Méthodes algorithmiques

Sophie Pinchinat
sophie.pinchinat@irisa.fr

IRISA, Université de Rennes

UE ALG – année 2023-2024

Contenu

Conception et analyse des algorithmes

- Importance du choix d'un modèle technologique d'exécution.

  Nous prenons le modèle RAM (Random Access Memory) à processeur unique

Conception et analyse des algorithmes

- Importance du choix d'un modèle technologique d'exécution.

  Nous prenons le modèle RAM (Random Access Memory) à processeur unique

- Conception : formuler le problème à résoudre avec une précision mathématique suffisante de manière à poser une question concrète et définir un algorithme permettant de résoudre ce problème.

Conception et analyse des algorithmes

- Importance du choix d'un modèle technologique d'exécution.

  Nous prenons le modèle RAM (Random Access Memory) à processeur unique

- Conception : formuler le problème à résoudre avec une précision mathématique suffisante de manière à poser une question concrète et définir un algorithme permettant de résoudre ce problème.
  En pratique on s'appuie sur quelques techniques de conception fondamentales, très utiles pour évaluer la complexité inhérente du problème et pour formuler un algorithme qui le résout.

Conception et analyse des algorithmes

- Importance du choix d'un modèle technologique d'exécution.

  Nous prenons le modèle RAM (Random Access Memory) à processeur unique

- Conception : formuler le problème à résoudre avec une précision mathématique suffisante de manière à poser une question concrète et définir un algorithme permettant de résoudre ce problème.
  En pratique on s'appuie sur quelques techniques de conception fondamentales, très utiles pour évaluer la complexité inhérente du problème et pour formuler un algorithme qui le résout.

- Analyse d'un algorithme : montrer sa correction (il fait bien ce qu'on veut) et donner une borne supérieure sur son temps d'exécution et/ou l'espace mémoire qu'il utilise pour établir son efficacité.

Conception et analyse des algorithmes

- Importance du choix d'un modèle technologique d'exécution.

  Nous prenons le modèle RAM (Random Access Memory) à processeur unique

- Conception : formuler le problème à résoudre avec une précision mathématique suffisante de manière à poser une question concrète et définir un algorithme permettant de résoudre ce problème.
  En pratique on s'appuie sur quelques techniques de conception fondamentales, très utiles pour évaluer la complexité inhérente du problème et pour formuler un algorithme qui le résout.

- Analyse d'un algorithme : montrer sa correction (il fait bien ce qu'on veut) et donner une borne supérieure sur son temps d'exécution et/ou l'espace mémoire qu'il utilise pour établir son efficacité.Trois questions systématiques face à un algorithme :

  1. Mon algorithme est-il correct ? termine-t-il ?

  2. Quelle est son temps d'exécution ? de quelle mémoire a-t-il besoin ?

  3. Est-il optimal ou peut-on résoudre le problème plus efficacement ?

Objectifs

Maîtriser les techniques fondamentales de conception d'algorithmes.

Pour cela, il faut savoir que :

- La familiarisation avec ces techniques est un processus progressif
- Reconnaître/flairer le "genre" du problème requiert de l'expérience
- Il existe des nuances subtiles de la nature du problème qui induisent des effets déterminants sur sa difficulté.

Exemple 1

"Le problème du Circuit Eulérien est facile" vs. "Le problème du Cycle Hamiltonien est difficile".

Vous devez être actifs :

À vous de jouer !

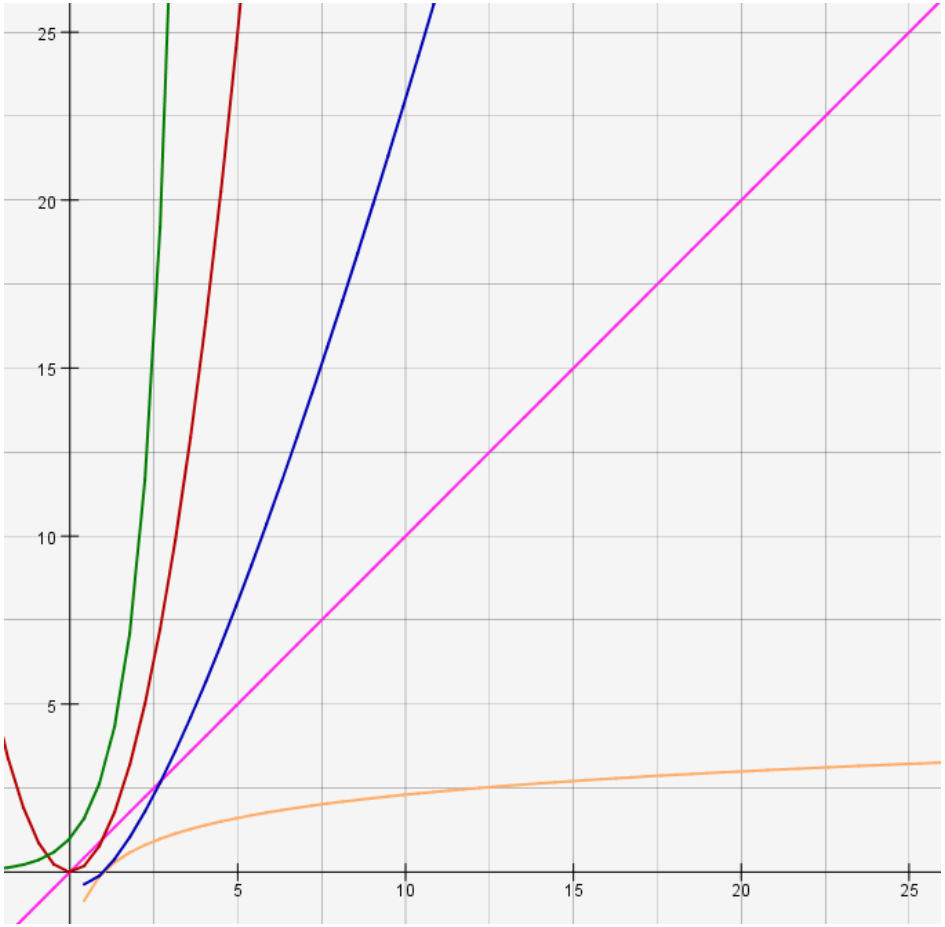
Voir par exemple dans [DPV06] la définition de ces deux problèmes et ce qu'on sait à leur sujet.

Plan du cours

Nous ébaucherons :

1. Des rappels sur les ordres de grandeur
2. La méthode Diviser pour Régner
3. Les approches gloutonnes
4. La méthode par Programmation Dynamique
5. La méthode par Programmation Linéaire
6. La notion de problèmes NP-complets

Quelques fonctions classiques



Complexités classiques
(ordre de grandeur)

$$\begin{cases} O(e^n) \\ O(n^2) \\ O(nlog(n)) \\ O(n) \\ O(log(n)) \end{cases}$$

À vous de jouer !

Qui est la courbe de qui ?

Nous partirons du principe
que vous avez TOUS
le bagage mathématique décrit dans

la Section 3.2 “NOTATIONS STANDARD ET FONCTIONS CLASSIQUES”
de

T.H. Cormen, C.E. Leiserson, and R.L. Rivest.
Introduction à l'algorithmique.
Dunod, 1996.

surtout pour la partie logarithme, par exemple le fait que changer la base d'un logarithme ne modifie la valeur du logarithme que d'un facteur constant

SI VOUS NE CONNAISSEZ PAS CES CHOSES-LÀ,
IL EST **ENCORE TEMPS** DE LES ACQUÉRIR

Ordre de grandeurs/Notations asymptotiques

Définition 2

Soit $g : \mathbb{N} \to R^+$ une fonction donnée.

$$\Theta(g(n)) = \{f : \mathbb{N} \to R^+ \mid \text{il existe des constantes } c_1, c_2 > 0 \text{ et } n_0 \in \mathbb{N} \text{ telles que } c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ pour tout } n_0 \leq n\}$$

$$O(g(n)) = \{f : \mathbb{N} \to R^+ \mid \text{il existe des constantes } c > 0 \text{ et } n_0 \in \mathbb{N} \text{ telles que } f(n) \leq c g(n), \text{ pour tout } n_0 \leq n\}$$

$$\Omega(g(n)) = \{f : \mathbb{N} \to R^+ \mid \text{il existe des constantes } c > 0 \text{ et } n_0 \in \mathbb{N} \text{ telles que } c g(n) \leq f(n), \text{ pour tout } n_0 \leq n\}$$

On note $f \in \Theta(g), f \in O(g), f \in \Omega(g)$.

Exemple 3

$n^2 + 10n = \Theta(n^2)$ car il suffit de prendre les constantes $c_1 = 1$, $c_2 = 2$, et les inégalités requises sont vraies à partir de $n_0 = 10$.

À vous de jouer !

Entraînez-vous pour vous convaincre que vous avez compris en choisissant des f et g.

Encore à vous de jouer !

Il existe une foultitude d'exercices dans les ouvrages de la liste bibliographique du module permettant de maîtriser les notions d'ordres de grandeur :

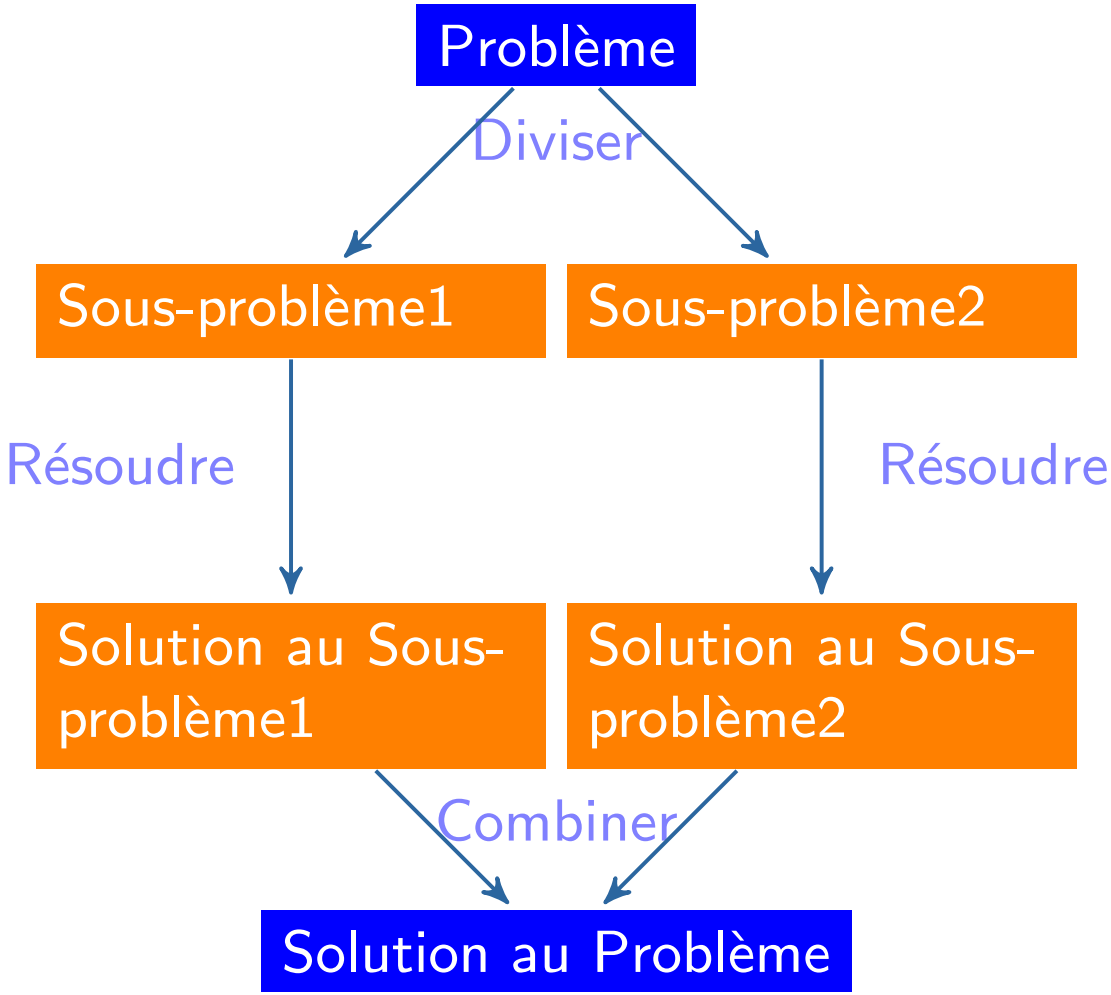
- Page 48 et suivantes de Introduction à l'algorithmique, T. H. Cormen, C. E. Leiserson, R. L. Rivest, Dunod, 1994. [CLR96].
- Page 18 et suivantes de Algorithms. Sanjoy Dasgupta, Christos H. Papadimitriou, Umesh Vazirani. Mcgraw Hill Book Co, 2006. [DPV06]
- Page 65 et suivantes de Algorithm Design. Jon Kleinberg, Éva Tardos. Addison Wesley, 2005. [KT05]

À vous de jouer !

Montrer que si q est un réel positif, alors $f(n) = 1 + q + q^2 + \ldots + q^n$ est dans :

1. $\Theta(1)$ si $q < 1$.
2. $\Theta(n)$ si $q = 1$.
3. $\Theta(q^n)$ si $q > 1$.

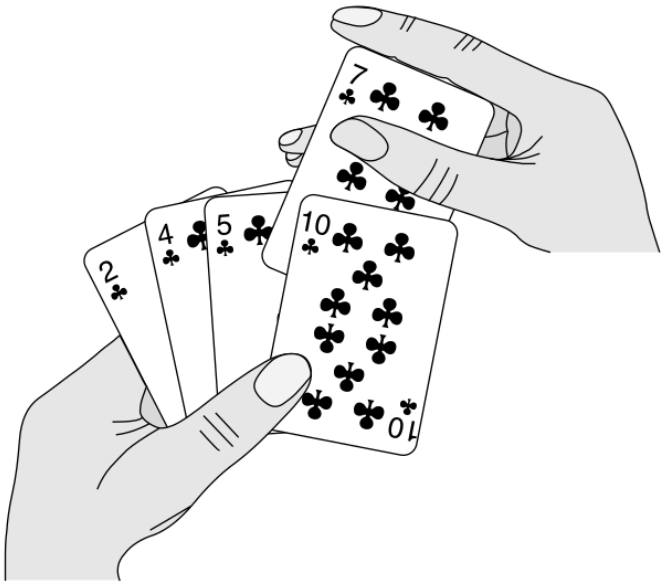
Diviser pour Régner



Le problème du tri

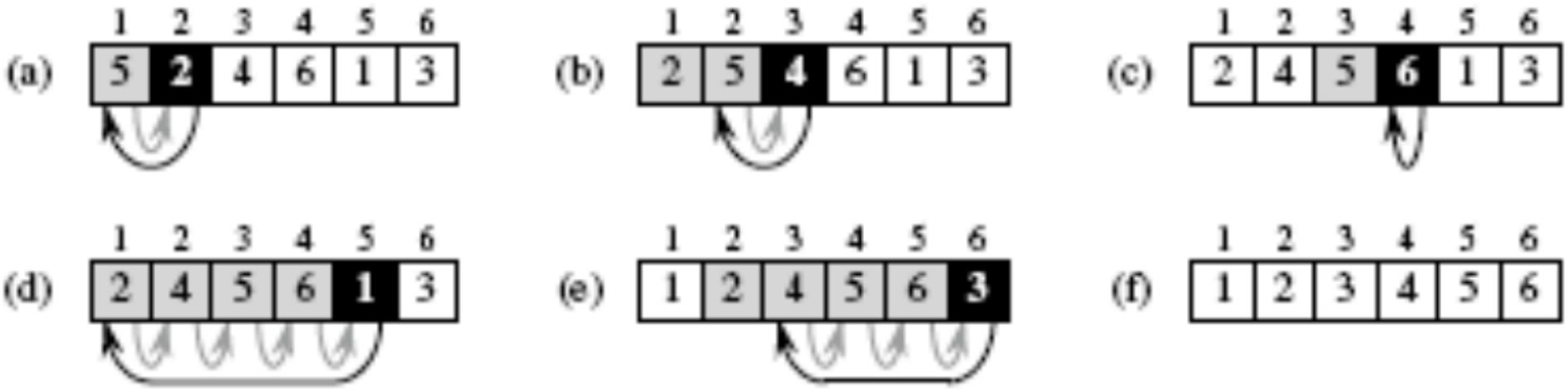
(Le tri)

Entrée : un tableau A d'entiers
Sortie : une permutation de A triée

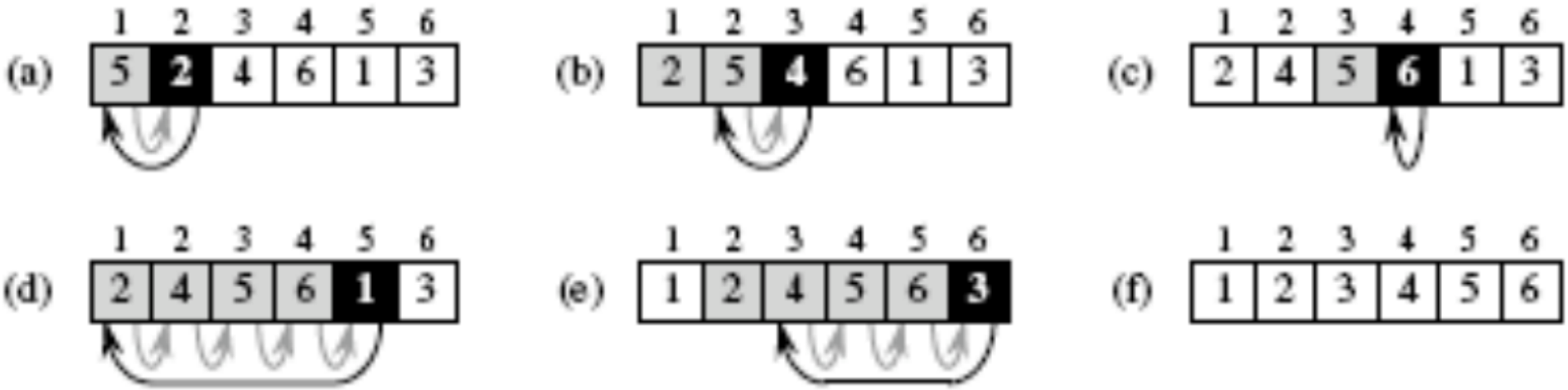


Une approche naturelle

Un algorithme naturel : le tri par insertion



Un algorithme naturel : le tri par insertion



TRI-INSERTION (A)

1 **pour** $j \leftarrow 2$ **à** *longueur*$[A]$
2 **faire** *clé* $\leftarrow A[j]$
3 ▷ Insère $A[j]$ dans la suite triée $A[1 \,..\, j-1]$.
4 $i \leftarrow j-1$
5 **tant que** $i > 0$ et $A[i] >$ *clé*
6 **faire** $A[i+1] \leftarrow A[i]$
7 $i \leftarrow i-1$
8 $A[i+1] \leftarrow$ *clé*

Correction du tri par insertion par invariant de boucle

Les invariants de boucle sont des propriétés qui aident à comprendre/expliquer/justifier pourquoi un algorithme (itératif) est correct. Nous verrons plus tard comment prouver la correction d'un algorithme récursif.

Correction du tri par insertion par invariant de boucle

Les invariants de boucle sont des propriétés qui aident à comprendre/expliquer/justifier pourquoi un algorithme (itératif) est correct. Nous verrons plus tard comment prouver la correction d'un algorithme récursif.

Exemple 4 (un invariant pour l'algorithme de tri par insertion)

"Au début de chaque itération de la boucle **pour** *les lignes 1–8, le sous-tableau $A[1..j-1]$ se compose des éléments qui occupaient initialement les positions 1 à $j-1$, mais qui sont maintenant triés."*

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3           ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4           i ← j − 1
5           tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                   i ← i − 1
8           A[i + 1] ← clé
```

Correction du tri par insertion par invariant de boucle

Les invariants de boucle sont des propriétés qui aident à comprendre/expliquer/justifier pourquoi un algorithme (itératif) est correct. Nous verrons plus tard comment prouver la correction d'un algorithme récursif.

Exemple 4 (un invariant pour l'algorithme de tri par insertion)

"Au début de chaque itération de la boucle **pour** *les lignes 1–8, le sous-tableau $A[1..j-1]$ se compose des éléments qui occupaient initialement les positions 1 à $j-1$, mais qui sont maintenant triés."*

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3          ▷ Insère A[j] dans la suite
                 triée A[1 . . j − 1].
4          i ← j − 1
5          tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                   i ← i − 1
8          A[i + 1] ← clé
```

Une fois l'invariant énoncé, il faut établir trois choses le concernant :

1. **Initialisation** : l'invariant est vrai avant la première itération de la boucle.
2. **Conservation** : si l'invariant est vrai avant une itération de la boucle, il le reste avant l'itération suivante.
3. **Terminaison** : une fois la boucle terminée (!), l'invariant fournit une propriété utile pour montrer la correction de l'algorithme.

Invariant de boucle pour le tri par insertion

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3           ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4           i ← j − 1
5           tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                   i ← i − 1
8           A[i + 1] ← clé
```

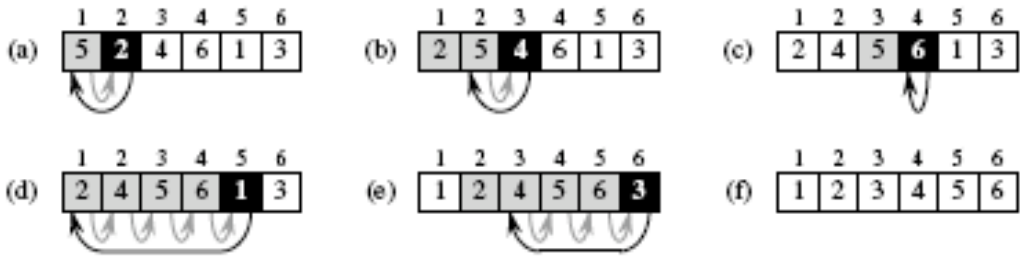


Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

Invariant :

"Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau $A[1..j-1]$ *se compose des éléments qui occupaient initialement les positions* $A[1..j-1]$ *mais qui sont maintenant triés."*

Invariant de boucle pour le tri par insertion

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3          ▷ Insère A[j] dans la suite
                 triée A[1 . . j − 1].
4          i ← j − 1
5          tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                   i ← i − 1
8          A[i + 1] ← clé
```

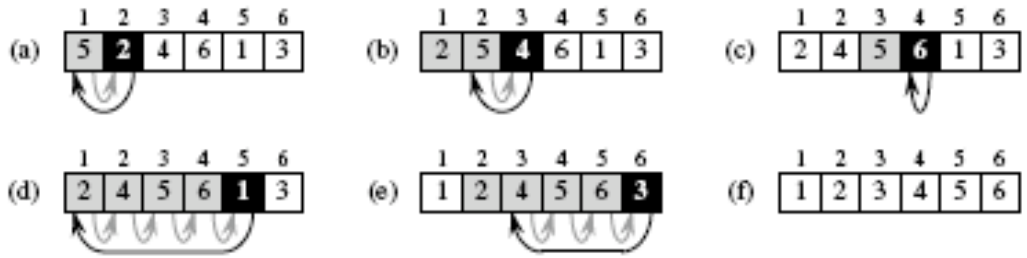


Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = (5, 2, 4, 6, 1, 3)$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

Invariant :

"Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau $A[1..j-1]$ se compose des éléments qui occupaient initialement les positions $A[1..j-1]$ mais qui sont maintenant triés."

Initialisation j = 2 et le tableau $A[1..j-1]$ est $A[1]$ qui est trivialement trié.

Invariant de boucle pour le tri par insertion

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3          ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4          i ← j − 1
5          tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← clé
```

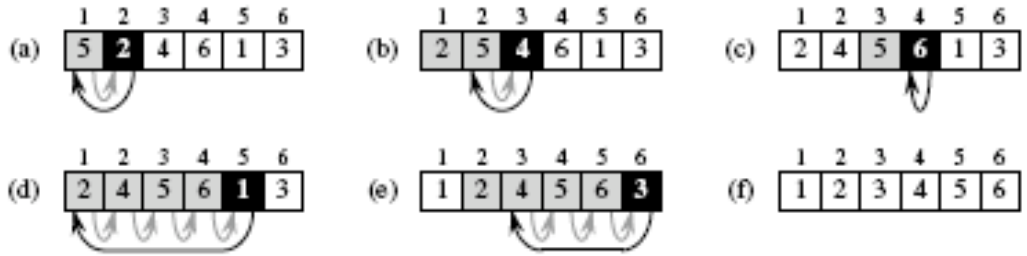


Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

Invariant :

"Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau $A[1..j-1]$ *se compose des éléments qui occupaient initialement les positions* $A[1..j-1]$ *mais qui sont maintenant triés."*

Conservation : (on pourrait être plus formel).

Le corps de la boucle **tant que** consiste à déplacer d'une position vers la droite $A[j-1]$, $A[j-2]$, $A[j-3]$, etc. jusqu'à ce qu'on trouve la bonne position pour A[j] (lignes 4–7). On insère alors la valeur de A[j] (ligne 8) à la bonne place, et le tout est parfaitement trié.

Invariant de boucle pour le tri par insertion

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3          ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4          i ← j − 1
5          tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← clé
```

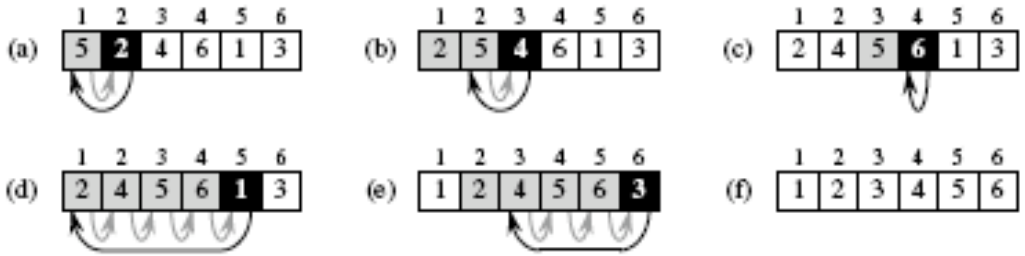


Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

Invariant :

"Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau $A[1..j-1]$ se compose des éléments qui occupaient initialement les positions $A[1..j-1]$ mais qui sont maintenant triés."

Terminaison : La boucle **pour** extérieure prend fin quand j dépasse n, c-à-d. $j = n + 1$.

L'invariant de boucle devient :

Le sous-tableau $A[1..n]$ se compose des éléments qui appartenaient originellement à $A[1..n]$ mais qui sont maintenant triés.

Comme l'algorithme termine, il est correct.

Complexité du tri par insertion (dans le pire cas)

TRI-INSERTION (A)	*coût*	*fois*
1 **pour** $j \leftarrow 2$ **à** *longueur*$[A]$	c_1	n
2 **faire** *clé* $\leftarrow A[j]$	c_2	$n-1$
3 ▷ Insère $A[j]$ dans la suite triée $A[1\,..\,j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **tant que** $i > 0$ et $A[i] >$ *clé*	c_5	$\sum_{j=2}^{n} t_j$
6 **faire** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] \leftarrow$ *clé*	c_8	$n-1$

Complexité du tri par insertion (dans le pire cas)

TRI-INSERTION (A)		*coût*	*fois*
1	**pour** $j \leftarrow 2$ **à** *longueur*$[A]$	c_1	n
2	**faire** *clé* $\leftarrow A[j]$	c_2	$n-1$
3	▷ Insère $A[j]$ dans la suite triée $A[1\,..\,j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**tant que** $i > 0$ et $A[i] >$ *clé*	c_5	$\sum_{j=2}^{n} t_j$
6	**faire** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow$ *clé*	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j-1) + c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1)$$

Complexité du tri par insertion (dans le pire cas)

TRI-INSERTION (A)		coût	fois
1	**pour** $j \leftarrow 2$ **à** *longueur*$[A]$	c_1	n
2	**faire** *clé* $\leftarrow A[j]$	c_2	$n-1$
3	▷ Insère $A[j]$ dans la suite triée $A[1..j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**tant que** $i > 0$ et $A[i] >$ *clé*	c_5	$\sum_{j=2}^{n} t_j$
6	**faire** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow$ *clé*	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j-1) + c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1)$$

On se place dans le cas le plus défavorable, le pire cas : le tableau A est trié en ordre décroissant et donc $t_j = j$. Or

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad \text{et} \qquad \sum_{j=2}^{n}(j-1) = \frac{(n-1)n}{2}$$

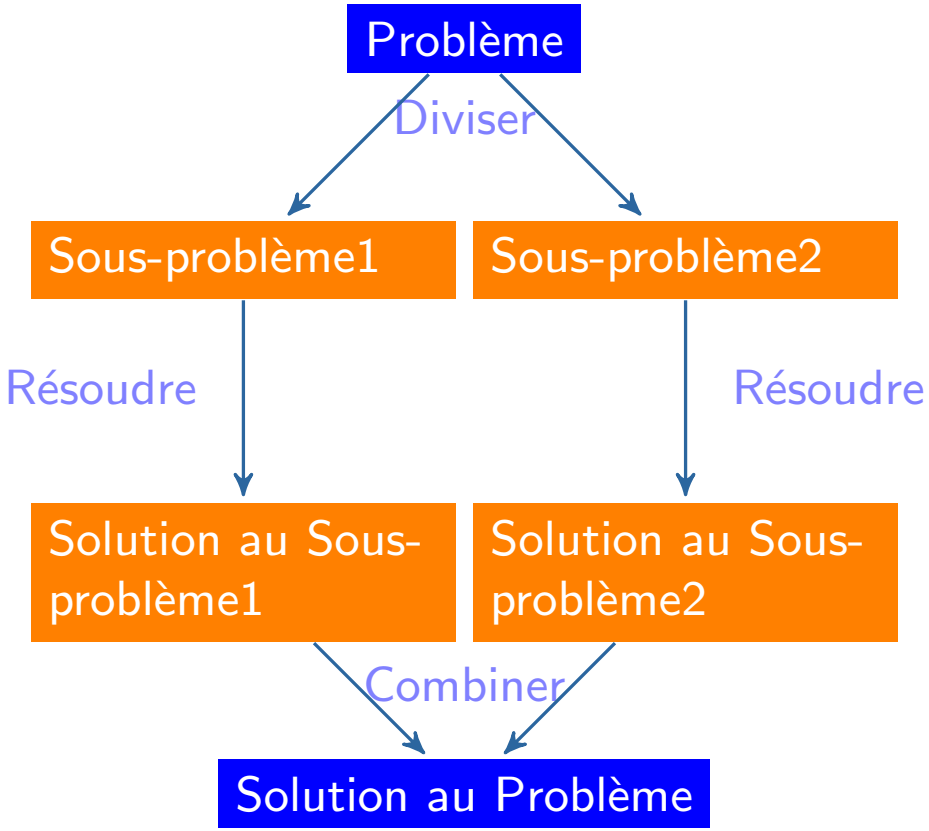
À vous de jouer !

Finir le raisonnement pour en déduire que $T(n) \in O(n^2)$.

Principe de la méthode Diviser-pour-Régner

Quand le problème s'y prête :

1. On divise le problème en plusieurs sous-problèmes
2. On résoud chaque sous-problème
3. On combine les solutions des sous-problèmes en une solution du problème



On peut ainsi espérer un facteur *logarithmique*.

Un exemple classique : l'algorithme Tri-Fusion

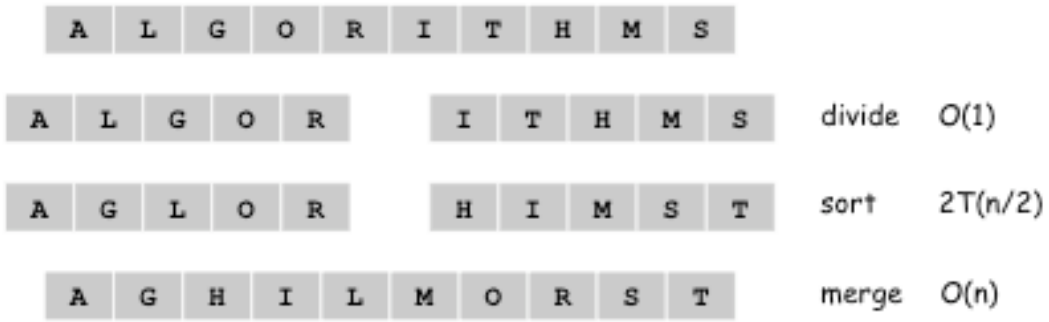
Pour un tableau $A[1..n]$ de clés, on appelle Tri-Fusion$(A, 1, n)$, où pour chaque $1 \leq p$ et chaque $r \leq n$, Tri-Fusion(A, p, r) est défini par :

Algorithm 1 Tri-Fusion(A, p, r)

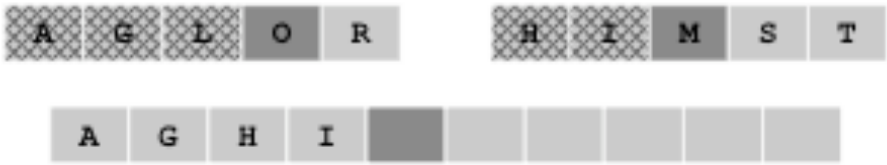
```
1: if p < r then                                   ▷ sinon il n'y a rien à faire
2:     q ← ⌊(p + r)/2⌋                             ▷ q est l'indice du milieu du tableau A[p..r]
3:     Tri-Fusion(A, p, q)                         ▷ on trie le sous-tableau de gauche
4:     Tri-Fusion(A, q + 1, r)                     ▷ on trie le sous-tableau de droite
5:     Fusion(A, p, q, r)                          ▷ on entrelace les deux sous-tableaux maintenant triés
6: end if
```

Tri fusion : un exemple

- TRI-FUSION$(A, 1, n)$ de coût $T(n)$:



- FUSION$(A, 1, \lfloor (n+1)/2 \rfloor, n)$:



TRI-FUSION(A, p, r)

```
1  si p < r
2     alors q ← ⌊(p + r)/2⌋
3           TRI-FUSION(A, p, q)
4           TRI-FUSION(A, q + 1, r)
5           FUSION(A, p, q, r)
```

Lemme 5 (voir la preuve plus loin)

L'appel de FUSION(A, p, q, r) *prend un temps dans* $\Theta(r - p + 1)$*, donc dans* $O(n)$*, si le tableau est de taille* n*.*

Équations de récurrence

En admettant que l'appel de $\textsc{Fusion}(A, 1, \lfloor (n+1)/2 \rfloor, n)$ prend un temps dans $\Theta(n)$, l'appel de $\textsc{Tri-Fusion}(1, n)$ prend un temps $T(n)$ vérifiant :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \quad // \text{ Il n'y a rien à trier} \\ 2T(\lfloor (n+1)/2 \rfloor) + cn & \text{si } n > 1 \quad // \text{ 2 sous-problèmes + la fusion} \end{cases}$$

De manière générale un algorithme de type DR qui résoud un problème en combinant a sous-problèmes de tailles b fois plus petits ont une complexité en temps vérifiant :

Patron des complexités de la méthode DR

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \quad \text{(les problèmes de base)} \\ aT(\lfloor n/b \rfloor) + f(n) & \text{si } n > 1 \quad \text{(les problèmes plus grands)} \end{cases}$$

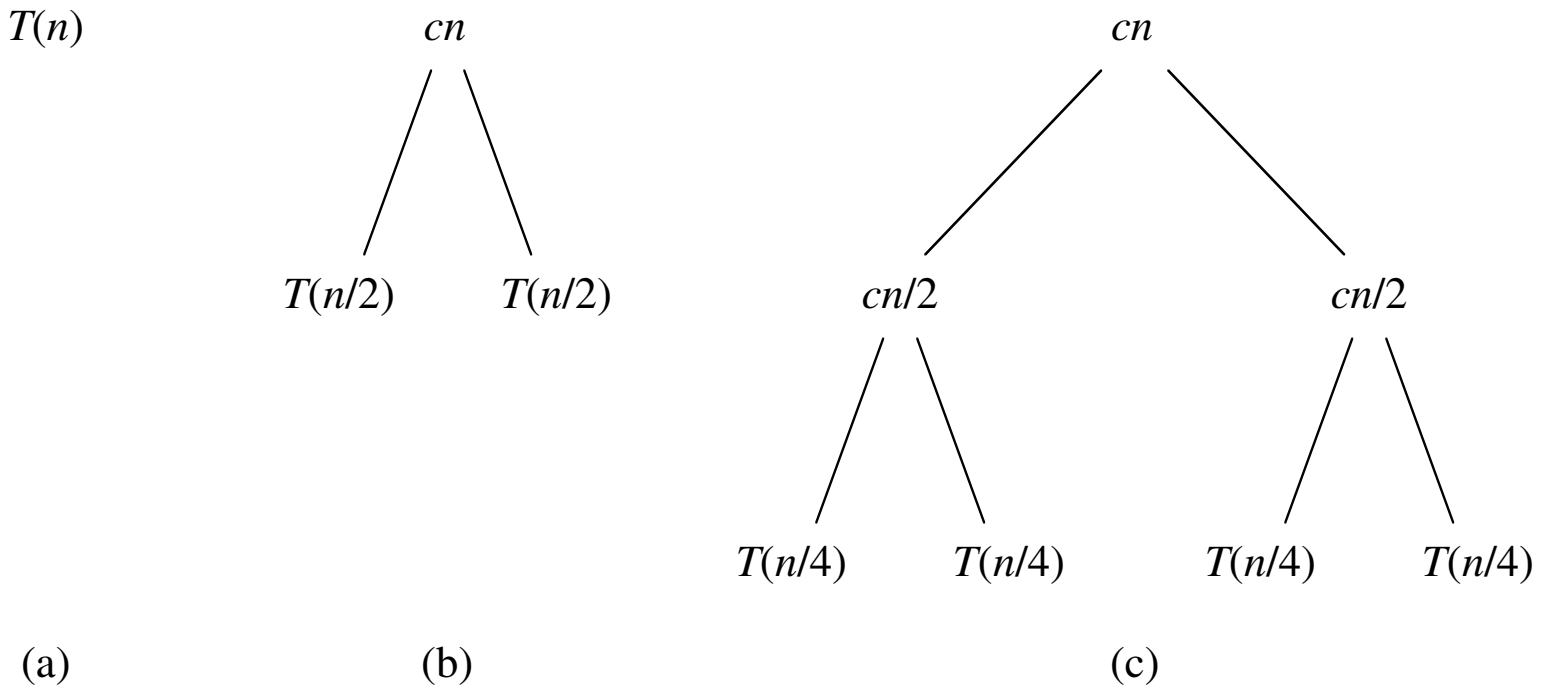
où $a \geq 1, b > 1$ et $f(n)$ est la coût de décomposer en a sous-problèmes et de recombiner ces a sous-problèmes.

Il s'agit maintenant de calculer $T(n)$?

Complexité de Tri-Fusion

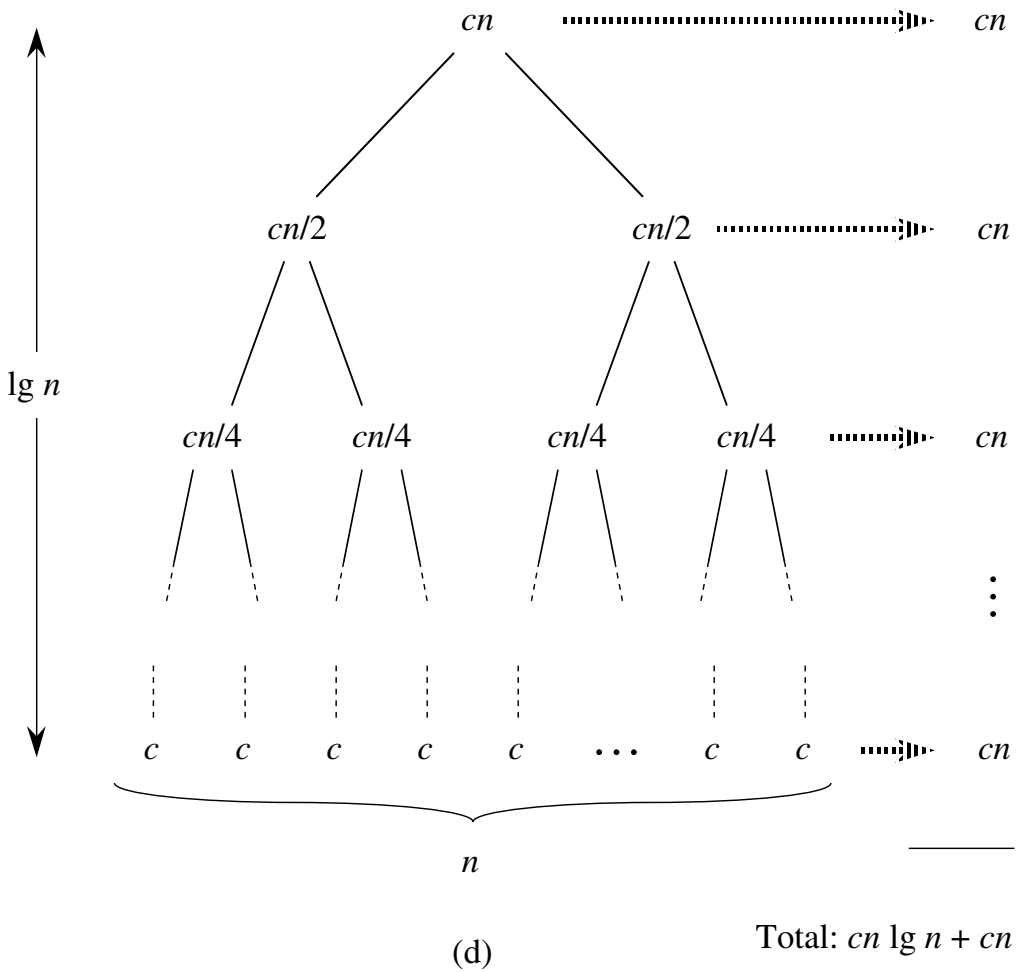
On suppose $n = 2^k$ où $k \in \mathbb{N}$, de sorte que $log_2(n)$ est un entier.

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \quad \text{(cas d'un tableau à 1 élément)} \\ 2T(\lfloor n/2 \rfloor) + cn & \text{si } n > 1 \quad \text{(tri des 2 sous-tableaux et leur fusion)} \end{cases}$$

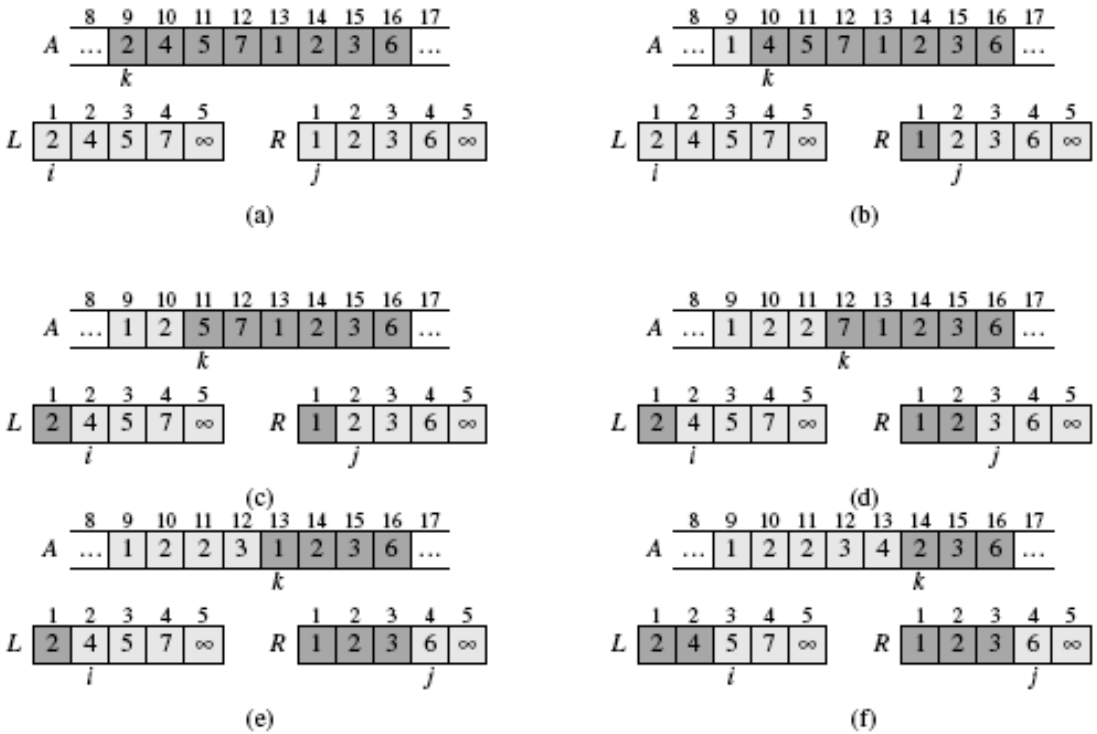


Complexité de TRI-FUSION

Avec $log_2(n) \in \mathbb{N}$, l'arbre des appels récursifs a $log_2(n) + 1$ niveaux.



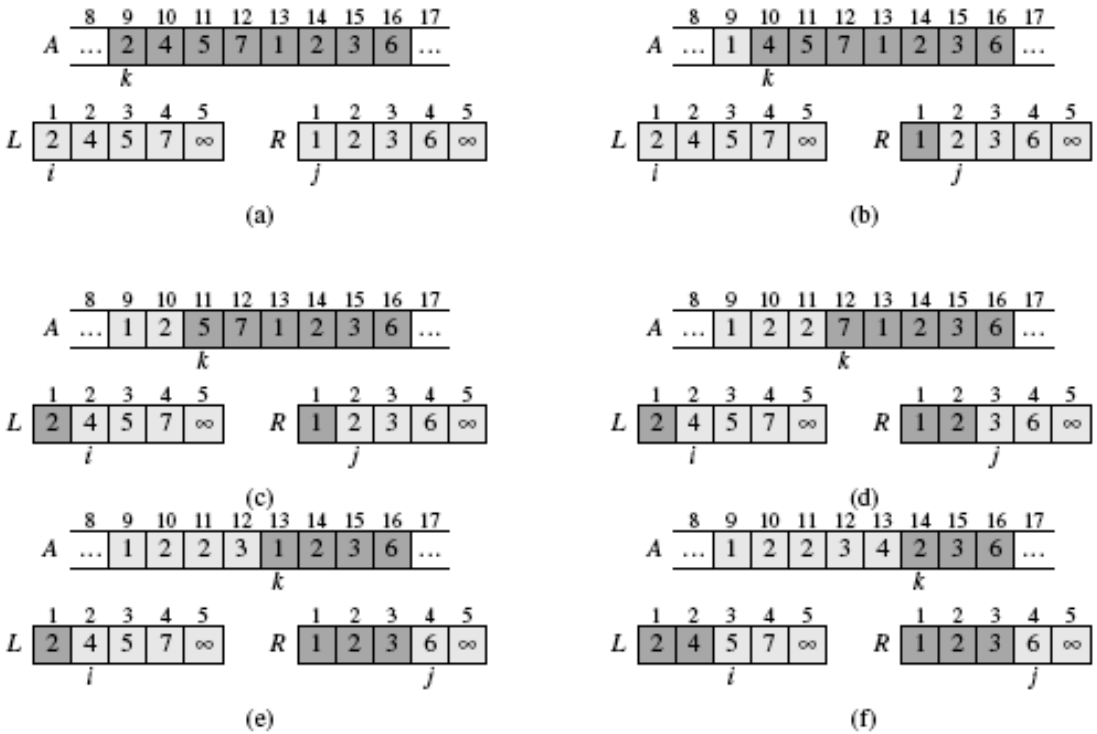
Algorithme de fusion FUSION(A, p, q, r) : analyse



à continuer ...

Algorithme de fusion FUSION(A, p, q, r) : analyse

```
FUSION(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  créer tableaux L[1 .. n1 + 1] et R[1 .. n2 + 1]
 4  pour i ← 1 à n1
 5      faire L[i] ← A[p + i − 1]
 6  pour j ← 1 à n2
 7      faire R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  pour k ← p à r
13      faire si L[i] ⩽ R[j]
14                alors A[k] ← L[i]
15                     i ← i + 1
16                sinon A[k] ← R[j]
17                     j ← j + 1
```



à continuer ...

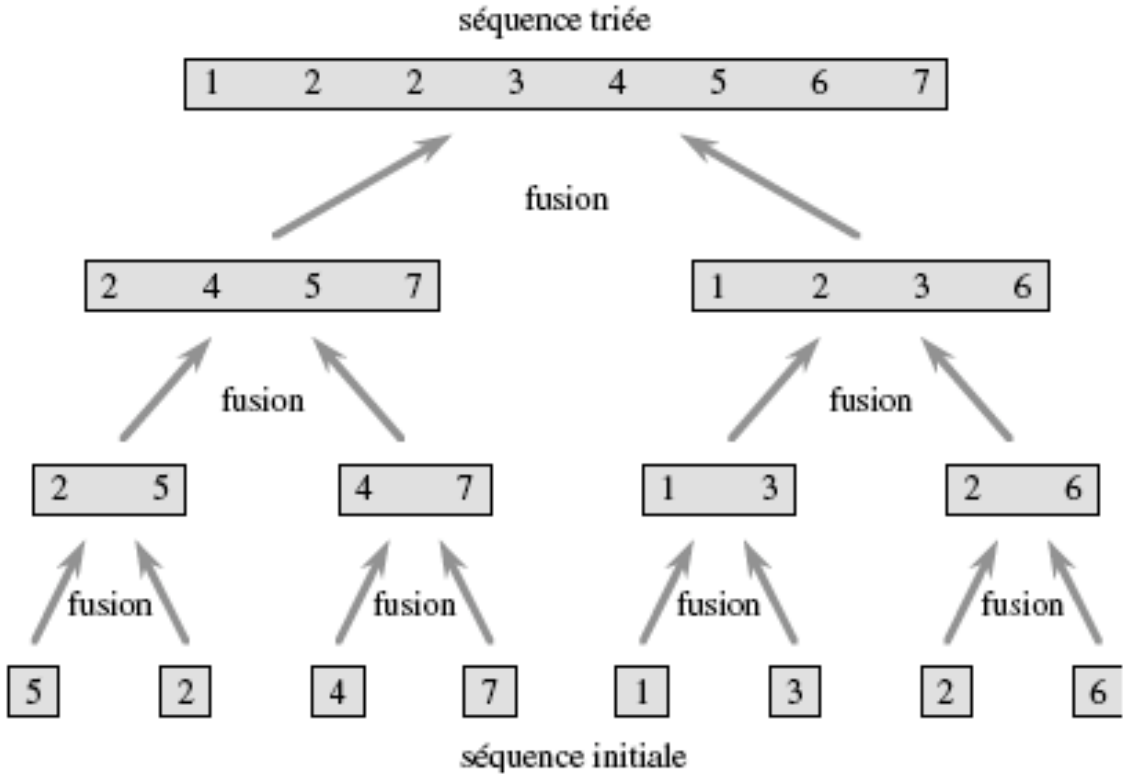


Figure 2.4 Le fonctionnement du tri par fusion sur le tableau $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. Les longueurs des séquences triées en cours de fusion augmentent à mesure que l'algorithme remonte du bas vers le haut.

Correction de l'algorithme FUSION(A, p, q, r)

```
FUSION(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  créer tableaux L[1 .. n1 + 1] et R[1 .. n2 + 1]
 4  pour i ← 1 à n1
 5      faire L[i] ← A[p + i − 1]
 6  pour j ← 1 à n2
 7      faire R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  pour k ← p à r
13      faire si L[i] ⩽ R[j]
14              alors A[k] ← L[i]
15                   i ← i + 1
16              sinon A[k] ← R[j]
17                   j ← j + 1
```

À vous de jouer !

Montrer que l'algorithme FUSION(A, p, q, r) est correct en utilisant l'invariant de boucle :

Au début de chaque itération de la boucle pour des lignes 12-17, le sous-tableau $A[p..k-1]$ contient les $k-p$ plus petits éléments de $L[1..n_1+1]$ et $R[1..n_2+1]$.

Méthodes pour résoudre les systèmes de récurrences

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ aT(\lfloor n/b \rfloor) + f(n) & \text{si } n > 1 \end{cases}$$

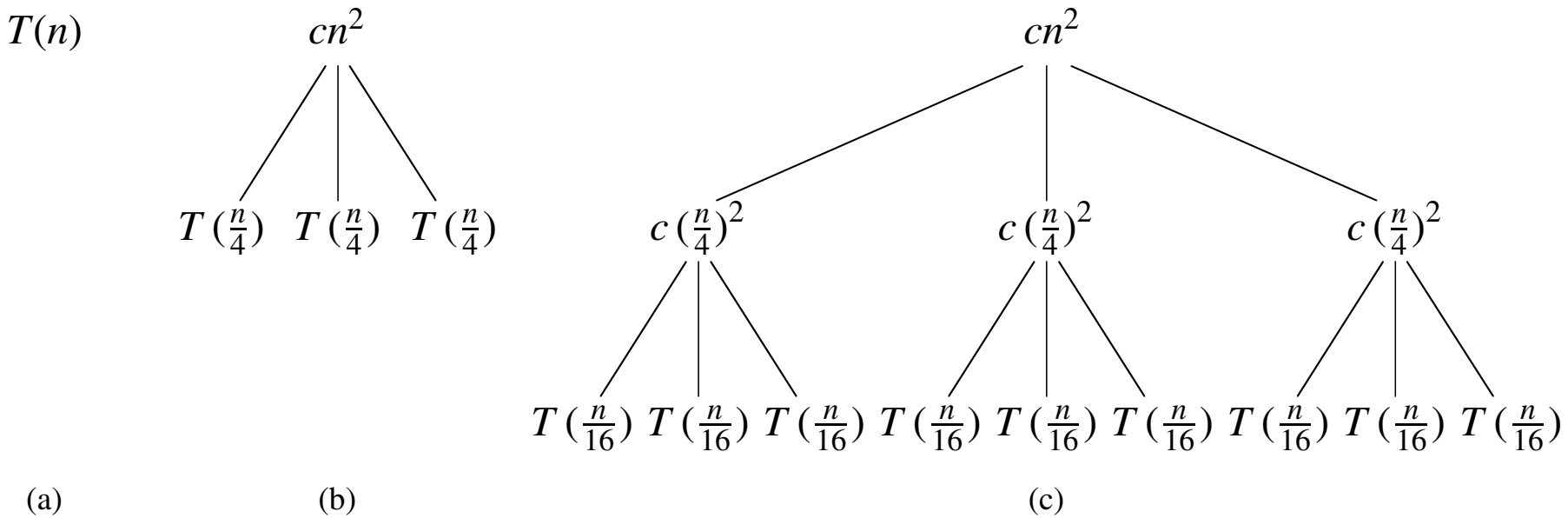
où $a \geq 1, b > 1$ et $f(n)$ est une fonction donnée.

À vous de jouer !

Voir le Chapitre 4 de "Introduction à l'algorithmique", T. H. Cormen, C. E. Leiserson, R. L. Rivest, Dunod, 1994.

- Dépliage de la récurrence.
- Méthode par substitution : on devine une solution et on vérifie qu'elle marche.
- Méthode par substitution partielle : on devine une solution mais sans préciser les constantes.

Cas de $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$



Cas de $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

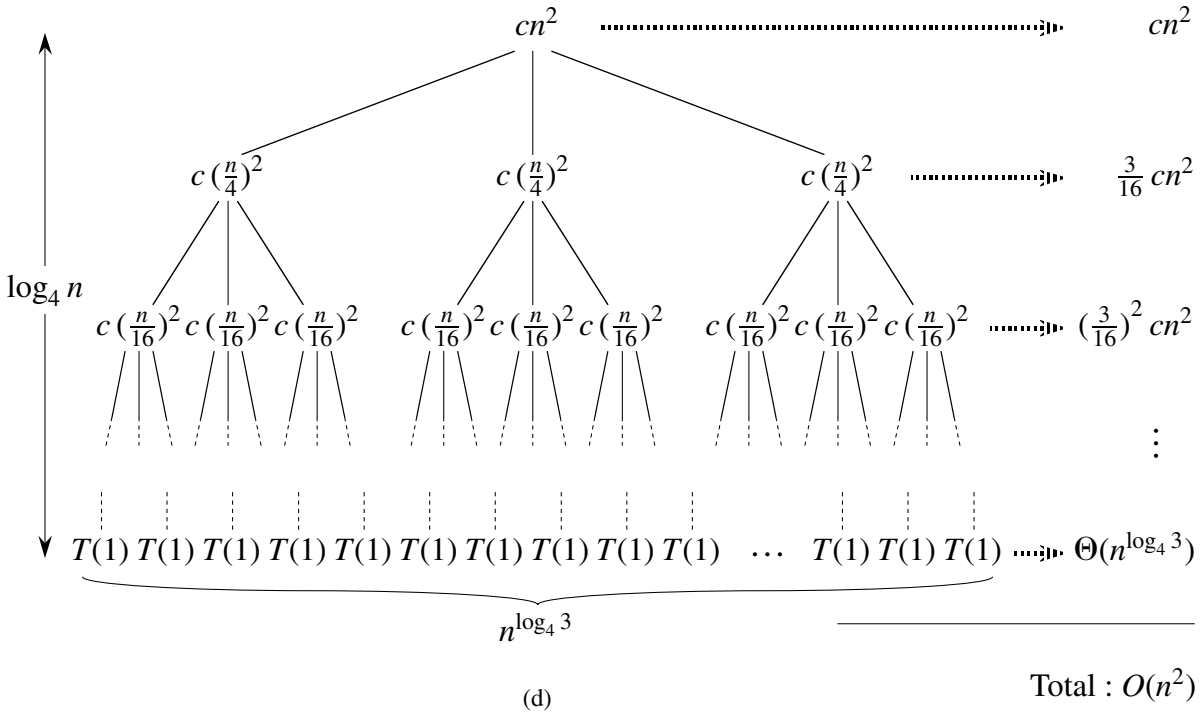


Figure 4.1 La construction d'un arbre récursif pour la récurrence $T(n) = 3T(n/4) + cn^2$. La partie **(a)** montre $T(n)$, progressivement développé dans les parties **(b)–(d)** pour former l'arbre récursif. L'arbre entièrement développé, en partie (d), a une hauteur de $\log_4 n$ (il comprend $\log_4 n + 1$ niveaux).

Le dernier niveau (profondeur $log_4 n$) a $3^{log_4 n} = n^{log_4 3}$ feuilles, qui ont chacunes un coût $T(1)$, ce qui donne un coût total de $n^{log_4 3} T(1)$, qui est dans $\Theta(n^{log_4 3})$.

Cas de $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

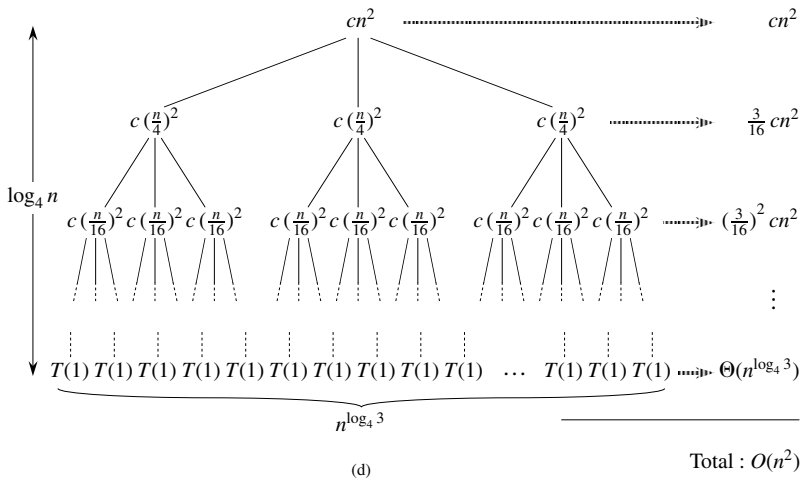


Figure 4.1 La construction d'un arbre récursif pour la récurrence $T(n) = 3T(n/4) + cn^2$. La partie **(a)** montre $T(n)$, progressivement développé dans les parties **(b)–(d)** pour former l'arbre récursif. L'arbre entièrement développé, en partie (d), a une hauteur de $\log_4 n$ (il comprend $\log_4 n + 1$ niveaux).

On cumule maintenant les coûts de tous les niveaux, afin de calculer le coût global de l'arbre :

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \,. \end{aligned}$$

Cas de $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ (suite)

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) = \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) = O(n^2) \,.
\end{aligned}
$$

Pour vérifier qu'une borne supérieure de la récurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ est $O(n^2)$, *i.e.* $T(n) \leq dn^2$ pour une certaine constante $d > 0$, on utilise la méthode de substitution :

$$
\begin{aligned}
T(n) &\leqslant 3T(\lfloor n/4 \rfloor) + cn^2 \leqslant 3d \lfloor n/4 \rfloor^2 + cn^2 \\
&\leqslant 3d(n/4)^2 + cn^2 = \frac{3}{16} dn^2 + cn^2 \leqslant dn^2
\end{aligned}
$$

Cas de $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ (suite)

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) = \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) = O(n^2)\,. \end{aligned}$$

Pour vérifier qu'une borne supérieure de la récurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ est $O(n^2)$, *i.e.* $T(n) \leq dn^2$ pour une certaine constante $d > 0$, on utilise la méthode de substitution :

$$\begin{aligned} T(n) &\leqslant 3T(\lfloor n/4 \rfloor) + cn^2 \leqslant 3d \lfloor n/4 \rfloor^2 + cn^2 \\ &\leqslant 3d(n/4)^2 + cn^2 = \frac{3}{16} dn^2 + cn^2 \leqslant dn^2 \end{aligned}$$

La dernière inéquation étant vraie dès $n > n_0 = 0$ en choisissant $d \geq (\frac{16}{13})c$.

Cas de $T(n) = T(\lceil n/3 \rceil) + T(\lceil 2n/3 \rceil) + cn$

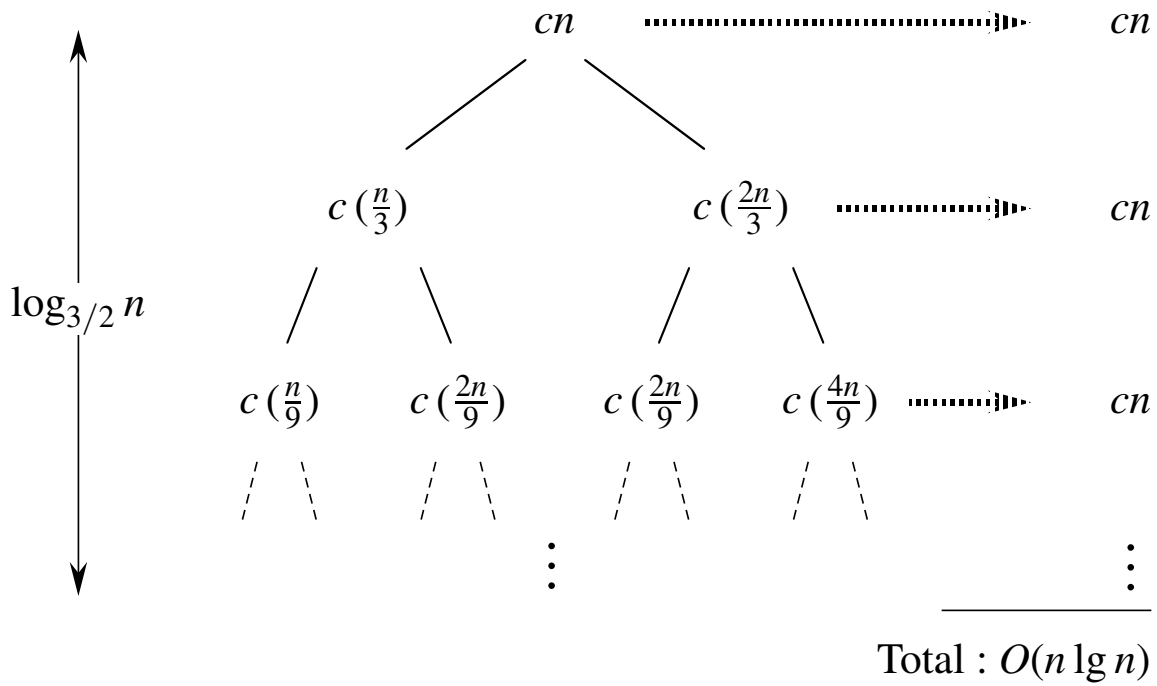


Figure 4.2 Un arbre récursif pour la récurrence $T(n) = T(n/3) + T(2n/3) + cn$.

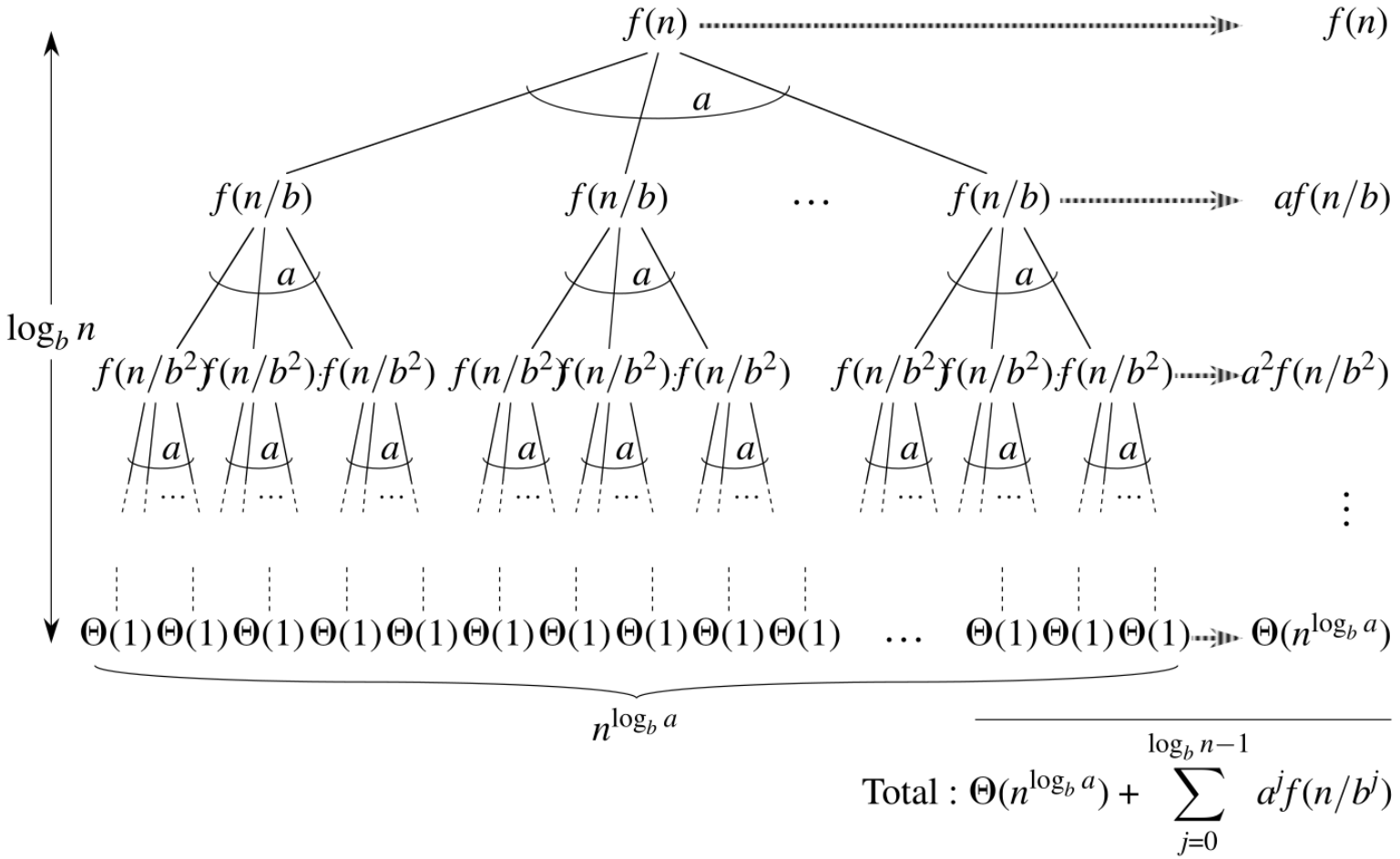
À vous de jouer !

Quel est l'ordre de grandeur de $T(n)$?

Entraînez-vous !

Voir l'Exercice 4.1. du Cormen [CLR96]

Cas général $T(n) = aT(\lceil n/b \rceil) + f(n)$ avec $a > 0, b > 1$



Les cas $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ avec $a > 0, b > 1, d \geq 0$

Théorème 3.1 (Théorème fondamental)

Si $T(n) = aT(n/b) + O(n^d)$ avec $a \geq 1, b > 1, d \geq 0$, alors

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

À vous de jouer !

Voir la preuve du Théorème 3.1 en Section 4.4 de [CLRS01], mais nous en verrons des cas simples un peu plus loin.

À vous de jouer !

Appliquer le Théorème 3.1 (bien pratique !) aux exemples dans l'Exercice 4.1. du Cormen [CLR96] que vous avez traités pour vérifier que vous aviez juste.

Nous allons voir de nombreux exemples d'utilisation de DR

1. Le tri rapide (Quicksort)
2. Calcul du médian ou problème de sélection
3. Multiplications de nombres à n digits
4. Multiplications de matrices $n \times n$
5. Les deux points les plus rapprochés (si le temps)

mais il y a aussi de nombreuses applications en traitement du signal (l'algorithme FFT, pour "Fast Fourrier Transformation"), d'autres cas en géométrie (enveloppe convexe) menant à de grands progrès en conception et dessins assistés par ordinateur.

Le tri rapide/Quicksort

(Le tri)

Entrée : un tableau A d'entiers
Sortie : une permutation de A triée par ordre croissant

Découvert en 1962 par Tony Hoare (11 Janvier 1934-) : Tri-Rapide$(A, 1, longeur[A])$

Algorithm 2 Tri-Rapide(A, p, r)

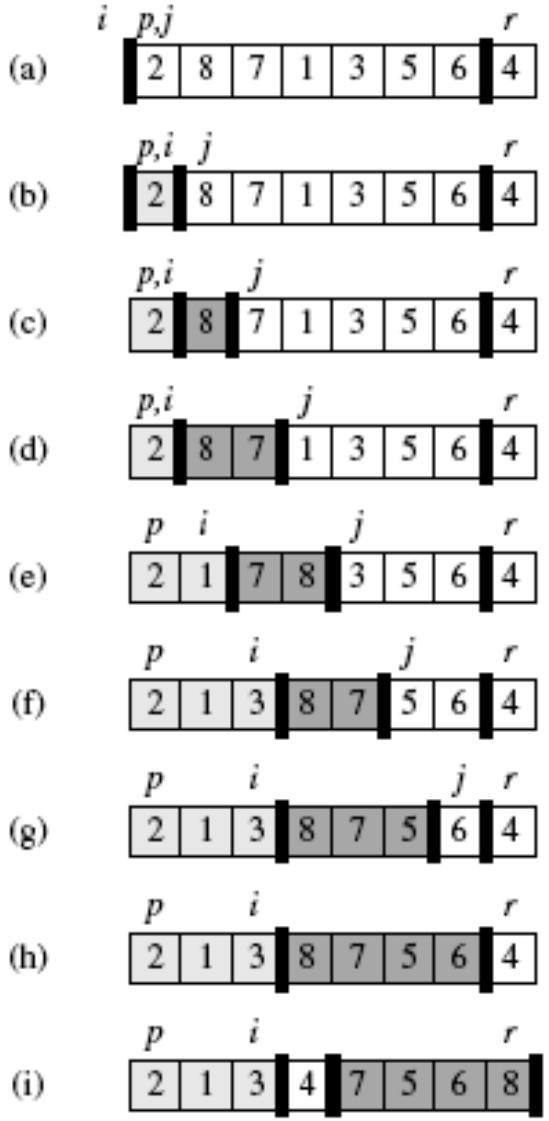
1: **if** $p < r$ **then** ▷ sinon il n'y a rien à faire
2: $q \leftarrow$ Partition(A, p, q, r) ▷ on trouve la place finale de $A[r]$ quand $A[p..r]$ sera trié
3: Tri-Rapide$(A, p, q-1)$ ▷ on trie le sous-tableau de gauche
4: Tri-Rapide$(A, q+1, r)$ ▷ on trie le sous-tableau de droite
5: **end if**

Lemme 6

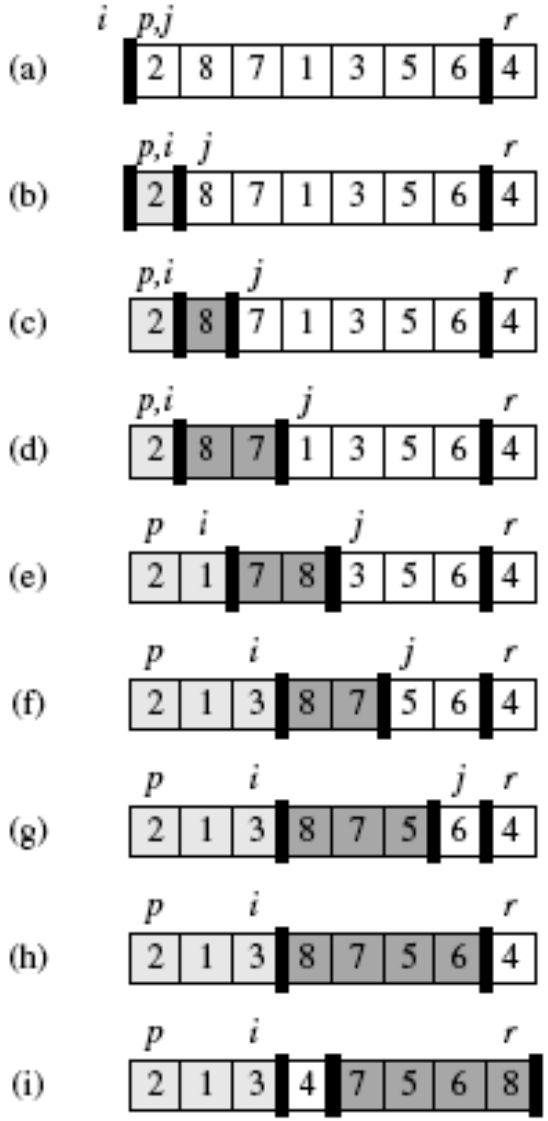
Après l'appel Partition(A, p, q, r), $A[q]$ *est à sa place :*

$$\text{pour tout } p \leq k < q, A[k] \leq A[q], \text{ et pour tout } q < k \leq r, A[q] < A[k]$$

Algorithme de PARTITION(A, p, r)



Algorithme de PARTITION(A, p, r)



PARTITION(A, p, r)

1 $x \leftarrow A[r]$
2 $i \leftarrow p - 1$
3 **pour** $j \leftarrow p$ **à** $r - 1$
4 **faire si** $A[j] \leqslant x$
5 **alors** $i \leftarrow i + 1$
6 permuter $A[i] \leftrightarrow A[j]$
7 permuter $A[i + 1] \leftrightarrow A[r]$
8 **retourner** $i + 1$

Clairement PARTITION(A, p, r) s'exécute en $\Theta(r - p + 1)$.

Correction de PARTITION(A, p, r)

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  pour j ← p à r − 1
4      faire si A[j] ⩽ x
5                alors  i ← i + 1
6                       permuter A[i] ↔ A[j]
7  permuter A[i + 1] ↔ A[r]
8  retourner i + 1
```

Correction de PARTITION(A, p, r)

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  pour j ← p à r − 1
4      faire si A[j] ⩽ x
5          alors i ← i + 1
6                permuter A[i] ↔ A[j]
7  permuter A[i + 1] ↔ A[r]
8  retourner i + 1
```

Invariant de boucle

$$\forall i', p \leq i' \leq i, A[i'] \leq x \text{ et } \forall j', i+1 \leq j' \leq j-1, A[j'] > x$$

Correction de PARTITION(A, p, r)

Invariant de boucle

$$\forall i', p \leq i' \leq i, A[i'] \leq x \text{ et } \forall j', i+1 \leq j' \leq j-1, A[j'] > x$$

Conservation

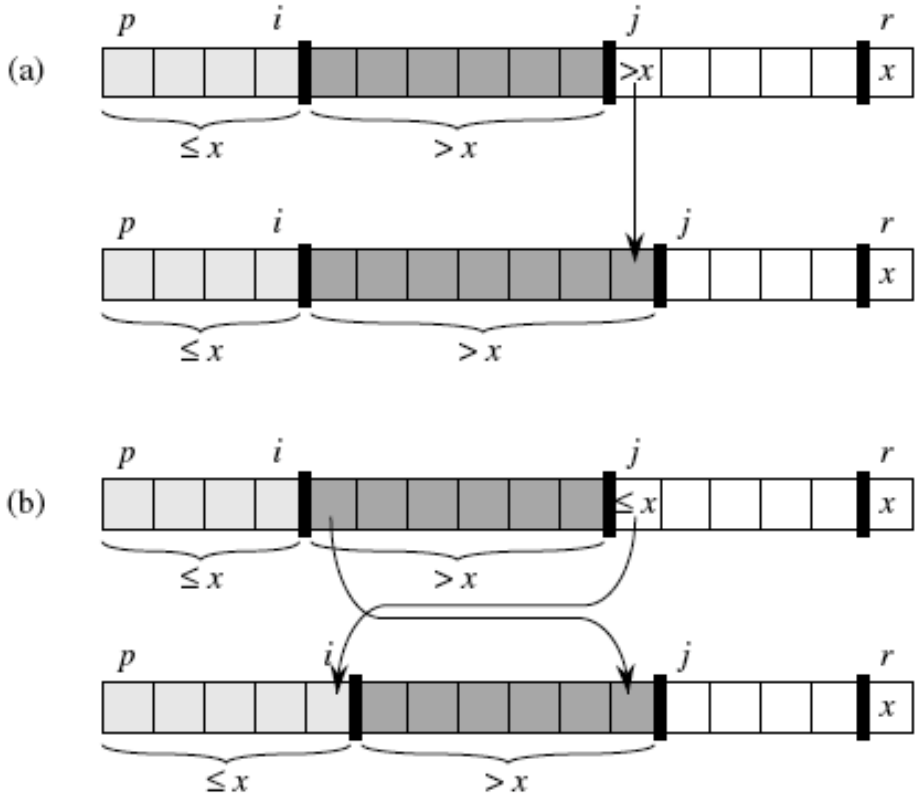


Figure 7.3 Les deux cas pour une itération de la procédure PARTITION. **(a)** Si $A[j] > x$, l'unique action est d'incrémenter j, ce qui conserve l'invariant de boucle. **(b)** Si $A[j] \leqslant x$, l'indice i est incrémenté, $A[i]$ et $A[j]$ sont échangés, puis j est incrémenté. Ici aussi, l'invariant de boucle est conservé.

Complexité du tri rapide

- Le cas le plus défavorable intervient pour le tri rapide quand PARTITION produit un sous-problème à $n-1$ éléments et un autre avec 0 élément. Cas d'une séquence déjà triée (croissante ou décroissante). La récurrence devient alors

$$T(n) = T(n-1) + \Theta(n)$$

de sorte que $T(n) = \Theta(n^2)$.

- Le cas le plus favorable s'obtient lorsque le pivot vient toujours se placer "au milieu" : la récurrence devient

$$T(n) = 2T(n/2) + \Theta(n)$$

dont on déduit par le Théorème Fondamental que $T(n) = O(nlogn)$.

Remarque 3.1

Dans [CLRS01, Chapitre 7] consacré aux algorithmes probabilistes, il est montré que la complexité en moyenne est $O(nlogn)$ si le pivot est choisi aléatoirement.

Complexité du tri rapide

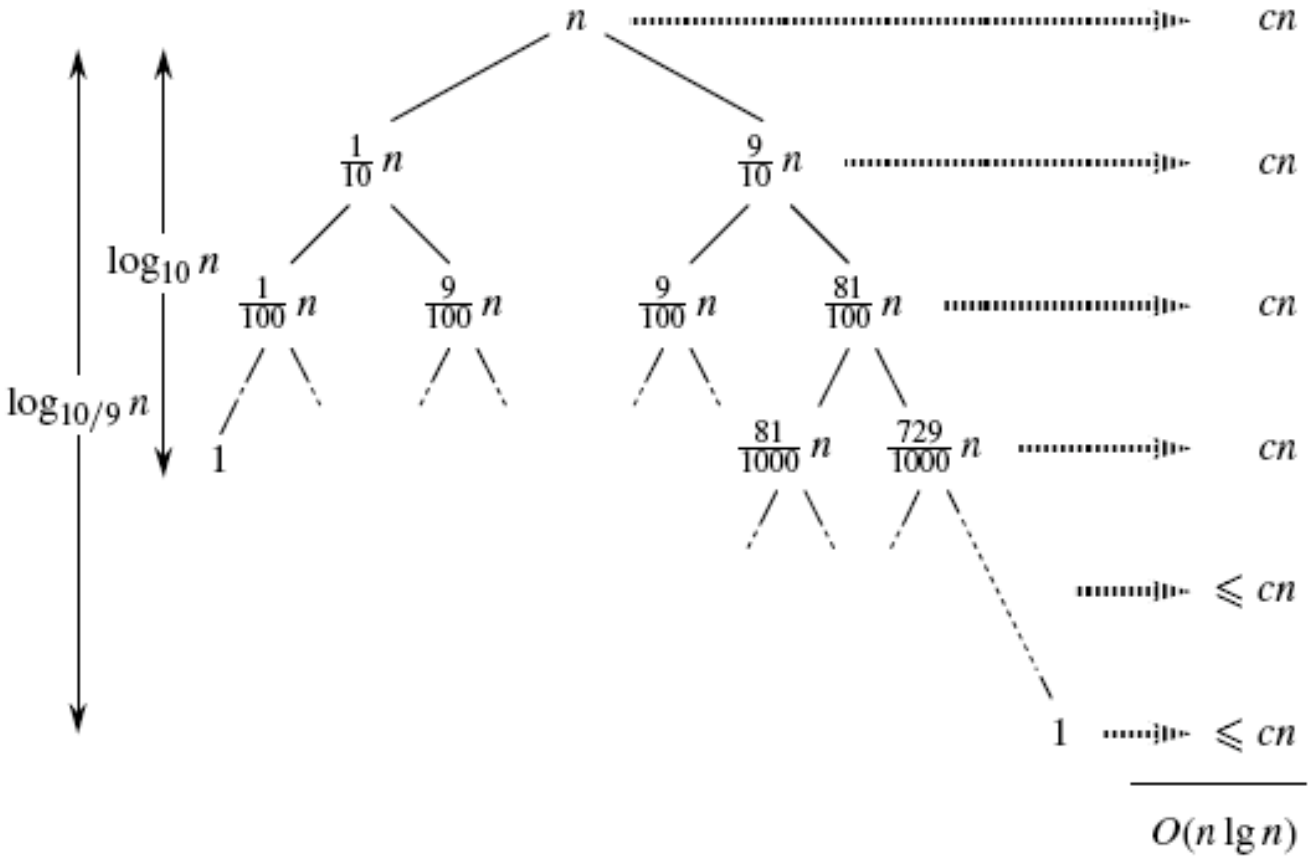


Figure 7.4 Arbre récursif de TRI-RAPIDE dans lequel PARTITION produit toujours une décomposition 9-1, donnant ainsi un temps d'exécution $O(nlgn)$. Les nœuds montrent les tailles de sous problème, les coûts par niveau figurant à droite. Les coûts par niveau incluent la constante c implicitement contenue dans le terme $\Theta(n)$

Complexité intrinsèque du tri par comparaison

On s'appuie sur les *arbres de décision*.

P. ex. celui du tri par insertion pour un tableau à 3 éléments.

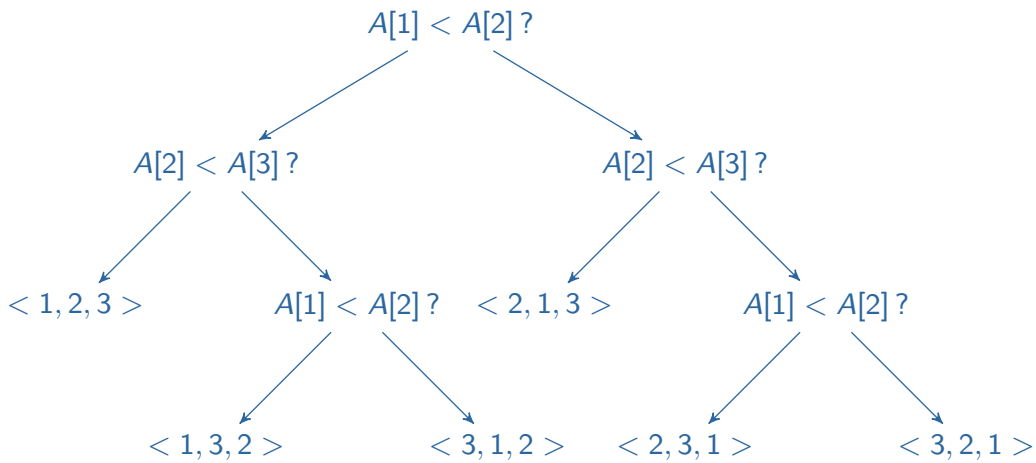
```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3           ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4           i ← j − 1
5           tant que i > 0 et A[i] > clé
6               faire A[i + 1] ← A[i]
7                    i ← i − 1
8           A[i + 1] ← clé
```

Complexité intrinsèque du tri par comparaison

On s'appuie sur les *arbres de décision*.

P. ex. celui du tri par insertion pour un tableau à 3 éléments.

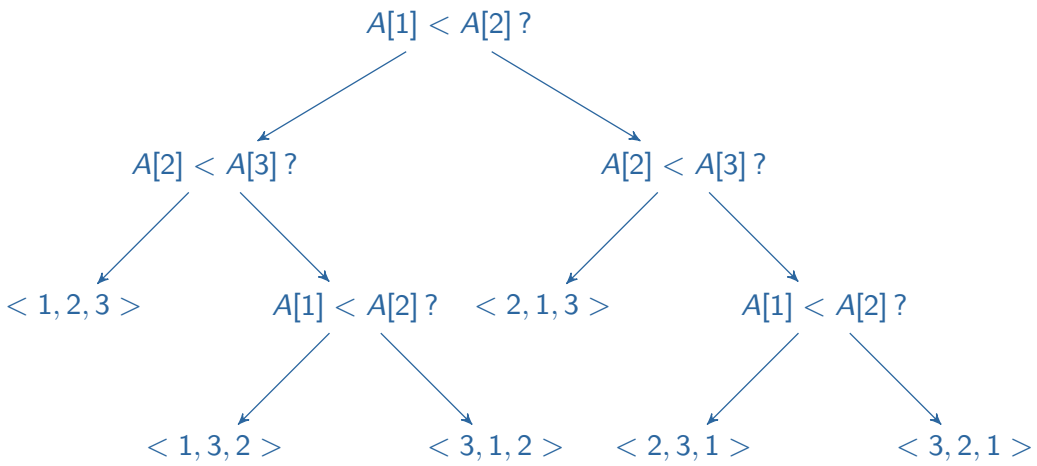
```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3          ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4          i ← j − 1
5          tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← clé
```



Complexité intrinsèque du tri par comparaison

On s'appuie sur les *arbres de décision*.

P. ex. celui du tri par insertion pour un tableau à 3 éléments.



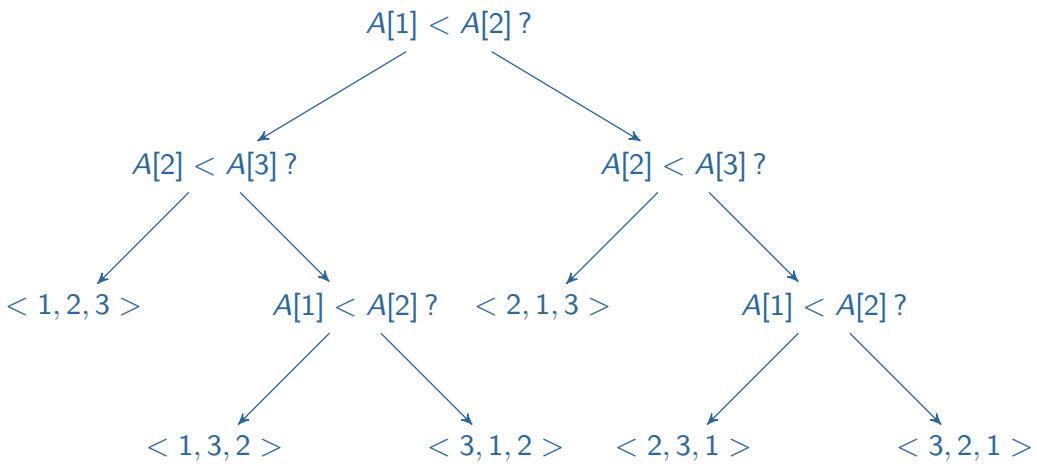
- Un arbre binaire de hauteur h a au plus 2^h feuilles.
- Donc l'arbre de décision doit avoir une hauteur d'au moins $log(n!)$.
- Or $n! \geq (\frac{n}{2})^{(\frac{n}{2})}$ donc $log(n!) \geq cnlog(n)$ (avec $c > 0$) d'où

$$log(n!) \in \Omega(nlog(n)).$$

Complexité intrinsèque du tri par comparaison

On s'appuie sur les *arbres de décision*.

P. ex. celui du tri par insertion pour un tableau à 3 éléments.



- Un arbre binaire de hauteur h a au plus 2^h feuilles.
- Donc l'arbre de décision doit avoir une hauteur d'au moins $log(n!)$.
- Or $n! \geq (\frac{n}{2})^{(\frac{n}{2})}$ donc $log(n!) \geq cnlog(n)$ (avec $c > 0$) d'où

$$log(n!) \in \Omega(nlog(n)).$$

Théorème ([DPV06, page 63])

Tout arbre de décision d'un tri pour n éléments a une hauteur en $\Omega(nlog(n))$.

Le problème de Sélection

Définition 7

Le i-ème rang d'une liste de n entiers est le i-ème plus petit élément.

Exemple 8

- Pour la liste $S = [10, 20, 30, 40, 10]$ l'élément de rang 3 est 20, et celui de rang 1 est 10.
- Un minimum est l'élément de rang 1, et un maximum est l'élément de rang n.
- Le médian est l'élément "au milieu" :
  Si n est impair, il est unique et c'est celui de rang $(n+1)/2$, sinon (n est pair) et on convient de prendre $n/2$.

  Ainsi le médiant est l'élément de rang $\lfloor (n+1)/2 \rfloor$.

Le problème de Sélection

Définition 7

Le i-ème rang d'une liste de n entiers est le i-ème plus petit élément.

Exemple 8

- Pour la liste $S = [10, 20, 30, 40, 10]$ l'élément de rang 3 est 20, et celui de rang 1 est 10.
- Un minimum est l'élément de rang 1, et un maximum est l'élément de rang n.
- Le médian est l'élément "au milieu" :
  Si n est impair, il est unique et c'est celui de rang $(n+1)/2$, sinon (n est pair) et on convient de prendre $n/2$.

  Ainsi le médiant est l'élément de rang $\lfloor (n+1)/2 \rfloor$.

(Le problème de Sélection)

Entrée : une liste d'entiers S et un entier i
Sortie : l'élément de rang i

Calcul de l'élément de rang i

(Le problème de Sélection)

Entrée : une liste d'entiers S et un entier i
Sortie : l'élément de rang i

- Solution 1, le coup de marteau : on trie S et on prend le i^{e} élément.

  Dans $O(nlogn)$, or on fait plus que demandé puisqu'on trie tous les éléments.

  Donc peut-on faire mieux ?

  Typiquement peut-on le faire en $O(n)$?

Calcul de l'élément de rang i

(Le problème de Sélection)

Entrée : une liste d'entiers S et un entier i
Sortie : l'élément de rang i

- Solution 1, le coup de marteau : on trie S et on prend le i^{e} élément.

  Dans $O(nlogn)$, or on fait plus que demandé puisqu'on trie tous les éléments.

  Donc peut-on faire mieux ?

  Typiquement peut-on le faire en $O(n)$?

- Solution 2 : Diviser-pour-Régner.

Un algorithme Diviser-pour-Régner pour le problème de sélection

Pour une valeur v fixée, on scinde S en 3 sous-listes d'éléments :

- $S_{<v}$ ceux plus petits que v,
- $S_{=v}$ ceux égaux à v,
- $S_{>v}$ ceux plus grands que v.

Exemple 9

Soit $S = [2, 36, 6, 21, 8, 13, 11, 20, 6, 4, 1]$ et $v = 6$.

On a $S_{<v} = [2, 4, 1]$ et $S_{=v} = [6, 6]$ et $S_{>v} = [36, 21, 8, 13, 11, 20]$

On remarque que $EltDeRang(S, 8) = EltDeRang(S_{>v}, 3)$.

Plus généralement, pour toute valeur v, on a :

$$EltDeRang(S, i) = \begin{cases} EltDeRang(S_{<v}, i) & \text{if } i \leq |S_{<v}| \\ v & \text{if } |S_{<v}| < i \leq |S_{<v}| + |S_{=v}| \\ EltDeRang(S_{>v}, i - (|S_{<v}| + |S_{=v}|)) & \text{if } i > |S_{<v}| + |S_{=v}| \end{cases}$$

Complexité du calcul du médian $EltDeRang(S, \lfloor (n+1)/2 \rfloor)$ (où $n = |S|$)

On a

$$EltDeRang(S,i) = \begin{cases} EltDeRang(S_{<v}, i) & \text{if } i \leq |S_{<v}| \\ v & \text{if } |S_{<v}| < i \leq |S_{<v}| + |S_{=v}| \\ EltDeRang(S_{>v}, i - (|S_{<v}| + |S_{=v}|)) & \text{if } i > |S_{<v}| + |S_{=v}| \end{cases}$$

et on est libre de choisir la valeur v.

- Le calcul de $S_{<v}$, $S_{=v}$ et $S_{>v}$ se fait en $O(n)$
  et on peut même faire ce calcul en place (c-à-d. sans mémoire auxiliaire).

Complexité du calcul du médian $EltDeRang(S, \lfloor (n+1)/2 \rfloor)$ (où $n = |S|$)

On a

$$EltDeRang(S,i) = \begin{cases} EltDeRang(S_{<v}, i) & \text{if } i \leq |S_{<v}| \\ v & \text{if } |S_{<v}| < i \leq |S_{<v}| + |S_{=v}| \\ EltDeRang(S_{>v}, i - (|S_{<v}| + |S_{=v}|)) & \text{if } i > |S_{<v}| + |S_{=v}| \end{cases}$$

et on est libre de choisir la valeur v.

- Le calcul de $S_{<v}$, $S_{=v}$ et $S_{>v}$ se fait en $O(n)$
  et on peut même faire ce calcul en place (c-à-d. sans mémoire auxiliaire).

- Imaginons qu'à chaque appel récursif on choisit une valeur v de sorte que la taille de $S_{<v}|$ et de $|S_{>v}|$ soit de l'ordre de $\frac{1}{2}|S|$, de sorte que la complexité du calcul de $EltDeRang(S, \lfloor (n+1)/2 \rfloor)$ est :

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

  Donc un algorithme en $O(n)$:-)

Complexité du calcul du médian $EltDeRang(S, \lfloor (n+1)/2 \rfloor)$ (où $n = |S|$)

On a

$$EltDeRang(S,i) = \begin{cases} EltDeRang(S_{<v}, i) & \text{if } i \leq |S_{<v}| \\ v & \text{if } |S_{<v}| < i \leq |S_{<v}| + |S_{=v}| \\ EltDeRang(S_{>v}, i - (|S_{<v}| + |S_{=v}|)) & \text{if } i > |S_{<v}| + |S_{=v}| \end{cases}$$

et on est libre de choisir la valeur v.

- Le calcul de $S_{<v}$, $S_{=v}$ et $S_{>v}$ se fait en $O(n)$
  et on peut même faire ce calcul en place (c-à-d. sans mémoire auxiliaire).

- Imaginons qu'à chaque appel récursif on choisit une valeur v de sorte que la taille de $S_{<v}|$ et de $|S_{>v}|$ soit de l'ordre de $\frac{1}{2}|S|$, de sorte que la complexité du calcul de $EltDeRang(S, \lfloor (n+1)/2 \rfloor)$ est :

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

  Donc un algorithme en $O(n)$:-)

  On peut établir qu'il suffit de choisir v aléatoirement à chaque appel récursif.
  (hors cours, voir [DPV06, p. 65])

Multiplications de nombres à n digits

(Multiplications de nombres à n digits)

Entrée : deux entiers codés sur n digits
Sortie : leur produit

La multiplication à l'école :

$$
\begin{array}{r}
31415962 \\
\times\ 27182818 \\
\hline
251327696 \\
31415962 \\
251327696 \\
62831924 \\
251327696 \\
31415962 \\
219911734 \\
62831924 \\
\hline
853974377340916
\end{array}
$$

donne un algorithme en $O(n^2)$, qu'on peut améliorer avec une approche DR.

On décompose les nombres

Exemple 10

$31415962 = 10^4 * 3141 + 5962$ et $27182818 = 10^4 * 2718 + 2818$

On remarque que

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

On en dérive un algorithme :

Pour le cas où n est impair, par exemple $n = 3$, il faut comprendre que les nombres a et c, étant de longueur 1, sont préfixés par 0 pour être bien formés dans l'appel $\mathrm{MULTIPLY}(a, c, 2)$.
Pour le voir, appliquer l'algorithme à par exemple à $x = 342$ et $y = 756$.

```
MULTIPLY(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m
      d ← ⌊y/10^m⌋;  c ← y mod 10^m
      e ← MULTIPLY(a, c, m)
      f ← MULTIPLY(b, d, m)
      g ← MULTIPLY(b, c, m)
      h ← MULTIPLY(a, d, m)
      return 10^{2m} e + 10^m (g + h) + f
```

On obtient ainsi $T(n) = 4T(\lceil n/2 \rceil) + O(n)$ et $T(1) = 1$, mais c'est dans $O(n^2)$.

Anatolii Karatsuba in 1962 (à la Gauss 1800) + Donald Knuth

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

mais si on remarque que $bc + ad = ac + bd - (a - b)(c - d)$,
on peut ne calculer que trois produits :

$$= 10^{2m} ac + 10^m (ac + bd - (a - b)(c - d)) + bd$$

À vous de jouer !

Appliquez l'algorithme pour calculer 1237×2587.

Algorithme de Karatsuba

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (ac + bd - (a-b)(c-d)) + bd$$

```
FastMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m
      d ← ⌊y/10^m⌋;  c ← y mod 10^m
      e ← FastMultiply(a, c, m)
      f ← FastMultiply(b, d, m)
      g ← FastMultiply(a − b, c − d, m)
      return 10^{2m} e + 10^m (e + f − g) + f
```

$$T(n) = 3T(\lceil n/2 \rceil) + O(n) \text{ et } T(1) = 1$$

cela donne un algorithme en $O(n^{log_2 3})$, or $log_2 3 \approx 1.585 < 2$.

Multiplications de matrices $n \times n$

(Multiplications de matrices $n \times n$)

Entrée : deux matrices carrées X et Y de dimension n
Sortie : $X * Y$

Remarque 3.2

C'est très bien expliqué dans `https://fr.wikipedia.org/wiki/Algorithme_de_Strassen`.

Par un produit habituel

Les trois matrices A, B et C sont divisées en matrices par blocs de taille égale :

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

où

$$\mathbf{A}_{i,j}, \mathbf{B}_{i,j}, \mathbf{C}_{i,j} \in F^{n/2} \times F^{n/2}.$$

On a alors

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \\ \mathbf{C}_{1,2} &= \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\ \mathbf{C}_{2,1} &= \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} \\ \mathbf{C}_{2,2} &= \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2} \end{aligned}$$

Cette méthode nécessite 8 multiplications de matrices pour calculer les $C_{i,j}$, comme dans le produit classique.

On a alors $T(n) = 8\,T(n/2) + \Theta(n^2)$, et donc $T(n) \in O(n^{\log_2 8}) = O(n^3)$

Par la méthode de Strassen

La force de l'algorithme de Strassen réside dans un ensemble de sept nouvelles matrices M_i qui vont servir à exprimer les $C_{i,j}$ avec seulement 7 multiplications au lieu de 8 :

$$\mathbf{M}_1 := (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2})$$
$$\mathbf{M}_2 := (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1}$$
$$\mathbf{M}_3 := \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2})$$
$$\mathbf{M}_4 := \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1})$$
$$\mathbf{M}_5 := (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2}$$
$$\mathbf{M}_6 := (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2})$$
$$\mathbf{M}_7 := (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})$$

Les $C_{i,j}$ sont alors exprimées comme

$$\mathbf{C}_{1,1} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$$
$$\mathbf{C}_{1,2} = \mathbf{M}_3 + \mathbf{M}_5$$
$$\mathbf{C}_{2,1} = \mathbf{M}_2 + \mathbf{M}_4$$
$$\mathbf{C}_{2,2} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$$

On a alors $T(n) = 7T(n/2) + \Theta(n^2)$, et donc $T(n) \in O(n^{\log_2 7}) \approx O(n^{2,807})$

Le problème des deux points les plus rapprochés

Preparata (Franco P.) and Shamos (Michael Ian). Computational Geometry : An Introduction. Springer-Verlag, 1985.

Remarque 3.3

À regarder par vous-même.

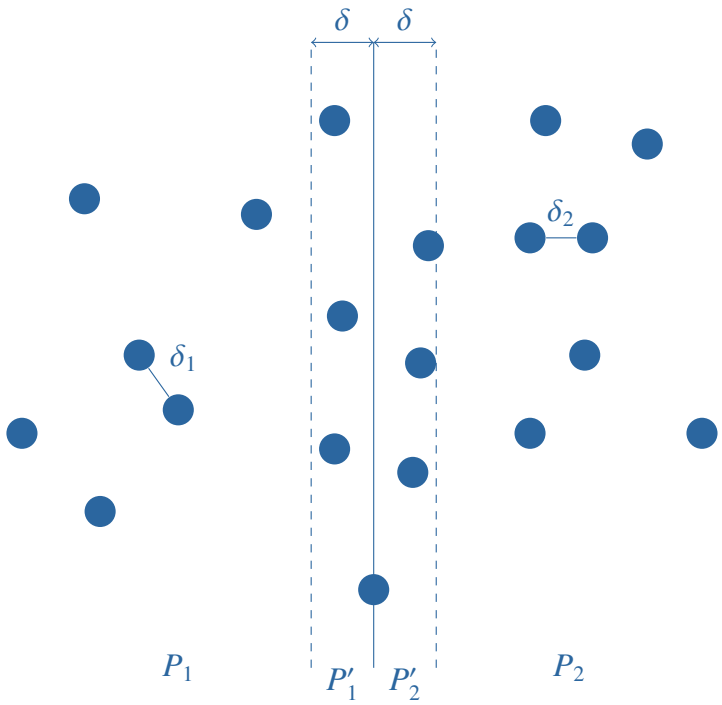
(Les deux points les plus rapprochés)

Entrée : un tableau P de points dans le plan
Sortie : $min_{a,b \in T, a \neq b} d(a,b)$

où $d(a,b) = \sqrt{(a.x - b.x)^2 + (a.y - b.y)^2}$, noté classiquement $\|a - b\|_2$

Diviser pour Régner : on coupe le plan en deux et on calcule δ_1 et δ_2 pour les deux sous-ensembles de points.

$\delta = min\{\delta_1, \delta_2\}$

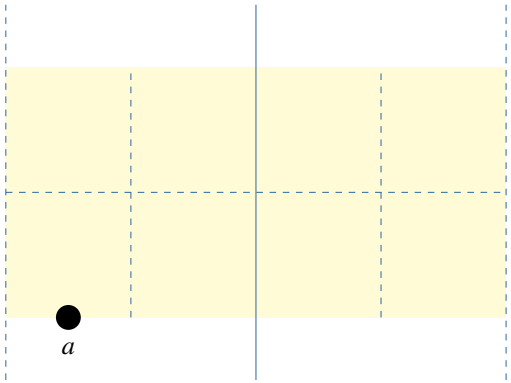


On cherche la distance minimale entre deux points distincts de la bande verticale $P' = P'_1 \cup P'_2$.

Calcul efficace dans la bande P'

On trie les points par ordonnées croissantes et on montre que pour un point a, il suffit de calculer les distances entre a et b où b est parmi les 7 points qui succèdent à a dans P'.

Si un point au dessus du point a est distant de moins de δ de a alors il est dans le rectangle jaune de taille $2\delta \times \delta$.



Lemme 11 (non démontré)

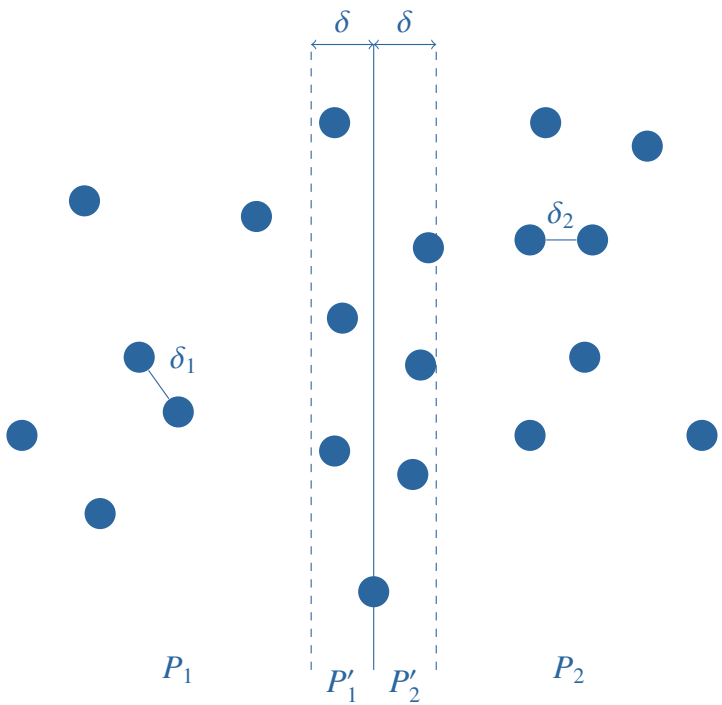
Au plus 8 points de P peuvent se trouver à l'intérieur de ce rectangle $2\delta \times \delta$.

Ainsi, il suffit de tester a avec les 7 autres points suivants pour l'ordre d'ordonnées croissantes.

Algorithme *plusRapproches*(P)

Entrées : Un tableau de points P du plan
Sorties : La distance minimale, i.e. $min_{a,b \in T | a \neq b} \|a - b\|_2$

```
1 X := tri(P, par abscisses croissantes) ;
2 Y := tri(P, par ordonnées croissantes) ;
3 plusRapprochesRec(X, Y)
```



Algorithme *plusRapprochesRec*(X, Y)

Entrées : Deux tableaux X, Y qui contiennent les même points, X est trié par absisse croissante et Y est trié par ordonnée croissante

Sorties : La distance minimale, i.e. $min_{a,b \in X | a \neq b} ||a - b||_2$

1 **si** card$(P) \leq 3$ **alors**
2 **retourner** *méthode naïve*
3 **sinon**
4 $X_G := X[1, \lfloor \frac{|X|}{2} \rfloor]$
5 $X_D := X[\lfloor \frac{|X|}{2} \rfloor + 1, |X|]$
6 $x_{sep} := X_G[\frac{|X|}{2}].x$
7 $Y_G :=$ extraire les éléments de Y d'abcsisse $\leq x_{sep}$
8 $Y_D :=$ extraire les éléments de Y d'abcsisse $> x_{sep}$
9 $\delta_G := plusRapprochesRec(X_G, Y_G)$
10 $\delta_D := plusRapprochesRec(X_D, Y_D)$
11 $\delta := min(\delta_G, \delta_D)$
12 $Y' :=$ extraire les éléments de Y qui sont dans la bande verticale d'absisse x_{sep} et de largeur 2δ ;
13 **retourner** $min(\delta, plusRapprochesBande(Y', \delta))$

Les fonctions non données ici sont laissés en exercice.

Complexité de l'algorithme *plusRapproches*(P)

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$$
$$T(3) = T(2) = T(1) = \Theta(1)$$

cela donne un algorithme en $O(nlogn)$.

Retour sur le Théorème fondamental pour les complexités

Théorème 3.2

Si $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ avec $a > 0, b > 1, d \geq 0$, alors

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

Quelques rappels mathématiques

sur les logarithmes

- $c^{log_c m} = m$
- $log_c(mn) = log_c m + log_c n$
- $log_c(\frac{m}{n}) = log_c m - log_c n$
- $log_c(m^n) = n log_c m$
- Changement de base : $log_c m = \frac{log_s m}{log_s c}$

sur les séries

- Série arithmétique : $\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$.
- Série géométrique : $\sum_{i=0}^{n} q^i = 1 + q + q^2 + \ldots + q^n = \begin{cases} \frac{q^{n+1}-1}{q-1} & \text{si } q \neq 1 \\ i+1 & \text{si } q = 1 \end{cases}$

Complexités réductions simples 1/2

$$\begin{cases} T(n) = aT(n-1) + cn^k \\ T(1) = d \end{cases} \quad a \geq 1, c \geq 1, k \geq 0, d \geq 0 \text{ sont des constantes.}$$

$$\begin{aligned} T(n) &= aT(n-1) + cn^k \\ aT(n-1) &= a^2T(n-2) + ca(n-1)^k \\ a^2T(n-2) &= a^3T(n-3) + ca^2(n-2)^k \\ &\dots \\ a^{n-2}T(2) &= a^{n-1}T(1) + ca^{n-2}2^k \\ a^{n-1}T(1) &= a^{n-1}d \end{aligned}$$

$$T(n) = da^{n-1} + c\sum_{i=0}^{n-2} a^i(n-i)^k$$

Complexités réductions simples 2/2

Réduction simple : $T(n) = da^{n-1} + c\sum_{i=0}^{n-2} a^i (n-i)^k$

Cas particuliers intéressants :

RS1 : $k = 0, a = 1 T(n) = d + c(n-1) = O(n)$	[linéaire]
RS2 : $k = 0, a > 1 T(n) = da^{n-1} + c\frac{a^{n-1}-1}{a-1} = O(a^n)$	[exponentiel]
RS3 : $k = 1, a = 1 T(n) = d + c\frac{(n-1)(n+2)}{2} = O(n^2)$	[polynomial]

Complexités réductions logarithmiques 1/2

$$\begin{cases} T(n) = aT(\frac{n}{b}) + cn^k \\ T(1) = d \end{cases} \quad a \geq 1, b \geq 2, c \geq 1, k \geq 0, d \geq 0 \text{ sont des constantes.}$$

On suppose $n = b^m$, donc $m = log_b n$ et $a^m = a^{log_b n} = n^{log_b a}$

$$\begin{array}{rl} T(b^m) & = aT(b^{m-1}) + cb^{km} \\ aT(b^{m-1}) & = a^2 T(b^{m-2}) + cab^{k(m-1)} \\ & \dots \\ a^{m-1}T(b) & = a^m T(1) + ca^{m-1}b^k \\ a^m T(1) & = a^m d \\ \hline T(n) & = T(b^m) = da^m + c\sum_{i=1}^{m} a^{m-i}b^{ki} = da^m + ca^m \sum_{i=1}^{m} \frac{b^{ki}}{a^i} \end{array}$$

Lorsqu'on s'intéresse seulement à l'ordre de grandeur des complexités, on peut simplifier la formule en posant $d = c$:

Réduction logarithmique : $T(n) = ca^m \sum_{i=0}^{m} \frac{b^{ki}}{a^i}$

Complexités réductions logarithmiques 1/2

Réduction logarithmique : $T(n) = ca^m \sum_{i=0}^{m} \frac{b^{ki}}{a^i}$

d'où 3 cas possibles en fonction de la valeur de $q = \frac{b^k}{a}$:

$$
\begin{array}{lll}
q < 1 : & T(n) = ca^m \frac{1-q^{m+1}}{1-q} = O(a^m) = O(n^{\log_b a}) \\
q = 1 : & T(n) = ca^m(m+1) = O(ma^m) = O(\log_b n \; n^{\log_b a}) \\
 & = O(\log_b n \; n^{\log_b b^k}) = O(n^k \log_b a) \\
q > 1 : & T(n) = ca^m \frac{q^{m+1}-1}{q-1} = O(a^m q^m) = O(a^m \frac{b^{km}}{a^m}) = O(n^k)
\end{array}
$$

RL1 : $a > b^k$	$T(n) = O(n^{\log_b a})$	[polynomial]
RL2 : $a = b^k$	$T(n) = O(n^k \log_b n)$	[en $O(n^{k+1})$ donc polynomial]
RL3 : $a < b^k$	$T(n) = O(n^k)$	[polynomial]

Pratique ...

À vous de jouer !

Pratiquez le théorème fondamental en revérifiant les résultats annoncés pour les systèmes d'équations de :

- Calcul du médian ou problème de sélection $T(n) = T(n/2) + O(n)$, donc un algorithme en ...
- Multiplications de nombres à n digits : 1) $T(n) = 4T(\lceil n/2 \rceil) + O(n)$ et $T(1) = 1$ donc un algorithme en ...puis $T(n) = 3T(\lceil n/2 \rceil) + O(n)$ et $T(1) = 1$, un algorithme en ...
- Les deux points les plus rapprochés $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$ et $T(3) = T(2) = T(1) = \Theta(1)$ donc en ...
- Multiplications de matrices $n \times n$ ...

Preuve de terminaison et de correction

Pour les algorithmes obéissant au paradigme Diviser-pour-Régner, donc intrinsèquement récursifs, on procède de la façon suivante.

- Terminaison : on exhibe un ordre *bien fondé* sur les problèmes, par exemple leur taille, et on montre que les appels récursifs ont des paramètres d'entrée plus petits que celui de l'appel initial.
- Correction (partielle) : on procède par induction. On vérifie
  (1) que le résultat est correct pour le(s) cas de base de la récurrence, et
  (2) en supposant que le résultat retourné par chacun des appels récursifs est correct, on montre que le résultat retourné par l'appel principal l'est aussi.

Cette technique s'applique à tout algorithme récursif, sans qu'il soit nécessairement de type Diviser-pour-Régner. Nous verrons de tels exemples dans les parties suivantes (algorithmes gloutons, Programmation Dynamique, etc.).

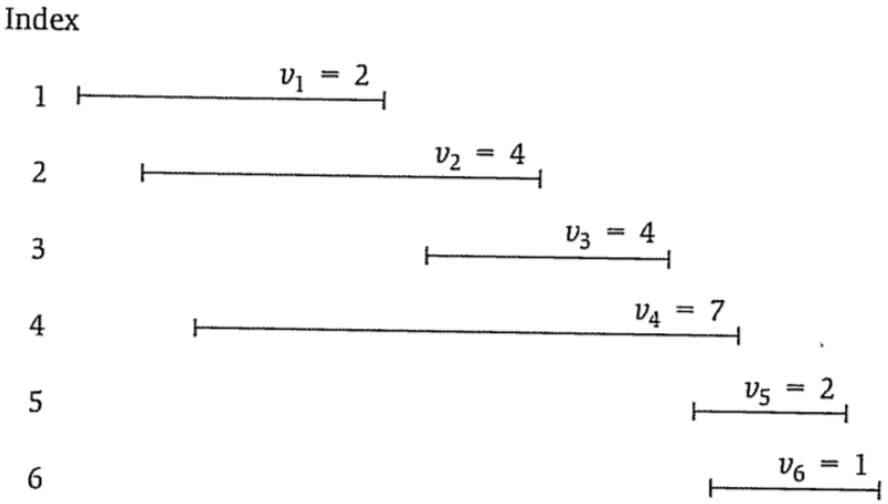
Programmation Dynamique

- Le concept a été introduit au début des années 1950 par Richard Bellman.
- Le terme “programmation” signifie planification/ordonnancement, e.g., *programmation d'un festival de musique*.

Weighted Interval Scheduling

(Weighted Interval Scheduling)

Entrée : n requêtes étiquetées $1, \ldots, n$, avec pour chaque requête i un intervalle $[s_i, f_i]$ d'exécution, et un poids v_i.
Sortie : Un sous-ensemble $S \subseteq \{1, \ldots, n\}$ de requêtes *compatibles* (leurs intervalles ne se chevauchent pas) qui maximise $\sum_{i \in S} v_i$.

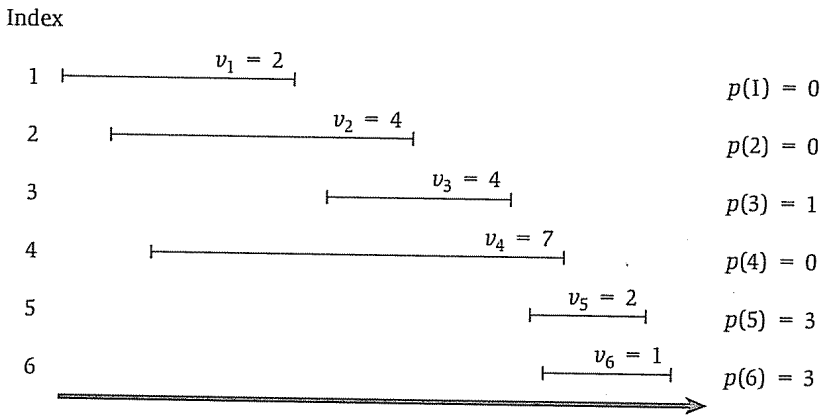


À vous de jouer !

Quel est le lien entre le problème Weighted Interval Scheduling et le problème des activités vu au Chapitre précédent ?

Weighted Interval Scheduling : une procédure récursive

- On suppose les requêtes numérotées selon l'ordre $f_1 \leq f_2 \ldots \leq f_n$. On dira que i est avant j si $f_i \leq f_j$, donc, avec la numérotation choisie, lorsque $i < j$.
- Pour un intervalle j, on définit $p(j)$ (pour précédent de j) comme le plus grand indice $i < j$ tel que i et j soient des intervalles compatibles, et il vaut 0 si aucun tel i n'existe.
  (Faire le lien avec le problème des activités)



  Par définition de $p(j)$, j n'est compatible avec aucune des requêtes $p(j)+1, \ldots, j-1$.
- Une solution optimale O est telle que :
  - si $n \in O$, alors les intervalles $p(n)+1, \ldots, n-1$ ne sont pas dans O, et O doit contenir une solution optimale du sous-problème pour les requêtes $\{1, \ldots, p(n)\}$;
  - sinon ($n \notin O$) O contient une solution optimale pour le sous-problème $\{1, \ldots, n-1\}$.

Une solution optimale O est telle que :

- si $n \in O$, alors les intervalles $p(n)+1, \ldots, n-1$ ne sont pas dans O, et O doit contenir une solution optimale du sous-problème pour les requêtes $\{1, \ldots, p(n)\}$;
- sinon ($n \notin O$) O contient une solution optimale pour le sous-problème $\{1, \ldots, n-1\}$.

Alors la valeur de la solution O optimale, notée $opt(n)$ est la meilleure parmis les deux cas précédents : $opt(n) = max(v_n + opt(p(n)), opt(n-1))$.
Notons P_j $(1 \leq j \leq n)$ le problème du Weighted Interval Scheduling pour l'ensemble des intervalles de 1 à j – le problème que l'on veut résoudre est P_n et notons O_j une solution optimale de P_j et de valeur $opt(j)$ – la solution optimale O du paragraphe précédent est donc O_n de valeur $opt(n)$. On convient que $opt(0) = 0$.

Une solution optimale O est telle que :

- si $n \in O$, alors les intervalles $p(n)+1, \ldots, n-1$ ne sont pas dans O, et O doit contenir une solution optimale du sous-problème pour les requêtes $\{1, \ldots, p(n)\}$;
- sinon ($n \notin O$) O contient une solution optimale pour le sous-problème $\{1, \ldots, n-1\}$.

Alors la valeur de la solution O optimale, notée $opt(n)$ est la meilleure parmis les deux cas précédents : $opt(n) = max(v_n + opt(p(n)), opt(n-1))$.
Notons P_j $(1 \leq j \leq n)$ le problème du Weighted Interval Scheduling pour l'ensemble des intervalles de 1 à j – le problème que l'on veut résoudre est P_n et notons O_j une solution optimale de P_j et de valeur $opt(j)$ – la solution optimale O du paragraphe précédent est donc O_n de valeur $opt(n)$. On convient que $opt(0) = 0$.

Les valeurs $opt(j)$ sont obtenues avec le même raisonnement que pour n (voir plus haut) en considérant les deux cas $j \in O_j$ et $j \notin O_j$:

$$opt(j) = max(v_j + opt(p(j)), opt(j-1)) \quad (1)$$

Ainsi

$$j \in O_j \text{ ssi } v_j + opt(p(j)) \geq opt(j-1) \quad (2)$$

Algorithme récursif

Algorithm 3 Algorithm $Opt(j)$

```
1: if j = 0 then
2:     return 0
3: else
4:     return max(v_j + Opt(p(j)), Opt(j − 1))
5: end if
```

Théorème

$Opt(j)$ calcule $opt(j)$ correctement pour tout $j \in \{0, \ldots, n\}$.

On raisonne par récurrence (**forte**) sur j pour la propriété : "$Opt(j)$ calcule $opt(j)$ correctement".
Par définition $opt(0) = 0$, et $Opt(0)$ retourne bien 0.
Soit $j > 0$ et supposons que $Opt(i)$ calcule $opt(i)$ correctement **pour tout** $i < j$. On a

$$opt(j) \quad = \quad max(v_j + opt(p(j)), opt(j-1)) \qquad // \text{ d'après (1)}$$

Algorithme récursif

Algorithm 3 Algorithm $Opt(j)$

```
1: if j = 0 then
2:     return 0
3: else
4:     return max(v_j + Opt(p(j)), Opt(j − 1))
5: end if
```

Théorème

$Opt(j)$ calcule $opt(j)$ correctement pour tout $j \in \{0, \ldots, n\}$.

On raisonne par récurrence (**forte**) sur j pour la propriété : "$Opt(j)$ calcule $opt(j)$ correctement".
Par définition $opt(0) = 0$, et $Opt(0)$ retourne bien 0.
Soit $j > 0$ et supposons que $Opt(i)$ calcule $opt(i)$ correctement **pour tout** $i < j$. On a

$$\begin{array}{rcll} opt(j) & = & max(v_j + opt(p(j)), opt(j-1)) & // \text{ d'après (1)} \\ & = & max(v_j + Opt(p(j)), Opt(j-1)) & // \text{ par hyp. réc.} \end{array}$$

Algorithme récursif

Algorithm 3 Algorithm $Opt(j)$

```
1: if j = 0 then
2:     return 0
3: else
4:     return max(v_j + Opt(p(j)), Opt(j − 1))
5: end if
```

Théorème

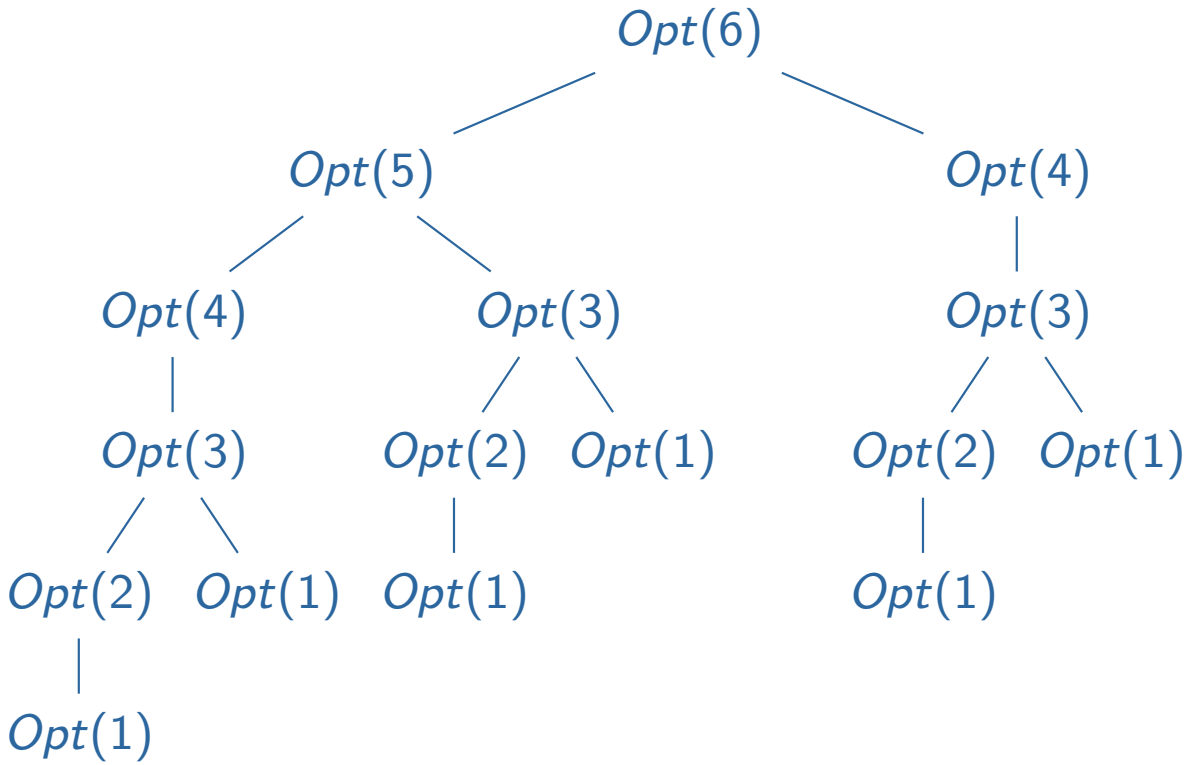
$Opt(j)$ calcule $opt(j)$ correctement pour tout $j \in \{0, \ldots, n\}$.

On raisonne par récurrence (**forte**) sur j pour la propriété : "$Opt(j)$ calcule $opt(j)$ correctement".
Par définition $opt(0) = 0$, et $Opt(0)$ retourne bien 0.
Soit $j > 0$ et supposons que $Opt(i)$ calcule $opt(i)$ correctement **pour tout** $i < j$. On a

$$\begin{array}{rcll} opt(j) & = & max(v_j + opt(p(j)), opt(j-1)) & // \text{ d'après (1)} \\ & = & max(v_j + Opt(p(j)), Opt(j-1)) & // \text{ par hyp. réc.} \\ & = & Opt(j) & // \text{ Algorithm 3} \end{array}$$

Inefficacité de la version récursive

Appels récursifs pour $Opt(6)$:



L'appel $Opt(j)$ engendre des appels récursifs indépendants à des sous-problèmes de taille $j-1$ et $j-2$, et on grandit exponentiellement

Voir le cas de la suite de Fibonacci $u_k = u_{k-1} + u_{k-2}$.

Mémoization

Calculer $Opt(n)$ demande de résoudre $n+1$ sous-problèmes : $Opt(0), Opt(1), \ldots Opt(n)$.
On mémorise ses résultats dans $M[0..n]$, où $M[i] = Opt(i)$.

Algorithm 4 $MemOpt(j)$

```
1:  if j = 0 then
2:      return 0
3:  else
4:      if M[j] is not empty then
5:          return M[j]
6:      else
7:          M[j] := max(v_j + MemOpt(p(j)), MemOpt(j − 1))
8:          return M[j]
9:      end if
10: end if
```

Théorème

Le temps d'exécution de $MemOpt(n)$ est en $O(n)$, **si on suppose que les intervalles sont triés par ordre croissant de date de fin**.

Analyse de la version *mémoïsée*

Théorème

Le temps d'exécution de $MemOpt(n)$ est en $O(n)$, si on suppose que les requêtes/intervalles sont triés par ordre croissant de date de fin.

Démonstration.

Le temps utilisé pour un appel à *MemOpt* est $O(1)$ hormis les appels récursifs qu'il engendre. Donc le temps d'exécution de *MemOpt* est borné par une constante fois le nombre d'appels récursifs issus de *MemOpt*.
On cherche une bonne mesure de "progrès" pour borner ce nombre d'appels : soit $ne(M)$ le nombre d'entrées de M qui ne sont pas vides.
Initialement, $ne(M) = 0$.
À chaque invocation de la récurrence, c'est à dire dans l'évaluation de $max(v_j + MemOpt(p(j), MemOpt(j-1))$ en ligne 7, on remplit une nouvelle entrée de M (en fait $M(j)$), donc $ne(M)$ vaut 1 de plus. Or $ne(M) \leq n+1$, il ne peut donc y avoir qu'au plus $O(n)$ appels à *MemOpt*. □

Calcul de la solution en plus de sa valeur

Simultanément au calcul du tableau M, on peut maintenir un tableau S tel que $S[j]$ contient un ensemble optimal de requêtes/intervalles dans $\{1, \ldots, j\}$.

- Un calcul naïf de S induit un facteur $O(n)$ à l'agorithme : si la mise à jour d'un calcul de M est en $O(1)$, celui de S est en $O(n)$ car il faut écrire tout un sous-ensemble de $\{1, \ldots, n\}$.
- On peut faire plus efficace en ne calculant pas *explicitement* S mais en retrouvant la solution optimale à partir des valeurs sauvées pour le calcul de M.

On a vu que $j \in O_j$ ssi $v_j + opt(p(j)) \geq opt(j-1)$, d'où :

Algorithm 5 *FindSolutionOpt*(j)

```
1: if j = 0 then
2:     return ∅
3: else
4:     if v_j + M[p(j)] ≥ M[j − 1] then        ▷ Il faut sélectionner l'intervalle j
5:         return {j} ∪ FindSolutionOpt(p(j))
6:     else                                    ▷ Il ne faut pas sélectionner l'intervalle j
7:         return FindSolutionOpt(j − 1)
8:     end if
9: end if
```

Complexité de *FindSolutionOpt*

Algorithm 6 *FindSolutionOpt*(j)

```
1: if j = 0 then
2:     return ∅
3: else
4:     if v_j + M[p(j)] ≥ M[j − 1] then        ▷ Il faut sélectionner l'intervalle j
5:         return {j} ∪ FindSolutionOpt(p(j))
6:     else                                    ▷ Il ne faut pas sélectionner l'intervalle j
7:         return FindSolutionOpt(j − 1)
8:     end if
9: end if
```

Puisque *FindSolutionOpt*(n) s'appelle récursivement sur des valeurs strictement plus petites, il y a au plus $O(n)$ appels récursifs. Sachant que le reste des instructions s'exécute en temps constant, on a :

Théorème

Étant donné le tableau M des valeurs optimales des sous-problèmes P_j $(0 \leq j \leq n)$, l'algorithme *FindSolutionOpt*(n) retourne une solution optimale en temps $O(n)$.

Sur l'exemple de Weighted Interval Scheduling

Jusqu'à présent nous avons trouvé un algorithme polynomial en plusieurs étapes

1. une version récursive
2. sa conversion par mémoïzation en un algo récursif efficace qui utilise un tableau global M des valeurs optimales des sous-problèmes

On va voir qu'en organisant les sous-problèmes par strates, on obtient un algorithme qui est plus efficace en mémoire.

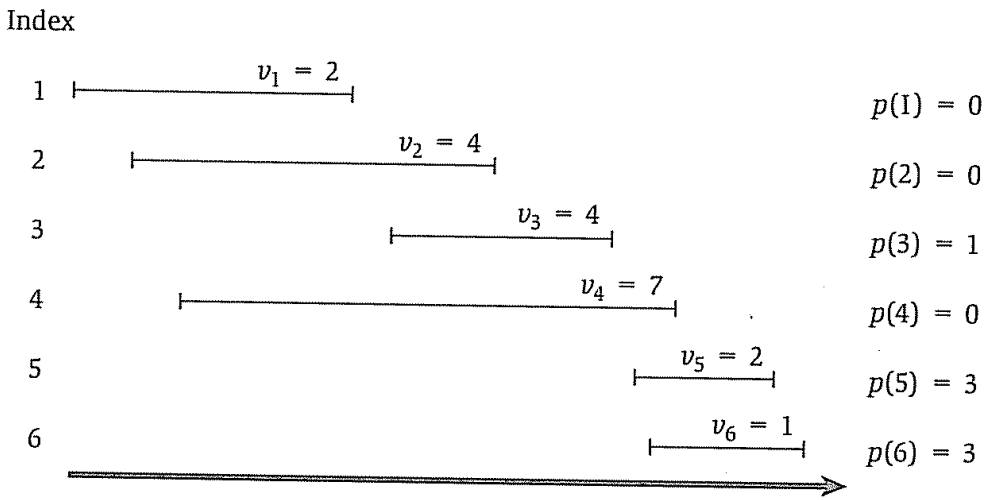
Itération de sous-problèmes au lieu de Mémoïzation Récursive

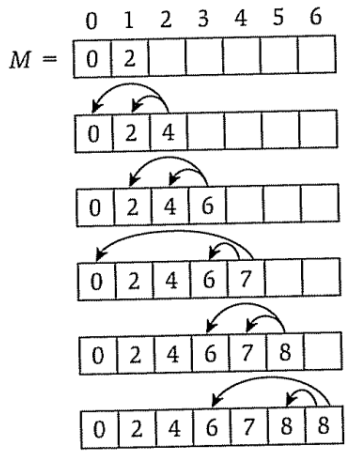
Algorithm 7 *IterativeOpt*$(v_1, \ldots, v_n)$

1: $M[0] = 0$
2: **for** $j = 1, 2 \ldots, n$ **do**
3: $\quad M[j] = max(v_j + M[p(j)], M[j-1])$
4: **end for**

À vous de jouer !

Justifiez que l'algorithme *IterativeOpt*(n) est correct et que sa complexité est en temps $O(n)$.





Complexité de notre algorithme pour Weighted Interval Scheduling

À vous de jouer !

Veuillez soigneusement répondre à cette question :

- Donnez-vous une chance d'y réfléchir.
- Puis, en consulter la littérature, par exemple en lisant `http://farazdagi.com/blog/2013/weighted-interval-scheduling/`.

Principes de la Programmation Dynamique (PrDy)

Nous allons voir des exemples avec des solutions du type itération de résolution de sous-problèmes, mais toutes les solutions peuvent être reformulées en mémoïzation récursive au prix d'une utilisation d'un espace mémoire important.

Pour développer des algorithmes du type PrDy, il faut avoir une collection de sous-problèmes basés sur le problème d'origine qui satisfait :

(a) On trouve une formulation de la solution du problème qui exhibe des sous-problèmes, en un nombre idéalement pas trop gros, c-à-d. polynomial) ;

(b) Le problème d'origine est un cas particulier de ces sous-problèmes ;

(c) On peut organiser ces sous-problèmes des plus petits aux plus grands de sorte que chaque sous-problème se résoud à partir de solutions de sous-problèmes plus petits ; les sous-problèmes minmaux se résolvent directement.

Exemple 12 (Weighted Scheduling Intervals)

$$opt(j) = max(v_j + opt(p(j)), opt(j-1)) \text{ et } j \in O_j \text{ ssi } v_j + opt(p(j)) \geq opt(j-1)$$

Retrouver les hypothèses (a), (b) et (c) pour ce cas.

Subset Sum Problem (SSP)

On suppose une machine qui doit exécuter des requêtes $1, ..., n$; chaque requête i s'exécute en un temps w_i donné. On veut exploiter la machine au maximum, sachant mais qu'elle ne fonctionne que sur une durée limitée W, et qu'elle ne peut exécuter qu'une requête à la fois.

Plus formellement, on considère le problème classique appelé Subset Sum Problem défini par :

(Subset Sum Problem)

Entrée : Un ensemble $E = \{1, \ldots, n\}$ d'items, avec un poids $w_i \geq 0$ pour chacun, et une borne W.
Sortie : Un sous-ensemble $S \subseteq \{1, \ldots, n\}$ des items tel que $\sum_{i \in S} w_i$ est maximale tout en respectant $\sum_{i \in S} w_i \leq W$.

Identification des sous-problèmes et des solutions optimales(1/2)

On se fixe une instance $(\{1,\ldots,n\}, W)$ de SSP. Soit $opt(n, W)$ une solution optimale pour $(\{1,\ldots,n\}, W)$.

- Si $n \notin O$, alors $opt(n, W) = opt(n-1, W)$, (c'est le cas si $W < w_n$)
- sinon $opt(n, W) = w_n + opt(n-1, W - w_n)$.

On voit apparaître le besoin de connaître la valeur de $opt(i, w)$ où $i \in \{1,\ldots,n\}$ et $w \in \{0, 1, \ldots W\}$, instance du problème SSP que l'on note $I_{i,w}$, pour lequel on cherche un sous-ensemble d'items dans $\{1,\ldots,i\}$ de poids maximum sans dépasser le poids autorisé w. Autrement dit,

$$opt(i, w) = max_{\{S \subseteq \{1,\ldots,i\} \mid \sum_{j \in S} w_j \leq w\}} \sum_{j \in S} w_j$$

Résoudre le problème pour l'instance $(\{1,\ldots,n\}, W)$ de SSP revient à résoudre le problème $I_{n,W}$ en calculant $opt(n, W)$.

Remarque 4.1

Contrairement au problème Weighted Interval Scheduling, on ne peut pas simplement considérer $opt(i)$ car le fait de prendre l'item i change le contexte pour les sous-problèmes à résoudre à cause de la quantité w.

Identification des sous-problèmes et des solutions optimales (2/2)

Une solution optimale $O \subseteq \{1, \ldots, i\}$ de $I_{i,w}$ est telle que

- Si $i \notin O$, alors $opt(i, w) = opt(i-1, w)$, (c'est le cas si $w < w_i$)
- sinon $opt(i, w) = w_i + opt(i-1, w-w_i)$.

Autrement écrit,

$$opt(i, w) = \begin{cases} \max\{opt(i-1, w), w_i + opt(i-1, w-w_i)\} & \text{si } w_i \leq w \\ opt(i-1, w) & \text{sinon} \end{cases} \quad (3)$$

Un algorithme déduit de la relation de récurrence

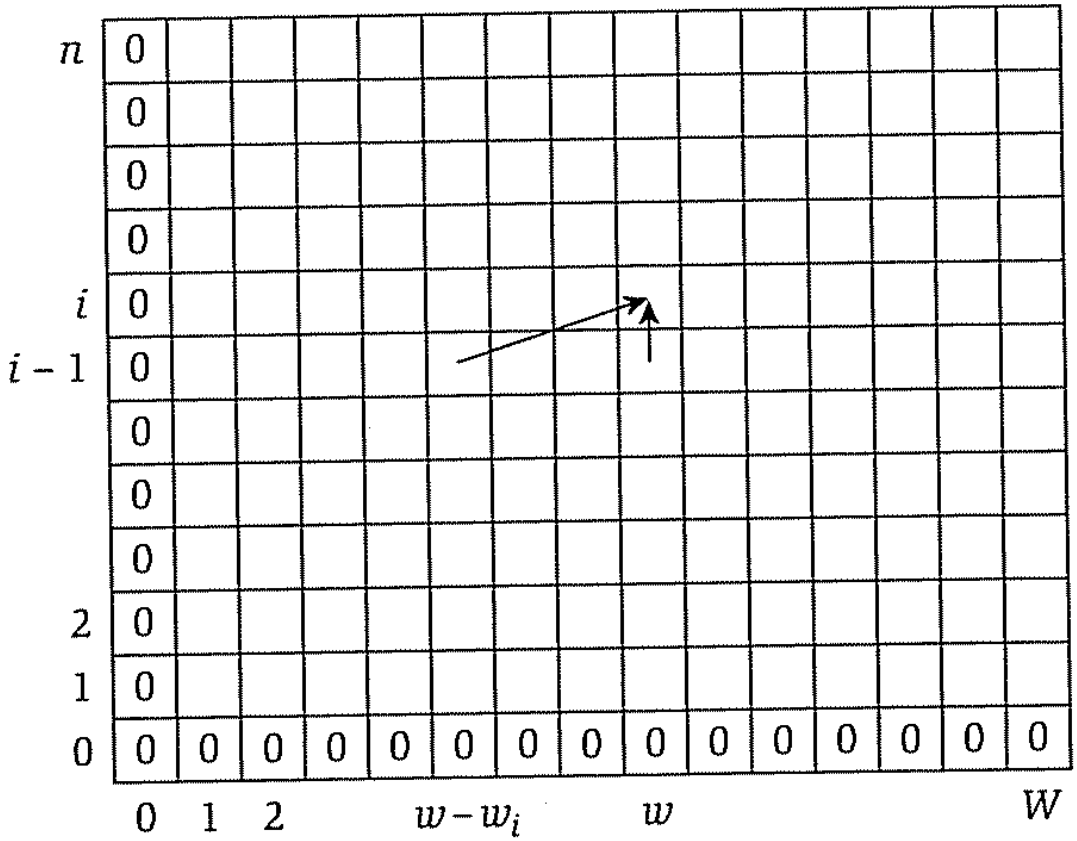
$$opt(i, w) = \begin{cases} max(opt(i-1, w), w_i + opt(i-1, w-w_i)) & \text{si } w_i \leq w \\ opt(i-1, w) & \text{sinon} \end{cases} \quad (4)$$

Algorithm 8 *SubsetSum*(n, W)

```
1: Array M[0..n, 0..W]
2: for w = 0, 1 ... W do
3:     M[0, w] = 0
4: end for
5: for i = 1, 2 ..., n do
6:     for w = 0, ..., W do
7:         Utiliser (4) pour définir M[i, w]
8:     end for
9: end for
```

Remplissage de la table $M[0..n, 0..W]$, à deux dimensions

L'entrée $M[i, w]$ est une fonction des entrées $M[i-1, w]$ et $M[i-1, w-w_i]$.



Exécution de *SubsetSum*

Algorithm 9 *SubsetSum*(n, W)

```
1: Array M[0..n, 0..W]
2: for w = 0, 1 ... W do
3:     M[0, w] = 0
4: end for
5: for i = 1, 2 ..., n do
6:     for w = 0, ..., W do
7:         Utiliser (4) pour définir M[i, w]
8:     end for
9: end for
```

À vous de jouer !

Exécutez l'algorithme *SubsetSum* pour $w_1 = w_2 = 2$, $w_3 = 3$ et $W = 6$.

Analyse de l'algorithme $SubsetSum(n, W)$

Algorithm 10 $SubsetSum(n, W)$

```
1:  Array M[0..n, 0..W]
2:  for w = 0, 1 ... W do
3:      M[0, w] ← 0
4:  end for
5:  for i = 1, 2 ..., n do
6:      for w = 0, ..., W do
7:          if w[i] ≤ w then
8:              M[i, w] ← M[i − 1, w]
9:          else
10:             M[i, w] ← max(M[i − 1, w], w_i + M[i − 1, w − w_i])
11:         end if
12:     end for
13: end for
```

Théorème 13

L'algorithme SubsetSum termine, est correct, et s'exécute en $O(nW)$.

Analyse de l'algorithme $SubsetSum(n, W)$

Algorithm 10 $SubsetSum(n, W)$

```
1:  Array M[0..n, 0..W]
2:  for w = 0, 1 ... W do
3:      M[0, w] ← 0
4:  end for
5:  for i = 1, 2 ..., n do
6:      for w = 0, ..., W do
7:          if w[i] ≤ w then
8:              M[i, w] ← M[i − 1, w]
9:          else
10:             M[i, w] ← max(M[i − 1, w], w_i + M[i − 1, w − w_i])
11:         end if
12:     end for
13: end for
```

Théorème 13

L'algorithme SubsetSum termine, est correct, et s'exécute en $O(nW)$.

Remarque 4.2

L'algorithme est pseudo-polynomial à cause de W qui dépend de la nature de l'entrée et pas uniquement du nombre n d'items.

Multiplication de matrices

On rappelle que si A est une matrice (de dimension) $p \times q$ et B est une matrice $q \times r$, le produit $C = AB$ de dimension $p \times r$ est calculé comme suit :
pour tous $1 \leq i \leq p$ et $1 \leq j \leq r$, on a

$$c_{ij} = \sum_{k=1}^{q} a_{ik} \times b_{kj}$$

Il y a donc $p \times r$ coefficients scalaires c_{ij} qui se calculent avec q multiplications scalaires ($a_{ik} \times b_{kj}$ pour $1 \leq k \leq q$), soit en tout $p \times q \times r$ multiplications scalaires.

(Multiplication de matrices)

Entrée : Une suite finie de matrices $A_1, \ldots, A_n$, où pour tout $i = 1, \ldots, n$, A_i est de dimension $p_{i-1} \times p_i$
Sortie : Un parenthésage du produit $A_1 A_2 \ldots A_n$ qui minimise le nombre de multiplications scalaires

Multiplication de matrices : bien parenthéser

Exemple 14

On considère A_1, A_2, A_3 où la matrice A_i est de dimension $p_{i-1} \times p_i$, avec $p_0 = 10$, $p_1 = 100$, $p_2 = 5$ et $p_3 = 50$.

- si on parenthèse $((A_1 A_2)A_3)$ on obtient un coût de $10 \times 100 \times 5$ produits pour calculer $A_1 A_2$, puis $10 \times 5 \times 50$ produits pour multilier ce résultat avec A_3. Soit 7500 multiplications scalaires.
- si on parenthèse $(A_1(A_2 A_3))$ $100 \times 5 \times 50 + 10 \times 100 \times 50$, soit 75000 multiplications scalaires.

Comptabilisons le nombre $P(n)$ façons possibles de parenthéser une séquence de n matrices. Clairement $P(1) = 1$. Pour $n > 1$, un produit matriciel entièrement parenthésé est le produit de deux sous-produits matriciels entièrement parenthésés, et la démarcation entre les deux sous-produits peut intervenir entre les $k$ème et $(k+1)$ème matrices, pour tout $k = 1, 2, ..., n-1$. On obtient donc la récurrence

$$P(n) = \begin{cases} 1 & \text{si } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \geq 2. \end{cases}$$

Les $P(n)$ sont les nombres de Catalan notés C_n, égaux à $\frac{(2n)!}{(n+1)!n!} \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$, qui est en $\Omega(2^n)$, de sorte qu'une recherche naïve n'est pas viable.

Structure d'un parenthésage optimal

Étape 1 de la PrDy. On cherche la sous-structure optimale et on s'en sert.
On notera $A_{i\ldots j}$ le produit $A_i A_{i+1} \ldots A_j$.

À vous de jouer !

Établir que si on a un parenthésage optimal pour $A_{i\ldots j}$, alors il fractionne ce produit entre A_k et A_{k+1} pour un $i \leq k < j$. Alors les parenthésages pour $A_{i\ldots k}$ $A_{k+1\ldots j}$ sont forcément optimaux.

Il faut donc chercher où fractionner le produit $A_{i\ldots j}$.

Étape 2 de la PrDy. On définit récursivement le coût d'une solution optimale en fonction du coût des solutions optimales de sous-problèmes.

On utilise notre sous-structure optimale pour montrer que nous pouvons construire une solution optimale du problème à partir de solutions optimales de sous-problèmes.
On prend comme sous-problèmes les produits $A_{i\ldots j}$ pour $1 \leq i \leq j \leq n$, et soit $m[i,j]$ le coût optimal pour calculer ce produit.

On a

$$\begin{cases} m[i,i] = 0, \text{ pour tout } i = 1, \ldots, n \\ m[i,j] = min_{i \leq k < j} m[i,k] + m[k+1,j] + p_{i-1} \times p_k \times p_j, \text{ si } i < j \end{cases}$$

Pour le problème principal est $A_{1\ldots n}$, et le résultat cherché est $m[1,n]$.

L'algorithme

On note $s[i,j]$ la (plus petite) valeur k qui minimise $m[i,j]$, et on suppose que l'entrée est une séquence $p = \langle p_0, p_1, \ldots, p_n \rangle$, où $longueur[p] = n+1$.

On organise les calculs en faisant d'abord grandir la longueur du produit $A_{i\ldots j}$, c'est à dire la valeur $l = j - i + 1$, puis la valeur i.

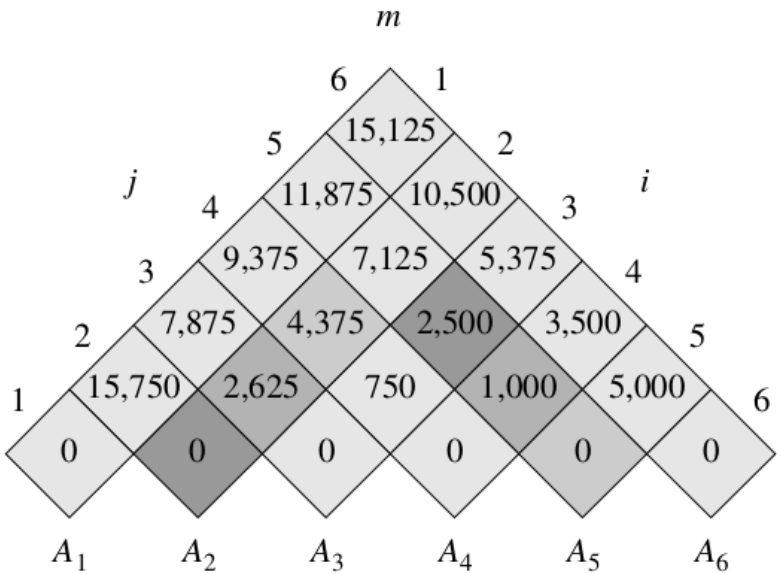
```
ORDRE-CHAÎNE-MATRICES(p)
 1  n ← longueur[p] − 1
 2  pour i ← 1 à n
 3      faire m[i, i] ← 0
 4  pour l ← 2 à n                ▷ l est la longueur de la chaîne.
 5      faire pour i ← 1 à n − l + 1
 6              faire j ← i + l − 1
 7                   m[i, j] ← ∞
 8                   pour k ← i à j − 1
 9                       faire q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                            si q < m[i, j]
11                              alors m[i, j] ← q
12                                    s[i, j] ← k
13  retourner m et s
```

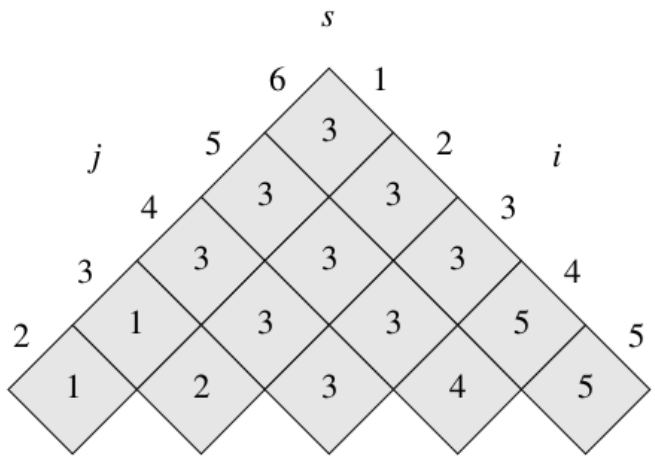
On obtient un algorithme en $O(n^3)$.

Un exemple du produit de matrices

matrice	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2\,500 + 35 \cdot 15 \cdot 20 = 13\,000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2\,625 + 1\,000 + 35 \cdot 5 \cdot 20 = 7\,125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4\,375 + 0 + 35 \cdot 10 \cdot 20 = 11\,375 \end{cases}$$
$$= 7125\,.$$





Calcul du parenthésage

On sait que la dernière multiplication matricielle sera le produit de $A_{1\ldots s[1,n]}$ avec $A_{s[1,n]+1\ldots n}$.

AFFICHAGE-PARENTHÉSAGE-OPTIMAL(s, i, j)

```
1  si i = j
2      alors afficher« A »_i
3      sinon afficher« ( «
4            AFFICHAGE-PARENTHÉSAGE-OPTIMAL(s, i, s[i, j])
5            AFFICHAGE-PARENTHÉSAGE-OPTIMAL(s, s[i, j] + 1, j)
6            print » ) »
```

Le problème du mot pour les grammaires algébriques

(Problème du mot)

Entrée : Une grammaire algébrique $G = (T, N, S, R)$ en forme normale de Chomsky et $s \in T^*$
Sortie : $s \in L(G)$

Ce n'est pas un problème d'optimisation, mais in problème de décision.

Définition 15

On rappelle que $G = (T, N, S, R)$ est en *forme normale de Chomsky*, si toutes les règles sont de la forme (a) $X \to YZ$ avec $Y, Z \in N$, ou (b) $X \to a$ avec $a \in T$ (On peut effectivement mettre toute grammaire algébrique sous cette forme), et éventuellement une règle $S \to \epsilon$.

Théorème 16

*Pour toute grammaire algébrique, il existe une grammaire en forme normale de Chomsky qui lui soit équivalente (*i.e. *, qui engendre le même langage).*

Le problème du mot pour les grammaires algébriques

Remarque 4.3

Si $G = (T, N, S, R)$ en forme normale de Chomsky, alors $s \in T^*$ est engendrable avec la règle $X \to YZ$ ssi on peut le séparer en $s = tt'$ avec t (resp. t') engendrable à partir de Y (resp. Z).

On va donc s'intéresser au sous-problème de savoir si un facteur de s, de la forme $s[i..j]$, est engendrable à partir d'un non terminal X, ce que l'on note $E[i, j, X]$ qui vaudra "vrai" ou "faux".

$$E[i, i, X] \text{ vaut vrai s'il existe une règle } X \to a \text{ et } s[i] = a,$$
$$\text{sinon } E[i, j, X] = \bigvee_{X \to YZ \in R} \left(\bigvee_{k=i}^{j-1} E[i, k, Y] \wedge E[k+1, j, Z] \right)$$

Alors

$$s \in L(G) \text{ si, et seulement si, } E[1, |s|, S] \text{ vaut vrai}$$

Le problème du mot pour les grammaires algébriques

Soit $G = (T, N, S, R)$ une grammaire algébrique en Forme Normale de Chomsky et $s \in T^*$.

On calcule les $E[i, j, X]$, où $1 \leq i \leq j \leq |s$ et $X \in N$ par ordre croissant de $j - i$ selon l'équation :

$$\begin{cases} E[i,i,X] \leftarrow \text{vrai, s'il existe une règle } X \to a \text{ et } s[i] = a \\ E[i,j,X] \leftarrow \bigvee_{X \to YZ \in R} \left(\bigvee_{k=i}^{j-1} E[i,k,Y] \wedge E[k+1,j,Z] \right) \end{cases}$$

Après avoir fini de calculer, on retourne

$$s \in L(G) \text{ si, et seulement si, } E[1, |s|, S] \text{ vaut vrai}$$

La complexité du calcul de $E[1, |s|, S]$ en temps est $O(|s|^3 |G|^2)$:

- on compte le nombre de $E[i, j, X]$: on a $O(|s|^2)$ couples $i < j$ à multiplier par le nombre de non-terminaux de la grammaire $|N| \in O(|G|)$.
- le calcul de chacun des $E[i, j, X]$ est en $O(|s|.|G|)$: on doit chercher pour toutes les valeurs intermédaires $i \leq k < j$, donc en $O(|s|)$, parmi toutes les règles où X apparaît dans le membre gauche, donc en $O(|G|)$.

Le problème du mot pour les grammaires algébriques

À vous de jouer !

- Alternativement, on peut definir $E[i,j] \subseteq N$ comme l'ensemble des non-terminaux de G à partir desquels on peut engendrer $s[i..j]$. Alors $s \in L(G)$ ssi $S \in E[1,|s|]$. Donner les équations de récurrence pour calculer les $E[i,j]$.
- Écrire l'algorithme de programmation dynamique qui en découle : c'est l'algorithme de Cocke-Younger-Kasami.

À vous de jouer !

Écrire un algorithme de mise sous forme normale de Chomsky. Quelle est sa complexité ? Regarder `http://compilation.irisa.fr/`.

Le problème de la distance d'édition de deux séquences

Objectif : On cherche à "aligner" les deux séquences "SUNNY" et "SNOW".

S	–	N	O	W	Y
S	U	N	N	–	Y

Cost: 3

–	S	N	O	W	–	Y
S	U	N	–	–	N	Y

Cost: 5

Le caractère "-" indique un vide.

Définition 17 (Coût d'un alignement)

C'est le nombre de colonnes dans lesquelles les lettres diffèrent.

Définition 18 (Distance d'édition entre deux séquences)

C'est le coût de leur meilleur alignement.

Distance d'édition

On parle de distance d'édition parce qu'on recherche un nombre minimum d'opérations appelées *éditions* de type

- insertion
- effacement
- substitutions

Exemple 19

S	–	N	O	W	Y		–	S	N	O	W
S	U	N	N	–	Y		S	U	N	–	–
			Cost: 3							Cost: 5	

Pour l'alignement de gauche,
on réalise trois éditions "insère U", "substitue O par N", "efface W".

À vous de jouer !

Donnez une suite d'éditions pour le cas de droite.

Distance d'édition/Alignement de séquences

(Distance d'édition)

Entrée : Deux séquences $x[1..m]$ et $y[1..n]$
Sortie : La distance d'édition entre x et y.

Pour une solution par programmation dynamique, on doit répondre à :

1. Quels sont les sous-problèmes ?
2. Dans quel ordre organiser leur calcul de sorte que l'on puisse calculer les sous-problèmes "grand" à partir de sous-problèmes "plus petits" ?

Quels sont les sous-problèmes ?

Les sous-problèmes portent sur le meilleur alignement des séquences $x[1..i]$ et $y[1..j]$, dont on note $E[i,j]$ le coût. Initialement, on veut calculer $E[m,n]$.

Exemple 20

Le sous-problème $E[7,5]$:

E	X	P	O	N	E	N	T	I	A	L
P	O	L	Y	N	O	M	I	A	L	

Le meilleur alignement de $x[1..i]$ et $y[1..j]$ est tel que sa dernière colonne est une des trois formes suivantes :

$$\begin{matrix} x[i] \\ - \end{matrix} \quad \textbf{or} \quad \begin{matrix} - \\ y[j] \end{matrix} \quad \textbf{or} \quad \begin{matrix} x[i] \\ y[j] \end{matrix}$$

Définissons $sontDiff(i,j) = 0$ si $x[i] = y[j]$, et $sontDiff(i,j) = 1$ sinon.
Ce qui permet d'écrire pour $1 \leq i \leq m$, $1 \leq j \leq n$

$$E[i,j] = min\{1 + E[i-1,j], 1 + E[i,j-1], sontDiff(i,j) + E[i-1,j-1])\}$$

et $E[i,0] = i$ et $E[0,j] = j$.

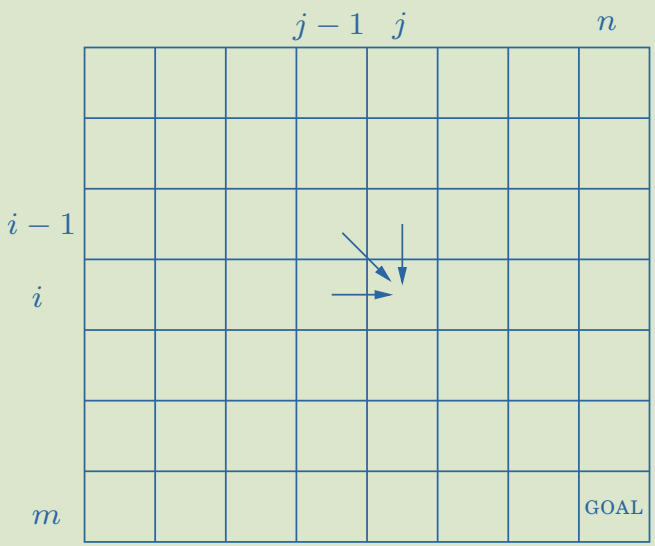
Dans quel ordre organiser les calculs ?

Les entrées $E[i-1,j], E[i,j-1], E[i-1,j-1]$ sont nécessaires pour remplir l'entrée $E[i,j]$.

Exemple 21 (avec $x =$ EXPONENTIAL et $y =$POLYNOMIAL)

(a) La table des sous-problèmes.
(b) Les valeurs finales de la table trouvées par la programmation dynamique.

(a)



(b)

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

L'algorithme de calcul de la distance d'édition

```
for i = 0, 1, 2, ..., m:
    E(i, 0) = i
for j = 1, 2, ..., n:
    E(0, j) = j
for i = 1, 2, ..., m:
    for j = 1, 2, ..., n:
        E(i, j) = min{E(i − 1, j) + 1, E(i, j − 1) + 1, E(i − 1, j − 1) + diff(i, j)}
return E(m, n)
```

On a une solution en $O(mn)$.

Pour EXPONENTIAL et POLYNOMIAL on trouve $E[11, 10] = 6$:

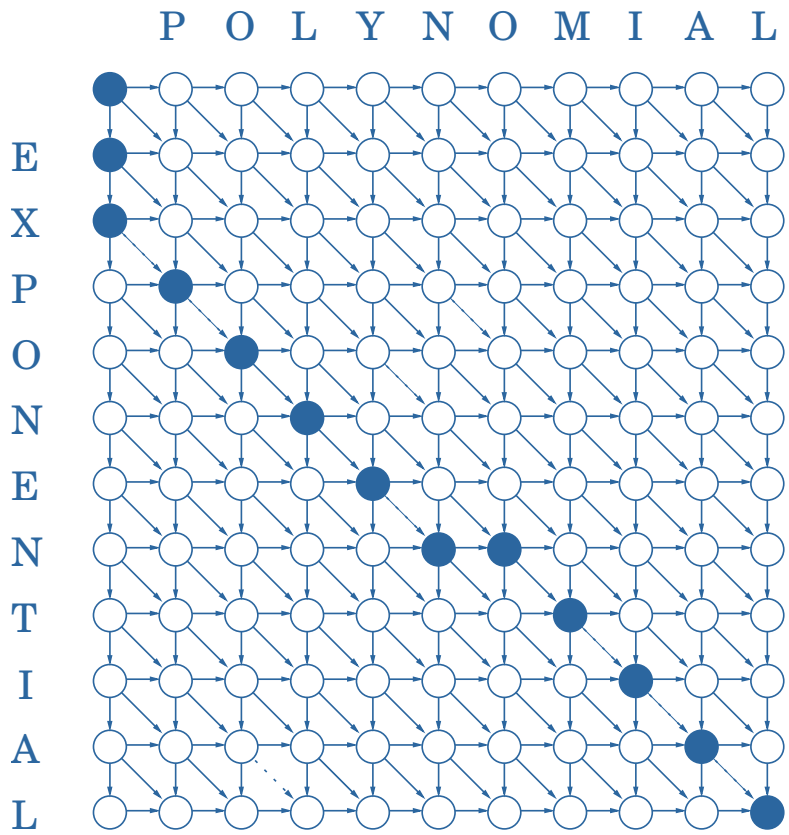
E	X	P	O	N	E	N	−	T	I	A	L
−	−	P	O	L	Y	N	O	M	I	A	L

À vous de jouer !

Quelle est la complexité en espace de cet algorithme de PrDy ?

Un problème de graphe

On convient que les flèches pointillées coûtent 0 (cas $x[i] = y[j]$), et que les flèches pleines vallent 1.

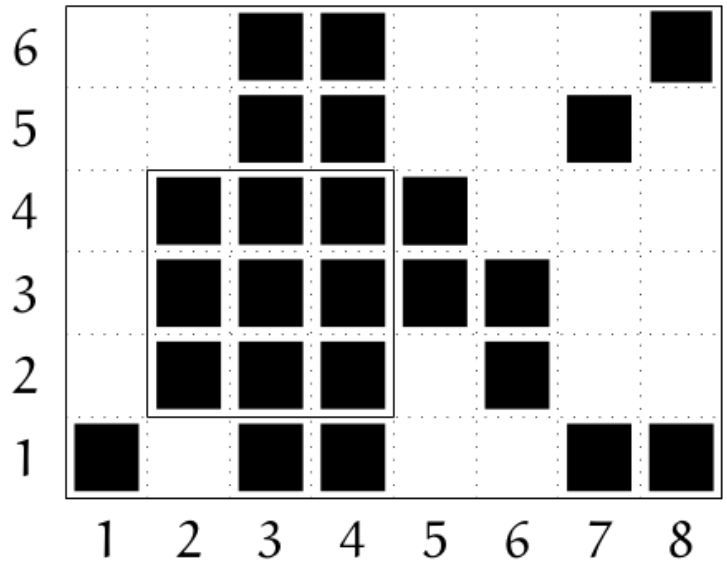


À vous de jouer !

Reprendre le tout en paramétrant les coûts des trois opérations d'édition.

Le plus grand carré noir (à concevoir ensemble)

Soit une image rectangulaire de largeur n et de hauteur m composée de pixels noirs (1) et blancs (0) représentée par la matrice $IMG[1..m, 1..n]$. On cherche le côté c de la plus grande sous-image carrée de IMG complètement noire. Par exemple, dans l'image ci-dessous, où $m = 6$ et $n = 8$, le plus grand carré noir est unique. Il a pour côté 3 et s'étend sur les lignes 2 à 4 et les colonnes 2 à 4 (avec la convention de numérotation des lignes de bas en haut et des colonnes de gauche à droite qui est utilisée.



Limitations de la Programmation Dynamique

On considère le problème suivant :

(Longest Simple Path)

Entrée : Un graphe pondéré $G = (V, E)$ et deux sommets s et t.
Sortie : Le chemin élémentaire de s à t de poids maximum.

On définit $LP[i, j]$ comme la valeur du plus long chemin élémentaire de i à j. Le dernier arc visité est de la forme (x, j), et de coût $c(x, j)$, ce qui suggère naïvement

$$LP[i, j] = max_{(x,j)\in E} LP[i, x] + c(x, j)$$

Mais comment garantit-on que la valeur $LP[i, x]$ ne correspond pas à une sous-solution où j est dans le plus long chemin élémentaire de i à x ?
Aussi, quel ordre d'évaluation appliquer entre les différents sous-problèmes ?

Dans ce cas, il faut non seulement garantir le principe d'optimalité, mais la nature même des sous-solutions optimales intervient dans ce calcul, et pas seulement leur coût.

Quand la programmation dynamique est-elle efficace/inefficace ?

La complexité d'un algorithme de PrDy repose sur trois choses :

1. pour le temps,
   1. le nombre de sous-problèmes à évaluer
   2. le temps nécessaire pour évaluer chaque sous-problème, et
2. pour l'espace, le nombre de sous-problèmes dont il faut garder trace à chaque étape.

En général c'est 1. qui est critique.

Dans le cas du Longest Simple Path, on pourrait choisir les sous-problèmes qui consistent en un chemin P entre deux sommets : $LP[i, j, P]$ dénote le plus long chemin élémentaire entre i et j, où P est la séquence exacte des sommets intermédiaires entre i et j (exclu) sur ce chemin.

$$LP[i, j, P.x_0] = max_{(x,j)\in E, x, j\notin P} LP[i, x, P] + c(x, j)$$

où $x_0 = argmax_{(x,j)\in E, x, j\notin P} LP[i, x, P] + c(x, j)$.
Cette formulation est correcte, mais malheureusement les chemins P sont en grand nombre, et on n'obtient pas un nombre polynomial de sous-problèmes.

À vous de jouer !

Combien y a-t-il de sous-problèmes ?

Approches Gloutonnes

Le problème des activités

- un ensemble d'activités $a_1, \ldots, a_n$;
- a_i commence à s_i et termine à f_i ($s_i < f_i$).

Le problème est d'**affecter une salle à chaque activité, en respectant la contrainte que "à un instant donné, il peut y avoir au plus une seule activité par salle" et en minimisant le nombre de salles utilisées**.
Rmq : Pas de temps de battement entre deux activités dans une même salle.
On considère une version plus abstraite du problème :

Définition 22 (Activités compatibles)

Les activités a et b sont compatibles si a commence après b ou b commence après a :

$$s_a \geq f_b \text{ ou } s_b \geq f_a \qquad (a \text{ n'est pas compatible avec elle-même}).$$

(Les activités)

Entrée : Un ensemble $E = \{a_1, \ldots, a_n\}$ de n activités, et pour chaque activité $a_i \in E$, ses valeurs $s_i \leq f_i$ de début et de fin.
Sortie : Un ensemble d'activités mutuellement compatibles de taille maximale.

Exemple du problème des activités

Exemple 23

a_k	1	2	3	4	5	6	7	8	9	10	11
s_k	1	3	0	5	3	5	6	8	8	2	12
f_k	4	5	6	7	8	9	10	11	12	13	14

Le sous-ensemble $\{3, 9, 11\}$ est solution, mais il n'est pas optimal car

on a aussi $\{1, 4, 8, 11\}$ ou encore $\{2, 4, 9, 11\}$.

Remarque : les activités sont numérotées par ordre croissant d'heure de fin.

Exemple du problème des activités

Exemple 23

a_k	1	2	3	4	5	6	7	8	9	10	11
s_k	1	3	0	5	3	5	6	8	8	2	12
f_k	4	5	6	7	8	9	10	11	12	13	14

Le sous-ensemble $\{3, 9, 11\}$ est solution, mais il n'est pas optimal car

on a aussi $\{1, 4, 8, 11\}$ ou encore $\{2, 4, 9, 11\}$.

Remarque : les activités sont numérotées par ordre croissant d'heure de fin.

Un critère intuitif pour sélectionner les activités

Si on vient de choisir une activité a, il faut choisit une autre activité parmi toutes les activités non encore sélectionnées et compatibles avec a. Parmi ces activités, on se propose de prendre **celle qui se termine le plus tôt**, car ainsi on se laisse plus de chance de pouvoir en sélectionner encore beaucoup d'autres.

On conçoit un algorithme sur la base de cette intuition.

Un algorithme glouton

- les activités sont $a_1, a_2, ..., a_n$, avec $s[0..n+1]$ les heures de début des activités et $f[1..n]$ les heures de fin des activités, **triés en ordre croissant** (comme dans notre exemple).
- on ajoute 2 activités "fictives" a_0 avec $s_0 = f_0 = 0$ et a_{n+1} avec $s_{n+1} = f_{n+1} = \infty$. On remarque que $f[0..n+1]$ est aussi trié par ordre croissant.

Algorithm 11 CHOIX-D'ACTIVITÉS-GLOUTON-RECURSIF(s, f, k, n)

Require: a_k est la dernière activité sélectionnée
Sortie : un ensemble maximal d'activités compatibles parmi $a_{k+1}, \ldots, a_n$

```
1:  if k ≤ n then
2:      m ← k + 1                ▷ on cherche la première activité après a_k qui est compatible
3:      while s[m] < f[k] do                          ▷ a_m n'est pas compatible avec a_k
4:          m ← m + 1
5:      end while                                     ▷ m ≤ n + 1 car s[n+1] = +∞ ≥ f[k]
6:                                  ▷ a_m est la première activité après a_k qui est compatible
7:      return {a_m} ∪ CHOIX-D'ACTIVITÉS-RÉCURSIF(s, f, m, n)
8:  else
9:      return ∅
10: end if
```

Appel principal : **Return** $\{a_0\} \cup$ CHOIX-D'ACTIVITÉS-RÉCURSIF$(s, f, 0, n)$

Principe des algorithmes gloutons

Construction incrémentale d'une solution pour une entrée E :

- un critère de sélection d'un morceau de la solution en **temps polynomial**
  ce choix est dit glouton ca
- \+
- on complète la solution en résolvant un sous-problème plus petit (il y a un nombre polyn

La méthode est intrinsèquement *récursive terminale*, c-à-d. qu'il n'y a aucune instruction après les appels récursifs.

De fait, la méthode peut aussi se décrire de façon itérative !

Écriture itérative de la version récursive pour le problème des activités

On rappelle que :

Algorithm 12 CHOIX-D'ACTIVITÉS-GLOUTON-RECURSIF(s, f, k, n)

Require: a_k est la dernière activité sélectionnée
Sortie : un ensemble maximal d'activités compatibles parmi $a_{k+1}, \ldots, a_n$

```
1:  if k ≤ n then
2:      m ← k + 1          ▷ on cherche la première activité après a_k qui est compatible
3:      while s[m] < f[k] do          ▷ a_m n'est pas compatible avec a_k
4:          m ← m + 1
5:      end while          ▷ m ≤ n + 1 car s[n + 1] = +∞ ≥ f[k]
6:                         ▷ a_m est la première activité après a_k qui est compatible
7:      return {a_m} ∪ CHOIX-D'ACTIVITÉS-RÉCURSIF(s, f, m, n)
8:  else
9:      return ∅
10: end if
```

Appel principal : **Return** $\{a_0\} \cup$ CHOIX-D'ACTIVITÉS-RÉCURSIF$(s, f, 0, n)$

Clairement l'algorithme CHOIX-D'ACTIVITÉS-RÉCURSIF(s, f, k, n) est récursif *terminal*.
On en donne une version itérative...

Un algorithme itératif pour le problème des activités

Algorithm 13 CHOIX-D'ACTIVITÉS-ITERATIF(s, f)

Require: A est un ensemble d'activités.
Sortie : un ensemble maximal d'activités compatibles parmi $a_{k+1}, \ldots, a_n$

```
1:  n ← longueur[s]                          ▷ n est le nombre d'activités
2:  A ← {a_0}                                ▷ on sélectionne a_0 (celle qui se termine le plus tôt)
3:  k ← 0
4:  while k ≤ n do                           ▷ a_k est la dernière activité sélectionnée
5:      m ← k + 1                            ▷ on cherche la première activité après a_k qui est compatible
6:      while s[m] < f[k] do                 ▷ a_m n'est pas compatible avec a_k
7:          m ← m + 1
8:      end while                            ▷ m ≤ n + 1 car s[n + 1] = +∞ ≥ f[k]
9:      A ← A ∪ {a_m}                        ▷ et c'est celle qui se termine le plus tôt
10:     k ← m
11: end while                                ▷ k = n + 1
12: return A \ {a_{n+1}}
```

À vous de jouer !

CHOIX-D'ACTIVITÉS-RÉCURSIF$(s, f, 0, n)$ et CHOIX-D'ACTIVITÉS-ITERATIF(s, f) font le

Correction d'un algorithme glouton

Il consiste à justifier le choix glouton.

P. ex, pour le problème des activités c'est le choix de a_m.

Puisque l'approche est récursive, il suffit de montrer que qu'il existe une solution optimale qui repose sur notre critère de sélection .

Pour cela on considère une solution optimale que l'on "déforme" en la forçant à contenir l'élément de notre critère de sélection et on montre que l'objet obtenu est

(Sol) une solution au problème, et

(Opt) qu'elle est optimale.

Illustration de ce principe sur le problème des activités...

Correction de Choix-D'activités-Récursif$(s, f, 0, n)$

Théorème 5.1

- Choix-D'activités-Récursif$(s, f, 0, n)$ termine ; ✓
- La sélection de a_m (la boucle en lignes 3-5) prend un temps polynomial ; ✓
- L'algorithme Choix-D'activités-Récursif$(s, f, 0, n)$ est optimal.

Algorithm 14 Choix-D'activités-Glouton-Recursif(s, f, k, n)

Require: a_k est la dernière activité sélectionnée
Sortie : un ensemble maximal d'activités compatibles parmi $a_{k+1}, \ldots, a_n$

1: **if** $k \leq n$ **then**
2: $m \leftarrow k + 1$ ▷ on cherche la première activité après a_k qui est compatible
3: **while** $s[m] < f[k]$ **do** ▷ a_m n'est pas compatible avec a_k
4: $m \leftarrow m + 1$
5: **end while** ▷ $m \leq n + 1$ car $s[n + 1] = +\infty \geq f[k]$
6: ▷ a_m est la première activité après a_k qui est compatible
7: **return** $\{a_m\} \cup$ Choix-D'activités-Récursif(s, f, m, n)
8: **else**
9: **return** $\emptyset$
10: **end if**

Choix-D'activités-Glouton-Recursif(s, f, k, n) est optimal

Soit $O = \{a_\ell, a_{\ell+1}, \ldots\}$ une solution optimale pour le problème du choix d'activités "à partir de a_k".
On déforme O en un autre ensemble d'activités O' qui contient le même choix glouton que celui fait dans notre algorithme, à savoir a_m (l'activité qui se termine le plus tôt, compatible avec a_k).

On pose naturellement $O' = (O \setminus \{a_\ell\}) \cup \{a_m\}$, et on montre que O' est une solution.
Comme de plus $Card(O') = Card(O)$, on aura établi que O' est optimale.

O' est une solution dès lors que a_m est compatible avec toutes les autres activités de O', c'est à dire avec les $a_{\ell'} \in O$ où $\ell' > \ell$.
Or,

- puisque O est une solution, on sait que $f_\ell \leq s_{\ell'}$ pour tout $a_{\ell'} \in O$ où $\ell < \ell'$ (les activités dans O sont compatibles entre elles et sont triées par ordre croissant de date de fin), et
- d'après le choix de a_m, on a $m \leq \ell$, de sorte que $f_m \leq f_\ell$.

Ainsi $f_m \leq s_{\ell'}$ pour tout $a_{\ell'} \in O$ et $\ell' > \ell \geq m$, de sorte que

$$O' \text{ est une solution (optimale).}$$

La satisfaisabilité des formules de Horn

Définition 24

Une clause de Horn est une clause contenant au plus un litéral positif.

L'énoncé

"Si un animal vole et pond des oeufs alors c'est un oiseau
et
si un animal vole et qu'il est grand alors c'est un albatros"

se formalise en :

$$(vole \wedge pond \rightarrow oiseau) \wedge ((vole \wedge grand) \rightarrow albatros)$$

ou encore en clause de Horn :

$$\{\neg vole \vee \neg pond \vee oiseau, \neg vole \vee \neg grand \vee albatros\}$$

.

Définition 25

Les clauses sans aucun littéral positif sont dites négative pures, les autres sont des implications.

Exemple d'une entrée de HORN-SAT (satisfaisabilité de clauses de Horn)

Exemple 26

On se donne des énoncés de base :

- x pour "le meurtre a eu lieu dans la cuisine"
- y pour "le majordome est innocent"
- z pour " le colonnel dormait à 20h00"
- w pour "le meurtre a eu lieu à 20h00"
- u pour " le colonnel est innocent", v pour "

On pourra écrire des clauses telles que :

- des faits : $\top \to x$ pour "c'est la cas que le meutre a eu lieu dans la cuisine" ;
- des inférences, telle que puisque " le colonnel dormait à 20h00" et que "le meurtre a eu lieu à 20h00", alors " le colonnel est innocent" : $z \wedge w \to u$;
- des affirmations, telle que "le coupable est parmi eux" : $\neg u \vee \neg v \vee \neg y$.

Ceci nous donne l'ensemble de clauses de Horn suivant :

$$\mathcal{C} = \{\top \to x, x \wedge z \to w, \neg w \vee \neg x \vee \neg y, w \wedge y \wedge z \to x, x \to y, x \wedge y \to w, \neg z\}$$

Pour résoudre l'énigme, on cherche un modèle (une valuation) de cet ensemble $\mathcal{C}$.

Un algorithme glouton pour les clauses de Horn

(Horn-SAT)

Entrée : Un ensemble fini $\mathcal{C}$ de clauses de Horn.
Sortie : Si elle existe, une valuation ν telle que $\nu \models \mathcal{C}$.

Algorithm 15 Horn-SAT-Glouton($\mathcal{C}$)

Soit *Prop* l'ensemble des propositions qui apparaissent dans les clauses de $\mathcal{C}$.
Soit $\mathcal{I} \subseteq \mathcal{C}$ l'ensemble des clauses *implications*, comme $x \wedge z \rightarrow w$.
Soit $\mathcal{N} \subseteq \mathcal{C}$ l'ensemble des clauses *négatives pures*, comme $\neg w \vee \neg x \vee \neg y$.

1: **for all** $p \in Prop$ **do**
2: $\quad \nu(p) \leftarrow$ `false` ▷ Initialiser toutes les variables à faux
3: **end for**
4: **while** il existe une implication $C \in \mathcal{I}$ avec $\nu(C) =$ `false` **do**
5: $\quad$ Choisir $C \in \mathcal{I}$ tq $\nu(C) =$ `false` de la forme $C = \ldots \rightarrow p$
6: $\quad \nu(p) \leftarrow$ `true` ▷ modifier $\nu(p)$
7: **end while**
8: **if** pour chaque $C \in \mathcal{N}$, $\nu(C) =$ `true` **then**
9: $\quad$ **return** la valuation ν
10: **else**
11: $\quad$ **return** "la formule n'est pas satisfaisable"
12: **end if**

Correction de l'algorithme Horn-SAT-Glouton

Algorithm 16 Horn-SAT-Glouton($\mathcal{C}$)

Soit *Prop* l'ensemble des propositions qui apparaissent dans les clauses de $\mathcal{C}$.
Soit $\mathcal{I} \subseteq \mathcal{C}$ l'ensemble des clauses *implications*, comme $x \wedge z \rightarrow w$.
Soit $\mathcal{N} \subseteq \mathcal{C}$ l'ensemble des clauses *négatives pures*, comme $\neg w \vee \neg x \vee \neg y$.

1: **for all** $p \in Prop$ **do**
2: $\quad \nu(p) \leftarrow$ `false` ▷ Initialiser toutes les variables à faux
3: **end for**
4: **while** il existe une implication $C \in \mathcal{I}$ avec $\nu(C) =$ `false` **do**
5: $\quad$ Choisir $C \in \mathcal{I}$ tq $\nu(C) =$ `false` de la forme $C = \ldots \rightarrow p$
6: $\quad \nu(p) \leftarrow$ `true` ▷ modifier $\nu(p)$
7: **end while**
8: **if** pour chaque $C \in \mathcal{N}$, $\nu(C) =$ `true` **then**
9: $\quad$ **return** la valuation ν
10: **else**
11: $\quad$ **return** "la formule n'est pas satisfaisable"
12: **end if**

- L'algorithm Horn-SAT-Glouton termine car à chaque étape on diminue le nombre d'implications non satisfaites.
- L'algorithme Horn-SAT-Glouton s'exécute en temps polynomial dans la taille de $\mathcal{C}$, c-à-d. en $O(\sum_{C \in \mathcal{C}} |C|)$.

Correction de l'algorithme Horn-SAT-Glouton

Algorithm 17 Horn-SAT-Glouton($\mathcal{C}$)

Soit *Prop* l'ensemble des propositions qui apparaissent dans les clauses de $\mathcal{C}$.
Soit $\mathcal{I} \subseteq \mathcal{C}$ l'ensemble des clauses *implications*, comme $x \wedge z \rightarrow w$.
Soit $\mathcal{N} \subseteq \mathcal{C}$ l'ensemble des clauses *négatives pures*, comme $\neg w \vee \neg x \vee \neg y$.

1: **for all** $p \in Prop$ **do**
2: $\quad \nu(p) \leftarrow$ `false` ▷ Initialiser toutes les variables à faux
3: **end for**
4: **while** il existe une implication $C \in \mathcal{I}$ avec $\nu(C) =$ `false` **do**
5: $\quad$ Choisir $C \in \mathcal{I}$ tq $\nu(C) =$ `false` de la forme $C = \ldots \rightarrow p$
6: $\quad \nu(p) \leftarrow$ `true` ▷ modifier $\nu(p)$
7: **end while**
8: **if** pour chaque $C \in \mathcal{N}$, $\nu(C) =$ `true` **then**
9: $\quad$ **return** la valuation ν
10: **else**
11: $\quad$ **return** "la formule n'est pas satisfaisable"
12: **end if**

- Si l'algorithme retourne une valuation, cette valuation satisfait à la fois les implications et les clauses négatives pures, c'est donc bien une solution.

Correction de l'algorithme Horn-SAT-Glouton

- Si l'algorithme répond "la formule n'est pas satisfaisable", ne s'est-il pas un peu hâté ?

On établit un invariant.

Lemma 5.1

La propriété "*Si une variable est mise à vrai, alors elle vaut vrai dans toute valuation solution*" est un invariant de la boucle **while** des lignes 4-7.

À vous de jouer !

Prouver ce lemme.

Algorithm 18 Horn-SAT-Glouton($\mathcal{C}$)

Soit *Prop* l'ensemble des propositions qui apparaissent dans les clauses de $\mathcal{C}$.
Soit $\mathcal{I} \subseteq \mathcal{C}$ l'ensemble des clauses *implications*, comme $x \wedge z \rightarrow w$.
Soit $\mathcal{N} \subseteq \mathcal{C}$ l'ensemble des clauses *négatives pures*, comme $\neg w \vee \neg x \vee \neg y$.

1: **for all** $p \in Prop$ **do**
2: $\quad \nu(p) \leftarrow$ `false` ▷ Initialiser toutes les variables à faux
3: **end for**
4: **while** il existe une implication $C \in \mathcal{I}$ avec $\nu(C) =$ `false` **do**
5: $\quad$ Choisir $C \in \mathcal{I}$ tq $\nu(C) =$ `false` de la forme $C = \ldots \rightarrow p$
6: $\quad \nu(p) \leftarrow$ `true` ▷ modifier $\nu(p)$
7: **end while**
8: **if** pour chaque $C \in \mathcal{N}$, $\nu(C) =$ `true` **then**
9: $\quad$ **return** la valuation ν
10: **else**
11: $\quad$ **return** "la formule n'est pas satisfaisable"
12: **end if**

Donc, à la terminaison de la boucle, si la valuation calculée à la ligne 7 ne satisfait pas une des clauses négatives pures, il ne peut y avoir de valuation qui satisfait toutes les clauses de $\mathcal{C}$. L'algorithme qui dans ce cas répond "la formule n'est pas satisfaisable", est correct.

Intérêt pratique des clauses de Horn

Il existe un algorithme linéraire pour résoudre HORN-SAT-GLOUTON($\mathcal{C}$), le nôtre ne l'est pas (exercice), mais reste polynomial.

Remarque 5.1

Les clauses de Horn sont au coeur de Prolog ("programming by logic"), un langage dans lequel on programme les propriétés attendues en utilisant des expression logiques simples.

Le cheval de bataille des interpreteurs Prolog est notre algorithme glouton.

À vous de jouer !

Informez-vous un minimum sur Prolog.

Codage de Huffman

(Codage de Huffman)

Entrée : Un texte dont on connait l'alphabet fini des symboles, et pour chaque symbole son nombre d'occurrences dans le texte
Sortie : Un encodage qui minimise la longueur du code du texte

Exemple 27

On veut stocker un texte (une chaine de caratères) qui contient les symboles "A,B,C,D".

Lettre	Encodage
A	00
B	01
C	10
D	11

AABBCBCBDD $\rightsquigarrow$ 00000101100110010000011

Pour un texte tel que

Lettre	Nbre d'occurrences
A	70 millions
B	3 millions
C	20 millions
D	37 millions

On aura besoin de $70*2+3*2+20*2+31*2 = 248$ millions de bits pour stocker.

Peut-on faire mieux ?

Tous les codages ne marchent pas !

Exemple 28

Lettre	Encodage
A	0
B	001
C	01
D	11

Comment décoder 001 ?

Cela pourrait être autant le texte “B” que le texte “AC”.

Notion d'arbre préfixe et codage induit

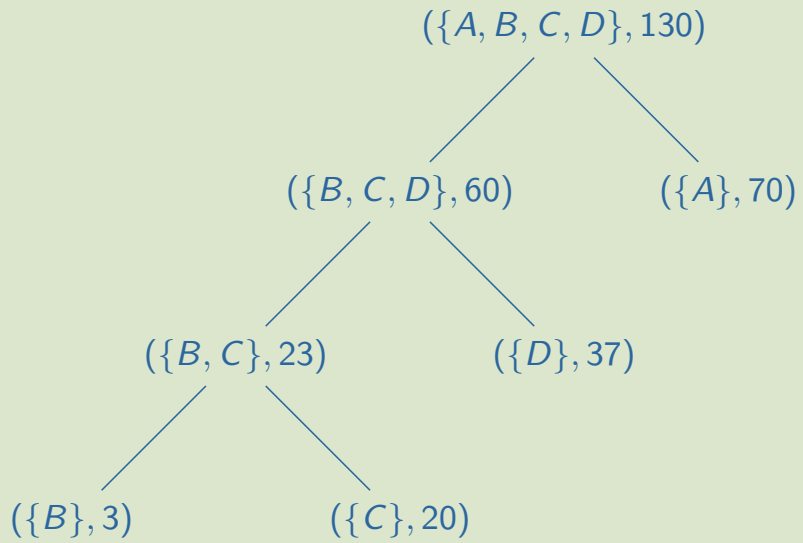
Exemple 29

lettre	fréquence
A	70
B	3
C	20
D	37

Notion d'arbre préfixe et codage induit

Exemple 29

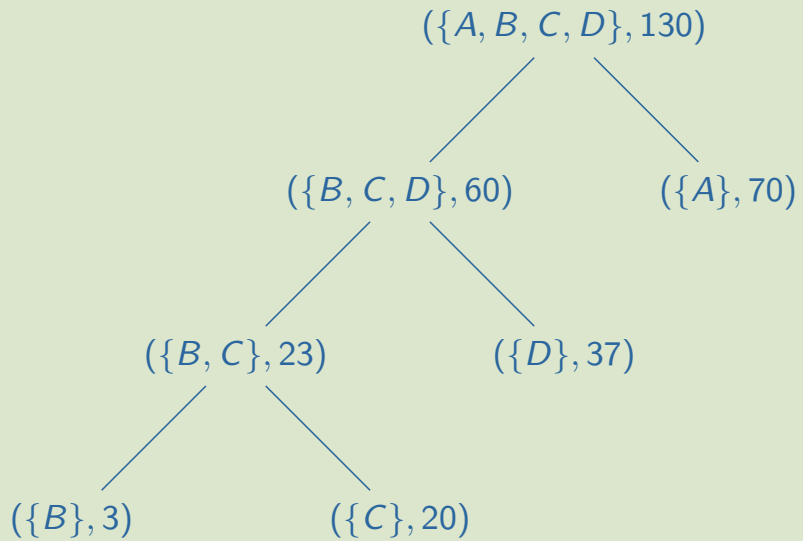
lettre	fréquence
A	70
B	3
C	20
D	37



Notion d'arbre préfixe et codage induit

Exemple 29

lettre	fréquence
A	70
B	3
C	20
D	37



(meta-)lettre	fréquence	Codage
{A,B,C,D}	130	-
{**A**}	70	**1**
{B,C,D}	60	0
{**D**}	37	**01**
{B,C}	23	00
{**C**}	20	**001**
{**B**}	3	**000**

Le codage est le chemin dans l'arbre pour atteindre la (meta-)lettre.
Par exemple, lorsqu'on lit un 0, on sait qu'il s'agit forcément de la meta-lettre $\{B, C, D\}$, c'est à dire que l'on commence le code soit de B, soit de C, soir de D.

Principe de l'algorithme de Huffman

On cherche à compresser un texte (fini) sur un alphabet Σ, où chaque lettre ℓ est codée sur un octet. Pour guider le choix du codage, on dispose d'un tableau `freq` indexé sur Σ où $\texttt{freq}_\ell > 0$ est la fréquence d'occurrence de la lettre ℓ dans le texte.

Initialisation : Pour chaque lettre ℓ, on crée un arbre-feuille d'étiquette $(\ell, \texttt{freq}_\ell)$.

On itère :

1. Classer chaque arbre par fréquence croissante ;
2. Retirer de la liste les deux arbres ayant les fréquences les plus faibles. Insérer, à la place, un nouvel arbre dont la racine pointe sur les racines des deux arbres retirés. Affecter à ce nouvel arbre binaire le couple $(\Lambda, \texttt{freq}_\Lambda)$ où $\Lambda \subseteq \Sigma$ la "meta-lettre" obtenue en rassemblant les meta-lettres de chacun des arbres retirés et $\texttt{freq}_\Lambda$ est la fréquence somme des fréquences des deux arbres retirés.
3. Recommencer en 1. jusqu'à ce qu'il ne reste plus qu'un unique arbre.

À la fin : L'arbre binaire ainsi obtenu est un arbre préfixe qui fournit le code de chaque lettre ℓ comme l'adresse de la feuille $(\ell, \texttt{freq}_\ell)$ dans cet arbre ("0" pour gauche, et "1" pour droite).

Mise en œuvre de cet algorithme de codage

On utilise une file de priorité $\mathcal{F}$, structure de donnée pour un ensemble d'arbres avec les méthodes `defiler` et `enfiler` permettant respectivement d'extraire un arbre de fréquence minimale, ou d'ajouter un élément.
Initialement, on forme la file $\mathcal{F}_0$ contenant tous les arbre-feuilles d'étiquette $(\{\ell\}, \texttt{freq}_\ell)$, où ℓ est une lettre de l'alphabet du texte.

Mise en œuvre de cet algorithme de codage

On utilise une file de priorité $\mathcal{F}$, structure de donnée pour un ensemble d'arbres avec les méthodes `defiler` et `enfiler` permettant respectivement d'extraire un arbre de fréquence minimale, ou d'ajouter un élément.
Initialement, on forme la file $\mathcal{F}_0$ contenant tous les arbre-feuilles d'étiquette $(\{\ell\}, \texttt{freq}_\ell)$, où ℓ est une lettre de l'alphabet du texte.
On appelle *Huffman*($\mathcal{F}_0$), avec :

Algorithm 19 Fonction *Huffman*($\mathcal{F}$)

1: **if** *Card*($\mathcal{F}$) > 1 **then** ▷ il y a au moins deux arbres dans la file $\mathcal{F}$.
2: $T_1 \leftarrow \mathcal{F}$.`defiler` ▷ extraction d'un arbre de fréquence minimale,
3: $T_2 \leftarrow \mathcal{F}$.`defiler` ▷ extraction d'un arbre de fréquence (2ème) minimale,
4: $T \leftarrow T_1 \oplus T_2$
5: $T.\Lambda \leftarrow T_1.\Lambda \cup T_2.\Lambda$ ▷ Meta-lettre de T
6: $T.\texttt{freq} \leftarrow T_1.\texttt{freq} + T_2.\texttt{freq}$ ▷ Fréquence de la meta-lettre de T
7: $\mathcal{F}$.`enfiler`(T) ▷ T est ajouté à $\mathcal{F}$
8: **return** *Huffman*($\mathcal{F}$) ▷ appel récursif avec une file plus petite de 1
9: **else**
10: **return** $\mathcal{F}$.`defiler` ▷ l'unique arbre de $\mathcal{F}$
11: **end if**

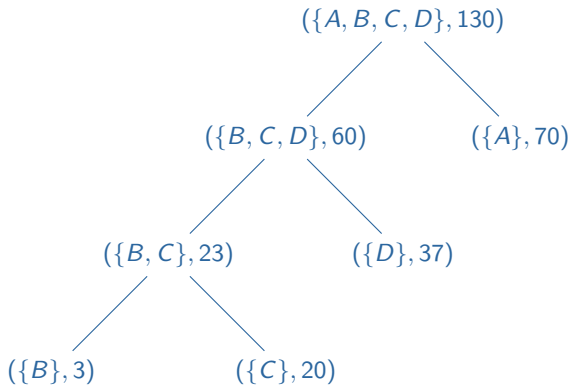
Terminaison de l'algorithme *Huffman*($\mathcal{F}$)

Algorithm 20 Fonction *Huffman*($\mathcal{F}$)

```
1:  if Card(F) > 1 then                          ▷ il y a au moins deux arbres dans la file F.
2:      T1 ← F.defiler                           ▷ extraction d'un arbre de fréquence minimale,
3:      T2 ← F.defiler                           ▷ extraction d'un arbre de fréquence (2ème) minimale,
4:      T ← T1 ⊕ T2
5:      T.Λ ← T1.Λ ∪ T2.Λ                        ▷ Meta-lettre de T
6:      T.freq ← T1.freq + T2.freq               ▷ Fréquence de la meta-lettre de T
7:      F.enfiler(T)                             ▷ T est ajouté à F
8:      return Huffman(F)                        ▷ appel récursif avec une file plus petite de 1
9:  else
10:     return F.defiler                         ▷ l'unique arbre de F
11: end if
```

L'algorithme *Huffman*($\mathcal{F}$) termine car dans l'appel récursif (l'alphabet de) la file a un cardinal strictement plus petit.

Coût $c(T)$ d'un arbre préfixe T



La profondeur d'une lettre est la longueur de son code. Le codage est directement issu de la construction de l'arbre préfixe.

Lettre	Encodage	Profondeur dans l'arbre
A	1	1
D	01	2
B	000	3
C	001	3

L'efficacité d'un arbre est la longueur total du texte encodé, ce qu'on cherche à minimiser. On associe donc un coût à un arbre préfixe :

Définition 30

Soit Σ l'alphabet du texte, et T un arbre préfixe où pour chaque lettre $\ell \in \Sigma$, on note $\texttt{prof}_\ell^T$ sa profondeur dans T.

Le coût de T est $c(T) = \begin{cases} \texttt{freq}_\ell & \text{si } \Sigma = \{\ell\} \\ \sum_{\ell \in \Sigma} \texttt{freq}_\ell \times \texttt{prof}_\ell^T & \text{sinon} \end{cases}$

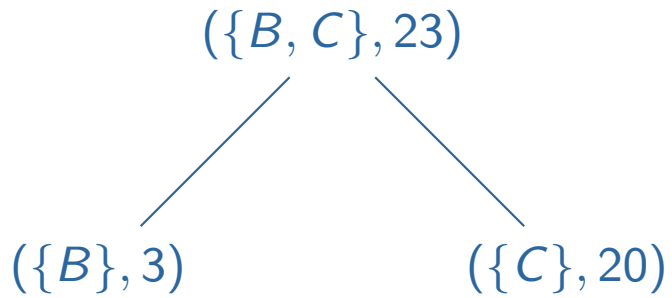
Exemple 31

Pour l'arbre préfixe T en haut, on a $c(T) = 3 \times 3 + 20 \times 3 + 37 \times 2 + 70 \times 1 = 213$.

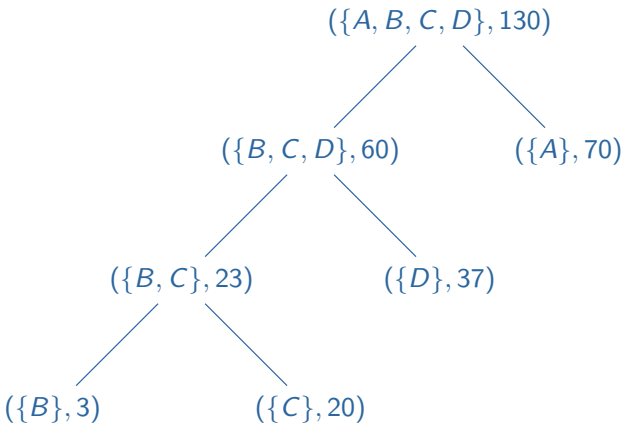
File $\mathcal{F}$ et $\mathcal{F}$-arbre

Pour un état de la file $\mathcal{F}$, on peut aussi voir un arbre comme un $\mathcal{F}$-arbre en arrêtant le dépliage de cet arbre aux noeuds qui sont les meta-lettres des arbres de $\mathcal{F}$.

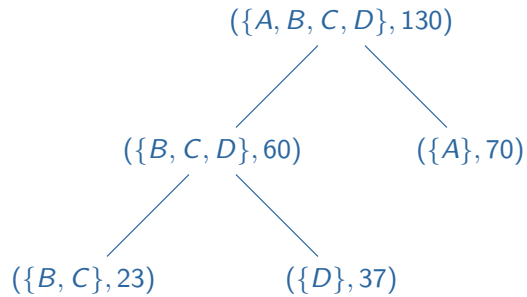
Par exemple, si dans $\mathcal{F}$ on a les trois les arbres :



$(\{D\}, 37)$ $(\{A\}, 70)$.



peut être vu comme le $\mathcal{F}$-arbre



Coût d'un $\mathcal{F}$-arbre

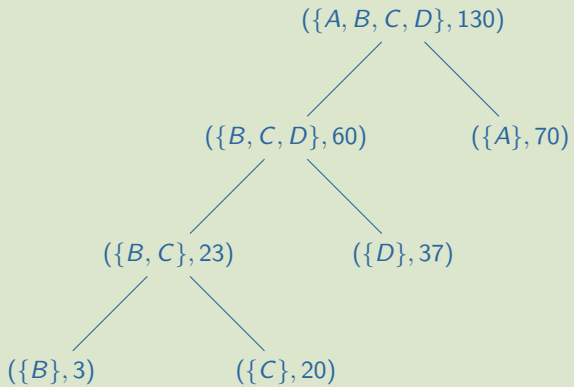
Définition 32

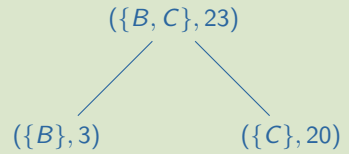
Soit T un $\mathcal{F}$-arbre, le $\mathcal{F}$-coût de T est $c_{\mathcal{F}}(T) = \begin{cases} \mathtt{freq}_\ell & \text{si } \Sigma = \{\ell\} \\ \sum_{\Lambda \in \mathcal{F}} \mathtt{freq}_\Lambda \times \mathtt{prof}_\Lambda^T & \text{sinon} \end{cases}$

Un $\mathcal{F}$-arbre est $\mathcal{F}$-optimal si $c_{\mathcal{F}}(T)$ est minimal parmi tous les $\mathcal{F}$-arbres.

En particulier $c_{\mathcal{F}_0}(T) = c(T)$.

Exemple 33

Pour $T =$ 

et $\mathcal{F} = \{$  $, (\{D\}, 37), (\{A\}, 70)\ \}$

on a : $c_{\mathcal{F}}(T) = \mathtt{freq}_{\{B,C\}} \times 2 + \mathtt{freq}_{\{D\}} \times 2 + \mathtt{freq}_{\{A\}} \times 1 = 23 \times 2 + 37 \times 2 + 70 \times 1 = 190.$

Remarquons que $c(T) = c_{\mathcal{F}}(T) + (\mathtt{freq}_{\{B\}} + \mathtt{freq}_{\{C\}})$.

Correction de l'algorithme *Huffman*($\mathcal{F}$)

Théorème 5.2

L'algorithme *Huffman*($\mathcal{F}_0$) retourne un arbre de coût minimal.

Pour cela, on établit (par récurrence sur cardinal de $\mathcal{F}$) que :

Théorème 5.3

L'algorithme *Huffman*($\mathcal{F}$) retourne un arbre $\mathcal{F}$-optimal

Preuve au tableau.

À vous de jouer !

Pourquoi le Théorème 5.4 entraîne le Théorème 5.2 ?

Correction de Huffman

Théorème 5.4

L'algorithme $Huffman(\mathcal{F})$ retourne un arbre $\mathcal{F}$-optimal.

Si $Card(\mathcal{F}) = 1$, il n'y a qu'un unique $\mathcal{F}$-arbre, qui est donc optimal.

Correction de Huffman

Théorème 5.4

L'algorithme $Huffman(\mathcal{F})$ retourne un arbre $\mathcal{F}$-optimal.

Si $Card(\mathcal{F}) = 1$, il n'y a qu'un unique $\mathcal{F}$-arbre, qui est donc optimal.
Supposons $Card(\mathcal{F}) > 1$.

Correction de Huffman

Théorème 5.4

L'algorithme *Huffman*($\mathcal{F}$) retourne un arbre $\mathcal{F}$-optimal.

Si $Card(\mathcal{F}) = 1$, il n'y a qu'un unique $\mathcal{F}$-arbre, qui est donc optimal.
Supposons $Card(\mathcal{F}) > 1$.

À l'appel *Huffman*($\mathcal{F}$), on note $\mathcal{F}'$ le nouvel état de la file obtenu en remplaçant dans $\mathcal{F}$ les deux arbres T_1 et T_2 de fréquences minimales par un seul arbre, et on a $Card(\mathcal{F}') = Card(\mathcal{F}) - 1$.
Par hypothèse de récurrence, *Huffman*($\mathcal{F}'$) retourne un $\mathcal{F}'$-arbre T' qui est $\mathcal{F}'$-optimal.
Cet $\mathcal{F}'$-arbre T' peut aussi être vu comme un $\mathcal{F}$-arbre et on a :

$$c_{\mathcal{F}}(T') = c_{\mathcal{F}'}(T') + T_1.\texttt{freq} + T_2.\texttt{freq}$$

On compare $c_{\mathcal{F}}(T')$ au coût $c_{\mathcal{F}}(T^*)$ d'un $\mathcal{F}$-arbre T^* qui est $\mathcal{F}$-optimal.

Lemme 34

On peut supposer que dans T^* les feuilles avec les fréquences minimales sont soeurs et de profondeur maximale.

Preuve du lemme

Lemme 35

On peut supposer que dans T^* les feuilles avec les fréquences minimales sont soeurs et de profondeur maximale.

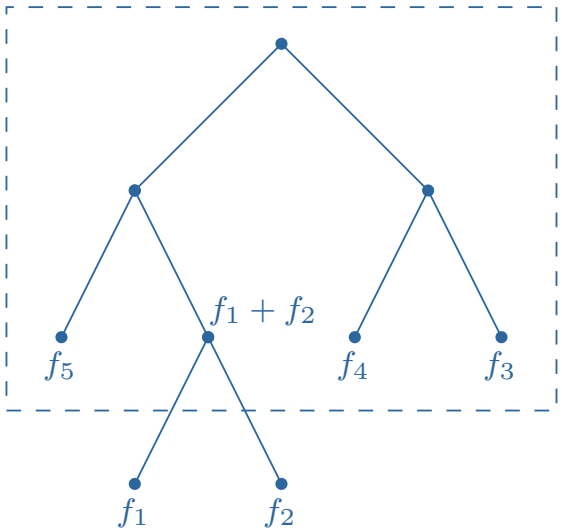
En effet, on prend X et Y deux feuilles soeurs de profondeur maximale $\mathtt{prof}^T_{max}$ dans T^* et on les échange avec les feuilles avec T_1 et T_2 (de profondeurs $\mathtt{prof}^T_1$ et $\mathtt{prof}^T_2$ dans T^*, resp.). Alors le $\mathcal{F}$-arbre obtenu a un $\mathcal{F}$-coût de :

$$\begin{aligned} & c(T^*) - \mathtt{freq}_X \times \mathtt{prof}^T_{max} + \mathtt{freq}_X \times \mathtt{prof}^T_1 - \mathtt{freq}_Y \times \mathtt{prof}^T_{max} + \mathtt{freq}_Y \times \mathtt{prof}^T_2 \\ & - \mathtt{freq}_{T_1} \times \mathtt{prof}^T_1 + \mathtt{freq}_{T_1} \times \mathtt{prof}^T_{max} - \mathtt{freq}_{T_2} \times \mathtt{prof}^T_2 + \mathtt{freq}_{T_2} \times \mathtt{prof}^T_{max} \\ = \quad & c(T^*) \\ & - [(\mathtt{freq}_X - \mathtt{freq}_{T_1}) \times (\mathtt{prof}^T_{max} - \mathtt{prof}^T_1) + (\mathtt{freq}_Y - \mathtt{freq}_{T_2}) \times (\mathtt{prof}^T_{max} - \mathtt{prof}^T_2)] \\ \leq \quad & c(T^*) \qquad \text{(car l'expression en rouge est positive)} \end{aligned}$$

Ainsi l'arbre obtenu est également optimal.

suite preuve

D'après le Lemme, on peut supposer que T_1 et T_2 sont des feuilles soeurs dans le $\mathcal{F}$-arbre T^*. On aussi peut voir T^* comme un $\mathcal{F}'$-arbre :



Notons que

$$c_{\mathcal{F}}(T^*) = c_{\mathcal{F}'}(T^*) + T_1.\texttt{freq} + T_2.\texttt{freq}$$

Comme T' est $\mathcal{F}'$-optimal, on a

$$c_{\mathcal{F}'}(T^*) \geq c_{\mathcal{F}'}(T')$$

Ainsi,

$$c_{\mathcal{F}}(T') = c_{\mathcal{F}'}(T') + T_1.\texttt{freq} + T_2.\texttt{freq} \leq c_{\mathcal{F}'}(T^*) + T_1.\texttt{freq} + T_2.\texttt{freq} = c_{\mathcal{F}}(T^*)$$

De sorte que T' est aussi $\mathcal{F}$-optimal.

Applications du codage de Huffman

Le codage de Huffman apparait pratiquement partout :

- dans les algorithmes de compression gzip, pkzip, winzip, bzip2 ;
- pour les images compressées jpeg, png ;
- pour l'audio compressée mp3.

Le problème du Sac à dos (fractionnaire)/Knapsack (Fractional) Problem

(Knapsack Problem (kp))

Entrée : n objets, pour chaque objet i deux valeurs w_i (≥ 0 son poids) et p_i (son profit/sa valeur), un poinds limite $K \geq 0$
Sortie : Des valeurs $x_1, x_2, \ldots, x_n$ dans $\{0, 1\}$ telles que $\Sigma_{i=1}^{n} w_i.x_i \leq K$ et qui maximise le profit $\Sigma_{i=1}^{n} p_i.x_i$.

À vous de jouer !

On cherche à maximiser $20.x_1 + 16.x_2 + 11.x_3 + 9.x_4 + 7.x_5 + x_6$
avec la contrainte $9.x_1 + 8.x_2 + 6.x_3 + 5.x_4 + 4.x_5 + x_6 \leq 12$
et que les x_is valent soit 0 soit 1.

Combien y a-t-il d'objets ? Quels sont les poids et valeurs de ces objets ?

Une variante de (KP)

(Knapsack Fractional Problem (kfp))

Entrée : n objets, pour chaque objet i deux valeurs w_i (≥ 0 son poids) et p_i (son profit/sa valeur), un poids limite $K \geq 0$
Sortie : Une valeur dans $[0, 1]$ pour les variables $x_1, x_2, \ldots, x_n$ telle que $\Sigma_{i=1}^{n} w_i.x_i \leq K$ et qui maximise le profit $\Sigma_{i=1}^{n} p_i.x_i$.

Définition 36

On dit que (KFP) est une relaxation de (KP), car les variables peuvent prendre des valeurs non entières.

Une approche naturelle pour résoudre ce (KFP) :

> *Le voleur prend d'abord tout l'or, puis tout l'argent et finit de remplir son sac avec le bronze qui peut encore y entrer. Il ne lui viendrait pas à l'idée de prendre moins d'or pour le remplacer par quelque chose de valeur moindre.*

Un algorithme glouton pour Knapsack Fractional Problem

Quitte à les trier, on suppose que les objets $1, \ldots, n$ sont numérotés selon le meilleur rapport profit/poids, c-à-d. $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \frac{p_n}{w_n}$. On décrit un algorithme qui est basé sur

notre approche "naturelle", où `Glouton-KFP`(i, k) résoud le sous-problème :

Entrée : les objets de i à n et $k \geq 0$ le poids restant dans le sac à dos
Sortie : Une valeur des variables $x_i, \ldots, x_n$ telle que $\Sigma_{j=i}^{n} w_j.x_j \leq k$ et qui maximise le profit $\Sigma_{j=i}^{n} p_j.x_j$.

Algorithm 21 `Glouton-KFP`(i, k)

```
1: if w_i ≤ k then
2:     x_i ← 1
3:     Glouton-KFP(i + 1, k − w_i)          ▷ appel du sous-problème
4: else
5:     x_i ← k/w_i          ▷ on prend la proportion qui finit de remplir le sac
6:     x[i + 1 . . . n] ← 0          ▷ le sac est plein on ne prend plus rien d'autre
7: end if
```

Pour résoudre le problème initial, on appelle `Glouton-KFP`$(1, K)$.

Correction de l'algorithme "GreedyKFP"

Algorithm 22 `Glouton-KFP`(i, k)

1: **if** $w_i \leq k$ **then**
2: $\quad x_i \leftarrow 1$
3: $\quad$ `Glouton-KFP`$(i+1, k-w_i)$ ▷ appel du sous-problème
4: **else**
5: $\quad x_i \leftarrow k/w_i$ ▷ on prend la proportion qui finit de remplir le sac
6: $\quad x[i+1 \ldots n] \leftarrow 0$ ▷ le sac est plein on ne prend plus rien d'autre
7: **end if**

On établit que :

1. notre premier choix glouton x_1 est inclus dans une solution optimale,
2. une solution optimale commençant par x_1 contient une solution optimale du KFP pour les objets $\{2, \ldots, n\}$ et la constante $K - w_1.x_1$.

1. Le choix glouton x_1 est inclus dans une solution optimale

Soit $O = (y_1, y_2, \ldots, y_n)$ une solution optimale du problème KFP, et soit $G = (x_1, x_2, \ldots, x_n)$ une solution de `Glouton-KFP`$(1, K)$.

1. Le choix glouton x_1 est inclus dans une solution optimale

Soit $O = (y_1, y_2, \ldots, y_n)$ une solution optimale du problème KFP, et soit $G = (x_1, x_2, \ldots, x_n)$ une solution de `Glouton-KFP`$(1, K)$.

- Clairement $x_1 \geq y_1$ car le choix glouton sélectionne une quantité maximum de l'objet 1 (de valeur maximum).

1. Le choix glouton x_1 est inclus dans une solution optimale

Soit $O = (y_1, y_2, \ldots, y_n)$ une solution optimale du problème KFP, et soit $G = (x_1, x_2, \ldots, x_n)$ une solution de `Glouton-KFP`$(1, K)$.

- Clairement $x_1 \geq y_1$ car le choix glouton sélectionne une quantité maximum de l'objet 1 (de valeur maximum).
- Si $y_1 = x_1$, alors on a fait dans G le même premier choix que pour la solution optimale O.

1. Le choix glouton x_1 est inclus dans une solution optimale

Soit $O = (y_1, y_2, \ldots, y_n)$ une solution optimale du problème KFP, et soit $G = (x_1, x_2, \ldots, x_n)$ une solution de `Glouton-KFP`$(1, K)$.

- Clairement $x_1 \geq y_1$ car le choix glouton sélectionne une quantité maximum de l'objet 1 (de valeur maximum).
- Si $y_1 = x_1$, alors on a fait dans G le même premier choix que pour la solution optimale O.
- sinon $(x_1 > y_1)$

1. Le choix glouton x_1 est inclus dans une solution optimale

Soit $O = (y_1, y_2, \ldots, y_n)$ une solution optimale du problème KFP, et soit $G = (x_1, x_2, \ldots, x_n)$ une solution de `Glouton-KFP`$(1, K)$.

- Clairement $x_1 \geq y_1$ car le choix glouton sélectionne une quantité maximum de l'objet 1 (de valeur maximum).
- Si $y_1 = x_1$, alors on a fait dans G le même premier choix que pour la solution optimale O.
- sinon $(x_1 > y_1)$, mais alors le poids de la solution O est :

$$w(O) = \sum_{i=1}^{n} w_i.y_i (\leq K)$$

1. Le choix glouton x_1 est inclus dans une solution optimale

Soit $O = (y_1, y_2, \ldots, y_n)$ une solution optimale du problème KFP, et soit $G = (x_1, x_2, \ldots, x_n)$ une solution de `Glouton-KFP`$(1, K)$.

- Clairement $x_1 \geq y_1$ car le choix glouton sélectionne une quantité maximum de l'objet 1 (de valeur maximum).
- Si $y_1 = x_1$, alors on a fait dans G le même premier choix que pour la solution optimale O.
- sinon $(x_1 > y_1)$, mais alors le poids de la solution O est :

$$w(O) = \sum_{i=1}^{n} w_i.y_i (\leq K)$$

mais qu'on peut réécrire :

$$w(O) = w_1.x_1 + \sum_{i=1}^{n} w_i.y_i - w_1.(x_1 - y_1)$$

1. Le choix glouton x_1 est inclus dans une solution optimale

Soit $O = (y_1, y_2, \ldots, y_n)$ une solution optimale du problème KFP, et soit $G = (x_1, x_2, \ldots, x_n)$ une solution de `Glouton-KFP`$(1, K)$.

- Clairement $x_1 \geq y_1$ car le choix glouton sélectionne une quantité maximum de l'objet 1 (de valeur maximum).
- Si $y_1 = x_1$, alors on a fait dans G le même premier choix que pour la solution optimale O.
- sinon $(x_1 > y_1)$, mais alors le poids de la solution O est :

$$w(O) = \sum_{i=1}^{n} w_i.y_i (\leq K)$$

mais qu'on peut réécrire :

$$w(O) = w_1.x_1 + \sum_{i=1}^{n} w_i.y_i - w_1.(x_1 - y_1)$$

On modifie alors O en la solution O_1 obtenue en retirant une quantité de poids $w_1.(x_1 - y_1)$ des objets les moins précieux, pour la reporter en objet 1 :

$$O_1 = (x_1, z_2, \ldots, z_n)$$

O_1 est aussi optimale puisque (a) par report des poids, on a $w(O_1) = w(O) \leq K$, et (b) $p(O_1) \geq p(O)$, puisque l'objet 1 est plus précieux que ceux qu'on a retirés.

2. Si $O_1 = (x_1, z_2, \ldots, z_n)$ est une solution optimale de KFP alors $(z_2, \ldots, z_n)$ est une solution optimale pour le sous-problème avec les objets $\{2, \ldots, n\}$ et le poids limite $K - w_1.x_1$.

2. Si $O_1 = (x_1, z_2, \ldots, z_n)$ est une solution optimale de KFP alors $(z_2, \ldots, z_n)$ est une solution optimale pour le sous-problème avec les objets $\{2, \ldots, n\}$ et le poids limite $K - w_1.x_1$.

On rappelle que O_1 est obtenue en prélevant un poids de $w_1.(x_1 - y_1)$ en partant des objets les moins précieux, que l'on rajoute en quantité de l'objet 1, et que O_1 est une solution optimale pour le problème initial avec les objets $\{1, 2, \ldots, n\}$ et le poids limite K (on le note $KFP(1)$).

On note $KFP(2)$ le sous-problème sur les objets $\{2, \ldots, n\}$ avec le poids limite $K - w_1.x_1$.

- $(z_2, \ldots, z_n)$ est une solution de $KFP(2)$ car

$$w((z_2, \ldots, z_n)) = w(O_1) - w_1.x_1 \leq K - w_1.x_1$$

2. Si $O_1 = (x_1, z_2, \ldots, z_n)$ est une solution optimale de KFP alors $(z_2, \ldots, z_n)$ est une solution optimale pour le sous-problème avec les objets $\{2, \ldots, n\}$ et le poids limite $K - w_1.x_1$.

On rappelle que O_1 est obtenue en prélevant un poids de $w_1.(x_1 - y_1)$ en partant des objets les moins précieux, que l'on rajoute en quantité de l'objet 1, et que O_1 est une solution optimale pour le problème initial avec les objets $\{1, 2, \ldots, n\}$ et le poids limite K (on le note $KFP(1)$).

On note $KFP(2)$ le sous-problème sur les objets $\{2, \ldots, n\}$ avec le poids limite $K - w_1.x_1$.

- $(z_2, \ldots, z_n)$ est une solution de $KFP(2)$ car

$$w((z_2, \ldots, z_n)) = w(O_1) - w_1.x_1 \leq K - w_1.x_1$$

- Supposons $(z_2, \ldots, z_n)$ non-optimale pour $KFP(2)$.

2. Si $O_1 = (x_1, z_2, \ldots, z_n)$ est une solution optimale de KFP alors $(z_2, \ldots, z_n)$ est une solution optimale pour le sous-problème avec les objets $\{2, \ldots, n\}$ et le poids limite $K - w_1.x_1$.

On rappelle que O_1 est obtenue en prélevant un poids de $w_1.(x_1 - y_1)$ en partant des objets les moins précieux, que l'on rajoute en quantité de l'objet 1, et que O_1 est une solution optimale pour le problème initial avec les objets $\{1, 2, \ldots, n\}$ et le poids limite K (on le note $KFP(1)$).

On note $KFP(2)$ le sous-problème sur les objets $\{2, \ldots, n\}$ avec le poids limite $K - w_1.x_1$.

- $(z_2, \ldots, z_n)$ est une solution de $KFP(2)$ car

$$w((z_2, \ldots, z_n)) = w(O_1) - w_1.x_1 \leq K - w_1.x_1$$

- Supposons $(z_2, \ldots, z_n)$ non-optimale pour $KFP(2)$.

  Alors il existe une autre solution $O_2 = (t_2, \ldots, t_n)$ telle que $p(O_2) > p((z_2, \ldots, z_n))$.

  Mais alors $(x_1, t_2, \ldots, t_n)$ est aussi une solution de $KFP(1)$, telle que

$$p((x_1, t_2, \ldots, t_n)) > p(O_1)$$

  ce qui contredit l'optimalité de O_1.

En récapitulant pour KFP

- Après le choix glouton x_1 pour $KFP(1)$, on est ramené à résoudre optimalement le sous-problème problème $KFP(2)$ (de même nature que le problème d'origine) ;
- pour $KFP(2)$ il existe une solution optimale qui fait le choix x_2 et on doit ensuite résoudre le sous-problème $KFP(3)$, celui sur les objets $\{3, \ldots, n\}$ avec le poids limite $K - x_1 w_1 - x_2 w_2$;
- et ainsi de suite, prouvant ainsi que la solution gloutonne est elle-même optimale.

De manière générale, l'optimalité d'un glouton s'établit en montrant :

1. qu'à chaque étape, il existe une solution optimale qui contient notre choix glouton, c'est la propriété du "choix glouton".
2. que toute solution optimale repose sur des solutions optimales de sous-problèmes, c'est la propriété de "sous-structure optimale".

Version itérative de l'algorithme glouton pour *KFP*

À vous de jouer !

Puisque `Glouton-KFP`(i, k) est récursif terminal, écrire une version itérative de `Glouton-KFP`$(1, K)$.

Knapsack Problem non fractionnaire

On se pose naturellement la question d'exploiter le principe de `Glouton-KFP` pour résoudre le problem KP, et en particulier en ne gardant que les items "pleins".
On obtient alors l'algorithme :

Algorithm 23 `Glouton-KP`(i, k)

```
1: if w_i ≤ k then
2:     x_i ← 1
3:     Glouton-KP(i + 1, k − w_i)          ▷ appel du sous-problème
4: else
5:     x_i ← 0
6:     Glouton-KP(i + 1, k)                ▷ appel du sous-problème
7: end if
```

À vous de jouer !

L'algorithme `Glouton-KP`$(1, K)$ est-il correct, c-à-d. retourne-t-il une solution optimale ?

Non, l'algorithme `Glouton-KP`$(1, K)$ n'est pas correct !

Exemple 37

On cherche à maximmiser $20.x_1 + 16.x_2 + 11.x_3 + 9.x_4 + 7.x_5 + x_6$

avec les contraintes $\begin{cases} 9.x_1 + 8.x_2 + 6.x_3 + 5.x_4 + 4.x_5 + x_6 \leq 12 \\ x_i \in \{0, 1\} \text{ pour } i = 1, \ldots, 6 \end{cases}$

En appliquant `Glouton-KP`$(1, K)$ à $w = [9, 8, 6, 5, 4, 1], p = [20, 16, 11, 9, 7, 1], K = 12$, on obtient successivement :

0. $k = 12$
1. $x_1 = 1$ et $k = 3$
2. $x_2 = 0$ et $k = 3$, puis ... puis $x_5 = 0$ et $k = 3$
3. $x_6 = 1$ et $k = 2$.

À la fin de l'algorithme $k = 2$, le sac n'est pas plein, et la solution $(1, 0, 0, 0, 0, 1)$ calculée à la valeur 21.

Or il existe une meilleure solution : la valeur de $(0, 1, 0, 0, 1, 0)$ et $z = 23$ (et le sac est plein).

Théorie de la NP-complétude

Modèles et anti-modèles d'algorithmes

Modèles d'algorithmes

- Gloutons
- Diviser pour Régner
- Programmation dynamique
- Programmation linéaire + Dualité (on le verra plus tard)

Modèles et anti-modèles d'algorithmes

Modèles d'algorithmes

- Gloutons
- Diviser pour Régner
- Programmation dynamique
- Programmation linéaire + Dualité (on le verra plus tard)
- Réductions

"Anti-modèles" d'algorithmes

- La NP-complétude d'un problème est critère pour dire qu'il n'y a (*sans doute*) aucun algorithme en temps polynomial qui le résoud.

Modèles et anti-modèles d'algorithmes

Modèles d'algorithmes

- Gloutons
- Diviser pour Régner
- Programmation dynamique
- Programmation linéaire + Dualité (on le verra plus tard)
- Réductions

"Anti-modèles" d'algorithmes

- La NP-complétude d'un problème est critère pour dire qu'il n'y a (*sans doute*) aucun algorithme en temps polynomial qui le résoud.
- Il existe d'autres classes de complexité de problèmes, telle que la classe des problèmes PSPACE-complets qui indique qu'un algorithme en espace polynomial est nécessaire.
- Enfin, le critère d'un problème indécidable indique qu'il n'y a aucun algorithme qui peut le résoudre.

À vous de jouer !

Regardez la preuve que le problème de Correspondance de Post est indécidable.

Classification des problèmes (décidables) en fonction de leur difficulté/complexité intrinsèque

Quels problèmes sommes-nous capables de résoudre en pratique ?

Classification des problèmes (décidables) en fonction de leur difficulté/complexité intrinsèque

Quels problèmes sommes-nous capables de résoudre en pratique ?

Une définition qui fait le consensus : Ceux qui ont des algorithmes polynomiaux

- Au niveau théorique : Définition vaste et robuste car elle se compose bien.
- Au niveau pratique : Les algorithmes polynomiaux qui passent à l'échelle

Classification des problèmes (décidables) en fonction de leur difficulté/complexité intrinsèque

Quels problèmes sommes-nous capables de résoudre en pratique ?

Une définition qui fait le consensus : Ceux qui ont des algorithmes polynomiaux

- Au niveau théorique : Définition vaste et robuste car elle se compose bien.
- Au niveau pratique : Les algorithmes polynomiaux qui passent à l'échelle

Problèmes pour lesquels on a exhibé un algo polynomial	**Problèmes pour lesquels on n'a sans doute pas d'algo polynomial**
plus court chemin	plus long chemin
2SAT	3SAT
4-coloration planaire	3-coloration planaire
couverture de sommets bipartite	couverture de sommets
matching	3D-matching
test de primalité	factorisation
programmation lineaire	programmation entière

Classification des problèmes (décidables) en fonction de leur difficulté/complexité intrinsèque

Quels problèmes sommes-nous capables de résoudre en pratique ?

Une définition qui fait le consensus : Ceux qui ont des algorithmes polynomiaux

- Au niveau théorique : Définition vaste et robuste car elle se compose bien.
- Au niveau pratique : Les algorithmes polynomiaux qui passent à l'échelle

MAIS COMMENT LE SAIT-ON ? !

↓

Problèmes pour lesquels on a exhibé un algo polynomial	**Problèmes pour lesquels on n'a sans doute pas d'algo polynomial**
plus court chemin	plus long chemin
2SAT	3SAT
4-coloration planaire	3-coloration planaire
couverture de sommets bipartite	couverture de sommets
matching	3D-matching
test de primalité	factorisation
programmation lineaire	programmation entière

Desiderata : Classifier les problèmes (décidables)

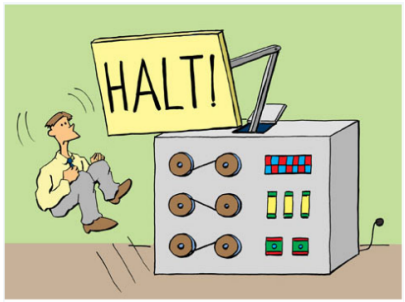
On sépare les problèmes que l'on sait résoudre en temps polynomial (faciles à résoudre) et ceux dont on conjecture qu'ils ne peuvent pas l'être – en anglais ces derniers sont dits intractable, en français on dira intraitables.

Desiderata : Classifier les problèmes (décidables)

On sépare les problèmes que l'on sait résoudre en temps polynomial (faciles à résoudre) et ceux dont on conjecture qu'ils ne peuvent pas l'être – en anglais ces derniers sont dits intractable, en français on dira intraitables.

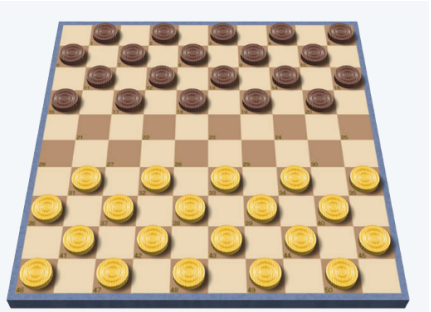
Exemples de problèmes intraitables :

- Étant donné un programme (ou algorithme, ou Machine de Turing déterministe), s'arrête-t-il en au plus k étapes ?



Alan designed the perfect computer

- Étant donné une position sur un échiquer $n \times n$ de dames, les noirs ont-ils une stratégie gagnante ?

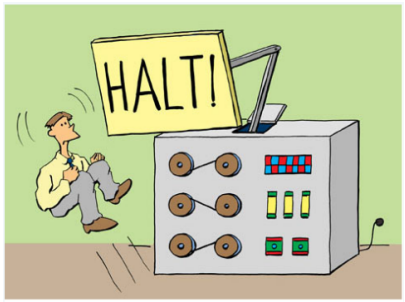


Desiderata : Classifier les problèmes (décidables)

On sépare les problèmes que l'on sait résoudre en temps polynomial (faciles à résoudre) et ceux dont on conjecture qu'ils ne peuvent pas l'être – en anglais ces derniers sont dits intractable, en français on dira intraitables.

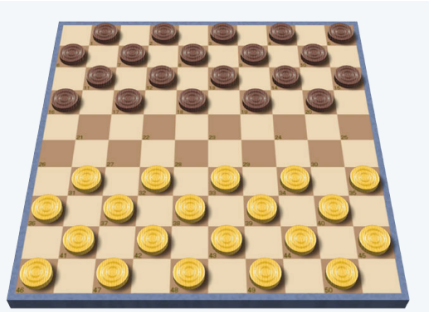
Exemples de problèmes intraitables :

- Étant donné un programme (ou algorithme, ou Machine de Turing déterministe), s'arrête-t-il en au plus k étapes ?



Alan designed the perfect computer

- Étant donné une position sur un échiquer $n \times n$ de dames, les noirs ont-ils une stratégie gagnante ?



Nouvelle frustrante Un nombre gigantesque de problèmes fondamentaux défient la classification depuis des décennies.

Mais d'ailleurs c'est quoi un "problème" ?

Définition 38

Un problème (de décision) $\mathcal{X}$ est une question **oui/non** sur des entrées.

LE NOM DU PROBLÈME

Entrée :	$\mathcal{I}$, appelée une *instance* (les données d'entrée)
Sortie :	la réponse (**oui**/**non**) à la question posée dans ce problème

Mais d'ailleurs c'est quoi un "problème" ?

Définition 38

Un problème (de décision) $\mathcal{X}$ est une question **oui/non** sur des entrées.

LE NOM DU PROBLÈME

Entrée : $\mathcal{I}$, appelée une *instance* (les données d'entrée)
Sortie : la réponse (**oui**/**non**) à la question posée dans ce problème

Par exemple (et présenté de façon informelle) :

- Le graphe a-t-il un cycle Hamiltonien ?
- Cette formule propositionnelle est-elle satifaisable ?
- Le sac à dos peut-il être rempli avec un profit d'au moins p ?

À vous de jouer !

Formaliser les problèmes ci-dessus.

Mais d'ailleurs c'est quoi un "problème" ?

Définition 38

Un problème (de décision) $\mathcal{X}$ est une question **oui/non** sur des entrées.

LE NOM DU PROBLÈME

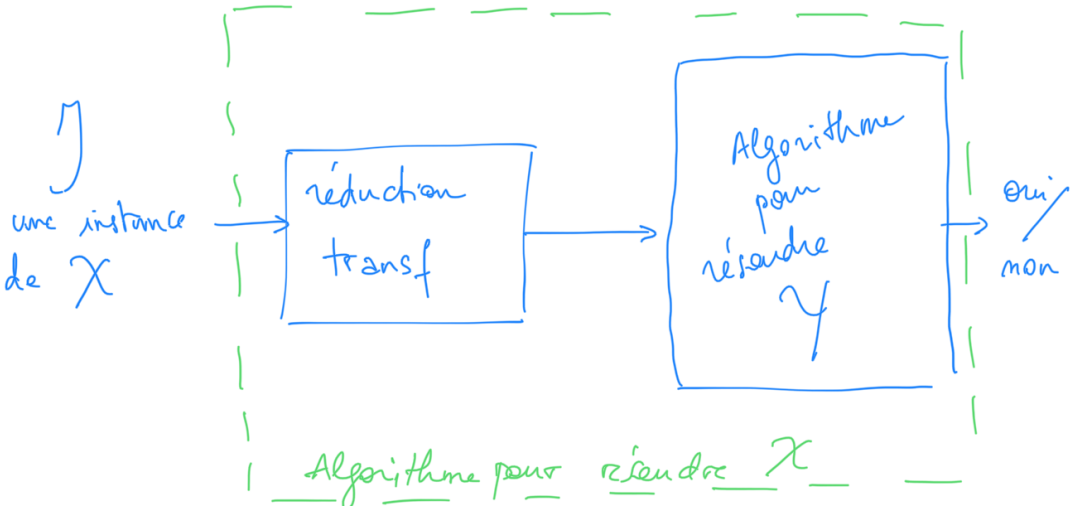
Entrée :	$\mathcal{I}$, appelée une *instance* (les données d'entrée)
Sortie :	la réponse (**oui**/**non**) à la question posée dans ce problème

On confond $\mathcal{X}$ avec l'ensemble des instances positives, *c-à-d.* celles pour lesquelles la réponse est **oui**.

On notera donc $\mathcal{I} \in \mathcal{X}$ pour indiquer que $\mathcal{I}$ est une instance positive de $\mathcal{X}$, et $\mathcal{I} \notin \mathcal{X}$ sinon.

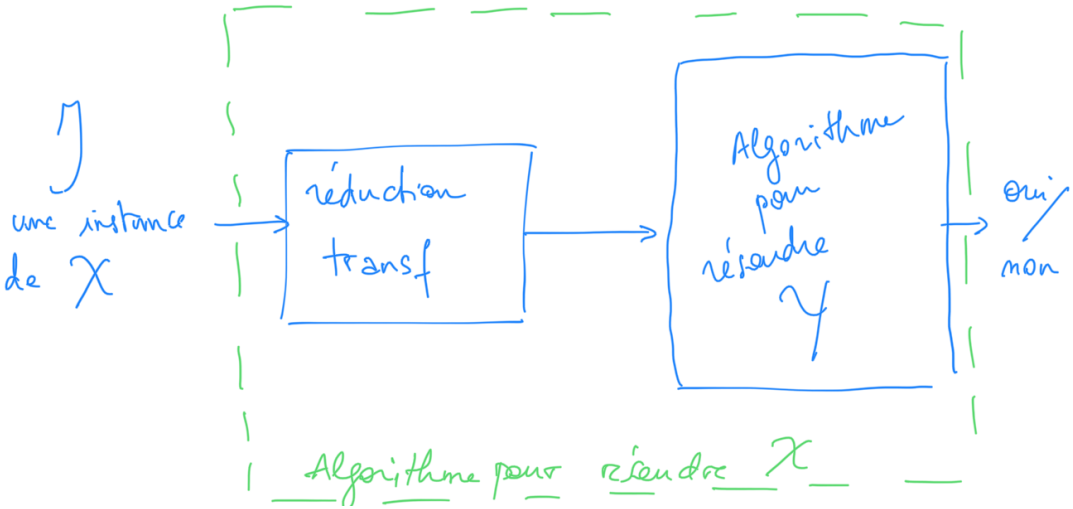
Réduction polynomiale

Desiderata Supposons que l'on sait résoudre un problème donné $\mathcal{Y}$ en temps polynomial. Que pourrions-nous résoudre d'autre en temps polynomial ?



Réduction polynomiale

Desiderata Supposons que l'on sait résoudre un problème donné $\mathcal{Y}$ en temps polynomial. Que pourrions-nous résoudre d'autre en temps polynomial ?



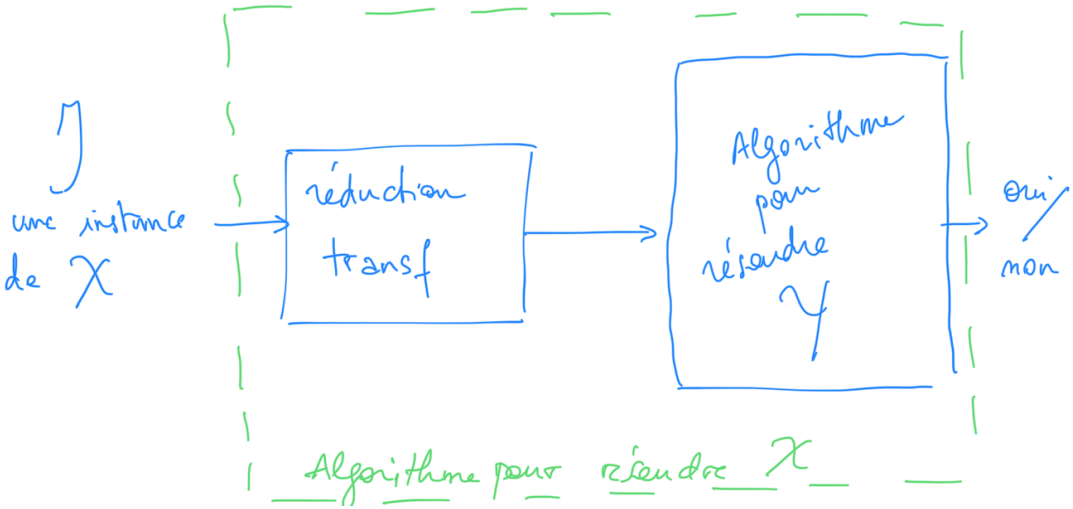
Définition 39 (Réduction polynomiale – "à la Karp")

Le problème $\mathcal{X}$ se réduit au problème $\mathcal{Y}$, noté $\mathcal{X} \leq_P \mathcal{Y}$, s'il existe un algorithme $\mathrm{transf}(.)$ appelé réduction qui transforme temps polynomial toute instance $\mathcal{I}$ de $\mathcal{X}$ en une instance $\mathrm{transf}(\mathcal{I})$ de $\mathcal{Y}$ en de sorte que

$$\mathcal{I} \in \mathcal{X} \text{ si, et seulement si, } \mathrm{transf}(\mathcal{I}) \in \mathcal{Y}$$

Réduction polynomiale

Desiderata Supposons que l'on sait résoudre un problème donné $\mathcal{Y}$ en temps polynomial. Que pourrions-nous résoudre d'autre en temps polynomial ?



Définition 39 (Réduction polynomiale – "à la Karp")

Le problème $\mathcal{X}$ se réduit au problème $\mathcal{Y}$, noté $\mathcal{X} \leq_P \mathcal{Y}$, s'il existe un algorithme $\mathrm{transf}(.)$ appelé réduction qui transforme temps polynomial toute instance $\mathcal{I}$ de $\mathcal{X}$ en une instance $\mathrm{transf}(\mathcal{I})$ de $\mathcal{Y}$ en de sorte que

$$\mathcal{I} \in \mathcal{X} \text{ si, et seulement si, } \mathrm{transf}(\mathcal{I}) \in \mathcal{Y}$$

On remarque que forcément $|\mathrm{transf}(\mathcal{I})| \leq p(|\mathcal{I}|)$ pour un certain polynôme $p(.)$.

Réduction polynomiale : pour la difficulté relative des problèmes

Théorème 40 (pour la conception d'algorithmes)

Si $\mathcal{X} \leq_P \mathcal{Y}$ et $\mathcal{Y}$ peut être résolu en temps polynomial, alors $\mathcal{X}$ peut être résolu en temps polynomial.

Réduction polynomiale : pour la difficulté relative des problèmes

Théorème 40 (pour la conception d'algorithmes)

Si $\mathcal{X} \leq_P \mathcal{Y}$ et $\mathcal{Y}$ peut être résolu en temps polynomial, alors $\mathcal{X}$ peut être résolu en temps polynomial.

Théorème 41 (pour établir l'intraitabilité)

Si $\mathcal{X} \leq_P \mathcal{Y}$ et $\mathcal{X}$ ne peut pas être résolu en temps polynomial, alors $\mathcal{Y}$ ne peut pas être résolu en temps polynomial.

Réduction polynomiale : pour la difficulté relative des problèmes

Théorème 40 (pour la conception d'algorithmes)

Si $\mathcal{X} \leq_P \mathcal{Y}$ et $\mathcal{Y}$ peut être résolu en temps polynomial, alors $\mathcal{X}$ peut être résolu en temps polynomial.

Théorème 41 (pour établir l'intraitabilité)

Si $\mathcal{X} \leq_P \mathcal{Y}$ et $\mathcal{X}$ ne peut pas être résolu en temps polynomial, alors $\mathcal{Y}$ ne peut pas être résolu en temps polynomial.

Définition 42

On note $\mathcal{X} \equiv_P \mathcal{Y}$ lorsque $\mathcal{X} \leq_P \mathcal{Y}$ et $\mathcal{Y} \leq_P \mathcal{X}$

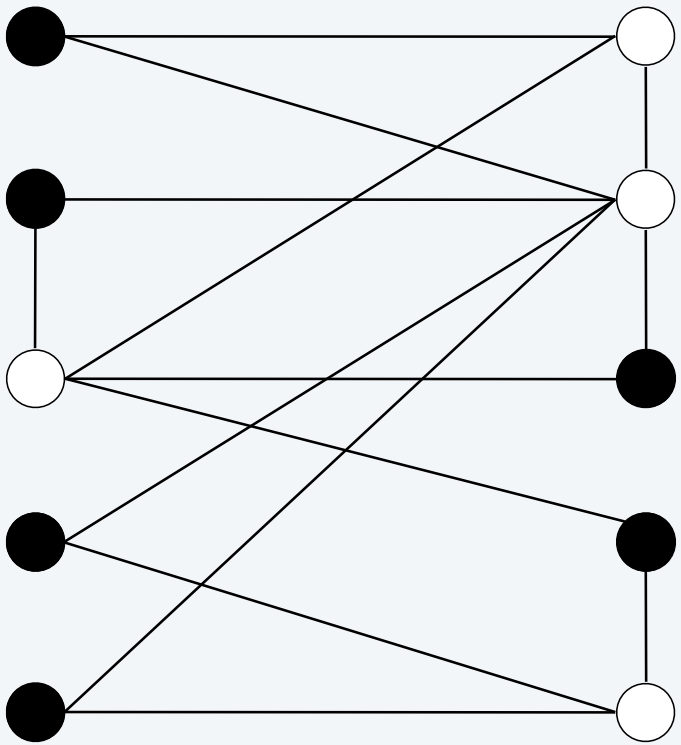
Théorème 43

Si $\mathcal{X} \equiv_P \mathcal{Y}$, alors
$\mathcal{X}$ peut être résolu en temps polynomial ssi $\mathcal{Y}$ peut être résolu en temps polynomial.

Des exemples de type Packing and covering problems

Independent Set

INDEPENDENT-SET Étant donné un graphe $G = (V, E)$ et un entier k, existe-t-il un sous-ensemble de sommets $I \subseteq V$ tel que $|I| \geq k$ et qu'aucune paire de sommets dans I n'est reliée par un arc ?

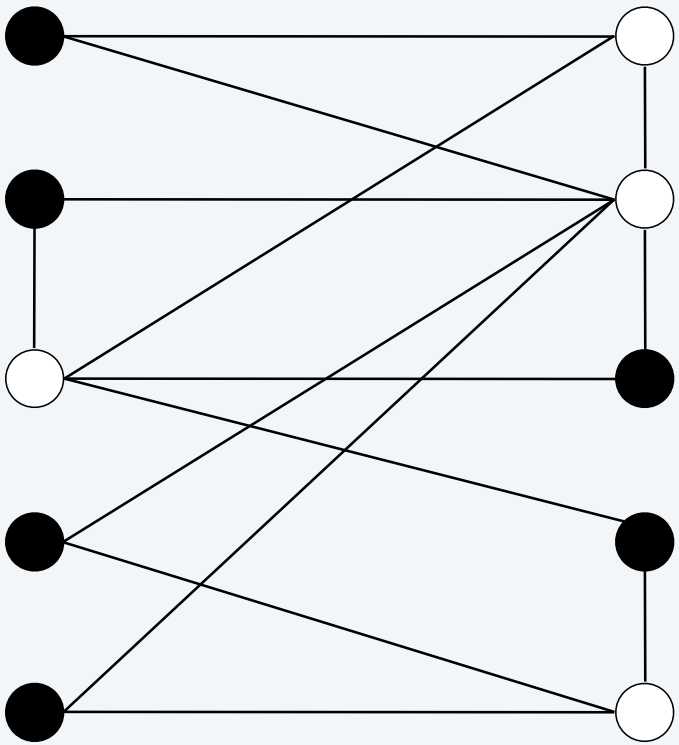


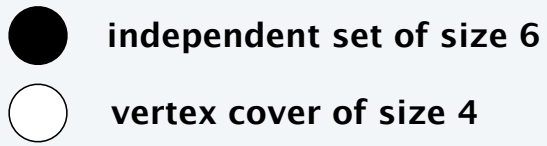
independent set of size 6

11

Vertex Cover (couverture de sommets)

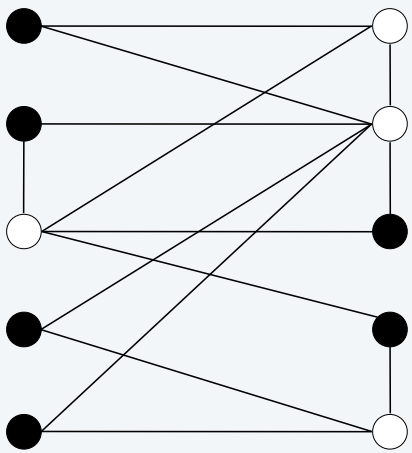
VERTEX-COVER Étant donné un graphe $G = (V, E)$ et un entier k, existe-t-il un sous-ensemble de sommets $C \subseteq V$ tel que $|C| \leq k$ et pour chaque arc, au moins une des deux extrémités est dans C (Autrement dit, pour tous $(u, v) \in E$, $u \in C$ ou $v \in C$) ?

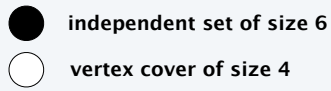




12

Vertex Cover et Independent Set se réduisent l'un à l'autre



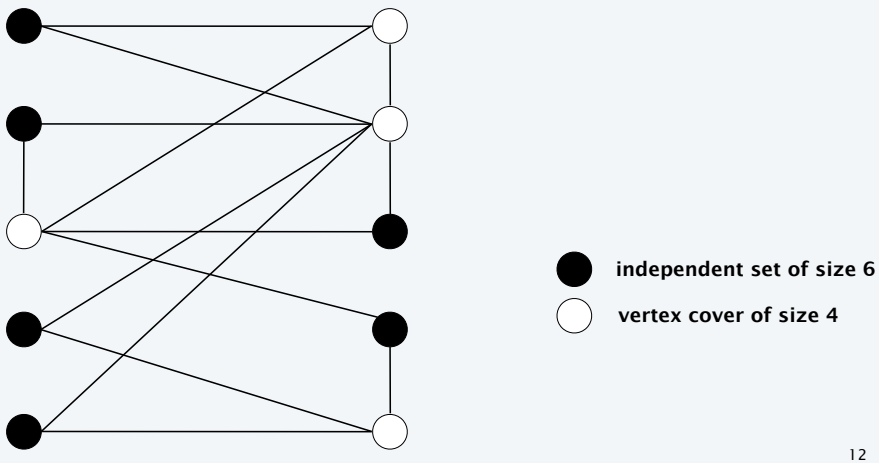


12

Théorème 44

VERTEX-COVER $\equiv_P$ INDEPENDENT-SET

Vertex Cover et Independent Set se réduisent l'un à l'autre



Théorème 44

VERTEX-COVER $\equiv_P$ INDEPENDENT-SET

Démonstration.

Elle repose sur le fait suivant.

C est un Vertex Cover de cardinal k ssi $V \setminus C$ est un Independent Set de cardinal $n - k$.

□

À vous de jouer !

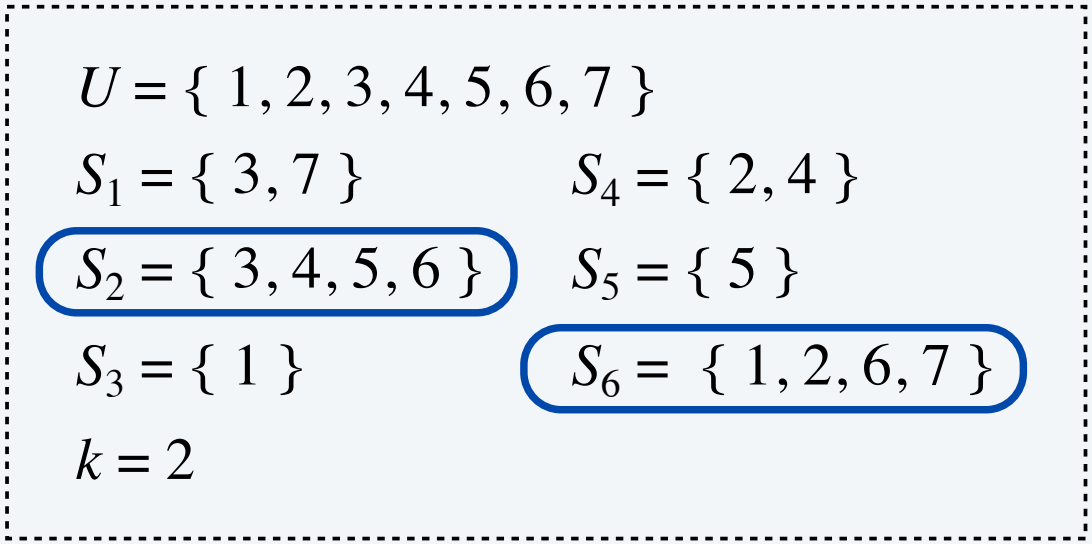
Terminer la preuve du Théorème 44.

Set Cover

SET-COVER Étant donné un ensemble $U = \{1, 2, \ldots, n\}$ d'éléments, une collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ de sous-ensembles de U, et un entier k, existe-t-il au plus k ensembles de $\mathcal{S}$ dont l'union est égale à U ?

Exemples d'application

- m modules logiciels.
- U un ensemble de n fonctionnalités que vous aimeriez que votre système ait.
- Le $i$ème module fournit l'ensemble $S_i \subseteq U$ de fonctionnalités.
- But : réaliser les n fonctionnalités en utilisant le moins de modules possible.



a set cover instance

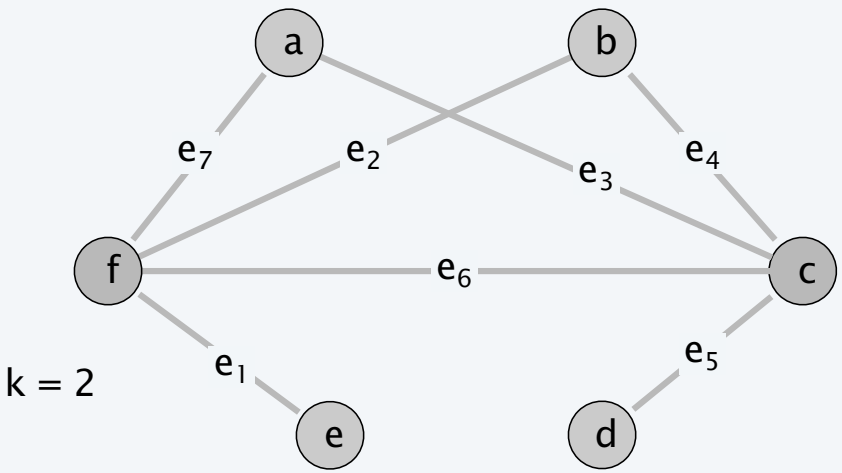
Vertex Cover se réduit à Set Cover

Théorème 45

VERTEX-COVER $\leq_P$ SET-COVER

Démonstration.

Étant donnée une instance $G = (V, E)$ (et k) de VERTEX-COVER, on construit l'instance $(U, \mathcal{S}, k)$ de SET-COVER de taille polynomial en la taille de G.



vertex cover instance
(k = 2)

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

set cover instance
(k = 2)

Vertex Cover se réduit à Set Cover

Théorème 45

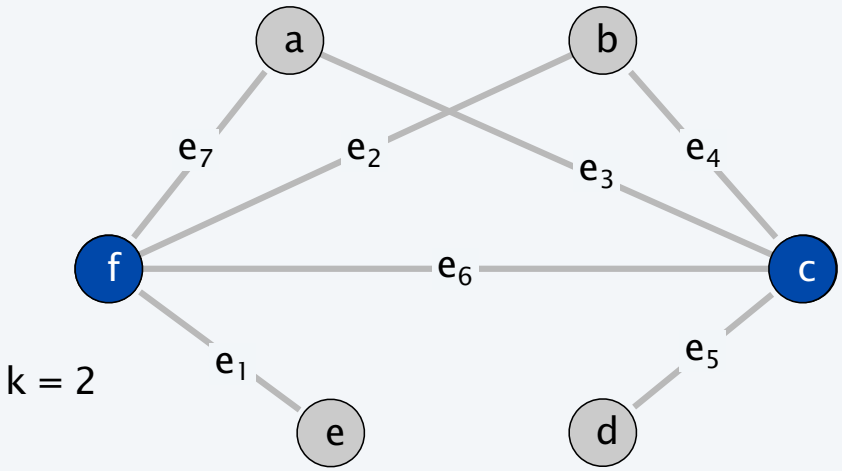
VERTEX-COVER $\leq_P$ SET-COVER

Démonstration.

Étant donnée une instance $G = (V, E)$ (et k) de VERTEX-COVER, on construit l'instance $(U, \mathcal{S}, k)$ de SET-COVER de taille polynomial en la taille de G.

Lemme 46

G a un Vertex Cover de taille k ssi $(U, \mathcal{S})$ a un Set Cover de taille k.



vertex cover instance (k = 2)

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$S_a = \{ 3, 7 \}$ $S_b = \{ 2, 4 \}$

$S_c = \{ 3, 4, 5, 6 \}$ $S_d = \{ 5 \}$

$S_e = \{ 1 \}$ $S_f = \{ 1, 2, 6, 7 \}$

set cover instance (k = 2)

Un exemple de type Constraint satisfaction problems

Satisfaisabilité

Littéral Une variable booléenne ou sa négation x_i ou $\bar{x}_i$

Clause Une disjonction de littéraux $C_j = x_1 \vee \bar{x}_2 \vee x_3$

Forme normale conjonctive (FNC) $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$
Une formule propositionnelle Φ qui est une conjonction de clauses

SAT-FNC Étant donnée une formule Φ en FNC, a-t-elle un modèle ?
3-SAT-FNC Le sous-problème de SAT-FNC où les clauses ont exactement 3 littéraux.

Par exemple

$$\Phi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

est une instance positive de 3-SAT-FNC, avec comme témoin/certificat

$$val(x_1) = \texttt{true}, val(x_2) = \texttt{true}, val(x_3) = \texttt{false}, val(x_4) = \texttt{false}$$

Application Electronic design automation (EDA).

3-SAT se réduit à Independent Set

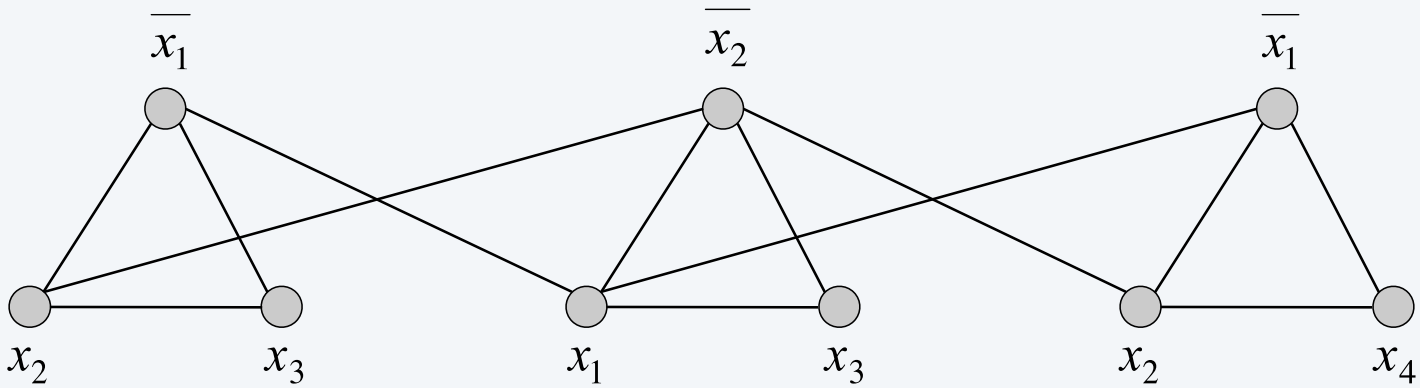
Théorème 47

3-SAT $\leq_P$ Independent-Set

Démonstration.

G contient 3 sommets pour chaque clause, un pour chacun de ses littéraux ;
On connecte ces 3 littéraux par un triangle ;
On connecte chaque littéral à sa négation.

G



k = 3

$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4\right)$$

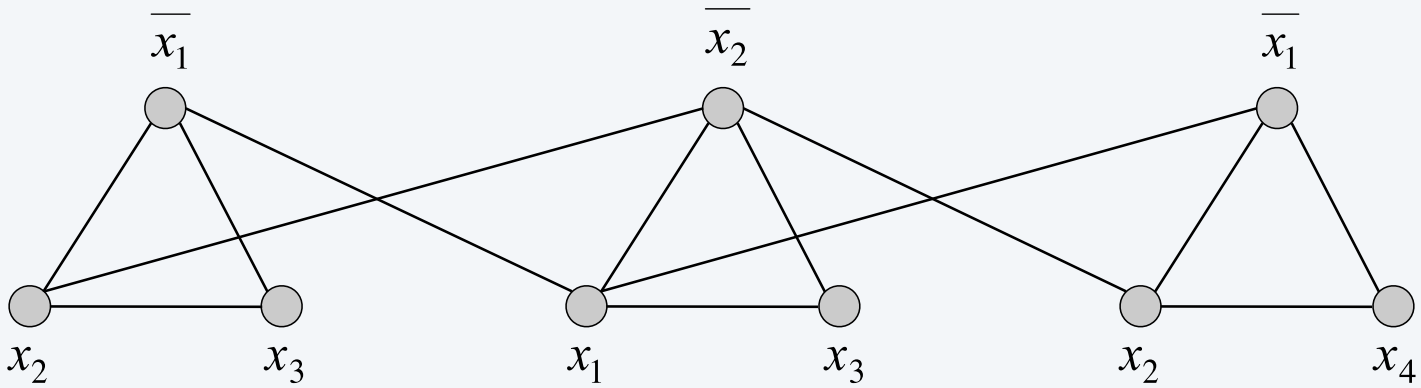
3-SAT se réduit à Independent Set

Théorème 47

3-SAT $\leq_P$ Independent-Set

Démonstration.

G



k = 3

$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4\right)$$

Lemme 48

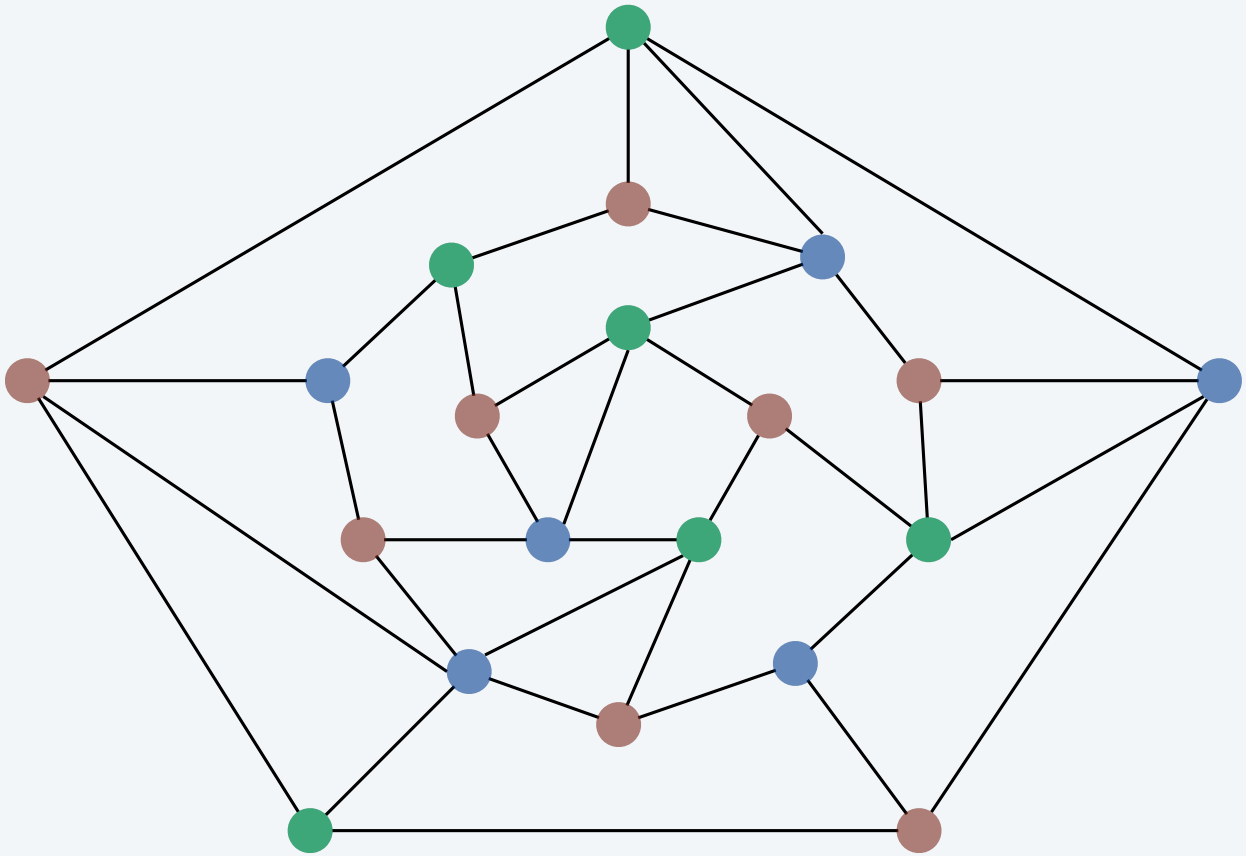
Soit k le nombre de clauses de Φ. Alors, G admet un Independent Set de taille k ssi Φ est satisfaisable.

□

Le cas des problèmes de type Graph coloring

3-coloration

3-Color Étant donné un graphe non-dirigé, peut-on colorier les sommets en rouge, vert et bleu de sorte que deux sommets adjacents aient des couleurs disjointes ?



yes instance

Application : allocation de registres

Allocation de registres (K-Register-Allocation) On se donne un entier k et un programme. Assigner les variables d'un programme à au plus k registres de sorte que deux variables utilisées au même moment ne soient pas assignées au même registre (p. ex. $print(x + y)$).

Graphe d'inférence Les sommets sont les variables du programme, et il y a un arc entre x et y ssi les variables x et y sont "vivantes" au même moment.

Observation [Chaitin 1982] On peut résoudre le problème d'allocation de registres ssi le graphe d'inference est k-coloriable.

REGISTER ALLOCATION & SPILLING VIA GRAPH COLORING

G. J. Chaitin
IBM Research
P.O.Box 218, Yorktown Heights, NY 10598

Fait 3-Color $\leq_P$ K-Register-Allocation pour toute constant $k \geq 3$.

Compte rendu

Stratégies de base de réduction

- Équivalences simples : Vertex-Cover $\equiv_P$ Independent-Set
- D'un cas particuler à un cas plus général : Vertex-Cover $\leq_P$ Set-Cover
- Encodage à l'aide de gadgets : 3-SAT $\leq_P$ Independent-Set

Lemme 49 (Transitivité de $\leq_P$)

Si $\mathcal{X} \leq_P \mathcal{Y}$ et $\mathcal{Y} \leq_P \mathcal{Z}$, alors $\mathcal{X} \leq_P \mathcal{Z}$.

Démonstration.

On compose les deux réductions □

Exemple

$$\text{3-SAT} \leq_P \text{Independent-Set} \leq_P \text{Vertex-Cover} \leq_P \text{Set-Cover}$$

Problèmes de recherche d'une solution

Problème de décision Existe-t-il un vertex cover de taille $\leq k$? (Vertex-Cover)

Problème de recherche d'une solution Trouver un vertex cover de taille $\leq k$ (Find-Vertex-Cover).

Exemple Pour trouver un vertex cover de taille $\leq k$:

1. Déterminer s'il existe un vertex cover de taille $\leq k$.
2. Si oui, trouver un sommet v tel que $G \setminus \{v\}$ a un vertex cover de taille $\leq k-1$ (récursivement).
   (tout sommet v d'un Vertex Cover de taille k est tel que $G \setminus \{v\}$ a un Vertex Cover de taille $\leq k-1$).
   Inclure v dans le Vertex Cover de taille $k-1$ de $G \setminus \{v\}$.
   Sinon répondre qu'il n'y a pas de solution.

Bilan Vertex-Cover et Find-Vertex-Cover ont la même complexité

Problèmes de recherche d'une solution

Problème de décision Existe-t-il un vertex cover de taille $\leq k$? (Vertex-Cover)

Problème de recherche d'une solution Trouver un vertex cover de taille $\leq k$ (Find-Vertex-Cover).

Problèmes d'optimisation Trouver un vertex cover de taille minimum (Minimum-Vertex-Cover).

Exemple Pour trouver un vertex cover de taille minimum :

- Recherche (dichotomique) pour la taille k^* du minimum vertex cover
- Résoudre le problème de recherche associé.

Bilan Vertex-Cover et Find-Vertex-Cover ont la même complexité, et qui est la même que celle de Minimum-Vertex-Cover

P vs. NP

Problèmes de décision

Définition 50

Soit $\mathcal{X}$ un problème de décision. On dit que l'algorithme $\mathfrak{A}$ résout le problème $\mathcal{X}$ lorsque

l'appel $\mathfrak{A}(\mathcal{I})$ retourne "oui" si $\mathcal{I} \in \mathcal{X}$, et "non" sinon.

Définition 51

L'algorithme $\mathfrak{A}$ s'exécute en temps polynomial s'il existe $p(.)$ un polynôme tel que pour toute instance $\mathcal{I}$, l'appel $\mathfrak{A}(\mathcal{I})$ termine en au plus $p(|\mathcal{I}|)$ "étapes".

Exemple 52

- Le problème $\textsc{Primes}= \{2, 3, 5, 7, 11, 13, 17, 23, 29, 31, 37, \ldots\}$
- Instance $\mathcal{I} = 592335744548702854681$.
- L'algorithme AKS résout $\textsc{Primes}$ en $O(|\mathcal{I}|^8)$ étapes.

Agrawal, Kayal, et Saxena ont eu le Gödel Prize 2006 et le Fulkerson Prize 2006 pour leur conception de l'algorithme AKS.
Voir par exemple `https://www.youtube.com/watch?v=D7AHbyAlgIA`.

La classe de complexité P (les problèmes faciles)

Définition 53

La classe P est la classe des problèmes de décision pour lesquels il existe un algorithme polynomial.

Problem	Description	Algorithm	yes	no
MULTIPLE	Is x a multiple of y ?	grade-school division	51, 17	51, 16
REL-PRIME	Are x and y relatively prime ?	Euclid (300 BCE)	34, 39	34, 51
PRIMES	Is x prime ?	AKS (2002)	53	51
EDIT-DISTANCE	Is the edit distance between x and y less than 5 ?	dynamic programming	niether neither	acgggt ttttta
L-SOLVE	Is there a vector x that satisfies $Ax = b$?	Gauss-Edmonds elimination	$\begin{bmatrix}0&1&1\\2&4&-2\\0&3&15\end{bmatrix}, \begin{bmatrix}4\\2\\36\end{bmatrix}$	$\begin{bmatrix}1&0&0\\1&1&1\\0&1&1\end{bmatrix}, \begin{bmatrix}1\\1\\1\end{bmatrix}$
ST-CONN	Is there a path between s and t in a graph G ?	depth-first search (Theseus)		

La classe de complexité NP (des problèmes moins faciles)

Pour la comprendre, nous allons introduire par des exemples les notions :

1. de certificat, et
2. de certifieur.

avec en filigrane des exigences de taille et d'exécution polynomiales.

Il faut pour cela être capable de s'imaginer un algorithme non-déterministe.

Si vous connaissez les automates d'états finis, vous avez appris qu'ils peuvent être non-déterministes, et que dans ce cas, un automate accepte un mot en entrée dès lors qu'il existe un calcul de l'automate sur ce mot qui se termine dans un état final.

C'est le même principe pour un algorithme non-déterministe : il "accepte" une instance $\mathcal{I}$ si un de ses calculs répond "oui", sinon l'algorithme "rejette" l'instance.

Un exemple d'algorithme non-déterministe pour résoudre SAT

(SAT)

Entrée : φ en logique propositionnelle
Sortie : φ est-elle satisfaisable ?

Algorithm 24 non-det$\mathrm{SAT}(\varphi)$

1: **Choisir** une valuation ν
2: ▷ se qui s'écrit comme une itération :
3: ▷ pour chaque $p \in Var(\varphi)$ faire $\nu(p) \leftarrow$ `true` **ou** $\nu(p) \leftarrow$ `false`
4: **if** $\nu(\varphi) =$ `true` **then**
5: **return** "Accept"
6: **else**
7: **return** "Reject"
8: **end if**

Il est clair que

$\varphi \in \mathrm{SAT}$ ssi il existe une exécution de cet algorithme sur l'entrée φ qui accepte.

Certificats et Certifieurs : cas du problème Composites (en anglais)

(Composites)

Entrée : Un entier n
Sortie : n est-il non-premier (on dit alors *composé*) ?

Pour répondre que n est composé, il faut trouver un diviseur p autre que 1 et n ; on dit que c'est un *facteur non-trivial de* n.
Cet entier p s'il existe est un *témoin/certificat* qui justifie de répondre **oui** pour l'instance n de Composites.

Certificats et Certifieurs : cas du problème Composites (en anglais)

(Composites)

Entrée : Un entier n
Sortie : n est-il non-premier (on dit alors *composé*) ?

Pour répondre que n est composé, il faut trouver un diviseur p autre que 1 et n ; on dit que c'est un *facteur non-trivial de* n.
Cet entier p s'il existe est un *témoin/certificat* qui justifie de répondre **oui** pour l'instance n de Composites.

Le principe consiste à engendrer un nombre (certificat) p de façon non-déterministe, puis de vérifier, avec un *certifier*, que p est bien un facteur non-trivial.

Certificats et Certifieurs : cas du problème Composites (en anglais)

(Composites)

Entrée : Un entier n
Sortie : n est-il non-premier (on dit alors *composé*) ?

Pour répondre que n est composé, il faut trouver un diviseur p autre que 1 et n ; on dit que c'est un *facteur non-trivial de* n.
Cet entier p s'il existe est un *témoin/certificat* qui justifie de répondre **oui** pour l'instance n de Composites.

Le principe consiste à engendrer un nombre (certificat) p de façon non-déterministe, puis de vérifier, avec un *certifier*, que p est bien un facteur non-trivial.

Donc pour le problème Composites, on a :

- Certificat : Un facteur non-trivial p de n existe ssi n est composé. De plus, $p \leq n$.
- Certifier = Vérifier que p divise n.

À vous de jouer !

Écrire l'algorithme (non-déterministe) qui en découle. Quelle est sa complexité ?

Certificats et Certifieurs

SAT Étant donnée une formule φ en FNC, a-t-elle un modèle ?

Certificat. Une valuation des n variables propositionnelles de φ.

Certifier. C'est vérifier que chaque clause de φ a au moins un littéral vrai.

Certificats et Certifieurs

SAT Étant donnée une formule φ en FNC, a-t-elle un modèle ?

Certificat. Une valuation des n variables propositionnelles de φ.

Certifier. C'est vérifier que chaque clause de φ a au moins un littéral vrai.

HAM-CYCLE Étant donné un graphe non-dirigé $G = (V, E)$, existe-t-il un cycle élémentaire dont les sommets forment V ?

Certificat. Une liste des n sommets du graphe.

Certifier. C'est vérifier que c'est une permutation (sauf pour le dernier qui est égal au premier) et que c'est une chemin, c-à-d. qu'il existe un arc entre deux sommets consécutifs de cette permutation.

Définition de la classe de complexité NP

Intuition. On considère un problème de décision $\mathcal{X}$.

- Un certifieur pour $\mathcal{X}$ est un algorithme $C(\mathcal{I}, c)$ où $\mathcal{I}$ est une instance de $\mathcal{X}$ et c est certificat (ou encore témoin) pour la question "$\mathcal{I} \in \mathcal{X}$?".
- Un certifieur $C(\mathcal{I}, c)$ ne détermine pas si $\mathcal{I} \in \mathcal{X}$ en soi, mais il utilise efficacement (c-à-d. en temps polynomial) le certificat c pour justifier que $\mathcal{I} \in \mathcal{X}$.

Définition 54

NP est l'ensemble des problèmes pour lesquels il existe un certifieur $C(\mathcal{I}, c)$ qui calcule en temps polynomial.

Puisque $C(\mathcal{I}, c)$ s'exécute en temps polynomial, le certificat c (ou ce qu'on en utilise) est de taille polynomial : $|c| \leq p(|\mathcal{I}|)$, où $p(.)$ est un polynôme.

Définition de la classe de complexité NP

Intuition. On considère un problème de décision $\mathcal{X}$.

- Un certifieur pour $\mathcal{X}$ est un algorithme $C(\mathcal{I}, c)$ où $\mathcal{I}$ est une instance de $\mathcal{X}$ et c est certificat (ou encore témoin) pour la question "$\mathcal{I} \in \mathcal{X}$?".
- Un certifieur $C(\mathcal{I}, c)$ ne détermine pas si $\mathcal{I} \in \mathcal{X}$ en soi, mais il utilise efficacement (c-à-d. en temps polynomial) le certificat c pour justifier que $\mathcal{I} \in \mathcal{X}$.

Définition 54

NP est l'ensemble des problèmes pour lesquels il existe un certifieur $C(\mathcal{I}, c)$ qui calcule en temps polynomial.

Puisque $C(\mathcal{I}, c)$ s'exécute en temps polynomial, le certificat c (ou ce qu'on en utilise) est de taille polynomial : $|c| \leq p(|\mathcal{I}|)$, où $p(.)$ est un polynôme. Étant donné $C(\mathcal{I}, c)$ pour $\mathcal{X}$, on exécute l'algorithme non-déterministe polynomial suivant :

1. On calcule/engendre de façon non-déterministe un certificat c (en temps polynomial puisque $|c| \leq p(|\mathcal{I}|)$) ;
2. On exécute l'algorithme $C(\mathcal{I}, c)$ (certifieur en temps polynomial) :
   - s'il retourne "oui" alors on retourne **accepte**,
   - s'il retourne "non" alors on retourne **rejette**.

Définition de NP

NP. Les problèmes de décision pour lesquels il existe un certifieur polynomial.

Problem	Description	Algorithm	yes	no
L-SOLVE	Is there a vector x that satisfies $Ax = b$?	Gauss-Edmonds elimination	$\begin{bmatrix} 0 & 1 & 1 \\ 2 & 4 & -2 \\ 0 & 3 & 15 \end{bmatrix}, \begin{bmatrix} 4 \\ 2 \\ 36 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$
COMPOSITES	Is x composite ?	AKS (2002)	51	53
FACTOR	Does x have a nontrivial factor less than y ?	?	(56159, 50)	(55687, 50)
SAT	Is there a truth assignment that satisfies the formula ?	?	$\neg x_1 \vee x_2$ $x_1 \vee x_2$	$\neg x_2$ $\neg x_1 \vee x_2$ $x_1 \vee x_2$
3-COLOR	Can the nodes of a graph G be colored with 3 colors?	?		
HAM-PATH	Is there a simple path between s and t that visits every node?	?		

Les classes de complexité P, NP, et EXPTIME

P : Les problèmes de décision pour lesquels il existe un algorithme polynomial en temps.
NP : Les problèmes de décision pour lesquels il existe un certifieur polynomial en temps.

Théorème 55

$\mathrm{P} \subseteq \mathrm{NP}$

Démonstration.

Soit $\mathcal{X} \in \mathrm{P}$.
Par définition, il existe une algorithme polynomial-time $\mathfrak{A}(.)$ qui résout $\mathcal{X}$.
On définit le certifieur $C(\mathcal{I}, c) = \mathfrak{A}(\mathcal{I})$ qui ignore le paramètre c. □

NP ⊆ EXPTIME

EXPTIME : Les problèmes de décision pour lesquels il existe un algorithme exponential en temps.

Théorème 56

NP ⊆ EXPTIME

Démonstration.

Soit un problème $\mathcal{X} \in \text{NP}$. Par définition, il existe une certifieur polynomial-time $C(\mathcal{I}, c)$ pour $\mathcal{X}$.
L'algorithme **déterministe** $\mathfrak{A}(\mathcal{I})$ en temps exponentiel pour résoudre le problème $\mathcal{X}$ parcours l'ensemble des certificats (qui sont en nombre exponenetiel) :

```
1: for each certificat c (où |c| ≤ p(|I|)) do
2:     if C(I, c) retourne "oui" then
3:         Return Accept
4:     end if
5: end for
6: Return Reject
```

La grande question : $\mathrm{P} = \mathrm{NP}$?

Q. Comment résout-on une instance de 3-SAT avec n variables ?

A. Par une recherche exhaustive : on essaie tous les 2^n valuations.

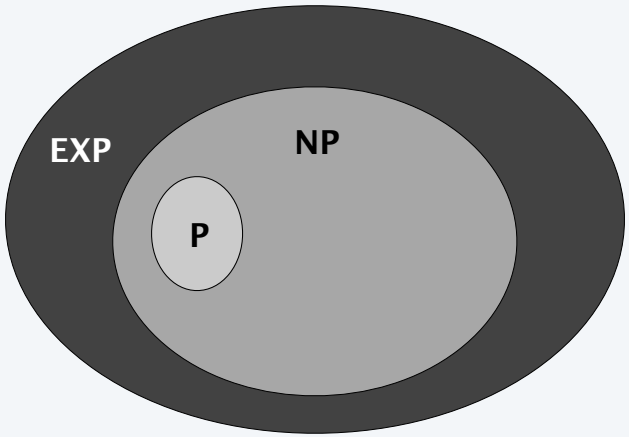
Q. Ne peut-on pas faire quelque chose de plus astucieux ?

Conjecture. *Il n'existe pas d'algorithme polynomial* pour 3-SAT.

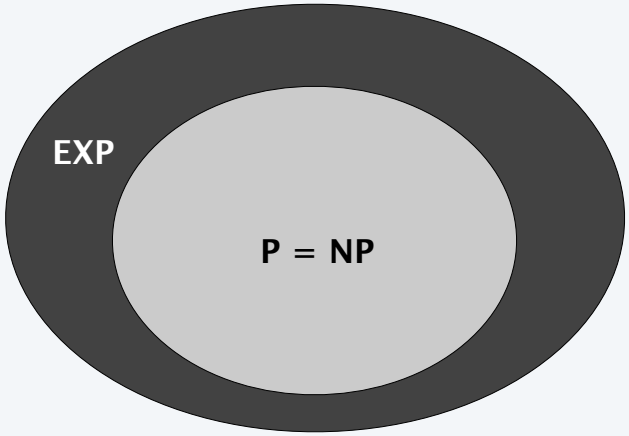
"intractable"

La grande question $\mathrm{P} = \mathrm{NP}$?

A-t-on $\mathrm{P} = \mathrm{NP}$? [Cook 1971]



If P ≠ NP



If P = NP

Si oui, alors nous avons des algorithmes efficaces pour 3-SAT, TSP, 3-Color, etc.
Si non, alors il n'existe pas d'algorithmes efficaces pour 3-SAT, TSP, 3-Color, etc.

Un consensus. Sans doute que la réponse est "non", c'est à dire $\mathrm{P} \neq \mathrm{NP}$.

NP-complétude

Polynomial transformation

Rappel

Définition 57 (Réduction "à la Karp")

Le problème $\mathcal{X}$ se réduit au problème $\mathcal{Y}$, noté $\mathcal{X} \leq_P \mathcal{Y}$, si il existe un polynôme $p(.)$ tel que pour toute instance arbitraire $\mathcal{I}$ du problème $\mathcal{X}$, on peut construire une instance $\text{transf}(\mathcal{I})$ du problème $\mathcal{Y}$ avec $|\text{transf}(\mathcal{I})| \leq p(|\mathcal{I}|)$ telle que

$$\mathcal{I} \in \mathcal{X} \text{ ssi } \text{transf}(\mathcal{I}) \in \mathcal{Y}$$

À vous de jouer !

Attention à la terminologie "se réduit", pourquoi ?

Polynomial transformation

Définition 57 (NP-complétude)

Un problème $\mathcal{Y}$ est NP-complet si
(1) $\mathcal{Y} \in \mathrm{NP}$, et
(2) pour tout problème $\mathcal{X} \in \mathrm{NP}$, on a $\mathcal{X} \leq_P \mathcal{Y}$.

Théorème 58

Soit $\mathcal{Y}$ un problème NP-complet. Alors, $\mathcal{Y} \in \mathrm{P}$ iff $\mathrm{P} = \mathrm{NP}$.

Démonstration.

$\Leftarrow$ Si $\mathrm{P} = \mathrm{NP}$, comme $\mathcal{Y} \in \mathrm{NP}$ alors $\mathcal{Y} \in \mathrm{P}$.
$\Rightarrow$ Supposons $\mathcal{Y} \in \mathrm{P}$.

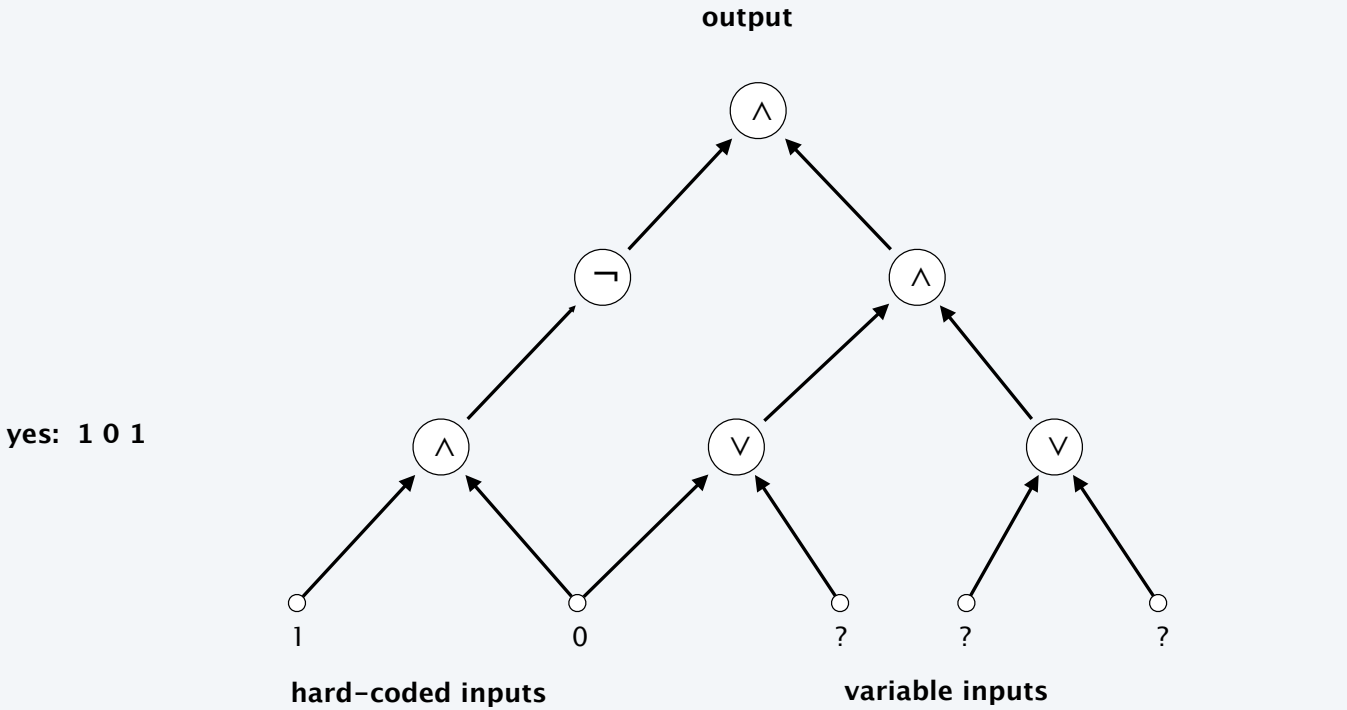
- Soit $\mathcal{X} \in \mathrm{NP}$. Puisque $\mathcal{X} \leq_P \mathcal{Y}$, on a $\mathcal{X} \in \mathrm{P}$
- Donc $\mathrm{NP} \subseteq \mathrm{P}$.
- Or on sait déjà que $\mathrm{P} \subseteq \mathrm{NP}$.

□

Question fondamentale. Existe-t-il des problèmes naturels NP-complets ?

Satisfaisabilité d'un circuit

CIRCUIT-SAT Étant donné un circuit combinatoire construit à partir de portes AND, OR et NOT, existe-t-il une valeur pour les entrées du circuit de sorte qu'il en resorte la valeur 1 ?



Théorème 59 (Cook 1971, Levin 1973)

CIRCUIT-SAT $\in$ NP-*complet.*

Admis.

Établir la NP-complétude d'un problème

Une fois qu'on a établi qu'un problème est NP-complet, on peut en récupérer plein d'autres.

Pour montrer que $\mathcal{Y} \in$ NP-complet.

1. On établit que $\mathcal{Y} \in \mathrm{NP}$.
2. On choisit un problème NP-complet $\mathcal{X}$.
3. On établit que $\mathcal{X} \leq_P \mathcal{Y}$.

Théorème 60

Si $\mathcal{X} \in$ NP-complet, $\mathcal{Y} \in \mathrm{NP}$, et $\mathcal{X} \leq_P \mathcal{Y}$, alors $\mathcal{Y} \in$ NP-complet.

Démonstration.

Soit un problème $\mathcal{Z} \in \mathrm{NP}$. Alors on a $\mathcal{Z} \leq_P \mathcal{X}$, et aussi $\mathcal{X} \leq_P \mathcal{Y}$.

- Par transitivité $\mathcal{Z} \leq_P \mathcal{Y}$
- Donc $\mathcal{Y} \in$ NP-complet.

□

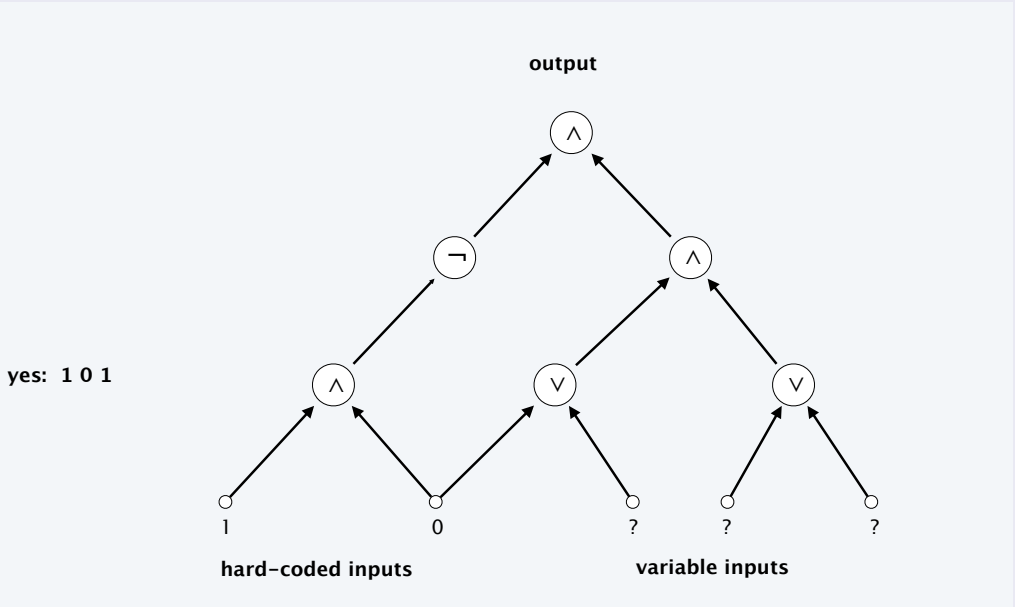
SAT est NP-complet

Théorème 61

SAT $\in$ NP-*complet.*

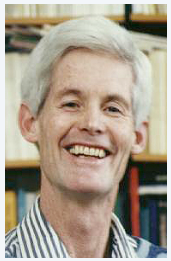
Démonstration.

Elle repose sur CIRCUIT-SAT $\leq_P$ SAT.



que l'on réduit à la satisfaisabilité de $(\neg(\texttt{true} \wedge \texttt{false})) \wedge ((\texttt{false} \vee p_1) \wedge (p_2 \vee p_3))$. □

Implications de Karp et Cook-Levin



Quelques problèmes NP-complets

- Packing + covering problems: Set-Cover, Vertex-Cover, Independent-Set.
- Constraint satisfaction problems: Circuit-Sat, Sat, 3-sat.
- Sequencing problems: Ham-Cycle, TSP.
- Partitioning problems: 3d-Matching, 3-Color.
- Numerical problems: Subset-Sum, Partition.

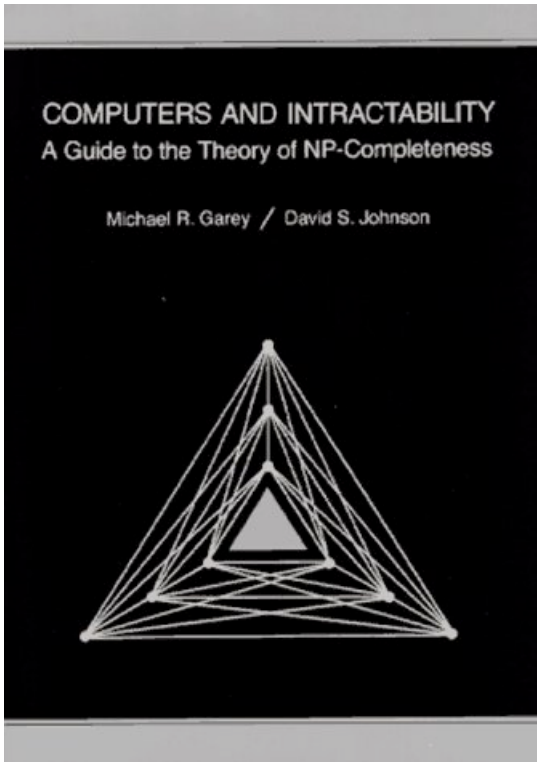
En pratique.
La plupart des problèmes NP sont connus pour être soit dans P, soit NP-complets.

Exceptions notables. Factor, Graph-Isomorphism, Nash Equilibrium.

Plus de problèmes NP-complets

Garey and Johnson

- Avec une annexe de plus de 300 problèmes NP-complets.
- La référence en Informatique la plus citée.



Plus de problèmes NP-complets

Aerospace engineering. Optimal mesh partitioning for finite elements.

Biology. Phylogeny reconstruction.

Chemical engineering. Heat exchanger network synthesis.

Chemistry. Protein folding.

Civil engineering. Equilibrium of urban traffic flow.

Economics. Computation of arbitrage in financial markets with friction.

Electrical engineering. VLSI layout.

Environmental engineering. Optimal placement of contaminant sensors.

Financial engineering. Minimum risk portfolio of given return.

Game theory. Nash equilibrium that maximizes social welfare.

Mathematics. Given integer a_1, ..., a_n, compute $\int_0^{2\pi} \cos(a_1\theta) \times \cos(a_2\theta) \times \cdots \times \cos(a_n\theta)\, d\theta$

Mechanical engineering. Structure of turbulence in sheared flows.

Medicine. Reconstructing 3d shape from biplane angiocardiogram.

Operations research. Traveling salesperson problem.

Physics. Partition function of 3d Ising model.

Politics. Shapley-Shubik voting power.

Recreation. Versions of Sudoko, Checkers, Minesweeper, Tetris.

Bibliographie I


G. Brassard and P. Bratley.
Algorithms : conception and analysis.
Masson editeur, 1987.


G. Brassard and P. Bratley.
Fundamentals of algorithmics, volume 524.
Prentice Hall New York, 1996.


D. Beauquier, J. Berstel, P. Chrétienne, et al.
Éléments d'algorithmique, volume 8.
Masson, 1992.


T.H. Cormen, C.E. Leiserson, and R.L. Rivest.
Introduction à l'algorithmique.
Dunod, 1996.


T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein.
Introduction to algorithms.
MIT press, 2001.

Bibliographie II


S. Dasgupta, C.H. Papadimitriou, and U. Vazirani.
Algorithms.
McGraw-Hill, Inc., 2006.


C. Froidevaux, M.C. Gaudel, and M. Soria.
Types de données et algorithmes.
Ediscience international, 1993.


D.C. Kozen.
The design and analysis of algorithms.
Springer, 1991.


J. Kleinberg and E. Tardos.
Algorithm Design.
Addison Wesley, 2005.