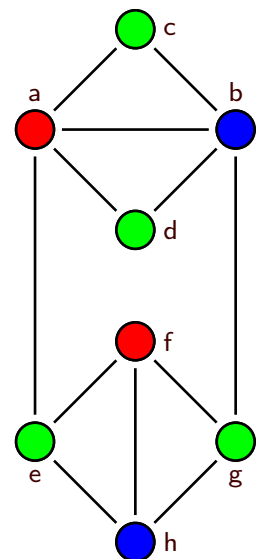
Nombre Chromatique

$\chi(G)$ est le plus petit entier tel que chaque sommet de G peut être associé à une couleur parmi $\chi(G)$ sans que les deux sommets d'une même arête ne partagent la même couleur.

Coloration

donnée: Un graphe $G = (S, A)$

question: Quel est le nombre chromatique $\chi(G)$ de G ?



Par couleur

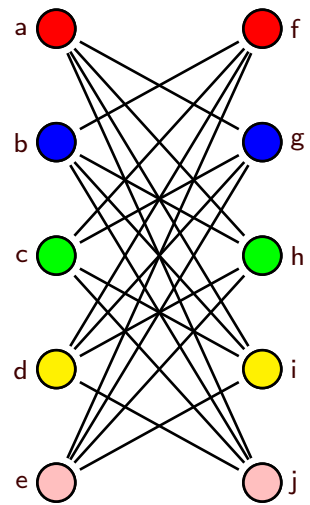
- Assigne la première couleur à un nombre maximal de sommets
- Heuristique : par ordre décroissant de degré

Peut surestimer $\chi(G)$ par un facteur $\Theta(n)$

Par sommet

- Explore les sommets dans un certain ordre et assigne la première couleur disponible pour ce sommet
- Heuristique : par ordre décroissant de degré *de saturation* (nombre de couleurs parmi les voisins)

Peut surestimer $\chi(G)$ par un facteur $\Theta(n)$



Modèle et espace de recherche

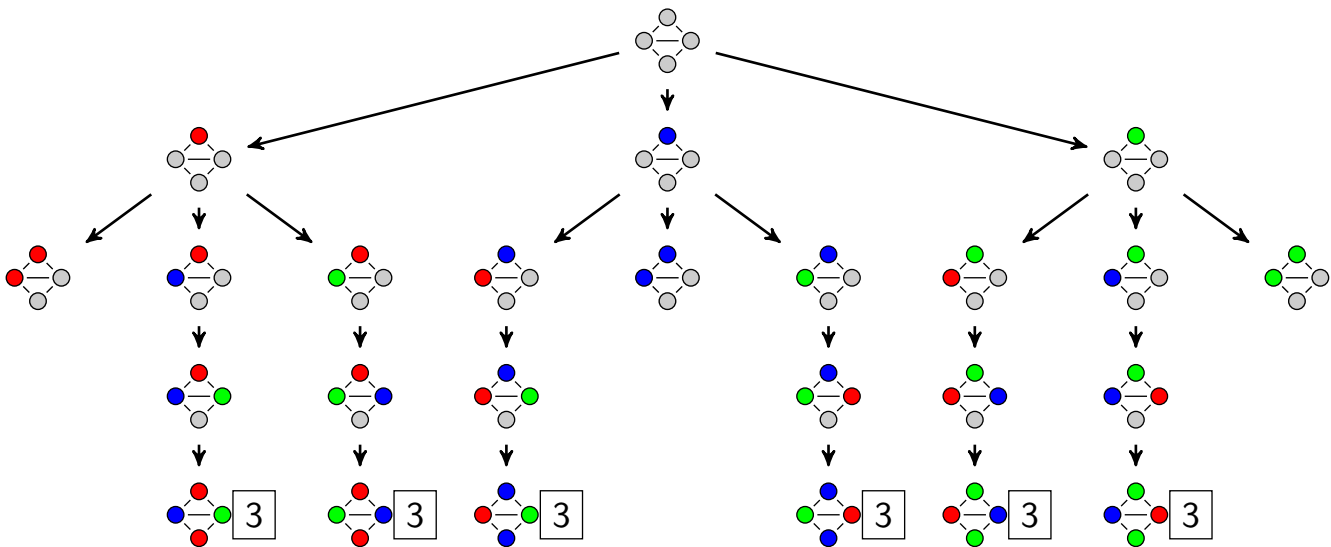
Dans un problème de minimisation P avec une *fonction d'objectif* f , étant donnée une *instance* $x \in P$, on cherche une solution y qui minimise $f(x, y)$

Un *modèle* est une représentation des choix possibles pour y , il définit un *espace de recherche* $\Sigma_{P,x}$ tel que $\forall x \in P, \arg \min_y f(x, y) \in \Sigma_{P,x}$

- Exemple : coloration de graphes
 - ▶ On peut représenter une solution y par une table *col* qui associe une couleur à chaque sommet
 - ▶ On peut déterminer une borne supérieure k sur le nombre de couleurs

Espace de recherche Σ

Pour une instance $x = (S, A)$, $\Sigma_{Col,x}$ est l'ensemble des tableaux de longueur $|S|$ dont les cases contiennent une couleur parmi k : taille de l'espace de recherche $|\Sigma_{Col,x}| = k^{|S|}$



Pour 3 couleurs : $3^4 = 81$ solutions, mais seulement 6 feuilles (élagage). $\chi(G) = 3$

Arbre de recherche

On définit un arbre par une racine r , une fonction N qui associe à chaque noeud de l'arbre la liste de ses descendants et une fonction booléenne ℓ telle que $\ell(x) = \text{vrai}$ si et seulement si x est une feuille.

Recherche (minimisation)

Données : Un arbre $\langle r, N, \ell \rangle$ et une fonction f qui associe un réel à chaque feuille de l'arbre

Résultat : Le minimum de f

$opt \leftarrow \infty$

$E \leftarrow \{r\}$

tant que E n'est pas vide **faire**

Retirer un élément x de E

si $\ell(x)$ **alors** $opt \leftarrow \min(f(x), opt)$

sinon

pour chaque $y \in N(x)$ **faire** Ajouter y à E

retourner opt

Coloration du graphe $G = (S, A)$ par C

- Noeud de l'arbre : une *coloration* (partielle) col
 - ▶ associe une couleur (ou \emptyset) $col[v]$ aux sommets $v \in S$
- Racine : la table t.q. $col[v] = \emptyset$ quelque soit $v \in S$
- ℓ : renvoie vrai si et seulement si col associe une couleur à *tous* les sommets
- N : pour un sommet v tel que $col[v] = \emptyset$, et pour chaque $c \in C \setminus \{col[w] \mid (v, w) \in A\}$:
 - ▶ contient col_c t.q. $col_c[w] = col[w]$ si $w \neq v$ et c sinon

Recherche en profondeur (DFS)

Données : Un arbre $\langle r, N, \ell \rangle$ et une fonction f qui associe un réel à chaque feuille de l'arbre

Résultat : Le minimum de f

$opt \leftarrow \infty$

$E \leftarrow \{r\}$

tant que E n'est pas vide **faire**

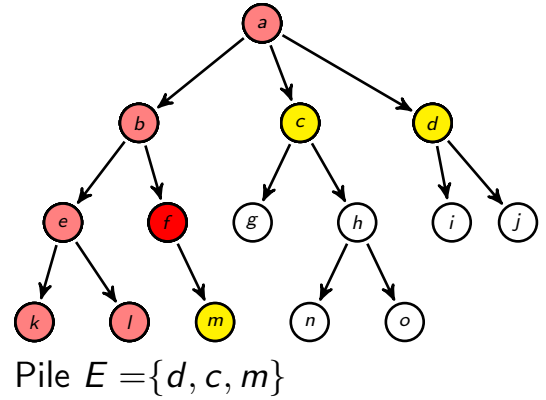
Dépiler le dernier élément x de E

si $\ell(x)$ **alors** $opt \leftarrow \min(f(x), f)$

sinon

pour chaque $y \in N(x)$ **faire** **Empiler** y **sur** E

retourner opt



Recherche en largeur (BFS)

Données : Un arbre $\langle r, N, \ell \rangle$ et une fonction f qui associe un réel à chaque feuille de l'arbre

Résultat : Le minimum de f

$opt \leftarrow \infty$

$E \leftarrow \{r\}$

tant que E n'est pas vide **faire**

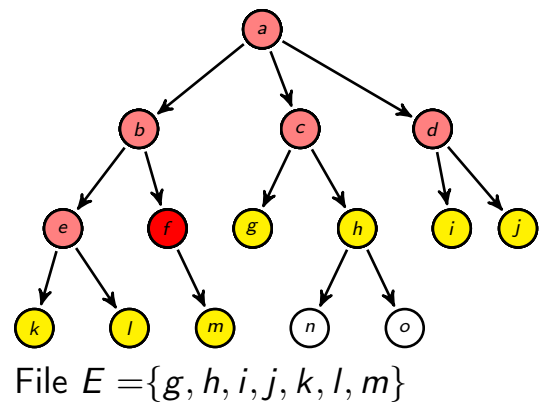
Choisir le premier élément x de la **file** E

si $\ell(x)$ **alors** $opt \leftarrow \min(f(x), opt)$

sinon

pour chaque $y \in N(x)$ **faire** **Insérer** y **dans** E

retourner opt



- Complexité en temps dans le pire des cas : identique
- BFS est *exponentiel* en mémoire : il faut stocker autant de points de l'espace de recherche que la largeur de l'arbre (même ordre de grandeur que l'arbre lui-même)
- DFS est *polynomial* en mémoire : il suffit de stocker une branche (logarithmique dans la taille de l'arbre)
- BFS (Breadth-first search) peut facilement devenir "Best-first search"
 - ▶ On peut explorer les branches les plus prometteuses en premier

- Soit x un noeud de l'arbre de recherche. Dans le cas de la coloration de graphe, ce noeud correspond à une coloration, qu'on appellera col_x .
 - ▶ Le coût propre au noeud x est dénoté $g(x)$. Par exemple le nombre de couleurs utilisées dans col_x .
 - ▶ La fonction $h(x)$ est une *heuristique admissible* si toute feuille l de l'arbre de recherche qui descend de x a un coût $g(l) = f(l)$ et tel que $g(l) \geq g(x) + h(x)$.

Algorithme A*

Explore en premier le noeud x de l'arbre de recherche tel que $g(x) + h(x)$ est minimum

Théorème

Si h est admissible, alors la première solution trouvée par Algorithme A* est optimale

A*

Données : Un arbre $\langle r, N, \ell \rangle$, une fonction f qui associe un réel à chaque feuille de l'arbre, deux fonctions g, h qui associent un coût à chaque noeud de l'arbre de recherche et telles que pour toute feuille l , on a $f(l) \geq g(x) + h(x)$

Résultat : Le minimum de f

$opt \leftarrow \infty$

$E \leftarrow \{r\}$

tant que E n'est pas vide **faire**

Choisir l'élément x dans E qui **minimise** $g(v) + h(v)$

si $\ell(x)$ **alors**

$opt \leftarrow \min(f(x), opt)$

retourner opt

sinon

pour chaque $y \in N(x)$ **faire** **Insérer** y **dans** E

- Soit x la première feuille de l'arbre de recherche, on a $g(x) = f(x)$
- Supposons qu'il existe une feuille x' telle que $g(x') = f(x') < g(x)$
- Tous les ascendants x'' de x' sont tels que $g(x'') + h(x'') \leq g(x') < g(x)$
- C'est une contradiction puisque au moins un de ces sommets est ouvert, et donc la branche qui mène à x' sera explorée avant la feuille x

Heuristique admissible pour la coloration de graphe

$h(x) =$ la taille d'une clique dans le sous-graphe de G induit par les sommets dont le voisinage contient au moins un sommet pour chacune des couleurs de col_x

Sous-graphe

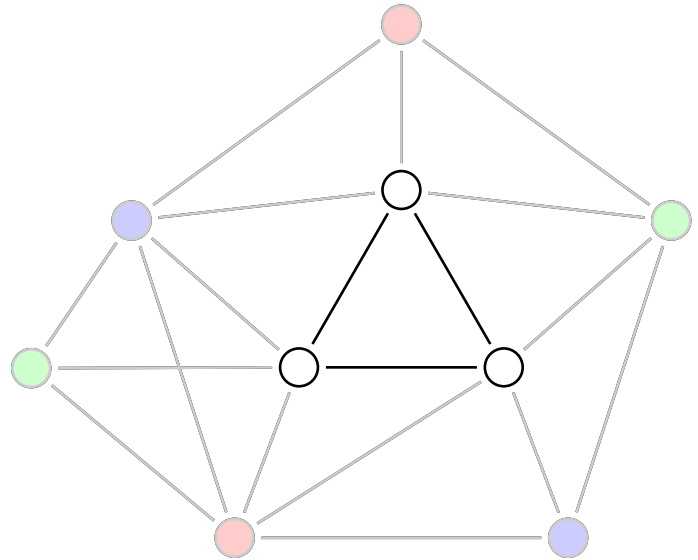
Le sous-graphe G_E induit par $E \subseteq S$ du graphe $G = (S, A)$:

- a pour ensemble de sommets E ;
- a une arête (v, w) si et seulement si $v \in E, w \in E$ et $(v, w) \in A$.

Clique

Un ensemble K est une clique du graphe G si et seulement si le sous-graphe de G induit par K est *complet* (contient une arête pour chaque paire de sommets dans K)

- Soit x un noeud de l'arbre de recherche correspondant au graphe partiellement coloré ci-contre
- 3 couleurs utilisées dans col_x donc $g(x) = 3$
- Le sous-graphe induit par les sommets avec un voisin de chacune des 3 couleurs contient une clique de taille 3
 - ▶ donc $h(x) = 3$, et $f(x) \geq g(x) + h(x) = 6$



Séparation et évaluation (Branch & Bound)

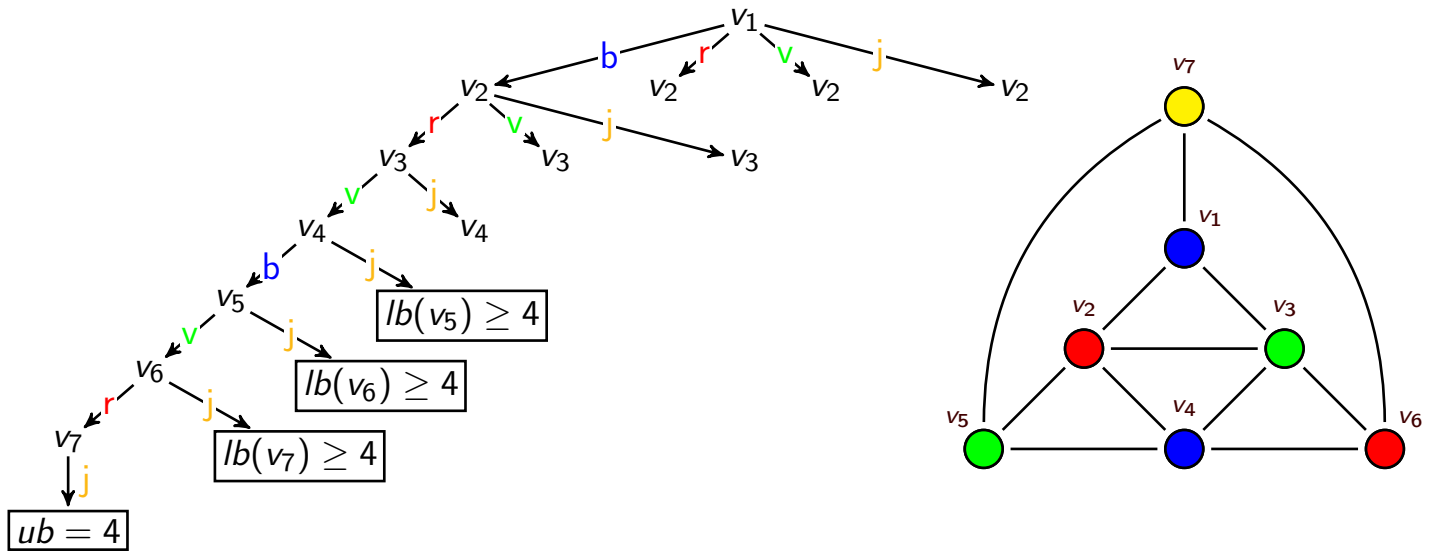
Borne inférieure

Pour un noeud x de l'arbre de recherche, $lb(x)$ est une borne inférieure de la fonction d'objectif f si et seulement si pour toute feuille l du sous-arbre enraciné en x , on a :

$$lb(x) \leq f(l)$$

Élagage

- Supposons qu'il existe une solution σ de coût $f(\sigma) = ub$.
 - ▶ Si $lb(x) \geq ub$ alors le sous-arbre enraciné en x ne contient pas de solutions meilleures que σ
 - ▶ Autrement dit, il est inutile d'explorer ce sous-arbre
- Les fonction g et h définies pour l'algorithme A^* sont telles que :
 - ▶ $g(x)$ et $g(x) + h(x)$ sont des bornes inférieures



Symétrie

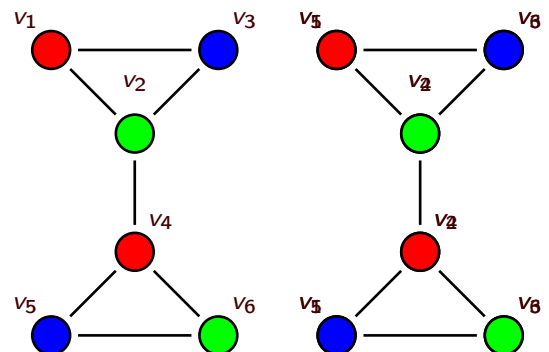
Une symétrie d'un espace de recherche Σ est une bijection $\varphi : \Sigma \mapsto \Sigma$ invariante pour la fonction d'objectif :

- Pour tout point l de l'espace de recherche (resp. noeud x de l'arbre) : $f(l) = f(\varphi(l))$ (resp. $f(x) = f(\varphi(x))$)
- Symétrie des couleurs : permutation des couleurs
 - ▶ Rouge \rightarrow Bleu \rightarrow Vert \rightarrow Rouge
- Symétrie du graphe : automorphisme du graphe
 - ▶ $v_1 \leftrightarrow v_5, v_2 \leftrightarrow v_4, v_3 \leftrightarrow v_6$

Composition de symétries

Si φ_1 et φ_2 sont des symétries, alors $\varphi_1 \circ \varphi_2$ en est une

Dans cet exemple, il y a $3! = 6$ symétries de couleur, et 2 symétries du graphe, donc $6 \times 2 = 12$ symétries au total

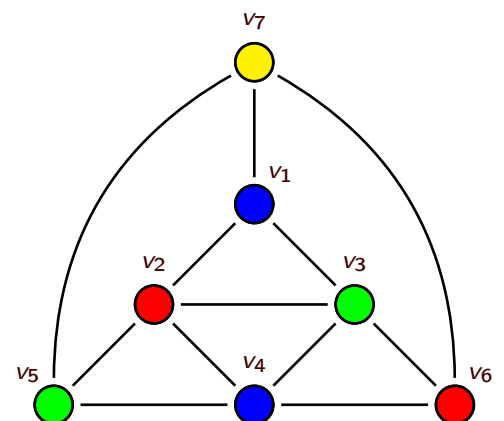
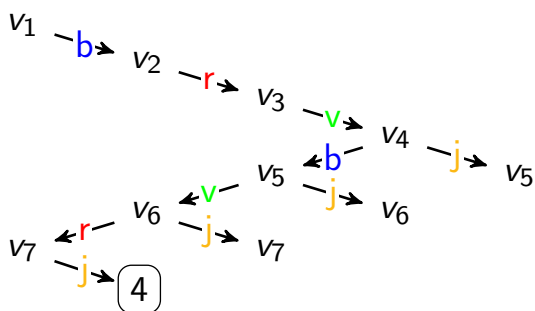


- Il est inutile d'explorer des régions symétriques de l'espace de recherche
 - ▶ Si le choix de couleur pour le premier sommet est v_1 = "rouge", il n'y a pas de solutions strictement meilleures dans le sous-arbre v_1 = "bleu"
 - ▶ La symétrique de cette solution par la transformation "bleu" → "rouge" a le même nombre de couleurs
- On peut parfois changer l'espace de recherche pour ne pas re-visiter des cas symétriques

Briser les symétries de couleur

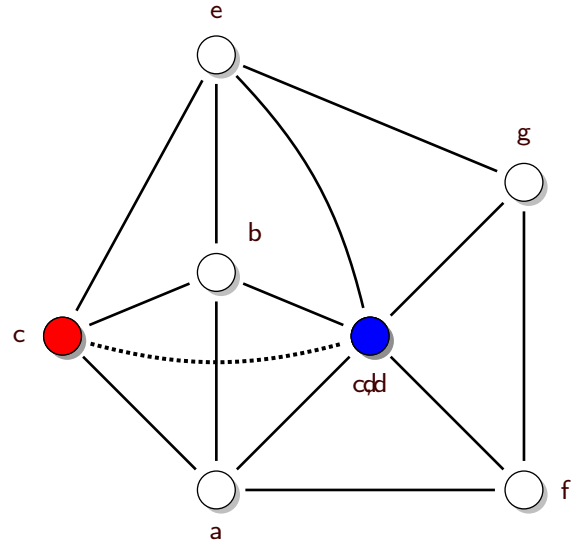
Étant donné un sommet v_i du graphe :

- Une branche pour chaque couleur *déjà utilisée*;
- Et une branche pour *une* nouvelle couleur.



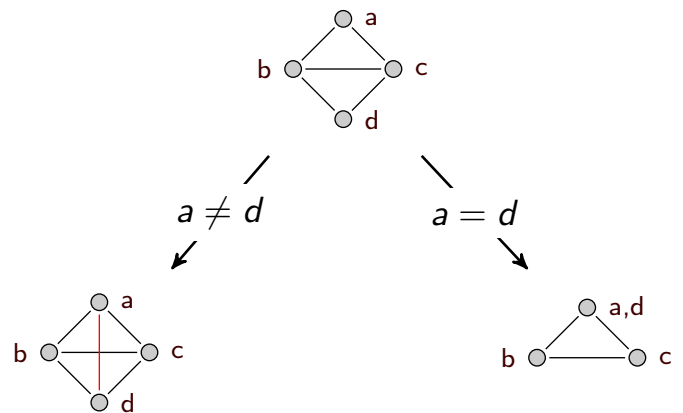
Changer l'espace de recherche

- L'espace de recherche n'est pas unique : exemple **Récurrence de Zykov**
- Soit col une coloration optimale de $G = (S, A)$. Pour toute paire de sommets v, w , tel que $(v, w) \notin A$:
 - ▶ Soit v et w ont des couleurs différentes ($col[v] \neq col[w]$)
 - ★ **Séparation** : rajouter l'arête (v, w) ne change pas $\chi(G)$
 - ▶ Soit v et w ont la même couleur ($col[v] = col[w]$)
 - ★ **Contraction** : fusionner v et w ne change pas $\chi(G)$



Récurrence de Zykov

- Pour chaque *non-arête* v, w
 - ▶ Une branche pour $v \neq w$ (rajouter l'arête (v, w))
 - ▶ Une branche pour $v = w$ (fusionner v et w)
- Ici, un seul point de choix : $a \neq d$ ou $a = d$
- Le noeud est une feuille si le graphe n'a aucune non-arête (aucune contraction ou séparation n'est possible)



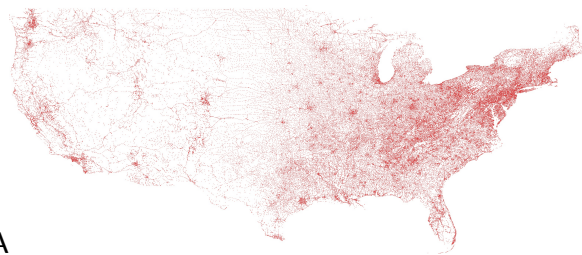
Quel algorithme ?

- Définition d'un modèle : **espace de recherche**
 - ▶ Table de couleurs? Choix des contractions/séparations?
- Définition de l'arbre de recherche : **branchement**
 - ▶ Un sous-arbre par couleur? Deux sous-arbres "couleur c / autre couleur" ?
- Définition de l'arbre de recherche : **feuilles**
- Définition de l'arbre de recherche : **DFS ou BFS**
- Methodes de raisonnement : **élagage**
 - ▶ Brisure de symétrie? Borne inférieure?

Quelle strategie ?

- Beaucoup de décisions sont *heuristiques*
 - ▶ Dans quel ordre doit-on visiter les sommets du graphe dans un problème de coloration ?
 - ▶ Dans quel ordre doit-on visiter les sous-arbres ?
- Ces décisions sont *très importantes* en pratique

Résoudre des problèmes NP-complets par recherche arborescente



- *Voyageur de Commerce* 115K villes des USA

- *Coloration de Graphe* "followers" Google+ (108K sommets, 12M arcs, $\chi = 326$)

