

# INF5130 Algorithmique

Coordinatrice : Louise Laforest  
Professeur : Samuele Giraudo

Département d'informatique

Hiver 2025

# 0. Présentation et informations générales

# Organisation

Cours	Lundi	13 h 30 - 16 h 30	PK-R310
Démonstrations	Mardi	13 h 30 - 16 h 30	PK-R310

Professeur : Samuele Giraudo, [giraudo.samuele@uqam.ca](mailto:giraudo.samuele@uqam.ca), bureau PK-4430

Auxiliaire de démonstration : Amanda Boatswain Jacques,  
[boatswain\\_jacques.amanda@courrier.uqam.ca](mailto:boatswain_jacques.amanda@courrier.uqam.ca)

Page du cours :

<https://igm.univ-mlv.fr/~giraudo/Teaching/INF5130/2025-01/INF5130.html>

## 0.1. Description et contenu

## Description et contenu

- ▶ Rappels sur la notation asymptotique.
- ▶ Complexité temporelle et spatiale, analyse probabiliste.
- ▶ Équations de récurrence et théorème fondamental.
- ▶ Algorithmes de force brute et voraces.
- ▶ Principe «diviser pour régner».
- ▶ Programmation dynamique.
- ▶ Algorithmes à retour arrière.
- ▶ Réduction de problèmes, NP-complétude.
- ▶ Introduction à la théorie de l'information.
- ▶ Entropie, information mutuelle et conditionnelle.
- ▶ Codes de longueur fixe/variable, théorème fondamental du codage de source.
- ▶ Détection et correction d'erreurs, distance de Hamming, codes linéaires.

## 0.2. Objectifs

# Objectifs

Ce cours est une initiation aux principes de base de la conception et de l'analyse des algorithmes séquentiels.

À la fin du cours, nous devrons être capables

- ▶ de connaître les algorithmes de base de l'informatique ;
- ▶ d'analyser la complexité et l'efficacité de différents types d'algorithmes ;
- ▶ de connaître les grands principes de la conception des algorithmes et de pouvoir les appliquer ;
- ▶ de comprendre la notion de problème NP-complet ;
- ▶ de comprendre les bases de la théorie de l'information.

### 0.3. Évaluation

# Évaluation

Description	Date	Pondération
Devoir 1	Lundi 17 février 2025 minuit	15 %
Examen 1	Lundi 24 février 2025	35 %
Devoir 2	Mardi 15 avril 2025 minuit	15 %
Examen 2	Mardi 22 avril 2025	35 %

- ▶ Examens
  - ▶ Individuels
  - ▶ Toute documentation papier est permise
  - ▶ Tout support électronique est interdit
  - ▶ Moyenne globale inférieure à 50 % aux examens = échec
- ▶ Devoirs :
  - ▶ Rédaction en `\LaTeX` obligatoire
  - ▶ Remise électronique sur Moodle (sources et pdf)
  - ▶ **Aucun** retard n'est autorisé.

## Autres informations importantes

- ▶ Intégrité académique (voir <http://r18.uqam.ca/> et <http://sciences.uqam.ca/etudiants/integrite-academique.html>)
- ▶ Absences aux examens (voir <http://info.uqam.ca/politiques/> et <https://info.uqam.ca/examendiffere>)
- ▶ Demande de modification de note (voir <https://etudier.uqam.ca/comment-faire-pour-demande-une-modification-notes>)

## 0.4. Références

## Références I

- ▶ Site web du cours sur moodle : <http://www.moodle.uqam.ca>
- ▶ Cormen, T., Leiserson, C., Rivest, R., Stein, C. – Algorithmique (3<sup>e</sup> édition) – Dunod (2010).
- ▶ Cormen, T. – Algorithmes - Notions de base – Dunod (2013).
- ▶ Neapolitan, R. et Naimipour, K. – Foundations of Algorithms Using Java Pseudocode – Jones and Bartlett Publishers, 2004.
- ▶ Weiss, M.A. – Data Structures and Algorithm Analysis in C++ (3<sup>e</sup> édition) – Addison Wesley, 2006.
- ▶ Levitin, A. – Introduction to The Design and Analysis of Algorithms (2<sup>e</sup> édition) – Addison Wesley, 2007.
- ▶ Aho, A.V., Hopcroft, J.E., Ullman, J.D. – Data Structures and Algorithms – Addison-Wesley, 1983.
- ▶ Aho, A.V., Ullman, J.D. – Foundations of Computer Science – Computer Science Press, 1992.

## Références II

- ▶ Baase, S. – Computer Algorithms : Introduction to the Design and Analysis of Algorithms – (3<sup>e</sup> édition), Addison-Wesley, 2000.
- ▶ Brassard, G., Bratley, P. – Fundamentals of Algorithmics – Prentice-Hall, 1996.
- ▶ Brassard, G., Bratley, P. – Algorithmique : conception et analyse – Masson, 1987.
- ▶ Goodrich, M.T. and Tamassia, T. – Data Structures and Algorithms in Java – John Wiley & Sons, 1998.
- ▶ Graham, R.L., Knuth, D.E., Patashnik, O. – Concrete Mathematics : a Foundation for Computer Science – Addison-Wesley, 1994.
- ▶ Harel, D. – Algorithmics, The Spirit of Computing – Addison-Wesley, 1987.
- ▶ Johnsonbaugh R. and Schaefer, M. – Algorithms – Pearson Education, 2004.
- ▶ Moret, B.M.E. – Towards a discipline of experimental algorithmics. In Proc. 5th DIMACS Challenge, volume DIMACS Monographs 59, pages 197-213 – American Mathematical Society, 2002.

## Références III

- ▶ Rosen, K.H. – Discrete Mathematics and its Applications – 1995 (version révisée en 1999).
- ▶ Sedgewick, R. – Algorithms (2<sup>e</sup> édition) – Addison-Wesley, 1988.

# 1. Problèmes et algorithmes

## 1.1. Introduction

# Introduction

- ▶ **Algorithme** : suite d'énoncés pour résoudre un problème. Se termine toujours.
  - ▶ Exemples : calcul du PGCD de deux entiers, recherche de la position d'un élément dans un tableau fini.
- ▶ **Processus** : suite d'énoncés pour résoudre un problème. Ne se termine pas forcément.
  - ▶ Exemples : calcul des antécédents de  $y \in Y$  relativement à une fonction  $f : X \rightarrow Y$  où  $X$  est infini, calcul de la suite de Syracuse d'un entier positif.
- ▶ **Programme** : traduction de l'algorithme en un langage compréhensible par l'ordinateur.

Qualités d'un bon algorithme :

- ▶ **correct** — résout le problème qu'il doit résoudre, pour toutes les données possibles
- ▶ **efficace** — consomme peu de ressources (temps, espace).

**Algorithme optimal** : est aussi efficace, sinon plus, que tous les autres algorithmes résolvant le même problème.

Un problème → un ou plusieurs algorithmes

- ▶ Entrée(s) : donnée(s) à fournir à l'algorithme
- ▶ Sortie(s) : résultat(s)

Déterminer :

- ▶ Ensemble des valeurs possibles
- ▶ Taille du problème

On évaluera les algorithmes en fonction de la taille du problème qu'ils résolvent.

## 1.2. Exemples de problèmes

## Exemples de problèmes

### ► Problème de la fouille → problème de décision

Soit  $x$  une valeur entière et  $T$  un tableau dont les indices sont compris entre 0 et  $n - 1$ . Existe-t-il un indice  $i$  compris entre 0 et  $n - 1$  tel que  $x = T[i]$  ?

#### ► Entrée

- ▶  $x$  : valeur entière
- ▶  $T$  : tableau indicé de 0 à  $n - 1$

#### ► Sortie

- ▶ booléen

- ▶ Taille du problème :  $n$  (taille du tableau)
- ▶ Exemplaire (instance) du problème :  $(7, [3, -7, 10, 67, 25, 100])$
- ▶  $P(7, [3, -7, 10, 67, 25, 100]) = \text{faux}$

► Problème de l'accessibilité → problème de décision

Soit  $G = (S, A)$  un graphe orienté. Soit  $s, t \in S$ . Le sommet  $t$  est-il accessible depuis le sommet  $s$  ?

► Entrée

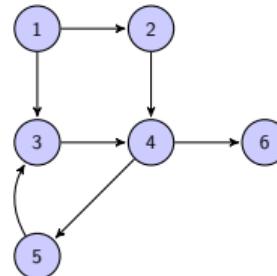
- $G$  : un graphe
- $s$  : un sommet de  $G$
- $t$  : un sommet de  $G$

► Sortie

- booléen

► Taille du problème : couple  $(|S|, |A|)$

► Exemplaire du problème : soit  $G$  le graphe orienté



Sa taille est  $(6, 7)$ .

Les tuples  $(G, 1, 6)$  et  $(G, 4, 1)$  sont des exemplaires de ce problème. La réponse au 1<sup>er</sup> est vrai et la réponse au 2<sup>e</sup> est faux.

## Problème de la satisfaisabilité (SAT) → problème de décision

**Définition :** Un littéral est une expression booléenne qui est soit une formule atomique (**V**, **F** ou une variable booléenne) ou la négation d'une formule atomique.

**Exemples :**  $A, \neg A, B, V, F$

**Définition :** Soit  $\phi$ , une expression booléenne.  $\phi$  est en forme normale conjonctive (FNC) si  $\phi$  est de la forme  $\bigwedge_{j=1}^m C_j$  où chaque  $C_j$  (clause de  $\phi$ ) est une disjonction de littéraux ( $C_j = \bigvee_{i=1}^n L_i$  où chaque  $L_i$  est un littéral).

**Exemple :**  $(B \vee \neg C) \wedge (\neg A \vee B \vee C) \wedge (A \vee \neg B \vee C) \wedge (\neg A \vee \neg B)$

**Problème 1 :** Soit  $\phi$  une expression booléenne en FNC. Existe-t-il une affectation aux variables qui rende  $\phi$  vraie ?

- ▶ Entrée
  - ▶  $\phi$  : une FNC
- ▶ Sortie
  - ▶ booléen
- ▶ Taille du problème :  $n$  le nombre d'occurrences de variables
- ▶ Exemplaire du problème : soit  $\phi$  l'expression  $(A \vee B) \wedge (\neg A \vee B \vee C)$ . Cette expression a pour taille 5. La réponse à  $\phi$  est vrai car l'affectation  $\{A \mapsto \text{vrai}, B \mapsto \text{faux}, C \mapsto \text{vrai}\}$  rend  $\phi$  vraie.

**Problème 2 (k-SAT) :** Soit  $\phi$  une expression booléenne en FNC où chaque clause contient  $k$  littéraux (avec un nombre quelconque de variables différentes).

Par exemple,  $(B \vee \neg C \vee D) \wedge (A \vee \neg B \vee \neg D)$  est une formule du problème 3-SAT. Existe-t-il une affectation aux variables qui rende  $\phi$  vraie ?

► Problème du flot maximal → problème d'optimisation (max)

**Définition** : Un réseau  $R = (S, A, s, t, c)$  est un graphe orienté où

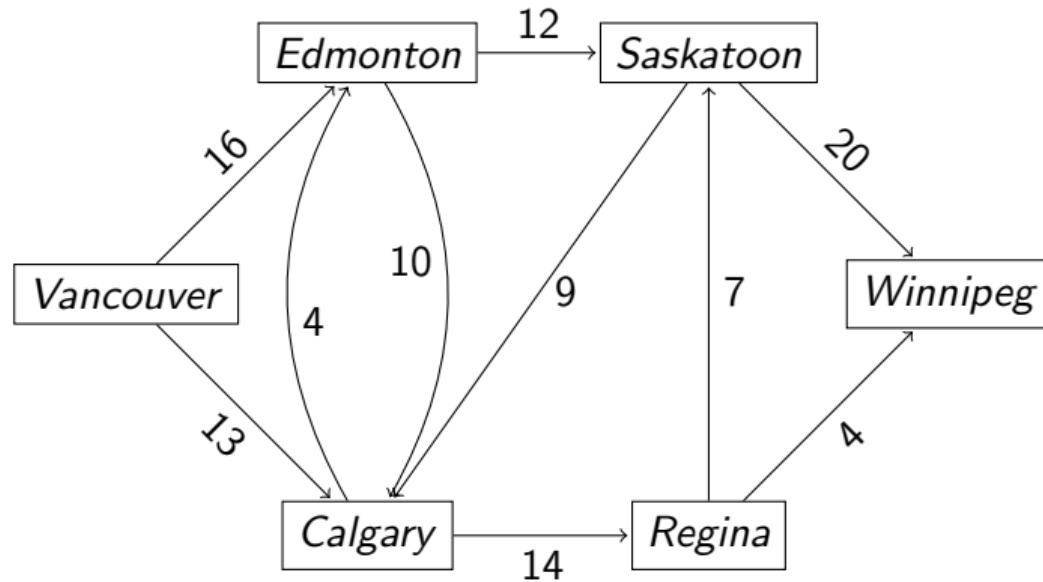
- ▶  $S$  : ensemble de sommets
- ▶  $A$  : ensemble d'arcs
- ▶  $s \in S$  : sommet source
- ▶  $t \in S$  : sommet puits
- ▶  $c$  : fonction capacité,  $c : A \rightarrow \mathbb{N}$

En sortie, on cherche

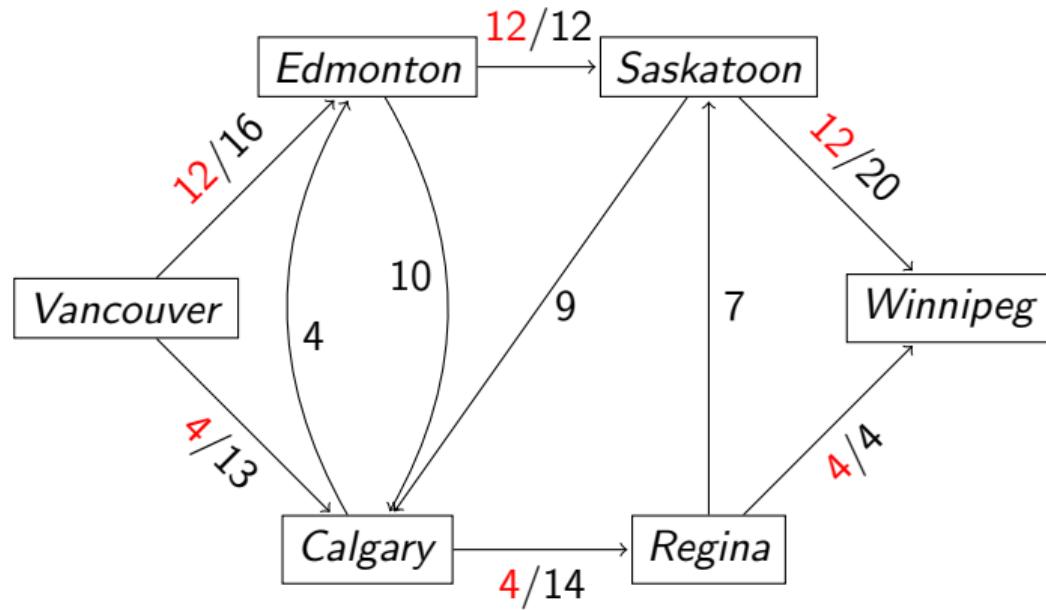
- ▶  $f$  : fonction flot,  $f : A \rightarrow \mathbb{N}$
- ▶  $\forall a \in A, f_a \leq c_a$ .
- ▶  $\forall i$  tel que  $i \neq s$  et  $i \neq t$ ,  $\sum_{(k,i) \in A} f_{(k,i)} = \sum_{(i,j) \in A} f_{(i,j)}$
- ▶ Valeur du flot :  $\sum_{(s,j) \in A} f_{(s,j)} = \sum_{(i,t) \in A} f_{(i,t)}$

**Problème** : étant donné un réseau  $R$ , quelle est la valeur maximale d'un flot dans  $R$ ? On cherche une fonction flot  $f$  telle que la valeur du flot est maximale. Il s'agit du flot entrant sur le puits (ou, de manière équivalente, du flot sortant de la source).

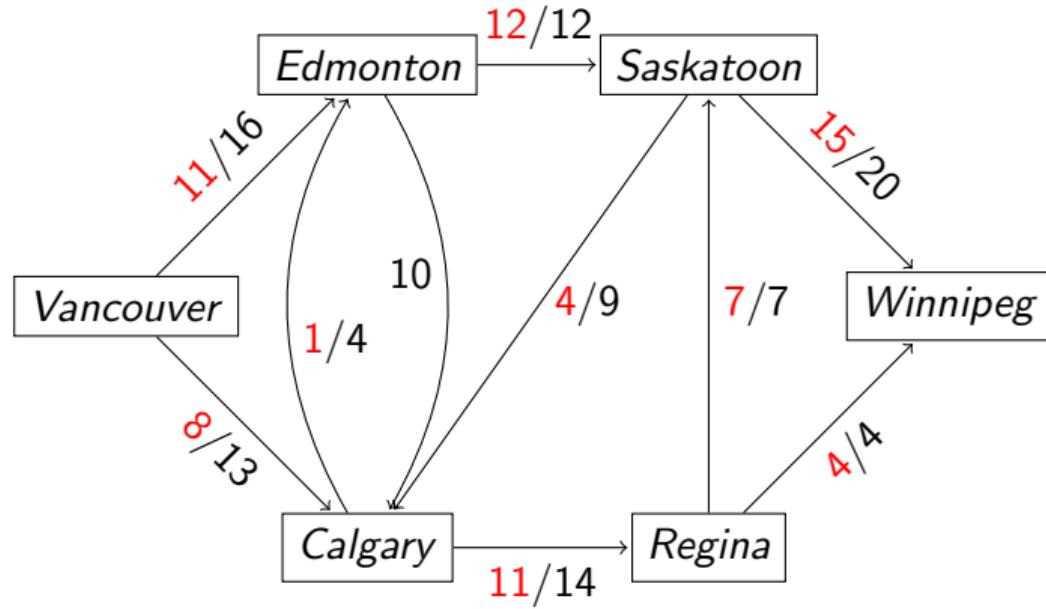
**Exemple** : la capacité est le nombre maximal de boîtes pouvant être livrées par jour.



**Question** : quel est le nombre maximal de boîtes qui peuvent être livrées à Winnipeg par jour ?



Valeur du **flot** = 16



Valeur du **flot** = 19

Le flot maximal est 23.

► Problème du commis voyageur → problème d'optimisation (min)

**Définition** : une permutation  $\pi$  de  $\{1, 2, \dots, n\}$  (notée  $[n]$ ) est une fonction bijective de  $[n]$  vers  $[n]$ .

**Exemple :**

$i$	1	2	3	4	5	6	7	8	9
$\pi(i)$	2	9	3	6	4	8	7	1	5

**Soient :**

- $V = \{1, 2, \dots, n\}$ , un ensemble de villes
- $d : V \times V \rightarrow \mathbb{N}$  une fonction distance (où  $d_{i,j}$  est la distance entre les villes  $i$  et  $j$ )

**Définition** : une tournée est une permutation de  $[n]$  notée  $\pi$ . La longueur de la tournée est la quantité  $\sum_{i=1}^n d_{\pi(i), \pi(i+1)}$  où  $\pi(1) = \pi(n + 1)$ .

**Problème** : soient  $V$  et  $d$ , quelle est la longueur minimale d'une tournée ?

► **Problème de la sélection**

- $T$  : Tableau de  $n$  éléments distincts.
- $i$  : Indice entre 1 et  $n$  inclusivement.

► **Problème :** Trouver une valeur  $x$  telle que

- $x \in T$ ,
- $i - 1$  éléments de  $T$  sont strictement inférieurs à  $x$ ,
- $n - i$  éléments de  $T$  sont strictement supérieurs à  $x$ .

**Exemple :**

- Soit  $T = [23, 12, 9, 5, 17, 45, 3, 29, 21, 37]$  et  $i = 6$ .
- $x = 21$  convient car
- $i - 1 = 5$  éléments ( $3, 5, 9, 12, 17$ ) sont strictement inférieurs à 21.
- $n - i = 4$  éléments ( $23, 29, 37, 45$ ) sont strictement supérieurs à 21.

## ► Problème du tri

- entrée :  $T$  : tableau de  $n$  éléments indicés de 1 à  $n$
- sortie :  $T'$  : éléments de  $T$  tels que  $T'[j] = T[\pi(j)]$  pour tout  $1 \leq j \leq n$  où  $\pi$  est une permutation et  $T'[j] \leq T'[j + 1]$  pour tout  $1 \leq j \leq n - 1$

## ► Exemple

$i$	1	2	3	4	5	6	7	8	9
$T$	18	3	7	12	23	11	17	15	3
$T'$	3	3	7	11	12	15	17	18	23
$\pi(i)$	2	9	3	6	4	8	7	1	5

## ► Problème de l'arrêt

- entrée :
  - $M$  : un programme
  - $\omega$  : une chaîne de caractères (qui est une entrée pour le programme  $M$ )
- sortie : **faux** si l'exécution de  $M$  avec l'entrée  $\omega$  provoque une boucle infinie, **vrai** dans le cas contraire.

Ce problème n'est pas résolvable par un algorithme → indécidable

## 2. Complexité temporelle des algorithmes

## 2.1. Introduction

# Introduction

**Définition :** La complexité d'un algorithme est le comportement de celui-ci pour de grandes valeurs de  $n$  (taille du problème).

- ▶ Complexité **spatiale** → espace mémoire utilisé
- ▶ Complexité **temporelle** → temps d'exécution

Le temps d'exécution dépend

- ▶ de la plate-forme (CPU)
- ▶ du langage de programmation
- ▶ du compilateur
- ▶ du programmeur
- ▶ des algorithmes utilisés

## 2.2. Complexité temporelle

## Complexité temporelle

- ▶ La complexité temporelle est indépendante des facteurs externes (CPU, langage de programmation, compilateur, programmeur)
- ▶ La complexité temporelle peut être étudiée
  - ▶ Dans le pire cas
  - ▶ Dans le cas moyen
  - ▶ Dans le meilleur cas

Tout langage de programmation possède des

1. opérations élémentaires

- ▶ arithmétiques : +, −, ...
- ▶ logiques : ∨, ∙, ...
- ▶ de comparaison : <, ≤, >, ...

2. énoncés élémentaires

- ▶ affectation : ←
- ▶ de branchement (appel de sous-programme) ...

3. énoncés structurés

- ▶ **tant que**
- ▶ **pour**
- ▶ **si**

On suppose que 1. et 2. consomment chacune 1 unité de temps à laquelle s'ajoute le temps d'évaluation des opérandes.

# Règles générales

## Notation :

- ▶  $T(expression)$  : temps pour évaluer l'expression
- ▶  $T(énoncés)$  : temps pour évaluer les énoncés

## Exemples :

- ▶  $T(x + 4) = 1$
- ▶  $T(y \leftarrow 2x) = 2$
- ▶  $T(x \leftarrow 0; \text{Lire } y; ) = 1 + T(\text{Lire } y)$
- ▶  $T(x = 0 \wedge y/x > 12) = 4$

Énoncé si

si *<condition>* alors

*< se<sub>1</sub>>*

sinon

*< se<sub>2</sub>>*

fin si

- ▶ Temps dans le pire cas :

$$T(<\text{condition}>) + \max\{T(<\text{se}_1>), T(<\text{se}_2>)\}$$

- ▶ Temps dans le meilleur cas :

$$T(<\text{condition}>) + \min\{T(<\text{se}_1>), T(<\text{se}_2>)\}$$

- ▶ Temps dans le cas moyen : La formule dépend de chaque problème et  
**n'est pas nécessairement**

$$T(<\text{condition}>) + \frac{1}{2}(T(<\text{se}_1>) + T(<\text{se}_2>))$$

## Énoncé tant que

tant que *<condition>* faire

*<se>*

fin tant que

- ▶  $N_{max}$  : nombre maximal d'itérations
- ▶  $N_{min}$  : nombre minimal d'itérations (pas nécessairement 0)
- ▶  $N_{moy}$  : nombre moyen d'itérations
- ▶ La condition est  $k$  fois vraie, 1 fois fausse,  $k \geq 0$
- ▶ Temps dans le pire cas :

$$(N_{max} + 1)T(<\text{condition}>) + N_{max}T(<\text{se}>)$$

- ▶ Temps dans le meilleur cas :

$$(N_{min} + 1)T(<\text{condition}>) + N_{min}T(<\text{se}>)$$

- ▶ Temps dans le cas moyen :

$$(N_{moy} + 1)T(<\text{condition}>) + N_{moy}T(<\text{se}>)$$

## Énoncé pour

**pour** *variable* = <val. init.> à <val. fin.> **faire**

< se >

**pour**

▶  $N$  : nombre d'itérations = <val. fin.> – <val. init.> + 1

▶ À chaque itération :

- ▶ mise à jour de la variable de contrôle
- ▶ comparaison avec la valeur finale

▶ Temps dans tous les cas :

$$(N + 1)T(\text{m.à.j. et comparaison}) + NT(< \text{se} >)$$

## Exemples

- Tri par sélection (extraction)

$\leq$	trié absolu						
1	$i - 1$	$i$	$n$				

5	18	3	2	15	7	11	10
5	<b>18</b>	3	2	15	7	11	10
5	<b>10</b>	3	2	15	7	11	<b>18</b>
5	10	3	2	<b>15</b>	7	11	<b>18</b>
5	10	3	2	<b>11</b>	7	<b>15</b>	<b>18</b>
5	10	3	2	<b>11</b>	7	<b>15</b>	<b>18</b>
5	10	3	2	<b>7</b>	<b>11</b>	<b>15</b>	<b>18</b>
5	<b>10</b>	3	2	<b>7</b>	<b>11</b>	<b>15</b>	<b>18</b>
5	<b>7</b>	3	2	<b>10</b>	<b>11</b>	<b>15</b>	<b>18</b>

**fonction** indiceDuPlusGrand ( *i*, *tab* ) **renvoie** entier

entrée :

*i* : indice entre 1 et *n*

*tab* : tableau indicé de 1 à *n* incl.

sortie :

indice où se trouve le plus grand élément dans *tab*[1..*i*]

début

1      *jMax*  $\leftarrow$  1

1 fois

2      pour *j*  $\leftarrow$  2 haut *i* faire

(*i* - 1) + 1 fois

3      si *tab*[*j*] > *tab*[*jMax*] alors

(*i* - 1) fois

4          *jMax*  $\leftarrow$  *j*

$\ell$  fois,  $0 \leq \ell \leq i - 1$

5          fin si

6      fin pour

7      renvoyer *jMax*

1 fois

fin indiceDuPlusGrand

temps d'exécution *c*<sub>1</sub> *c*<sub>2</sub> *c*<sub>3</sub> *c*<sub>4</sub>

## Analyse

$$T(\text{indiceDuPlusGrand}(i)) = c_1 + ic_2 + c_3(i - 1) + c_4\ell$$

- Pire cas :  $\ell = i - 1$

$$\begin{aligned} T(\text{indiceDuPlusGrand}(i)) &= c_1 + ic_2 + c_3(i - 1) + c_4(i - 1) \\ &= c_1 + ic_2 + (c_3 + c_4)(i - 1) \end{aligned}$$

- Meilleur cas :  $\ell = 0$

$$\begin{aligned} T(\text{indiceDuPlusGrand}(i)) &= c_1 + ic_2 + c_3(i - 1) + c_30 \\ &= c_1 + ic_2 + c_3(i - 1) \end{aligned}$$

```
procedure triSelection ( tab )
```

entrée :

tab : tableau indicé de 1 à n incl.

sortie :

tab : tableau indicé de 1 à n incl. trié

début

1      **pour**  $i \leftarrow n$  **bas 2 faire**

$(n - 1) + 1$  fois

2       $iMax \leftarrow \text{indiceDuPlusGrand}(i, tab)$

$n - 1$  fois

3      **si**  $i \neq iMax$  **alors**

$n - 1$  fois

4       $tab[i] \leftrightarrow tab[iMax]$

$p$  fois,  $0 \leq p \leq n - 1$

5      **fin si**

6       $\{ \text{Invariant} : tab[k] \leq tab[i] \text{ pour tout } 1 \leq k < i \text{ et } tab[i..n] \text{ trié} \}$

7      **fin pour**

8      **fin triSelection**

temps d'exécution  $c_5 \ c_6 \ c_7 \ c_8$

## Analyse — meilleur cas

$$T(\text{triSelection}) = nc_5 + (n-1)(c_6 + c_7) + pc_8 + \sum_{i=2}^n T(\text{iDPG}(i)), \quad p = 0$$

Simplifions le terme  $\sum_{i=2}^n T(\text{iDPG}(i))$  de cette somme. Dans le meilleur cas,

$$\begin{aligned}\sum_{i=2}^n T(\text{iDPG}(i)) &= \sum_{i=2}^n (c_1 + ic_2 + (i-1)c_3) \\&= \sum_{i=2}^n c_1 + \sum_{i=2}^n ic_2 + \sum_{i=2}^n (i-1)c_3 \\&= (n-2+1)c_1 + c_2 \sum_{i=2}^n i + c_3 \sum_{i=2}^n (i-1) \\&= (n-1)c_1 + c_2 \left( \frac{n(n+1)}{2} - 1 \right) + c_3 \left( \frac{n(n-1)}{2} \right) \\&= n^2 \left( \frac{c_2 + c_3}{2} \right) + n \left( \frac{2c_1 + c_2 - c_3}{2} \right) - c_1 - c_2.\end{aligned}$$

## Analyse — meilleur cas

Ainsi, dans le meilleur cas,

$$\begin{aligned} T(\text{triSelection}) &= nc_5 + (n - 1)(c_6 + c_7) + \cancel{0}c_8 + \sum_{i=2}^n T(\text{iDPG}(i)) \\ &= nc_5 + (n - 1)(c_6 + c_7) + n^2 \left( \frac{c_2 + c_3}{2} \right) + n \left( \frac{2c_1 + c_2 - c_3}{2} \right) - c_1 - c_2 \\ &= n^2 \left( \frac{c_2 + c_3}{2} \right) + n \left( \frac{2c_1 + c_2 - c_3 + 2c_5 + 2c_6 + 2c_7}{2} \right) - c_1 - c_2 - c_6 - c_7. \end{aligned}$$

## Analyse — pire cas

$$T(\text{triSelection}) = nc_5 + (n-1)(c_6 + c_7) + pc_8 + \sum_{i=2}^n T(\text{iDPG}(i)), \quad p = n - 1$$

Simplifions le terme  $\sum_{i=2}^n T(\text{iDPG}(i))$  de cette somme. Dans le pire cas,

$$\begin{aligned}\sum_{i=2}^n T(\text{iDPG}(i)) &= \sum_{i=2}^n (c_1 + ic_2 + (i-1)(c_3 + c_4)) \\&= \sum_{i=2}^n c_1 + \sum_{i=2}^n ic_2 + \sum_{i=2}^n (i-1)(c_3 + c_4) \\&= (n-2+1)c_1 + c_2 \sum_{i=2}^n i + (c_3 + c_4) \sum_{i=2}^n (i-1) \\&= (n-1)c_1 + c_2 \left( \frac{n(n+1)}{2} - 1 \right) + (c_3 + c_4) \left( \frac{n(n-1)}{2} \right) \\&= n^2 \left( \frac{c_2 + c_3 + c_4}{2} \right) + n \left( \frac{2c_1 + c_2 - c_3 - c_4}{2} \right) - c_1 - c_2.\end{aligned}$$

## Analyse — pire cas

Ainsi, dans le pire cas,

$$\begin{aligned} T(\text{triSelection}) &= nc_5 + (n - 1)(c_6 + c_7) + (n - 1)c_8 + \sum_{i=2}^n T(\text{iDPG}(i)) \\ &= nc_5 + (n - 1)(c_6 + c_7) + (n - 1)c_8 \\ &\quad + n^2 \left( \frac{c_2 + c_3 + c_4}{2} \right) + n \left( \frac{2c_1 + c_2 - c_3 - c_4}{2} \right) - c_1 - c_2 \\ &= n^2 \left( \frac{c_2 + c_3 + c_4}{2} \right) + n \left( \frac{2c_1 + c_2 - c_3 - c_4 + 2c_5 + 2c_6 + 2c_7 + 2c_8}{2} \right) \\ &\quad - c_1 - c_2 - c_6 - c_7 - c_8 \end{aligned}$$

	trié relatif				ordre quelconque				
► Tri par insertion	1		$i$	$i + 1$					$n$
	5	18	3	2	15	7	11	10	
	5	18	3	2	15	7	11	10	
	3	5	18	2	15	7	11	10	
	2	3	5	18	15	7	11	10	
	2	3	5	15	18	7	11	10	
	2	3	5	7	15	18	11	10	
	2	3	5	7	11	15	18	10	
	2	3	5	7	10	11	15	18	

**procedure** triInsertion ( *tab* )

entrée :

*tab* : tableau indicé de 1 à *n* incl.

sortie :

*tab* : tableau indicé de 1 à *n* incl., trié

début

1	<b>pour</b> <i>i</i> $\leftarrow 2$ <b>haut</b> <i>n</i> <b>faire</b>	$(n - 1) + 1$ fois
2	<i>x</i> $\leftarrow \text{tab}[i]$	<i>n</i> – 1 fois
3	<i>j</i> $\leftarrow i$	<i>n</i> – 1 fois
4	<b>tant que</b> ( <i>j</i> $\geq 2$ ) et <i>tab</i> [ <i>j</i> – 1] > <i>x</i> <b>faire</b>	$1 \leq q \leq i$ fois
5	<i>tab</i> [ <i>j</i> ] $\leftarrow \text{tab}[j - 1]$	<i>q</i> – 1 fois
6	<i>j</i> $\leftarrow j - 1$	<i>q</i> – 1 fois
7	<b>fin tant que</b>	
8	<i>tab</i> [ <i>j</i> ] $\leftarrow x$	<i>n</i> – 1 fois
9	<b>fin pour</b>	
	<b>fin triInsertion</b>	temps d'exécution <i>c</i> <sub>1</sub> <i>c</i> <sub>2</sub> <i>c</i> <sub>3</sub> <i>c</i> <sub>4</sub>

## Analyse

$$T(\text{triInsertion}) = c_1 + (n - 1)(c_1 + c_2) + \sum_{i=2}^n qc_3 + \sum_{i=2}^n (q - 1)c_4$$

► Pire cas :  $q = i$

$$\begin{aligned} &= c_1 + (n - 1)(c_1 + c_2) + \sum_{i=2}^n ic_3 + \sum_{i=2}^n (i - 1)c_4 \\ &= c_1 + (n - 1)(c_1 + c_2) + c_3\left(\frac{n(n + 1)}{2} - 1\right) + c_4\frac{n(n - 1)}{2} \\ &= n^2\left(\frac{c_3 + c_4}{2}\right) + n\left(c_1 + c_2 + \frac{c_3 - c_4}{2}\right) - (c_2 + c_3) \end{aligned}$$

## Analyse

$$T(\text{triInsertion}) = c_1 + (n - 1)(c_1 + c_2) + \sum_{i=2}^n qc_3 + \sum_{i=2}^n (q - 1)c_4$$

- ▶ Meilleur cas :  $q = 1$

$$\begin{aligned} &= c_1 + (n - 1)(c_1 + c_2) + \sum_{i=2}^n c_3 \\ &= n(c_1 + c_2 + c_3) - (c_2 + c_3) \end{aligned}$$

## ► Pire cas

---

$$\text{Sélection : } n^2 \left( \frac{c_2 + c_3 + c_4}{2} \right) + n \left( \frac{2c_1 + c_2 - c_3 - c_4 + 2c_5 + 2c_6 + 2c_7 + 2c_8}{2} \right) - c_1 - c_2 - c_6 - c_7 - c_8$$

$$\text{Insertion : } n^2 \left( \frac{c_3 + c_4}{2} \right) + n \left( c_1 + c_2 + \frac{c_3 - c_4}{2} \right) - (c_2 + c_3)$$

---

## ► Meilleur cas

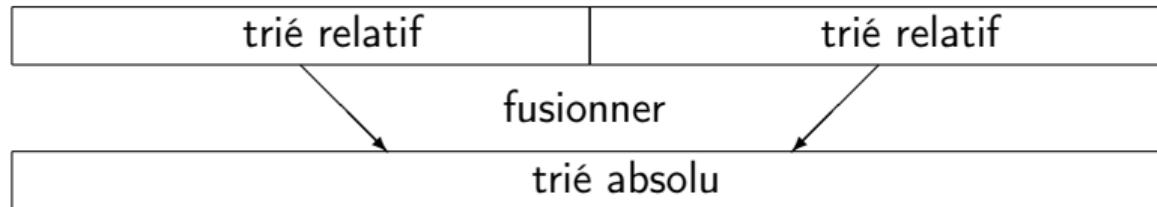
---

$$\text{Sélection : } n^2 \left( \frac{c_2 + c_3}{2} \right) + n \left( \frac{2c_1 + c_2 - c_3 + 2c_5 + 2c_6 + 2c_7}{2} \right) - c_1 - c_2 - c_6 - c_7$$

$$\text{Insertion : } n(c_1 + c_2 + c_3) - (c_2 + c_3)$$

---

► Tri fusion (*merge sort*)



Exemple :

1 4 6 10 11 18 29 40

2 2 6 7 12 15 28 39

1 2 2 4 6 6 7 10 11 12 15 18 28 29 39 40

**procedure** triFusion ( *tab, bi, bs* )

entrée :

*tab* : tableau indicé de 1 à *n* incl.

*bi, bs* : indices entre 1 et *n*

sortie :

*tab* : tableau indicé de 1 à *n* incl., trié

début

1	<b>si</b> <i>bi &lt; bs</i> <b>alors</b>	<b>1 fois</b>
2	<i>milieu</i> $\leftarrow$ ( <i>bi + bs</i> ) / 2	<b>1 fois</b>
3	triFusion ( <i>tab, bi, milieu</i> )	
4	triFusion ( <i>tab, milieu + 1, bs</i> )	
5	fusionner ( <i>tab, bi, milieu, bs, tabTemp</i> )	
6	<b>pour</b> <i>i</i> $\leftarrow$ <i>bi</i> <b>haut</b> <i>bs</i> <b>faire</b>	<i>bs – bi + 2</i> fois
7	<i>tab[i]</i> $\leftarrow$ <i>tabTemp[i]</i>	<i>bs – bi + 1</i> fois
	<b>fin pour</b>	
	<b>fin si</b>	
	<b>fin triFusion</b>	temps d'exécution <i>c<sub>1</sub> c<sub>2</sub> c<sub>3</sub></i>

**procedure** fusionner ( *tab, debut, fin<sub>1</sub>, fin<sub>2</sub>, temp* )

entrée :

*tab* : tableau indicé de 1 à *n* incl.

*debut, fin<sub>1</sub>, fin<sub>2</sub>* : indices entre 1 et *n*

sortie :

*temp* : tableau indicé de 1 à *n* incl.,

fusion de *tab[debut..fin<sub>1</sub>]* et *tab[fin<sub>1</sub> + 1..fin<sub>2</sub>]*

début

```
1   i  $\leftarrow$  debut ; j  $\leftarrow$  fin1 + 1           1 fois
2   pour k  $\leftarrow$  debut haut fin2 faire      fin2 – debut + 2 fois
3     si i  $\leq$  fin1 et (j > fin2 ou tab[i] < tab[j]) alors
4       temp[k]  $\leftarrow$  tab[i] ; i  $\leftarrow$  i + 1      fin2 – debut + 1 fois
5     sinon
6       temp[k]  $\leftarrow$  tab[j] ; j  $\leftarrow$  j + 1
7     fin si
8   fin pour
9 fin fusionner
```

temps d'exécution *c<sub>4</sub> c<sub>5</sub> c<sub>6</sub>*

## Analyse

$$T(\text{fusionner}) = c_4 + (fin_2 - debut + 2)c_5 + (fin_2 - debut + 1)c_6$$

► Pire cas et meilleur cas :

$$= c_4 + c_5 + (fin_2 - debut + 1)(c_5 + c_6)$$

Si  $debut = 1$  et  $fin_2 = n$  :

$$= n(c_5 + c_6) + c_4 + c_5$$

$$\begin{aligned}
 T(\text{triFusion}, bi, bs) = & c_1 + (bs - bi + 2)c_2 + (bs - bi + 1)c_3 \\
 & + T(\text{triFusion}, bi, (bi + bs)/2) + T(\text{triFusion}, (bi + bs)/2 + 1, bs) \\
 & + T(\text{fusionner}(tab, bi, (bi + bs)/2, bs, tabTemp))
 \end{aligned}$$

► Pire cas et meilleur cas :

$$\begin{aligned}
 &= c_1 + (bs - bi + 2)c_2 + (bs - bi + 1)c_3 + \\
 &\quad T(\text{triFusion}, bi, (bi + bs)/2) + T(\text{triFusion}, (bi + bs)/2 + 1, bs) + \\
 &\quad c_4 + c_5 + (bs - bi + 1)(c_5 + c_6)
 \end{aligned}$$

Si  $bi = 1$  et  $bs = n$  et  $n$  est pair :

$$\begin{aligned}
 &= c_1 + (n + 1)c_2 + nc_3 + 2T(\text{triFusion}, 1, n/2) + c_4 + c_5 + n(c_5 + c_6) \\
 &= 2T(\text{triFusion}, 1, n/2) + n(c_2 + c_3 + c_5 + c_6) + c_1 + c_2 + c_4 + c_5
 \end{aligned}$$

Notation plus simple :

$$T(n) = 2T(n/2) + n(c_2 + c_3 + c_5 + c_6) + c_1 + c_2 + c_4 + c_5$$

Le résolution de ce type d'**équations de récurrence** sera étudiée au chapitre 4.

## Opération ou instruction barométrique

la ligne exécuté la plus souvent ou celle la plus dispendieuse en terme de temps

**Opération barométrique** : opération ou instruction qui est exécutée au moins aussi souvent que n'importe quelle autre instruction ou opération de l'algorithme.

**Avantage** : simplifie l'analyse asymptotique d'un algorithme en laissant tomber les détails négligeables.

## Exemples

**fonction** indiceDuPlusGrand ( *i, tab* ) **renvoie** entier

entrée :

*i* : indice entre 1 et *n*

*tab* : tableau indicé de 1 à *n* incl.

sortie :

indice où se trouve le plus grand élément dans *tab[1..i]*

**début**

```
1      jMax ← 1
2      pour j ← 2 haut i faire
3          si tab[j] > tab[jMax] alors
4              jMax ← j
5          fin si
6      fin pour
7      renvoyer jMax
fin indiceDuPlusGrand
```

$$T(\text{iDPG}(i)) = i - 1$$

**procedure** triSelection ( *tab* )

entrée :

*tab* : tableau indicé de 1 à *n* incl.

sortie :

*tab* : tableau indicé de 1 à *n* incl. trié

**début**

1      **pour** *i*  $\leftarrow n$  bas 2 faire

2            *iMax*  $\leftarrow$  indiceDuPlusGrand(*i*, *tab*)

3            **si** *i*  $\neq$  *iMax* alors

4                *tab*[*i*]  $\leftrightarrow$  *tab*[*iMax*]

5                **fin si**

6                *{Invariant : tab*[*k*]  $\leq$  *tab*[*i*], 1  $\leq$  *k* < *i* et *tab*[*i* . . . *n*] trié}

7                **fin pour**

8      **fin** triSelection

$$T(n) = \sum_{i=2}^n T(\text{iDPG}(i)) = \sum_{i=2}^n (i - 1) = \frac{n(n - 1)}{2}$$

**procedure** trilnsertion ( *tab* )

entrée :

*tab* : tableau indicé de 1 à *n* incl.

sortie :

*tab* : tableau indicé de 1 à *n* incl., trié

début

1       **pour** *i*  $\leftarrow$  2 **haut** *n* **faire**

2            *x*  $\leftarrow$  *tab*[*i*]

3            *j*  $\leftarrow$  *i*

4       **tant que** (*j*  $\geq$  2) et *tab*[*j* - 1] > *x* **faire**

5            *tab*[*j*]  $\leftarrow$  *tab*[*j* - 1]

6            *j*  $\leftarrow$  *j* - 1

7       **fin tant que**

8            *tab*[*j*]  $\leftarrow$  *x*

9       **fin pour**

**fin** trilnsertion

Pire cas :

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

Meilleur cas :

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

### 3. Rappels sur la croissance des fonctions, les logarithmes et les sommes

### 3.1. Croissance des fonctions

- Le temps d'exécution  $T_A(n)$  d'un algorithme  $A$ , où  $n$  est la taille du problème, est une fonction

$$f : \mathbb{N} \rightarrow \mathbb{R}^+$$

- Il existe cinq constructeurs de familles de fonctions :

Constructeur	Concept intuitif	Appellation
$O$	$\leq$	Grand- $O$
$o$	$<$	Petit- $o$
$\Omega$	$\geq$	Oméga
$\omega$	$>$	Petit oméga
$\Theta$	$=$	Theta

- Ces constructeurs permettent d'avoir une idée du **comportement asymptotique** du temps d'exécution, c'est-à-dire pour de grandes valeurs de  $n$ .

$f \in O(g)$      $f$  croît au même rythme ou moins vite que  $g$      $\leq$

$f \in o(g)$      $f$  croît strictement moins vite que  $g$      $<$

$f \in \Omega(g)$      $f$  croît au même rythme ou plus vite que  $g$      $\geq$

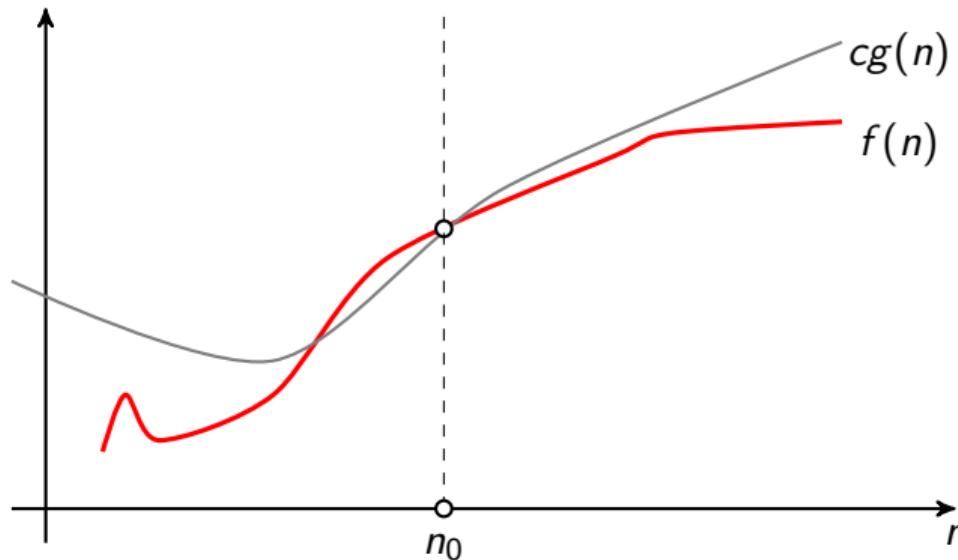
$f \in \omega(g)$      $f$  croît strictement plus vite que  $g$      $>$

$f \in \Theta(g)$      $f$  et  $g$  croissent au même rythme     $=$

## Définition

$f \in O(g)$  s'il existe des constantes  $c > 0$  et  $n_0 \geq 0$  telles que

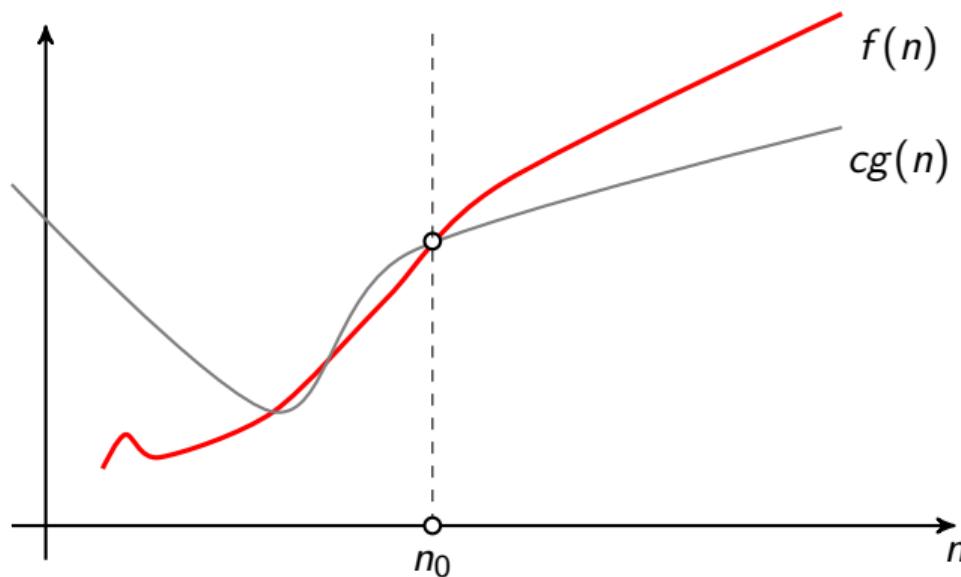
$$f(n) \leq cg(n) \quad \text{pour tout } n \geq n_0.$$



## Définition

$f \in \Omega(g)$  s'il existe des constantes  $c > 0$  et  $n_0 \geq 0$  telles que

$$f(n) \geq cg(n) \quad \text{pour tout } n \geq n_0.$$



## Définition

$f \in \Theta(g)$  si  $f \in O(g)$  et  $f \in \Omega(g)$ .

On a aussi

- ▶  $f \in O(g)$  si  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$
- ▶  $f \in o(g)$  si  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
- ▶  $f \in \Omega(g)$  si  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$
- ▶  $f \in \omega(g)$  si  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$
- ▶  $f \in \Theta(g)$  si  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lambda$  où  $\lambda \in \mathbb{R} \setminus \{0\}$ .

# Propriétés

## Transitivité

- ▶  $f \in O(g)$  et  $g \in O(h) \Rightarrow f \in O(h),$
- ▶  $f \in o(g)$  et  $g \in o(h) \Rightarrow f \in o(h),$
- ▶  $f \in \Omega(g)$  et  $g \in \Omega(h) \Rightarrow f \in \Omega(h),$
- ▶  $f \in \omega(g)$  et  $g \in \omega(h) \Rightarrow f \in \omega(h).$
- ▶  $f \in \Theta(g)$  et  $g \in \Theta(h) \Rightarrow f \in \Theta(h),$

## Reflexivité

- ▶  $f \in O(f)$ ,
- ▶  $f \in \Omega(f)$ ,
- ▶  $f \in \Theta(f)$ .

## Symétrie

- ▶  $f \in \Theta(g) \iff g \in \Theta(f)$ .

## Symétrie transposée

- ▶  $f \in O(g) \iff g \in \Omega(f)$ ,
- ▶  $f \in o(g) \iff g \in \omega(f)$ .

## Définition

Soit un entier  $d \geq 0$ . Un **polynôme** en  $n$  de **degré**  $d$  est une fonction  $p$  telle que

$$p(n) = \sum_{i=0}^d a_i n^i = a_0 n^0 + a_1 n^1 + a_2 n^2 + \cdots + a_d n^d = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d$$

où les constantes  $a_0, a_1, a_2, \dots, a_d$  sont les coefficients du polynôme et  $a_d \neq 0$ .

## Exemples

- ▶  $3n^4 + 28n - 4$  est un polynôme de degré 4.
- ▶  $n^3 + n^2 + n + 1$  est un polynôme de degré 3.
- ▶  $100n^{18}$  est un polynôme de degré 18.
- ▶  $-1$  est un polynôme de degré 0.

## Théorème

Si  $f(n) = \sum_{i=0}^d a_i n^i$ , un polynôme de degré  $d$  alors

- ▶  $f \in \Theta(n^d)$
- ▶  $f \in O(n^m)$  pour tout  $m \geq d$
- ▶  $f \in \Omega(n^m)$  pour tout  $0 \leq m \leq d$

## Exemple

Soient  $f(n) = 15n^5 - 4n^2 + 10n - 5$ .

- ▶  $f \in \Theta(n^5)$
- ▶  $f \in O(n^5), f \in O(n^6), f \in O(n^{1024})$
- ▶  $f \in \Omega(n^5), f \in \Omega(n^4), f \in \Omega(1)$

## Exemples

1. Montrer que  $f(n) = 2n^2 \in O(n^2)$

► **Avec la définition**

Trouver des constantes  $c$  et  $n_0$  telles que  $2n^2 \leq cn^2$  pour tout  $n \geq n_0$

$$2n^2 \leq cn^2 \text{ pour tout } n \geq 0$$

$$2 \leq c, \text{ pour tout } n \geq 0$$

Ainsi,  $c = 2$  et  $n_0 = 0$  conviennent.

► **Avec la limite**

$$\lim_{n \rightarrow \infty} \frac{2n^2}{n^2} = \lim_{n \rightarrow \infty} 2 = 2$$

2. Montrer que  $f(n) = 2n^2 \notin O(n)$

► Avec la définition

Il faut montrer qu'il n'existe pas de constantes  $c$  et  $n_0$  telles que  $2n^2 \leq cn$  pour tout  $n \geq n_0$

En supposant par l'absurde que  $c$  et  $n_0$  existent, nous aurions

$$2n^2 \leq cn, \text{ pour tout } n \geq n_0$$

$$2n \leq c, \text{ pour tout } n \geq n_0$$

Or, dès que  $n > \frac{c}{2}$ , l'inégalité précédente devient fausse. Ainsi, les constantes  $c$  et  $n_0$  n'existent pas.

► Avec la limite

$$\lim_{n \rightarrow \infty} \frac{2n^2}{n} = \lim_{n \rightarrow \infty} 2n = \infty$$

3. Montrer que  $f(n) = 2n^2 - 3n + 5 \in O(n^3)$

► Avec la définition

Trouver des constantes  $c$  et  $n_0$  telles que  $2n^2 - 3n + 5 \leq cn^3$  pour tout  $n \geq n_0$

$$2n^2 - 3n + 5 \leq 2n^2 + 3n + 5 \quad (\text{pour tout } n \geq 0)$$

$$\leq 2n^3 + 3n^3 + 5n^3 \quad (\text{pour tout } n \geq 1)$$

$$\leq 10n^3 \quad (\text{pour tout } n \geq 1)$$

Ainsi,  $c = 10$  et  $n_0 = 1$  conviennent.

► Avec la limite

$$\lim_{n \rightarrow \infty} \frac{2n^2 - 3n + 5}{n^3} = 0$$

4. Montrer que  $f(n) = 3n^4 - 2n^3 + 5n^2 - 2 \in \Omega(n^2)$

► Avec la définition

Supposons qu'il existe  $c > 0$  et  $n_0 \geq 0$  tels que

$$3n^4 - 2n^3 + 5n^2 - 2 \geq cn^2 \text{ pour tout } n \geq n_0$$

Ainsi,

$$3n^4 \geq 2n^3 + (c - 5)n^2 + 2 \text{ pour tout } n \geq n_0$$

En posant  $c = 5$ , nous avons  $3n^4 \geq 2n^3 + 2$ , ce qui est vrai pour tout  $n \geq n_0 = 2$ .

► Avec la limite

$$\lim_{n \rightarrow \infty} \frac{3n^4 - 2n^3 + 5n^2 - 2}{n^2} = \lim_{n \rightarrow \infty} 3n^2 = \infty$$

Remarque : Il est équivalent de montrer que  $n \in O(f)$ .

5. Montrer que  $f(n) = 2n^2 \in \Omega(n^2)$

► **Avec la définition**

Trouver des constantes  $c$  et  $n_0$  telles que  $2n^2 \geq cn^2$  pour tout  $n \geq n_0$

Ainsi,  $c = 1$  et  $n_0 = 0$  conviennent.

► **Avec la limite**

$$\lim_{n \rightarrow \infty} \frac{2n^2}{n^2} = \lim_{n \rightarrow \infty} 2 = 2$$

**Remarque :** Puisque  $2n^2 \in O(n^2)$  et  $2n^2 \in \Omega(n^2)$ , alors  $2n^2 \in \Theta(n^2)$ .

## 3.2. Logarithmes

## Logarithmes

- ▶  $x = \log_a y$  est la réciproque de  $y = a^x$
- ▶  $\log_a(a^x) = x$  et  $a^{\log_a y} = y$
- ▶  $\log_a b = \frac{1}{\log_b a}$
- ▶  $\log_a(xy) = \log_a x + \log_a y$  et  $\log_a(x/y) = \log_a x - \log_a y$
- ▶  $\log_a(x^n) = n \log_a x$  et  $\log_a(1/x) = -\log_a x$
- ▶  $\log_b x = \log_b a \cdot \log_a x$  et  $\log_a x = \frac{\log_b x}{\log_b a}$
- ▶  $a^{\log_b n} = n^{\log_b a}$

Ici,  $a > 1, b > 1$  et si  $x = \log_a y, y > 0$ .

Bases usuelles : 2, 10,  $e = \sum_{n \in \mathbb{N}} \frac{1}{n!} \simeq 2.71828182845904523536028747$

### 3.3. Fonctions utiles

## Fonctions utiles

- ▶  $f$  : fonction polynôme logarithmique :

$$a_0 + a_1 \log n + a_2(\log n)^2 + \cdots + a_k(\log n)^k, a_k > 0, k \geq 1$$

- ▶  $g$  : polynôme en  $n$ . Ex. :  $12n^4 - 3n + 7$
- ▶  $h$  : fonction exponentielle. Ex. :  $a^n$  où  $a$  est une constante  $> 1$

On a

- ▶  $f \in o(g)$
- ▶  $g \in o(h)$
- ▶  $n^a \in o(n^b), a < b$
- ▶  $a^n \in o(b^n), 1 < a < b$
- ▶ Toutes croissent plus lentement que  $n!$
- ▶  $n! \in o(n^n)$

Taux de croissance typiques, en ordre croissant :

Fonction	Nom
1	Constant
$\log n$	Logarithmique
$\log^2 n$	Log-carré
$n$	Linéaire
$n \log n$	Quasi linéaire
$n^2$	Quadratique
$n^3$	Cubique
$n^a$ ( $a > 1$ )	Polynomial
$b^n$ ( $b > 1$ )	Exponentiel
$n!$	Factoriel
$n^n$	Hyper-exponentiel

## 3.4. Sommes

## Suites arithmétiques

Suite de la forme  $s_n = a + bn$  où  $a, b \in \mathbb{R}$ .

Par exemple, pour  $a = 1$  et  $b = 2$ , la suite  $(s_n)_{n \in \mathbb{N}}$  vérifie

$$s_0 = 1 + 2 \times 0 = 1$$

$$s_1 = 1 + 2 \times 1 = 3$$

$$s_2 = 1 + 2 \times 2 = 5$$

$$s_3 = 1 + 2 \times 3 = 7$$

Résultat important, sur la somme des  $k + 1$  premiers termes de  $(s_n)_{n \in \mathbb{N}}$  :

$$\begin{aligned} \sum_{n=0}^k (a + bn) &= \sum_{n=0}^k a + \sum_{n=0}^k bn \\ &= (k + 1)a + b \sum_{n=0}^k n \\ &= (k + 1)a + b \frac{k(k + 1)}{2} \end{aligned}$$

## Suites géométriques

Suite de la forme  $s_n = ar^n$  où  $a \in \mathbb{R}$  et  $r \in \mathbb{R} \setminus \{0, 1\}$ .

Par exemple, pour  $a = 2$  et  $r = 3$ , la suite  $(s_n)_{n \in \mathbb{N}}$  vérifie

$$s_0 = 2 \times 3^0 = 2$$

$$s_1 = 2 \times 3^1 = 6$$

$$s_2 = 2 \times 3^2 = 18$$

$$s_3 = 2 \times 3^3 = 54$$

Résultat important, sur la somme des  $k + 1$  premiers termes de  $(s_n)_{n \in \mathbb{N}}$  :

$$\begin{aligned}\sum_{n=0}^k ar^n &= a \sum_{n=0}^k r^n \\ &= a \frac{1 - r^{k+1}}{1 - r}\end{aligned}$$

### 3.5. Plafond et plancher

## Plafond et plancher

- ▶ Plancher (partie entière par défaut)  $\lfloor r \rfloor$  : Le plus grand entier  $\leq r$

Exemples

- ▶ Plafond (partie entière excès)  $\lceil r \rceil$  : Le plus petit entier  $\geq r$

Exemples

### Propriétés

$$\left\lfloor \frac{\lfloor n/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{n}{ab} \right\rfloor$$

$$\left\lceil \frac{\lceil n/a \rceil}{b} \right\rceil = \left\lceil \frac{n}{ab} \right\rceil$$

$$n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$$

$$\lfloor n/2 \rfloor + 1 \geq \lceil n/2 \rceil$$

# 4. Résolution des équations de récurrence

## But

- ▶ Trouver la complexité asymptotique temporelle à partir d'une équation de récurrence pour  $T(n)$  mesurant le temps d'exécution d'un algorithme sur une entrée de taille  $n$  dans le pire cas.

Concrètement, trouver une fonction  $g$  telle que  $T \in \Theta(g)$ .

## Exemples de récurrences

- ▶  $T(n) = T(n - 1) + 1$
- ▶  $T(n) = T(n - 1) + n$
- ▶  $T(n) = 2T(n/2) + 1$
- ▶  $T(n) = 2T(n/2) + n$

## Types de récurrences étudiées

- ▶ récurrences aux différences finies
- ▶ récurrences aux divisions finies

## 4.1. Récurrences aux différences finies

## Différences finies

Considérons  $T$  qui vérifie la récurrence

$$T(n) = T(n - 1) + f(n), \quad n \geq 1,$$

$$T(0) = f(0),$$

où  $f : \mathbb{N} \rightarrow \mathbb{R}^+$  est une fonction.

### Exemples.



$$T(n) = T(n - 1) + 1, \quad n \geq 1,$$

$$T(0) = 1.$$



$$T(n) = T(n - 1) + n + 1, \quad n \geq 1,$$

$$T(0) = 1.$$

## Différences finies — exemples d'algorithmes

La mesure de complexité

$$T(n) = T(n - 1) + f(n), \quad n \geq 1,$$

$$T(0) = f(0)$$

peut par exemple apparaître pour un algorithme récursif de la forme

A( $x$ ) :

Si  $|x| = 0$  : cas terminal

$x' \leftarrow$  objet obtenu à partir de  $x$ , de taille  $|x| - 1$

Appel A( $x'$ )

Instructions de coût total  $f(|x|)$

Fin.

## Différences finies — résolution

Rappel :

$$\begin{aligned}T(n) &= T(n - 1) + f(n), \quad n \geq 1, \\T(0) &= f(0).\end{aligned}$$

Voyons quelques exemples :

$$T(0) = f(0)$$

$$T(1) = T(0) + f(1) = f(0) + f(1)$$

$$T(2) = T(1) + f(2) = f(0) + f(1) + f(2)$$

$$T(3) = T(2) + f(3) = f(0) + f(1) + f(2) + f(3)$$

## Différences finies — résultat général

Nous obtenons ainsi le résultat général suivant :

Si  $T$  vérifie la récurrence

$$T(n) = T(n - 1) + f(n), \quad n \geq 1,$$

$$T(0) = f(0),$$

où  $f : \mathbb{N} \rightarrow \mathbb{R}^+$  est une fonction, alors

$$T(n) = \sum_{i=0}^n f(i), \quad n \geq 0.$$

De cette dernière expression dans laquelle la récurrence a disparu, une étude classique peut s'en suivre pour déterminer sa classe de complexité.

## Différences finies — exemples 1/3

$$T(n) = T(n - 1) + f(n), \quad n \geq 1,$$

$$T(0) = f(0).$$

$$T(n) = \sum_{i=0}^n f(i), \quad n \geq 0.$$

- ▶ Lorsque  $f(n) = 1$  pour tout  $n \geq 0$ ,

$$\begin{aligned} T(n) &= \sum_{i=0}^n 1 \\ &= n + 1 \in \Theta(n) \end{aligned}$$

## Différences finies — exemples 2/3

$$T(n) = T(n - 1) + f(n), \quad n \geq 1,$$

$$T(0) = f(0).$$

$$T(n) = \sum_{i=0}^n f(i), \quad n \geq 0.$$

- ▶ Lorsque  $f(n) = n$  pour tout  $n \geq 0$ ,

$$\begin{aligned} T(n) &= \sum_{i=0}^n i \\ &= \frac{n(n+1)}{2} \in \Theta(n^2). \end{aligned}$$

## Différences finies — exemples 3/3

$$T(n) = T(n-1) + f(n), \quad n \geq 1,$$
$$T(0) = f(0).$$
$$T(n) = \sum_{i=0}^n f(i), \quad n \geq 0.$$

- ▶ Lorsque  $f(n) = n^2$  pour tout  $n \geq 0$ ,

$$\begin{aligned} T(n) &= \sum_{i=0}^n i^2 \\ &= \frac{n(n+1)(2n+1)}{6} \in \Theta(n^3). \end{aligned}$$

## 4.2. Récurrences aux divisions finies

## Divisions finies

Considérons  $T$  qui vérifie la récurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + f(n), \quad n \geq 2,$$

$$T(1) = f(1),$$

où  $f : \mathbb{N} \rightarrow \mathbb{R}^+$  est une fonction.

### Exemples.



$$T(n) = 2T\left(\frac{n}{2}\right) + 1, \quad n \geq 2,$$

$$T(1) = 1.$$



$$T(n) = 2T\left(\frac{n}{2}\right) + n, \quad n \geq 2,$$

$$T(1) = 1.$$

## Divisions finies — exemples d'algorithmes

La mesure de complexité

$$T(n) = 2T\left(\frac{n}{2}\right) + f(n), \quad n \geq 2,$$

$$T(1) = f(1)$$

peut par exemple apparaître pour un algorithme récursif de la forme

A( $x$ ) :

Si  $|x| = 1$  : cas terminal

$(x_1, x_2) \leftarrow$  découpage de  $x$  tel que  $|x_1| \simeq \frac{|x|}{2} \simeq |x_2|$

Appel A( $x_1$ )

Appel A( $x_2$ )

Instructions de coût total  $f(|x|)$

Fin.

## Divisions finies — résolution

Rappel :

$$T(n) = 2T\left(\frac{n}{2}\right) + f(n), \quad n \geq 2,$$
$$T(1) = f(1).$$

Voyons quelques exemples dans le cas où  $n$  est une puissance de deux  $2^p$  :

$$T(1) = f(1)$$

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

$$T(4) = 2T(2) + f(4) = 4f(1) + 2f(2) + f(4)$$

$$T(8) = 2T(4) + f(8) = 8f(1) + 4f(2) + 2f(4) + f(8)$$

$$T(16) = 2T(8) + f(16) = 16f(1) + 8f(2) + 4f(4) + 2f(8) + f(16)$$

## Divisions finies — résultat général

Nous obtenons ainsi le résultat général suivant :

Si  $T$  vérifie la récurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + f(n), \quad n \geq 2,$$

$$T(1) = f(1)$$

où  $f : \mathbb{N} \rightarrow \mathbb{R}^+$  est une fonction, alors

$$T(2^p) = \sum_{i=0}^p 2^i f(2^{p-i}), \quad p \geq 0.$$

De cette dernière expression dans laquelle la récurrence a disparu, une étude classique peut s'en suivre pour déterminer sa classe de complexité.

## Divisions finies — exemples 1/4

$$T(1) = f(1), T(n) = 2T\left(\frac{n}{2}\right) + f(n), n \geq 2, \quad T(2^p) = \sum_{i=0}^p 2^i f(2^{p-i}), p \geq 0, n = 2^p.$$

- ▶ Lorsque  $f(n) = 1$  pour tout  $n \geq 1$ ,

$$\begin{aligned} T(2^p) &= \sum_{i=0}^p 2^i \times 1 = \sum_{i=0}^p 2^i \\ &= 2^{p+1} - 1 = 2 \times 2^p - 1 \\ &= 2n - 1 \\ &\in \Theta(n) \end{aligned}$$

## Divisions finies — exemples 2/4

$$T(1) = f(1), T(n) = 2T\left(\frac{n}{2}\right) + f(n), n \geq 2, \quad T(2^p) = \sum_{i=0}^p 2^i f(2^{p-i}), p \geq 0, n = 2^p.$$

► Lorsque  $f(n) = \sqrt{n} = n^{\frac{1}{2}}$  pour tout  $n \geq 1$ ,

$$\begin{aligned} T(2^p) &= \sum_{i=0}^p 2^i \sqrt{2^{p-i}} = \sum_{i=0}^p 2^i (2^{p-i})^{\frac{1}{2}} \\ &= \sum_{i=0}^p 2^i (2^p)^{\frac{1}{2}} (2^{-i})^{\frac{1}{2}} = \sum_{i=0}^p (2^p)^{\frac{1}{2}} 2^i (2^{-i})^{\frac{1}{2}} \\ &= (2^p)^{\frac{1}{2}} \sum_{i=0}^p 2^i (2^{-i})^{\frac{1}{2}} = (2^p)^{\frac{1}{2}} \sum_{i=0}^p 2^i 2^{\frac{-i}{2}} = (2^p)^{\frac{1}{2}} \sum_{i=0}^p 2^{\frac{i}{2}} = (2^p)^{\frac{1}{2}} \sum_{i=0}^p \left(2^{\frac{1}{2}}\right)^i \\ &= (2^p)^{\frac{1}{2}} \frac{1 - \left(2^{\frac{1}{2}}\right)^{p+1}}{1 - 2^{\frac{1}{2}}} = \sqrt{n} \frac{1 - \sqrt{2}^{p+1}}{1 - \sqrt{2}} = \sqrt{n} \frac{1 - \sqrt{2}\sqrt{n}}{1 - \sqrt{2}} \in \Theta(n) \end{aligned}$$

## Divisions finies — exemples 3/4

$$T(1) = f(1), T(n) = 2T\left(\frac{n}{2}\right) + f(n), n \geq 2, \quad T(2^p) = \sum_{i=0}^p 2^i f(2^{p-i}), p \geq 0, n = 2^p.$$

- ▶ Lorsque  $f(n) = n$  pour tout  $n \geq 1$ ,

$$\begin{aligned} T(2^p) &= \sum_{i=0}^p 2^i 2^{p-i} = \sum_{i=0}^p 2^p \\ &= \sum_{i=0}^p n = n \sum_{i=0}^p 1 \\ &= n(p+1) = n \log_2(2n) \\ &\in \Theta(n \log(n)) \end{aligned}$$

## Divisions finies — exemples 4/4

$$T(1) = f(1), T(n) = 2T\left(\frac{n}{2}\right) + f(n), n \geq 2, \quad T(2^p) = \sum_{i=0}^p 2^i f(2^{p-i}), p \geq 0, n = 2^p.$$

► Lorsque  $f(n) = n^2$  pour tout  $n \geq 1$ ,

$$\begin{aligned} T(2^p) &= \sum_{i=0}^p 2^i (2^{p-i})^2 = \sum_{i=0}^p 2^i 2^{2p-2i} \\ &= \sum_{i=0}^p 2^{2p-i} = \sum_{i=0}^p 2^{2p} 2^{-i} = 2^{2p} \sum_{i=0}^p \left(\frac{1}{2}\right)^i \\ &= n^2 \frac{1 - \left(\frac{1}{2}\right)^{p+1}}{1 - \frac{1}{2}} = n^2 \frac{1 - \frac{1}{2} \left(\frac{1}{2}\right)^p}{1 - \frac{1}{2}} \\ &= n^2 \frac{1 - \frac{1}{2} \frac{1}{n}}{1 - \frac{1}{2}} = \frac{n^2}{2} \left(1 - \frac{1}{2n}\right) \\ &\in \Theta(n^2) \end{aligned}$$

## Divisions finies

Le **théorème général (Master Theorem)** s'applique aux équations de récurrence aux divisions finies générales de la forme

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

où  $a \geq 1$ ,  $b \geq 2$  et  $f : \mathbb{N} \rightarrow \mathbb{R}^+$ .

$a$  est le nombre de sous-problèmes engendrés.  $b$  est le facteur de réduction.

### Théorème

Soit  $c = \log_b(a)$ . L'ordre de grandeur de  $T$  se classe en trois catégories :

1. si  $f \in \mathcal{O}(n^{c-\epsilon})$  pour un  $\epsilon > 0$ , alors  $T \in \Theta(n^c)$ ;
2. si  $f \in \Theta(n^c \log(n)^k)$  pour un  $k \geq 0$ , alors  $T \in \Theta(n^c \log(n)^{k+1})$ ;
3. si  $f \in \Omega(n^{c+\epsilon})$  pour un  $\epsilon > 0$  et il existe  $0 < k < 1$  et  $n_0 \in \mathbb{N}$  tels que pour tout  $n \geq n_0$ ,  $af\left(\frac{n}{b}\right) \leq kf(n)$ , alors  $T \in \Theta(f)$ .

## Intuition et cas d'usage

Considérons le cas particulier

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

où  $f(n) \in \Theta(n^d)$  pour un  $d \in \mathbb{R}^+$ .

Dans cette équation de récurrence,

- ▶  $a$  est le nombre de **sous-problèmes** (nombre d'appels récursifs) ;
- ▶  $b$  est le facteur de réduction du problème ; **plus c'est grand plus c'est découpé**
- ▶  $d$  est la complexité de la recombinaison des résultats.

Pour connaître le comportement asymptotique de  $T$ , on compare  $c = \log_b(a)$  avec  $d$  :

1. si  $d < c$ , le comportement est **dominé par les appels récursifs** et  $T \in \Theta(n^c)$  ;
2. si  $d = c$ , il y a **équilibre entre les appels récursifs et la recombinaison** et  $T \in \Theta(n^c \log(n))$  ;
3. si  $d > c$ , le comportement est **dominé par la recombinaison** et  $T \in \Theta(n^d)$ .

## Exemples d'application

- ▶  $a = 2, b = 2$ . Ainsi,  $c = \log_b(a) = 1$ .
  - ▶  $f : n \mapsto 1$ .  $d = 0 < c$ . Cas 1.  $T \in \Theta(n)$ . complexité linéaire
  - ▶  $f : n \mapsto n^{\frac{1}{2}}$ .  $d = \frac{1}{2} < c$ . Cas 1.  $T \in \Theta(n)$ .
  - ▶  $f : n \mapsto n$ .  $d = 1 = c$ . Cas 2.  $T \in \Theta(n \log(n))$ .
  - ▶  $f : n \mapsto n^2$ .  $d = 2 > c$ . Cas 3.  $T \in \Theta(n^2)$ .
- ▶  $a = 2, b = 3$ . Ainsi,  $c = \log_b(a) \simeq 0.63$ .
  - ▶  $f : n \mapsto 1$ .  $d = 0 < c$ . Cas 1.  $T \in \Theta(n^{0.63})$ .
  - ▶  $f : n \mapsto n^{\frac{1}{2}}$ .  $d = \frac{1}{2} < c$ . Cas 1.  $T \in \Theta(n^{0.63})$ .
  - ▶  $f : n \mapsto n^c$ .  $d = c$ . Cas 2.  $T \in \Theta(n^{0.63} \log(n))$ .
  - ▶  $f : n \mapsto n$ .  $d = 1 > c$ . Cas 3.  $T \in \Theta(n)$ .
  - ▶  $f : n \mapsto n^2$ .  $d = 2 > c$ . Cas 3.  $T \in \Theta(n^2)$ .
- ▶  $a = 3, b = 2$ . Ainsi,  $c = \log_b(a) \simeq 1.58$ .
  - ▶  $f : n \mapsto n$ .  $d = 1 < c$ . Cas 1.  $T \in \Theta(n^{1.58})$ .
  - ▶  $f : n \mapsto n^c$ .  $d = c$ . Cas 2.  $T \in \Theta(n^{1.58} \log(n))$ .
  - ▶  $f : n \mapsto n^2$ .  $d = 2 > c$ . Cas 3.  $T \in \Theta(n^2)$ .

Table des premières valeurs  $\log_b(a)$  arrondies :

	$a = 1$	$a = 2$	$a = 3$	$a = 4$	$a = 5$	$a = 6$	$a = 7$	$a = 8$	$a = 9$
$b = 2$	0.00	1.00	1.58	2.00	2.32	2.58	2.81	3.00	3.17
$b = 3$	0.00	0.63	1.00	1.26	1.46	1.63	1.77	1.89	2.00
$b = 4$	0.00	0.50	0.79	1.00	1.16	1.29	1.40	1.50	1.58
$b = 5$	0.00	0.43	0.68	0.86	1.00	1.11	1.21	1.29	1.37
$b = 6$	0.00	0.39	0.61	0.77	0.90	1.00	1.09	1.16	1.23
$b = 7$	0.00	0.36	0.56	0.71	0.83	0.92	1.00	1.07	1.13
$b = 8$	0.00	0.33	0.53	0.67	0.77	0.86	0.94	1.00	1.06
$b = 9$	0.00	0.32	0.50	0.63	0.73	0.82	0.89	0.95	1.00

# 5. Méthode « diviser pour régner »

## Principe de diviser pour régner

Diviser : Des problèmes de tailles plus petites sont résolus récursivement.

Régner : Les solutions des sous-problèmes sont utilisées pour former la solution du problème original.

Résolution du problème en taille  $n$ .

- ▶ Division :  $a$  sous-problèmes, chacun sur une taille  $\frac{n}{b}$ .
- ▶ Recombinaison des solutions en coût  $f(n)$ .

Mène à une équation de récurrence aux divisions finies.

## 5.1. Exemples

# Exponentiation à la russe

**fonction puissance ( $x, n$ ) renvoie entier**

entrée :

$x$  : réel.

$n$  : naturel.

sortie :

$x$  élevé à la puissance  $n$

début

1      si  $n = 0$  alors

2            renvoyer 1

3      sinon si  $n$  pair alors

4            renvoyer puissance ( $x*x, n/2$ )

5      sinon

6            renvoyer  $x * \text{puissance} (x*x, \lfloor n/2 \rfloor)$

7      fin si

fin puissance

recursion exemple  $x^8$  va être:

$x * x * x * x * x * x * x * x$

qui devient :

$(x * x * x * x)^* (x * x * x * x)$

ainsi de suite. dichotomiquement

il y a aussi l'exponentielle dichotomique. faire le pseudo code

$T(n/2) + 1$

$T(\lfloor n/2 \rfloor) + 2$

$T(n) =$  nombre de multiplications (opération barométrique) effectuées pour calculer puissance( $x, n$ )

# Analyse

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ T(n/2) + 1 & \text{si } n \text{ est pair} \\ T(\lfloor n/2 \rfloor) + 2 & \text{si } n \text{ est impair} \end{cases}$$

- ▶  $T(n) = T(n/2) + \text{cste}$
- ▶ Ici,  $a = 1$ ,  $b = 2$  et  $f(n) = \text{cste}$
- ▶  $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$
- ▶  $f(n)$  et  $n^{\log_b a} = 1$  croissent au même rythme  $\rightarrow$  cas 2 du théorème général
- ▶  $T \in \Theta(\log n)$
- ▶ **Remarque :** de l'espace mémoire est utilisé par les appels récursifs

## Version non-réursive

**fonction puissance ( $x, n$ ) renvoie entier**

...

**début**

```
1       $y \leftarrow x; m \leftarrow n; res \leftarrow 1$ 
2      tant que  $m > 0$  faire
3          si  $m$  impair alors
4               $res \leftarrow res * y$            entre 1 et  $\lfloor \log_2 n \rfloor + 1$  fois
5          fin si
6           $y \leftarrow y * y$                   $\lfloor \log_2 n \rfloor + 1$  fois
7           $m \leftarrow m/2$ 
8      fin tant que
9      renvoyer  $res$ 
fin puissance
```

- ▶ Rappel : le nombre de bits nécessaires pour représenter  $n \geq 1$  en base deux est  $\lfloor \log_2 n \rfloor + 1$ .
- ▶ Cela correspond au nombre de fois que  $n$  peut être divisé par 2 avant d'être nul.
- ▶  $\lfloor \log_2 n \rfloor + 2 \leq T(n) \leq 2\lfloor \log_2 n \rfloor + 2$
- ▶  $T \in \Theta(\log n)$

## Application : Calcul des nombres de Fibonacci

$$F_n = F_{n-1} + F_{n-2}, \quad n \geq 2, \quad F_0 = 0 \text{ et } F_1 = 1.$$

### ► Algorithme 1

**fonction**  $f(n)$  **renvoie** entier

entrée :

$n$  : naturel.

sortie :

$F_n$

**début**

1      **si**  $n = 0$  ou  $1$  **alors**

2             $res \leftarrow n$

$$T(0) = T(1) = 0$$

3      **sinon**

4             $res \leftarrow f(n - 1) + f(n - 2)$

$$T(n - 1) + T(n - 2) + 1$$

5      **fin si**

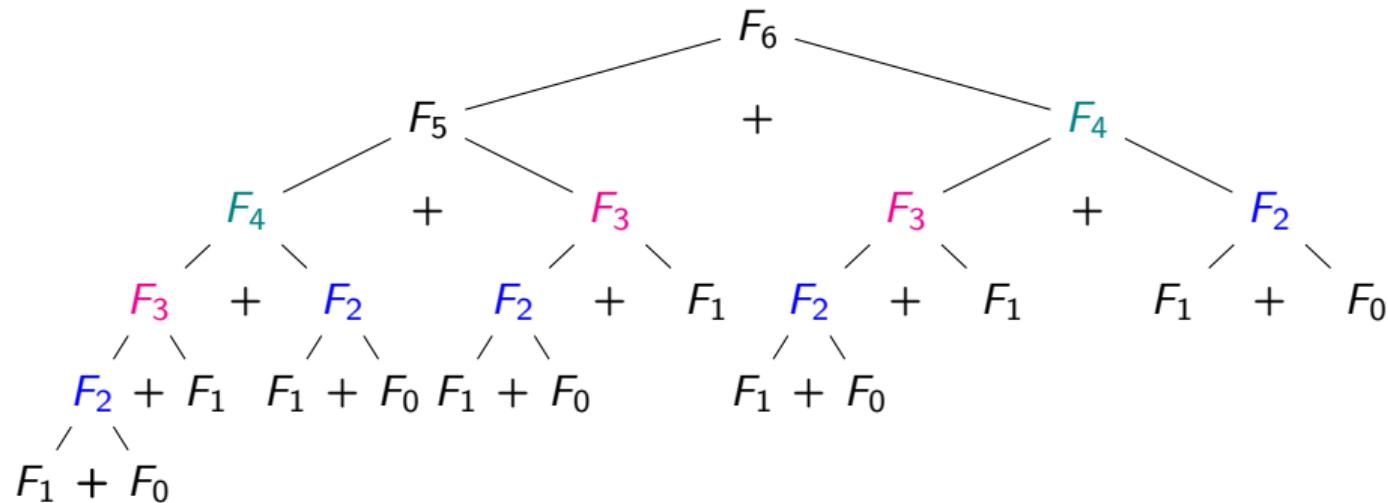
6      **renvoyer**  $res$

**fin f**

$T(n)$  = nombre d'additions (opération barométrique) effectuées pour calculer  $f(n)$

- ▶ Il est possible de démontrer par induction sur  $n$  que pour tout  $n \geq 3$ ,  $T(n) \geq \left(\frac{3}{2}\right)^{n-3}$ .
- ▶ Ainsi,  $T \in \Omega\left(\left(\frac{3}{2}\right)^{n-3}\right)$ , ce qui montre que  $T$  croît exponentiellement.
- ▶ **Remarque** : certaines valeurs de  $F$  sont calculées plusieurs fois

Nous avons l'arbre d'appels



## ► Algorithme 2

**fonction**  $f(n)$  **renvoie** entier

entrée :

$n$  : naturel.

sortie :

$F_n$

**début**

1      **si**  $n = 0$  ou  $1$  **alors**

2            **envoyer**  $n$

3      **sinon**

4             $fMoins2 \leftarrow 0$ ;  $fMoins1 \leftarrow 1$

5            **pour**  $i \leftarrow 2$  à  $n$  **faire**

6                 $tmp \leftarrow fMoins1$

7                 $fMoins1 \leftarrow fMoins1 + fMoins2$

8                 $fMoins2 \leftarrow tmp$

9            **fin pour**

10          **envoyer**  $fMoins1$

11          **fin si**

**fin f**

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

► Algorithme 3

Soit, pour tout  $n \geq 1$ , la matrice

$$M_n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

Nous avons

$$\begin{aligned} M_n \times M_1 &= \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} F_{n+1} + F_n & F_{n+1} \\ F_n + F_{n-1} & F_n \end{pmatrix} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix} \\ &= M_{n+1} \end{aligned}$$

On en déduit que

$$M_1^n = M_n.$$

Pour tout  $n \geq 1$ , la valeur de  $F_n$  se lit en coordonnées  $(1, 1)$  de la matrice  $M_1^{n-1}$ .

### ► Algorithme 3

Le calcul du produit de deux matrices de dimension  $2 \times 2$  nécessite  $4 \times 2 = 8$  multiplications et 4 additions. Ceci ne dépend pas de  $n$ .

Ainsi, il est possible de calculer  $M_1^{n-1}$  (et donc  $F_n$ ) via l'algorithme d'exponentiation à la russe.

La complexité  $T$  en temps du calcul est donc  $T \in \Theta(\log(n))$ .

Exercice : planter (dans n'importe quel langage de programmation) les trois algorithmes de calcul de  $F_n$  et expérimenter avec les différentes méthodes.

# Somme d'intervalle maximale

[fin cour](#)

## Problème

Soient  $n$  entiers  $a_1, a_2, \dots, a_n$ . Trouver la valeur maximale de  $S = \sum_{k=i}^j a_k$  où  $i$  et  $j$  sont des indices du tableau. Il s'agit de la plus grande obtenue en sommant les éléments d'une portion du tableau.

Si tous les entiers sont négatifs, le résultat renvoyé sera 0 par convention (ce qui est cohérent car il s'agit de la somme des éléments d'une portion vide du tableau).

## Exemples

- ▶ Tableau  $-2, 14, 22, 7, 18, -21, 5, 18$ . Somme maximale : 63 ( $= a_2 + \dots + a_8$ )
- ▶ Tableau  $-2, 11, -4, 13, -5, -2$ . Somme maximale : 20 ( $= a_2 + a_3 + a_4$ )

## ► Algorithme 1 – Naïf

Calculer  $S$  pour tous les intervalles possibles

**fonction** sommeMax (*tab*) **renvoie** entier

entrée :

*tab* : tableau d'entiers indicés de 1 à  $n$

**début**

```
1     sMax ← 0
2     pour i ← 1 haut n faire
3         pour j ← i haut n faire
4             sommeTemp ← 0
5             pour k ← i haut j faire    (Calcul de tab[i] + tab[i + 1] + ⋯ + tab[j])
6                 sommeTemp ← sommeTemp + tab[k]
7             fin pour
8             si sommeTemp > sMax alors
9                 sMax ← sommeTemp
10            fin si
11            fin pour
12        fin pour
13        renvoyer sMax
fin sommeMax
```

## Analyse

$T(n) = \text{Nombre d'additions effectuées à la ligne 6 lorsque } tab \text{ est de taille } n.$

$$\begin{aligned} T(n) &= \sum_{i=1}^n \sum_{j=i}^n \sum_{k=i}^j 1 \\ &= \sum_{i=1}^n \sum_{j=i}^n (j - i + 1) \\ &= \dots \\ &= \frac{n(n+1)(n+2)}{6} \\ &\in \Theta(n^3) \end{aligned}$$

## ► Algorithme 2 – Naïf amélioré

Calculer  $S$  pour tous les intervalles possibles en évitant les calculs inutiles

**fonction** sommeMax ( *tab* ) **renvoie** entier

entrée :

*tab* : tableau d'entiers indicés de 1 à *n*

**début**

```
1      sMax ← 0
2      pour i ← 1 haut n faire
3          Somme maximale de tous les intervalles qui débutent à l'indice i
4          sommeTemp ← 0
5          pour j ← i haut n faire
6              sommeTemp ← sommeTemp + tab[j]
7              si sommeTemp > sMax alors
8                  sMax ← sommeTemp
9                  fin si
10                 fin pour
11             fin pour
12             renvoyer sMax
fin sommeMax
```

## Analyse

$T(n)$  = Nombre d'additions effectuées à la ligne 6

$$\begin{aligned} T(n) &= \sum_{i=1}^n \sum_{j=i}^n 1 \\ &= \dots \\ &= \frac{n(n+1)}{2} \\ &\in \Theta(n^2) \end{aligned}$$

## ► Algorithme 3

Calculer  $S$  en utilisant la stratégie "diviser pour régner"

**fonction** sommeMax (*tab, inf, sup*) **renvoie** entier

entrée :

*tab* : tableau d'entiers indicés de 1 à  $n$

*inf* : début de la portion de tableau

*sup* : fin de la portion de tableau

**début**

```
1   si inf = sup alors
2       si tab[inf] > 0 alors
3           renvoyer tab[inf]
4       sinon
5           renvoyer 0
6   fin si
7   sinon
8       milieu  $\leftarrow$  (inf + sup)/2
9       Calculer la somme maximale dans chacune des deux moitiés
10      maxGauche  $\leftarrow$  sommeMax(tab, inf, milieu)
11      maxDroite  $\leftarrow$  sommeMax(tab, milieu + 1, sup)
```

Calculer la somme maximale pour les intervalles chevauchant les deux moitiés

```
11    maxSommeGauche ← 0
12    s ← 0
13    pour i ← milieu bas inf faire
14        s ← s + tab[i]
15        si s > maxSommeGauche alors
16            maxSommeGauche ← s
17        fin si
18    fin pour
19    maxSommeDroite ← 0
20    s ← 0
21    pour i ← milieu + 1 haut sup faire
22        s ← s + tab[i]
23        si s > maxSommeDroite alors
24            maxSommeDroite ← s
25        fin si
26    fin pour
27    fin si
28    renvoyer max(maxGauche, maxDroite, maxSommeGauche + maxSommeDroite)
fin sommeMax
```

## Analyse

Prenons les lignes 14 et 22 comme opérations barométriques. Posons  $T(n)$  le nombre de fois que ces opérations sont exécutées en fonction de la longueur  $n$  de *tab*.

À chaque appel récursif la longueur du tableau courant est divisée par 2. De plus, deux appels récursifs sont effectués sur chacune des deux parties. Nous avons donc à faire à une équation de récurrence aux divisions finies avec  $a = 2$  (nombre d'appels récursifs) et  $b = 2$  (facteur de division). Ainsi,

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + f(n) \\ &= 2T\left(\frac{n}{2}\right) + f(n) \end{aligned}$$

Il reste à déterminer la fonction  $f$ . Il y a un nombre linéaire d'additions qui sont effectuées par les deux boucles, en fonction de  $n$ . Ainsi,  $f \in \Theta(n)$ .

Nous sommes dans le 2<sup>e</sup> cas du théorème général. Ainsi,  $T \in \Theta(n \log(n))$ .

## Multiplication de grands nombres

**Problème** : multiplier les nombres entiers  $A$  et  $B$ . Le plus grand des deux nombres contient  $n$  chiffres.

Nous allons compter, en fonction de  $n$ , le nombre d'additions requises pour produire le résultat.

- ▶ Algorithme classique

Deux boucles imbriquées :  $T \in \Theta(n^2)$

► Algorithme "diviser pour régner"

On pose  $n = 2m$  et on suppose que  $n = 2^p$ ,  $p \geq 0$

A 

$A_g = A/2^m$	$A_d = A \bmod 2^m$
---------------	---------------------

B 

$B_g = B/2^m$	$B_d = B \bmod 2^m$
---------------	---------------------

C 

$C_g = C/2^m$	$C_d = C \bmod 2^m$
---------------	---------------------

$$A = (A/2^m)2^m + A \bmod 2^m = A_g 2^m + A_d$$

$$B = (B/2^m)2^m + B \bmod 2^m = B_g 2^m + B_d$$

Nous avons

$$\begin{aligned}C &= a \times b = (A_g 2^m + A_d)(B_g 2^m + B_d) \\&= A_g B_g 2^{2m} + (A_g B_d + A_d B_g) 2^m + A_d B_d \\&= A_g B_g 2^{2m} + (A_g B_d + A_d B_g + A_g B_g - A_g B_g + A_d B_d - A_d B_d) 2^m + A_d B_d \\&= A_g B_g 2^{2m} + (A_g B_g + A_g B_d + A_d B_g + A_d B_d - A_g B_g - A_d B_d) 2^m + A_d B_d \\&= A_g B_g 2^{2m} + (A_g(B_g + B_d) + A_d(B_g + B_d) - A_g B_g - A_d B_d) 2^m + A_d B_d \\&= A_g B_g 2^{2m} + ((A_g + A_d)(B_g + B_d) - A_g B_g - A_d B_d) 2^m + A_d B_d \\&= C_g 2^n + (A_s B_s - C_g - C_d) 2^m + C_d\end{aligned}$$

où

$$\begin{aligned}C_g &= A_g B_g \\C_d &= A_d B_d \\A_s &= A_g + A_d \\B_s &= B_g + B_d \\C_s &= A_s B_s \\C &= 2^n C_g + (C_s - C_g - C_d) 2^m + C_d\end{aligned}$$

**fonction** multiplier( $A$ ,  $B$ ) **renvoie** grand entier

entrée :

$A$ ,  $B$  : grand entier

sortie :

le grand entier produit de  $A$  et  $B$

**début**

1	$C_g \leftarrow \text{multiplier}(A_g, B_g)$	$T(n/2)$
2	$C_d \leftarrow \text{multiplier}(A_d, B_d)$	$T(n/2)$
3	$A_s \leftarrow \text{additionner}(A_g, A_d)$	$\Theta(n)$
4	$B_s \leftarrow \text{additionner}(B_g, B_d)$	$\Theta(n)$
5	$C_s \leftarrow \text{multiplier}(A_s, B_s)$	$T(n/2)$
5	$X \leftarrow \text{additionner}(\text{additionner}(C_s, -C_g), -C_d)$	$\Theta(n)$
6	$C \leftarrow \text{additionner}(\text{additionner}(C_g \times 2^n, X \times 2^m), C_d)$	$\Theta(n)$
7	<b>renvoyer</b> $C$	
	<b>fin</b> multiplier	

## Analyse

Posons  $T(n)$  le nombre d'additions de chiffres nécessaires au calcul de la multiplication de  $A$  par  $B$ .

À chaque appel récursif, le nombre de chiffres des nombres à multiplier est divisée par 2. De plus, trois appels récursifs sont effectués sur des nombres de taille  $\frac{n}{2}$ . Nous avons donc à faire à une équation de récurrence aux divisions finies avec  $a = 3$  et  $b = 2$ . Ainsi,

$$T(n) = 3T\left(\frac{n}{2}\right) + f(n)$$

où  $f$  est une fonction linéaire par rapport à  $n$ .

Nous sommes dans le 1<sup>er</sup> cas du théorème général. Ainsi,  $T \in \Theta(n^{\log_2(3)}) \simeq \Theta(n^{1.58})$ .

Il s'agit de la **multiplication de Karatsuba**.

# Multiplication de matrices de grande taille

**Problème :** multiplier les matrices de grandes tailles  $A_{m \times n}$  et  $B_{n \times p}$ .

Nous allons compter, en fonction de  $n$ ,  $m$  et  $p$ , le nombre d'opérations arithmétiques pour produire le résultat. Pour simplifier, nous pouvons nous restreindre au cas où  $n = m = p$  (matrices carrées).

## ► Algorithme classique

$$C_{m \times p} = A_{m \times n} \times B_{n \times p}$$

$$c_{i,j} = \sum_{k=1}^n a_{i,k} b_{k,j}$$

Trois boucles imbriquées :  $T \in \Theta(mnp)$ .

Algorithme de complexité cubique.

► Algorithme « diviser pour régner ».

Chaque matrice de dimensions  $n \times n$  est divisée en quatre parties, chacune de dimensions  $\frac{n}{2} \times \frac{n}{2}$ .

$$A = \left( \begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right), \quad B = \left( \begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array} \right),$$

$$A \times B = C = \left( \begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array} \right),$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

## Analyse

À chaque appel récursif, la dimension  $n$  des matrices carrées à multiplier est divisée par 2. De plus, huit appels récursifs sont effectués sur des matrices de dimensions  $\frac{n}{2}$ . Nous avons donc à faire à une équation de récurrence aux divisions finies avec  $a = 8$  et  $b = 2$ . Ainsi,

$$T(n) \in 8T\left(\frac{n}{2}\right) + f(n)$$

où  $f$  est une fonction quadratique par rapport à  $n$ . En effet, les morceaux de la matrice  $C$  sont obtenus par des additions de matrices de dimensions  $\Theta(n^2)$ .

Nous avons  $c = \log_b(a) = \log_2(8) = 3$ . Nous sommes dans le 1<sup>er</sup> cas du théorème général. Ainsi  $T \in \Theta(n^3)$ .

Cette méthode n'est donc pas meilleure que la méthode directe initiale.

► Algorithme de Strassen.

Avec le même découpage, posons

$$M_1 = (A_{21} + A_{22} - A_{11}) * (B_{22} - B_{12} + B_{11})$$

$$M_2 = A_{11} * B_{11}$$

$$M_3 = A_{12} * B_{21}$$

$$M_4 = (A_{11} - A_{21}) * (B_{22} - B_{12})$$

$$M_5 = (A_{21} + A_{22}) * (B_{12} - B_{11})$$

$$M_6 = (A_{12} - A_{21} + A_{11} - A_{22}) * B_{22}$$

$$M_7 = A_{22} * (B_{11} + B_{22} - B_{12} - B_{21})$$

On a

$$C_{11} = M_2 + M_3$$

$$C_{12} = M_1 + M_2 + M_5 + M_6$$

$$C_{21} = M_1 + M_2 + M_4 - M_7$$

$$C_{22} = M_1 + M_2 + M_4 + M_5$$

## Analyse

À chaque appel récursif, la dimension  $n$  des matrices carrées à multiplier est divisée par 2. De plus, sept appels récursifs sont effectués sur des matrices de dimensions  $\frac{n}{2}$ . Nous avons donc à faire à une équation de récurrence aux divisions finies avec  $a = 7$  et  $b = 2$ . Ainsi,

$$T(n) \in 7T\left(\frac{n}{2}\right) + f(n)$$

où  $f$  est une fonction quadratique par rapport à  $n$ . En effet, les morceaux de la matrice  $C$  sont obtenus par des additions de matrices de dimensions  $\Theta(n^2)$ .

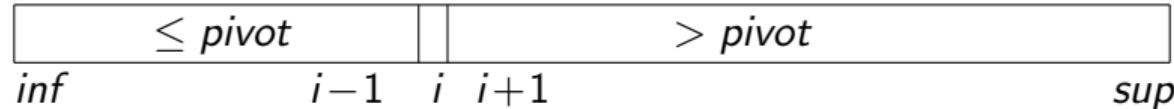
Nous avons  $c = \log_b(a) = \log_2(7) \simeq 2.81$ . Nous sommes dans le 1<sup>er</sup> cas du théorème général. Ainsi  $T \in \Theta(n^c) \simeq \Theta(n^{2.81})$ .

Cette méthode est donc bien meilleure que la méthode directe initiale et la méthode « diviser pour régner » directe. Un appel récursif de moins change beaucoup de choses.

## Quicksort

$pivot \leftarrow tab[inf]$

Partitionner :



Après :

$tab[i] = pivot$

Bien évidemment, le tableau n'est pas nécessairement séparé en deux parties de longueurs égales.

**fonction** partitionner (*tab, inf, sup*) **renvoie** indice

entrée :

*tab* : tableau d'entiers indicés de 1 à *n*

*inf* : début de la portion de tableau

*sup* : fin de la portion de tableau

sortie :

*indice* :  $tab[inf..indice] \leq tab[indice]$  et  $tab[inf..indice] < tab[indice + 1..sub]$

**début**

1      *pivot*  $\leftarrow tab[inf]$ ; *i*  $\leftarrow inf + 1$ ; *j*  $\leftarrow sup$

2      **boucle**

3          **tant que**  $i \leq j$  et  $tab[i] \leq pivot$  **faire**  $i \leftarrow i + 1$  **fin tant que**

4          **tant que**  $i \leq j$  et  $pivot < tab[j]$  **faire**  $j \leftarrow j - 1$  **fin tant que**

5          **si**  $i < j$  **alors**

6               $tab[i] \leftrightarrow tab[j]$ ; *i*  $\leftarrow i + 1$ ; *j*  $\leftarrow j - 1$

7          **fin si**

8          **tant que**  $i \leq j$

9              *indicePivot*  $\leftarrow j$ ;  $tab[inf] \leftrightarrow tab[indicePivot]$

10         **renvoyer** *indicePivot*

**fin** partitionner

**procédure** quickSort(*tab, inf, sup*)

entrée :

*tab* : tableau d'entiers indicés de 1 à *n*

*inf* : début de la portion de tableau

*sup* : fin de la portion de tableau

sortie :

*tab* : tableau d'entiers indicés de 1 à  $n$ , trié de *inf* à *sup*

début

1      si  $inf < sup$  alors

2             $indicePivot \leftarrow \text{partitionner}(tab, inf, sup)$              $\Theta(n)$

4            `quickSort(tab, indicePivot + 1, sup)`             $T(n_2)$

5 fin si

**fin quickSort**

**Meilleur cas** : le pivot sépare le tableau en deux parties de longueur similaires.

$$n_1 = n/2, n_2 = n - n_1 - 1$$

$\leq \text{pivot}$	$> \text{pivot}$
$n/2$	$n/2$

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

**Pire cas** : le pivot est une valeur minimale ou maximale du tableau.

$$n_1 = 0, n_2 = n - 1 \text{ ou } n_1 = n - 1, n_2 = 0$$

$=$	$> \text{pivot}$
1	$n - 1$

$$T(n) = T(0) + T(n - 1) + \Theta(n) \in \Theta(n^2)$$

## Statistique de rang $i$

**Problème :** Soit  $tab$  un tableau indicé de 1 à  $n$  non trié contenant des nombres, sans doublons. La statistique de rang  $i$  est la valeur dans le tableau telle que  $(i - 1)$  valeurs du tableau sont inférieures à elle.

**Exemples.** Soit  $tab = (80, 95, 70, 85, 90)$ .

- ▶ Pour  $i = 1$ , la réponse est 70.
- ▶ Pour  $i = 2$ , la réponse est 80.
- ▶ Pour  $i = 3$ , la réponse est 85.

Valeurs particulières :

$i$	Valeur renvoyée
1	minimum
$n$	maximum
$\lceil n/2 \rceil$	médiane

$T(n) =$  temps pour trouver la statistique de rang  $i$  dans un tableau de taille  $n$

► Algorithme naïf

Trier  $tab$   
**renvoyer**  $tab[i]$

**Analyse** :  $T(n, i)$  dépend de l'algorithme de tri utilisé. Si on prend le tri fusion,  
 $T(n) \in \Theta(n \log(n))$

## ► Algorithme basé sur le Quicksort

**fonction** selection (*tab, inf, sup, i*) **renvoie** entier

entrée :

*tab* : tableau d'entiers indicés de 1 à *n*

*inf* : début de la portion de tableau

*sup* : fin de la portion de tableau

*i* : le numéro du minimum voulu

antécédent :

$\text{inf} \leq i \leq \text{sup}$

**début**

```
1   si inf = sup alors
2       res  $\leftarrow$  tab[inf]
3   sinon
4       indicePivot  $\leftarrow$  partitionner (tab, inf, sup)            $\Theta(n)$ 
5       si i = indicePivot alors
6           renvoyer tab[indicePivot]
7       sinon si i < indicePivot alors
8           renvoyer selection(tab, inf, indicePivot - 1, i)       $T(n_1)$ 
9       sinon
10      renvoyer selection(tab, indicePivot + 1, sup, i)         $T(n_2)$ 
11   fin si
    fin selection
```

## Analyse

Étant donné que l'algorithme partitionner est linéaire par rapport à  $n$  et que l'algorithme selection réalise exactement un appel récursif sur un tableau de taille  $n' < n$ , nous avons

$$T(n) = T(n') + \Theta(n).$$

### Meilleur cas.

Arrive lorsque  $i = indicePivot$  est vrai immédiatement.  $T \in \Theta(n)$ .

### Pire cas.

Arrive lorsque  $n' = n - 1$ . C'est le cas le plus défavorable puisque l'espace de recherche dans le tableau reste le plus grand possible à chaque étape.  $T$  est une récurrence aux différences finies et donne  $T \in \Theta(n^2)$ .

### Cas moyen.

Arrive lorsque  $n' \simeq \frac{n}{2}$  à chaque étape.  $T$  est une équation de récurrence aux divisions finies avec  $a = 1$  (un seul sous-problème) et  $b = 2$  (en moyenne, l'espace de recherche est divisé de moitié). Par le théorème général,  $T \in \Theta(n)$ .

► Algorithme linéaire

Choix du pivot autre que  $tab[inf]$  pour avoir une meilleure partition

- 1 Séparer le tableau en groupes de 5 éléments consécutifs (sauf peut-être le dernier). Ceci donne  $\lceil n/5 \rceil$  groupes :  $nbGoupes \leftarrow \lceil n/5 \rceil$ .

$tab$

5 él.	5 él.	5 él.	5 él.	...	$\leq 5$ él.
$m_1$	$m_2$	$m_3$	$m_4$		$m_{\lceil n/5 \rceil}$

- 2 Calculer la médiane de chaque groupe.

- Pour un groupe :  $\Theta(1)$  (Tri de 5 éléments, prendre celui du milieu).
- Au total :  $\lceil n/5 \rceil \Theta(1) = \Theta(n)$

3 Placer les  $\lceil n/5 \rceil$  médianes dans un tableau de taille  $\lceil n/5 \rceil$

*tabMedIANes.*

$m_1$	$m_2$	$m_3$	$m_4$	$\cdots$	$m_{\lceil n/5 \rceil}$
-------	-------	-------	-------	----------	-------------------------

Trouver la médiane de ce tableau en utilisant l'algorithme selection :  
selection ( *tabMedIANes*, 1, *nbGroupes*,  $\lceil nbGroupes/2 \rceil$  )

4 Modifier partitionner en remplaçant  $pivot \leftarrow tab[inf]$  par  
 $pivot \leftarrow selection ( tabMedIANes, 1, nbGroupes, \lceil nbGroupes/2 \rceil )$

## Exemple

*tab, n = 31*

12 9 14 6 16 11 9 1 5 7 18 3 4 25 2 20 25 11 10 15 19 17 18 16 21 30 20 18 17 32 13

12 9 14 6 16 11 9 1 5 7 18 3 4 25 2 20 25 11 10 15 19 17 18 16 21 30 20 18 17 32 13

Tri de chaque groupe,  $nbGroupes \leftarrow \lceil 31/5 \rceil$

6	1	2	10	16	17	13
9	5	3	11	17	18	
12	7	4	15	18	20	
14	9	18	20	19	30	
16	11	25	25	21	32	

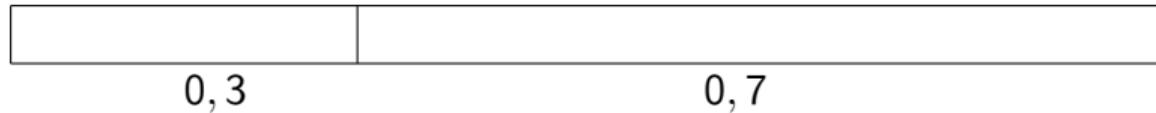
*tabMedIANES*      12    7    4    15    18    20    13

(Si *tabMedIANES* était trié : 4    7    12    13    15    18    20)

Médiane : 13  $\rightarrow$  pivot

42% des éléments sont inférieurs à 13.

On peut montrer que, au pire, on aura cette proportion pour la partition :



ou



Ceci donne

$$T(n) \leq T(0,7n) + T(0,2n) + \Theta(n) \in O(n)$$

où

- ▶  $T(0,7n)$  : Appel récursif sur la portion ( $\leq 0,7n$ ) qui contient la médiane.
- ▶  $T(0,2n)$  : Appel pour calculer le pivot = la médiane du tableau *tabMedIANes* de taille  $\lceil n/5 \rceil = 0,2n$ .
- ▶  $\Theta(n)$  : Faire la partition après avoir déterminé le pivot.

# 6. Programmation dynamique

## Principe de "diviser pour régner"

Diviser : Des problèmes de tailles plus petites sont résolus récursivement.

Régner : Les solutions des sous-problèmes sont utilisées pour former la solution du problème

La solution au problème original s'obtient de façon descendante (*top bottom*)

**Problème** : Si des sous-problèmes se chevauchent, la complexité temporelle peut devenir exponentielle.

**Exemples** : Les nombres de Fibonacci, les nombres binomiaux.

## Principe de la programmation dynamique

Diviser : On établit un schéma récursif comme dans la méthode "diviser pour régner".

Régner : Les sous-problèmes sont résolus de façon ascendante (*bottom up*) sans récursivité.

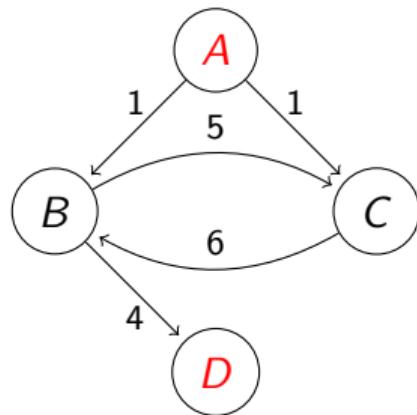
Des **tables** sont utilisées pour stocker les solutions des sous-problèmes.

Pour les problèmes d'**optimisation**, les solutions des sous-problèmes doivent être contenues dans la solution du problème.

## Définition

Le **principe d'optimalité** s'applique lorsque la solution optimale d'un problème contient toujours les solutions optimales des sous-problèmes.

**Problème :** Trouver le plus long chemin sans cycle entre deux sommets d'un graphe orienté.



Le chemin le plus long, sans cycle, de  $A$  à  $D$  est  $A - C - B - D$  de longueur 11. Le sous-chemin de la solution  $A - C$  a une longueur de 1 alors que le chemin le plus long de  $A$  à  $C$  est de 6.

## 6.1. Exemples

## Nombres binomiaux

Les nombres binomiaux sont donnés par la formule exacte

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 0 \leq k \leq n.$$



## Définition récursive :

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \text{ ou } k = n, \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{si } 0 < k < n. \end{cases}$$

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

$$\binom{n-1}{k} = \binom{n-2}{k} + \binom{n-2}{k-1}$$

$$\binom{n-2}{k} = \binom{n-3}{k} + \binom{n-3}{k-1}$$

$$\binom{n-3}{k} = \binom{n-4}{k} + \binom{n-4}{k-1}$$

$$\dots$$

**fonction** binomial (  $n$ ,  $k$  ) **retourne** entier

antécédent :  $0 \leq k \leq n$

**début**

```
1    $B[0][0] \leftarrow 1$ 
2   pour  $i \leftarrow 1$  haut  $n - 1$  faire
3        $B[i][0] \leftarrow 1$ 
4        $B[i][i] \leftarrow 1$ 
5       pour  $j \leftarrow 1$  haut  $i - 1$  faire
6            $B[i][j] \leftarrow B[i - 1][j - 1] + B[i - 1][j]$ 
7       fin pour
8   fin pour
9    $B[n][0] \leftarrow 1$ 
10   $B[n][n] \leftarrow 1$ 
11  pour  $j \leftarrow 1$  haut  $\min(n - 1, k)$  faire
12       $B[n][j] \leftarrow B[n - 1][j - 1] + B[n - 1][j]$ 
13  fin pour
14  retourner  $B[n][k]$ 
fin
```

## Multiplication chaînée de matrices

**Problème :** Trouver l'ordre dans lequel multiplier les matrices  $A_1, A_2, \dots, A_n$  afin de minimiser le nombre de multiplications scalaires. Autrement dit, où doit-on mettre des parenthèses pour minimiser le nombre de multiplications scalaires ? L'ordre des matrices doit être préservé.

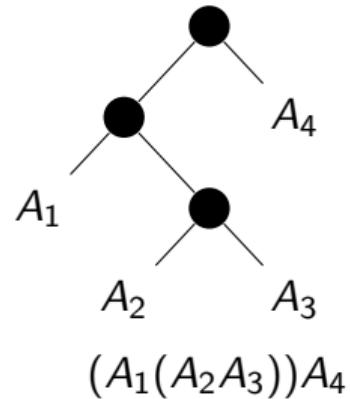
**Rappel :** Multiplier les deux matrices  $A(n \times p)$  et  $B(p \times q) = C(n \times q)$  nécessite  $n \times p \times q$  multiplications scalaires :  $c_{i,j} = \sum_{k=1}^p a_{i,k} \cdot b_{k,j}$ .

## Algorithme 1

Générer tous les parenthésages possibles et prendre celui qui donne le minimum de multiplications scalaires.

Exemple avec 4 matrices :

1.  $((A_1 A_2) A_3) A_4$
2.  $(A_1 (A_2 A_3)) A_4$
3.  $(A_1 A_2)(A_3 A_4)$
4.  $A_1 ((A_2 A_3) A_4)$
5.  $A_1 (A_2 (A_3 A_4))$



$A_1(13 \times 5)$ ,  $A_2(5 \times 89)$ ,  $A_3(89 \times 3)$  et  $A_4(3 \times 34)$

Arbre	Parenthésage	Nombre de multiplications
	$((A_1 A_2) A_3) A_4$	$A_1 A_2 : 13 \times 5 \times 89 = 5785$ $(A_1 A_2) A_3 : 5785 + 13 \times 89 \times 3 = 9256$ $((A_1 A_2) A_3) A_4 : 9256 + 13 \times 3 \times 34 = \mathbf{10582}$
	$(A_1 (A_2 A_3)) A_4$	$A_2 A_3 : 5 \times 89 \times 3 = 1335$ $A_1 (A_2 A_3) : 1335 + 13 \times 5 \times 3 = 1530$ $(A_1 (A_2 A_3)) A_4 : 1530 + 13 \times 3 \times 34 = \mathbf{2856}$
	$(A_1 A_2)(A_3 A_4)$	$A_1 A_2 : 13 \times 5 \times 89 = 5785$ $A_3 A_4 : 89 \times 3 \times 34 = 9078$ $(A_1 A_2)(A_3 A_4) : 5785 + 9078 + 13 \times 89 \times 34 = \mathbf{54201}$
	$A_1 ((A_2 A_3) A_4)$	$A_2 A_3 : 5 \times 89 \times 3 = 1335$ $(A_2 A_3) A_4 : 1335 + 5 \times 3 \times 34 = 1845$ $A_1 ((A_2 A_3) A_4) : 1845 + 13 \times 5 \times 34 = \mathbf{4055}$
	$A_1 (A_2 (A_3 A_4))$	$A_3 A_4 : 89 \times 3 \times 34 = 9078$ $A_2 (A_3 A_4) : 9078 + 5 \times 89 \times 34 = 24208$ $A_1 (A_2 (A_3 A_4)) : 24208 + 13 \times 5 \times 34 = \mathbf{26418}$

- ▶ L'arbre binaire qui représente le parenthésage des  $n$  matrices contient  $n - 1$  nœuds.
- ▶ Le nombre de parenthésages possibles de  $n$  matrices = nombre d'arbres binaires de  $n - 1$  nœuds.
- ▶ Nombre d'arbres binaires de  $n$  nœuds

$$A(n) = \sum_{i=0}^{n-1} A(i)A(n-1-i) \text{ avec } A(0) = 1, \text{ (nombres de Catalan).}$$

$n$	0	1	2	3	4	5	6	7	8	9
$A(n)$	1	1	2	5	14	42	132	429	1430	4862

$$A(n) = \frac{1}{n+1} \binom{2n}{n} \in \Theta\left(4^n/n^{3/2}\right)$$

**fonction** trouverParenthesageOptimalRec (  $p$ ,  $frontiere$ ,  $i$ ,  $j$  ) **retourne** entier

entrée :

$p$  : vecteur d'entiers positifs indicé de 0 à  $n$  donnant les dimensions des matrices

$A_i$  est de dimension  $p[i - 1] \times p[i]$

$i$  : indice de la première matrice

$j$  : indice de la dernière matrice

sortie :

$frontiere$  : matrice d'entiers positifs  $n \times n$

valeur retournée : nombre minimal de multiplications scalaires pour le produit

$A_i A_{i+1} \dots A_j$

antécédent :  $1 \leq i \leq j \leq n$

## Exemple

$A_1(13 \times 5)$ ,  $A_2(5 \times 89)$ ,  $A_3(89 \times 3)$  et  $A_4(3 \times 34)$

$i$	0	1	2	3	4
$p[i]$	13	5	89	3	34

**début**

```
1   minimum ← 0
2   si  $i \neq j$  alors
3       minimum ←  $+\infty$ 
4   pour  $k \leftarrow i$  haut  $j - 1$  faire  $(A_i A_{i+1} \dots A_k)(A_{k+1} A_{k+2} \dots A_j)$ 
5       gauche ← trouverParenthésageOptimalRec ( $p$ , frontiere,  $i$ ,  $k$ )
6       droite ← trouverParenthésageOptimalRec ( $p$ , frontiere,  $k + 1$ ,  $j$ )
7       total ← gauche + droite +  $p[i - 1] \times p[k] \times p[j]$ 
8       si  $total < minimum$  alors
9           minimum ← total
10      frontiereTemp ←  $k$ 
11      fin si
12  fin pour
13  frontiere[ $i, j$ ] ← frontiereTemp
    numéro de la dernière matrice du premier groupe qui minimise
    le nombre de multiplications scalaires
14  fin si
15  retourner  $minimum$ 
fin trouverParenthésageOptimalRec
```

## **Algorithme 2**

Utilisation de la programmation dynamique.

Les solutions des sous-problèmes seront placés dans une table M où

- ▶  $m_{i,j}$  : solution pour  $A_i A_{i+1} \dots A_j$ ,  $1 \leq i \leq j \leq n$ .
- ▶  $m_{i,i} = 0$ ,  $1 \leq i \leq n$
- ▶  $m_{i,j} = 0$ ,  $i > j$

## Calcul de $m_{i,j}$ , solution pour $A_i A_{i+1} \dots A_j$

$$\underbrace{(A_i A_{i+1} \dots A_k)}_{m_{i,k}} \underbrace{(A_{k+1} \dots A_j)}_{m_{k+1,j}}, i \leq k < j$$

- ▶  $(A_i A_{i+1} \dots A_k)$  de dimension  $p[i-1] \times p[k]$
- ▶  $(A_{k+1} \dots A_j)$  de dimension  $p[k] \times p[j]$
- ▶  $(A_i A_{i+1} \dots A_k)(A_{k+1} A_{k+2} \dots A_j)$  nécessite  $p[i-1] \times p[k] \times p[j]$  multiplications scalaires.

$$m_{i,j} = \min_{i \leq k < j} (m_{i,k} + m_{k+1,j} + p[i-1] \times p[k] \times p[j])$$

	1	2	$i$	$j$	$n$	
1	0					$m_{1,n}$
2	0	0				
	0	0	0			
$i$	0	0	0	0	$m_{i,j}$	
	0	0	0	0	0	
$j$	0	0	0	0	0	
	0	0	0	0	0	
$n$	0	0	0	0	0	0

$c - \ell = n - 1$

$c - \ell = n - 2$

$$m_{i,j} = \min_{i \leq k < j} (m_{i,k} + m_{k+1,j} + \dots)$$

$c$  = numéro de colonne

$\ell$  = numéro de ligne

$c - \ell = 2$

$c - \ell = 1$

$c - \ell = 0$

- ▶ Construire la table M diagonale par diagonale.
- ▶ Commencer par la plus longue diagonale.
- ▶ Éléments de la diagonale  $s$  : solutions optimales pour la suite de matrices consécutives  $A_i A_{i+1} \dots A_j$  où  $j - i = s$ .
- ▶ La diagonale  $s$  : éléments  $m_{i,j}$  tels que  $j - i = s$ ,  $0 \leq s \leq n - 1$ .
- ▶  $m_{i,i} = 0$ ,  $1 \leq i \leq n$ .
- ▶  $m_{i,j} = \min_{i \leq k < j} (m_{i,k} + m_{k+1,j} + p[i-1] \times p[k] \times p[j])$

**procédure** trouverParenthésageOptimal (  $n, p, m, frontiere$  )

entrée :

$n$  : entier positif

$p$  : vecteur d'entiers positifs indicé de 0 à  $n$

sortie :

$m$  : matrice d'entiers positifs  $n \times n$

$frontiere$  : matrice d'entiers positifs  $n \times n$

```

début
1   pour  $i \leftarrow 1$  haut  $n$  faire
2      $m[i, i] \leftarrow 0$ 
3   fin pour
4   pour  $s \leftarrow 1$  haut  $n - 1$  faire
5      $s + 1$  correspond au nombre de matrices à multiplier =  $j - i + 1$ 
6     pour  $i \leftarrow 1$  haut  $n - s$  faire
7        $j \leftarrow i + s$ 
8       calcul de  $m[i, j]$  correspondant au produit des  $s + 1$  matrices :  $A_i A_{i+1} \dots A_j$ 
9       minimum  $\leftarrow +\infty$ 
10      pour  $k \leftarrow i$  haut  $j - 1$  faire
11        temp  $\leftarrow m[i, k] + m[k + 1, j] + p[i - 1] \times p[k] \times p[j]$ 
12        si temp < minimum alors
13          minimum  $\leftarrow$  temp
14          frontiereTemp  $\leftarrow k$ 
15        fin si
16      fin pour
17       $m[i, j] \leftarrow$  minimum
18      frontiere[i, j]  $\leftarrow$  frontiereTemp
19    fin pour
20  fin pour
fin trouverParenthésageOptimal

```

## Analyse

$T(n)$  est relié au nombre de fois où la ligne 12 est exécutée.

$$T(n) = \sum_{s=1}^{n-1} \sum_{i=1}^{n-s} \sum_{k=i}^{j-1} 1 = \sum_{s=1}^{n-1} \sum_{i=1}^{n-s} j - i = \sum_{s=1}^{n-1} \sum_{i=1}^{n-s} s =$$

## Arbre binaire de recherche optimal

- ▶ **Cas dynamique** : On construit l'arbre au fur et à mesure de l'insertion des données. On peut utiliser un arbre AVL ou un arbre rouge-noir. La probabilité de recherche de chacune des clés est supposée équiprobable.
- ▶ **Cas statique** : On connaît toutes les clés à insérer ainsi que leur probabilité de recherche (ex. : mots réservés en Java). On construit l'arbre en tenant compte des probabilités de recherche.

**Problème** : Trouver l'arbre binaire de recherche qui minimise le temps moyen de comparaisons lors d'une recherche fructueuse dans le cas statique pour les clés  $x_1, x_2, \dots, x_n$  où  $x_1 < x_2 < \dots < x_n$ .

On pose

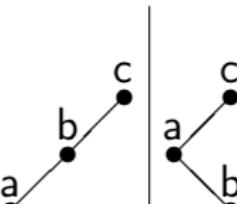
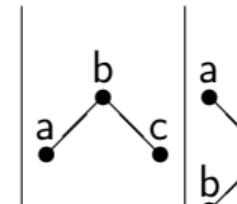
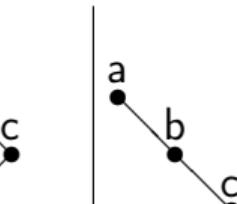
- ▶  $C_{1,n}$  : Nombre moyen de comparaisons pour une recherche fructueuse pour les clés  $x_1, x_2, \dots, x_n$ .
- ▶  $d_i$  : Profondeur (niveau) de  $x_i$  dans l'arbre.
- ▶  $p_i$  : Probabilité de chercher  $x_i$

On a

$$C_{1,n} = \sum_{i=1}^n p_i(d_i + 1), \quad \sum_{i=1}^n p_i = 1.$$

Le problème revient à trouver les valeurs  $d_i$  telles que  $C_{1,n}$  est minimal.

## Exemples

a	b	c					
$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	2	2	1,67	2	2
0,25	0,3	0,45	1,8	1,85	1,7	2,05	2,2
0,35	0,2	0,45	1,9	1,75	1,8	1,85	2,1

- ▶ Cas équiprobable :  $p_i = \frac{1}{n}, 1 \leq i \leq n$

$$C_{1,n} = \sum_{i=1}^n p_i(d_i + 1) = \frac{1}{n} \sum_{i=1}^n (d_i + 1) = 1 + \frac{1}{n} \sum_{i=1}^n d_i$$

- ▶  $2^{p-1} \leq n < 2^p$
- ▶ Hauteur minimale d'un arbre binaire contenant  $n$  nœuds est  $p = \lfloor \log_2 n \rfloor \rightarrow$  Tous les niveaux sont remplis sauf possiblement le dernier.
- ▶ Le nombre de nœuds au niveau  $k$  dans cet arbre est  $2^k \rightarrow$  la valeur  $d$  pour ces nœuds est  $k$
- ▶  $\sum_{k=0}^{\lfloor \log_2 n \rfloor - 1} k2^k \leq \sum_{i=1}^n d_i \leq \sum_{k=0}^{\lfloor \log_2 n \rfloor} k2^k$
- ▶ On peut montrer que cet arbre minimise la valeur de  $C_{1,n}$

## ► Cas non-équiprobable

### Algorithme 1

Générer tous les arbres possibles et prendre celui qui donne la valeur minimale pour  $C$ .

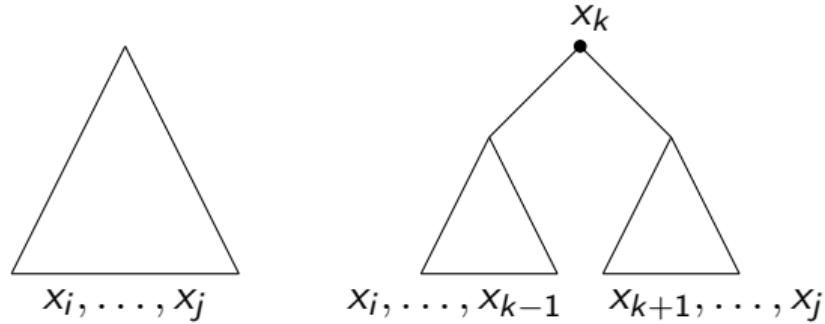
**Analyse** : Pour chacun des  $\frac{1}{n+1} \binom{2n}{n} \in \Theta(4^n/n^{3/2})$  arbres possibles on calcule  $C$  qui nécessite  $\Theta(n)$  opérations. On aura au total  $T(n) \in \Theta(4^n/n^{1/2})$ .

## Algorithme 2

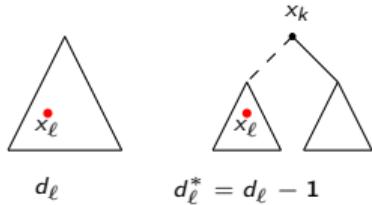
Utilisation de la programmation dynamique.

Établir le schéma récursif :

$C_{i,j}$  : Nombre moyen de comparaisons pour une recherche fructueuse pour les clés  $x_i, x_{i+1}, \dots, x_j$ .



$$i \leq k \leq j$$



$$\begin{aligned}
 C_{ij} &= \sum_{\ell=i}^j p_\ell(d_\ell + 1) = \sum_{\ell=i}^{k-1} p_\ell(d_\ell + 1) + p_k(d_k + 1) + \sum_{\ell=k+1}^j p_\ell(d_\ell + 1) \\
 &= \sum_{\ell=i}^{k-1} (\textcolor{blue}{p_\ell d_\ell} + \textcolor{red}{p_\ell}) + \textcolor{red}{p_k} + \sum_{\ell=k+1}^j (\textcolor{blue}{p_\ell d_\ell} + \textcolor{red}{p_\ell}) \\
 &= \sum_{\ell=i}^{k-1} \textcolor{blue}{p_\ell(d_\ell^* + 1)} + \sum_{\ell=i}^{k-1} \textcolor{red}{p_\ell} + \textcolor{red}{p_k} + \sum_{\ell=k+1}^j \textcolor{blue}{p_\ell(d_\ell^* + 1)} + \sum_{\ell=k+1}^j \textcolor{red}{p_\ell} \\
 &= C_{i,k-1} + C_{k+1,j} + \sum_{\ell=i}^j \textcolor{red}{p_\ell}.
 \end{aligned}$$

**Calcul de  $c_{i,j}$ , solution pour  $x_i, x_{i+1}, \dots, x_j$**

$$C_{ij} = \begin{cases} p_i & \text{si } i = j, \\ \sum_{\ell=i}^j p_\ell + \min_{i \leq k \leq j} \{C_{i,k-1} + C_{k+1,j}\} & \text{si } i < j. \end{cases}$$

	0	1	$i - 1$	$j$	$n$	
1	0	$p_1$				$c_{1,n}$
2		0	$p_2$			
$i$			0		$c_{i,j}$	$c_{i,j} = \min_{i \leq k \leq j} (c_{i,k-1} + c_{k+1,j} + \dots)$
$j + 1$				0		$c = \text{numéro de colonne}$ $\ell = \text{numéro de ligne}$
$n + 1$					0	$c - \ell = 1$ $c - \ell = 0$ $c - \ell = -1$

**procédure** arbreOptimal ( *p*, *r*, *C*, *racine* )

entrée :

*p* : vecteur des probabilités indicé de 1 à *n*

*r* : matrice telle que  $r_{ij} = \sum_{k=i}^j p_k$

sortie :

*C* : matrice donnant les espérances minimales des temps de recherche,  
lignes indicées de 1 à *n* + 1 et colonnes indicées de 0 à *n*

*racine* : matrice permettant de retrouver l'arbre optimal

**début**

1   **pour** *i*  $\leftarrow$  1 **haut** *n* **faire**

2      $C_{ii} \leftarrow p_i$

3      $C_{i,i-1} \leftarrow 0$

4      $racine_{ii} \leftarrow i$

5   **fin pour**

6      $C_{n+1,n} \leftarrow 0$

```

7   pour longueur  $\leftarrow 2 haut n faire           longueur =  $c - \ell$ 
8     pour i  $\leftarrow 1$  haut  $n - (\text{longueur} - 1)$  faire
9       j  $\leftarrow i + (\text{longueur} - 1)$ 
10      min  $\leftarrow +\infty$ 
11      pour k  $\leftarrow i$  haut j faire
12        temp  $\leftarrow C_{i,k-1} + C_{k+1,j}$ 
13        si temp < min alors
14          min  $\leftarrow \text{temp}$ 
15          rac  $\leftarrow k$ 
16        fin si
17      fin pour
18       $C_{i,j} \leftarrow min + r_{i,j}$ 
19       $racine_{ij} \leftarrow rac$ 
20    fin pour
21  fin pour
  fin$ 
```

## Analyse

$T(n)$  est relié au nombre de fois où le **si** de la ligne 13 est exécuté.

$$T(n) =$$

## Sous-suite commune

- ▶ Alphabet  $\Sigma$  (sigma) : Ensemble de symboles possibles.
- ▶ Deux suites

$$X = (x_1, x_2, \dots, x_m), m \geq 0, x_i \in \Sigma, 1 \leq i \leq m$$

$$Y = (y_1, y_2, \dots, y_n), n \geq 0, y_i \in \Sigma, 1 \leq i \leq n$$

**Problème :**

Trouver une sous-suite commune à  $X$  et  $Y$  de longueur maximale

$$Z = (z_1, z_2, \dots, z_k), 0 \leq k \leq \min(m, n) \text{ où}$$

$$z_1 = x_{i_1} = y_{j_1}, z_2 = x_{i_2} = y_{j_2}, \dots, z_k = x_{i_k} = y_{j_k}$$

$$1 \leq i_1 < i_2 < \dots < i_k \leq m$$

$$1 \leq j_1 < j_2 < \dots < j_k \leq n.$$

**Exemple** :  $X = (A, B, C, B, D, A, B)$  et  $Y = (B, D, C, A, B, A)$ .

$Z$  possibles :

- ▶ (),
- ▶ (A), (B), (C), (D),
- ▶ (A, A), (A, B), (B, A), (B, B), (B, C), (B, D), (C, A), (C, B), (D, A), (D, B),
- ▶ (A, B, A), (B, A, B), (B, B, A), (B, C, A), (B, C, B), (B, D, A), (B, D, B),  
(C, A, B), (C, B, A), (D, A, B),
- ▶ (B, C, A, B), (B, C, B, A), (B, D, A, B).

## Notation

- ▶ Préfixe de longueur  $k$  de  $X$ ,  $0 \leq k \leq m$  :  $X_k = (x_1, x_2, \dots, x_k)$
- ▶  $X = X_m$
- ▶  $X_0 = ()$  (suite vide)

## Exemples

## Propriétés de la SCLM

Soient  $X = (x_1, x_2, \dots, x_m)$ ,  $m \geq 0$ ,  $Y = (y_1, y_2, \dots, y_n)$ ,  $n \geq 0$  et soit  $Z = (z_1, z_2, \dots, z_k)$  une SCLM.

**si**  $x_m = y_n$  **alors**

$z_k = x_m = y_n$  et  $Z_{k-1}$  est une SCLM de  $X_{m-1}$  et  $Y_{n-1}$

**sinon**

**si**  $z_k \neq x_m$  **alors**

$Z$  est une SCLM de  $X_{m-1}$  et  $Y$

**fin si**

**si**  $z_k \neq y_n$  **alors**

$Z$  est une SCLM de  $X$  et  $Y_{n-1}$

**fin si**

**fin si**

## Schéma récursif de la solution

$L_{i,j}$  : longueur maximale de la SCLM pour les suites  $X_i$  et  $Y_j$

$$L_{i,j} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0, \\ 1 + L_{i-1,j-1} & \text{si } i > 0 \text{ et } j > 0 \text{ et } x_i = y_j, \\ \max(L_{i-1,j}, L_{i,j-1}) & \text{si } i > 0 \text{ et } j > 0 \text{ et } x_i \neq y_j. \end{cases}$$

Solution pour  $X$  et  $Y$  :  $L_{m,n}$

## Algorithme 1

**fonction** longueurMax (  $x, y, i, j$  ) **retourne** entier

entrée :

$x$  : suite de symboles numérotés de 1 à  $m$

$y$  : suite de symboles numérotés de 1 à  $n$

$i$  : entier tel que  $0 \leq i \leq m$

$j$  : entier tel que  $0 \leq j \leq n$

**début**

1      **si**  $i = 0$  **ou**  $j = 0$  **alors**

2           $res \leftarrow 0$

3          **sinon**  $i = y_j$  **alors**

4             $res \leftarrow 1 + \text{longueurMax} ( x, y, i - 1, j - 1 )$

5          **sinon**

6             $res \leftarrow \max ( \text{longueurMax} ( x, y, i - 1, j ), \text{longueurMax} ( x, y, i, j - 1 ) )$

7          **fin si**

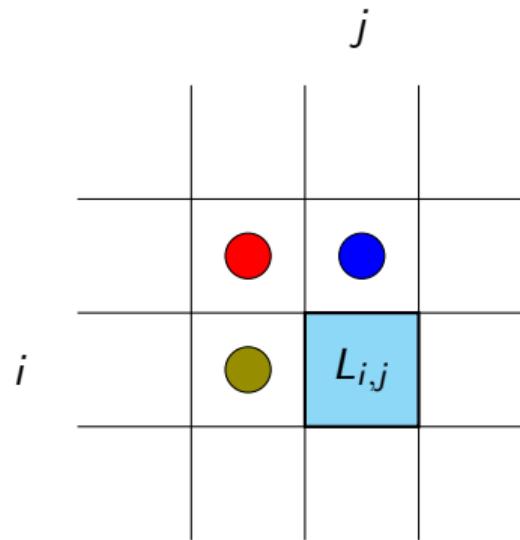
8          **retourner**  $res$

**fin** longueurMax

## Algorithme 2

Utilisation de la programmation dynamique.

$$L[i, j] = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0, \\ 1 + L[i - 1, j - 1] & \text{si } i > 0 \text{ et } j > 0 \text{ et } x_i = y_j, \\ \max(L[i - 1, j], L[i, j - 1]) & \text{si } i > 0 \text{ et } j > 0 \text{ et } x_i \neq y_j. \end{cases}$$



**fonction** trouverLongueurMax (  $x, y$  ) **retourne** entier

entrée :

$x$  : suite de symboles numérotés de 1 à  $m$

$y$  : suite de symboles numérotés de 1 à  $n$

**début**

1      **pour**  $i \leftarrow 0$  **haut**  $m$  **faire**

2         $L[i, 0] \leftarrow 0$

3      **fin pour**

4      **pour**  $j \leftarrow 0$  **haut**  $n$  **faire**

5         $L[0, j] \leftarrow 0$

6      **fin pour**

7      **pour**  $i \leftarrow 1$  **haut**  $m$  **faire**

8        **pour**  $j \leftarrow 1$  **haut**  $n$  **faire**

9            **si**  $x_i = y_j$  **alors**

10             $L[i, j] \leftarrow L[i - 1, j - 1] + 1$

11            **sinon si**  $L[i - 1, j] \geq L[i, j - 1]$  **alors**

12             $L[i, j] \leftarrow L[i - 1, j]$

13            **sinon**

14             $L[i, j] \leftarrow L[i, j - 1]$

15            **fin si**

16        **fin pour**

17      **fin pour**

18      **retourner**  $L[m, n]$

**fin** trouverLongueurMax

**procedure** trouverLongueurMaxPlus (  $x, y, L, V$  ) **retourne** entier

entrée :

$x$  : suite de symboles numérotés de 1 à  $m$

$y$  : suite de symboles numérotés de 1 à  $n$

sortie :

$L$  : matrice des longueurs maximales

$V$  : matrice des directions maximales

**début**

```
1   pour  $i \leftarrow 0$  haut  $m$  faire
2      $L[i, 0] \leftarrow 0$ ;  $V[i, 0] \leftarrow \text{ouest}$ 
3   fin pour
4   pour  $j \leftarrow 0$  haut  $n$  faire
5      $L[0, j] \leftarrow 0$ ;  $V[0, j] \leftarrow \text{ouest}$ 
6   fin pour
7   pour  $i \leftarrow 1$  haut  $m$  faire
8     pour  $j \leftarrow 1$  haut  $n$  faire
9       si  $x_i = y_j$  alors
10          $L[i, j] \leftarrow L[i - 1, j - 1] + 1$ ;  $V[i, j] \leftarrow \text{nord} - \text{ouest}$ 
11       sinon si  $L[i - 1, j] \geq L[i, j - 1]$  alors
12          $L[i, j] \leftarrow L[i - 1, j]$ ;  $V[i, j] \leftarrow \text{nord}$ 
13       sinon
14          $L[i, j] \leftarrow L[i, j - 1]$ ;  $V[i, j] \leftarrow \text{ouest}$ 
15       fin si
16     fin pour
17   fin pour
fin trouverLongueurMaxPlus
```

**fonction** obtenirSCLM (  $V, x, i, j$  ) **retourne** suite

entrée :

$V$  : matrice

$x$  : suite de symboles

$i$  : position dans une suite de symboles

$j$  : position dans une suite de symboles

**début**

1      **si**  $i = 0$  **ou**  $j = 0$  **alors**

2           $res \leftarrow$  suite vide

3      **sinon si**  $V[i, j] = nord - ouest$  **alors**

4           $res \leftarrow$  obtenirSCLM (  $V, x, i - 1, j - 1$  ) &  $x_i$       Concaténation

5      **sinon si**  $V[i, j] = nord$  **alors**

6           $res \leftarrow$  obtenirSCLM (  $V, x, i - 1, j$  )

7      **sinon**

8           $res \leftarrow$  obtenirSCLM (  $V, x, i, j - 1$  )

9      **fin si**

10     **retourner**  $res$

**fin** obtenirSCLM

**Analyse :**

## Exemple

Tableau  $L$

$X$	$Y$		$a$	$x$	$y$	$b$	$s$	$u$	$c$	$d$
		0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
$x$	1	0	0	1	1	1	1	1	1	1
$a$	2	0	1	1	1	1	1	1	1	1
$b$	3	0	1	1	1	2	2	2	2	2
$z$	4	0	1	1	1	2	2	2	2	2
$c$	5	0	1	1	1	2	2	2	3	3
$u$	6	0	1	1	1	2	2	3	3	3
$s$	7	0	1	1	1	2	3	3	3	3
$d$	8	0	1	1	1	2	3	3	3	4
$x$	9	0	1	2	2	2	3	3	3	4
$i$	10	0	1	2	2	2	3	3	3	4

Tableau V

$X$	$Y$		$a$	$x$	$y$	$b$	$s$	$u$	$c$	$d$
		$0$	$1$	$2$	$3$	$4$	$5$	$6$	$7$	$8$
	$0$	0	0	0	0	0	0	0	0	0
$x$	$1$	0	1	2	0	0	0	0	0	0
$a$	$2$	0	2	1	1	1	1	1	1	1
$b$	$3$	0	1	1	1	2	0	0	0	0
$z$	$4$	0	1	1	1	1	1	1	1	1
$c$	$5$	0	1	1	1	1	1	1	2	0
$u$	$6$	0	1	1	1	1	1	2	1	1
$s$	$7$	0	1	1	1	1	2	1	1	1
$d$	$8$	0	1	1	1	1	1	1	1	2
$x$	$9$	0	1	2	0	1	1	1	1	1
$i$	$10$	0	1	1	1	1	1	1	1	1

Tableau V

$0 \leftarrow$        $1 \uparrow$        $2 \nwarrow$

$X$	$Y$	$a$	$x$	$y$	$b$	$s$	$u$	$c$	$d$
$X$	0	0	0	0	0	0	0	0	0
$x$	1	0	1	<b>2</b>	<b>0</b>	0	0	0	0
$a$	2	0	2	1	<b>1</b>	1	1	1	1
$b$	3	0	1	1	1	<b>2</b>	<b>0</b>	<b>0</b>	0
$z$	4	0	1	1	1	1	<b>1</b>	1	1
$c$	5	0	1	1	1	1	1	<b>2</b>	0
$u$	6	0	1	1	1	1	<b>2</b>	<b>1</b>	1
$s$	7	0	1	1	1	2	1	<b>1</b>	1
$d$	8	0	1	1	1	1	1	1	<b>2</b>
$x$	9	0	1	2	0	1	1	1	<b>1</b>
$i$	10	0	1	1	1	1	1	1	<b>1</b>

## Distance d'édition (Distance de Levenshtein)

Nombre d'opérations élémentaires requises pour transformer la première suite  $X$  en la deuxième  $Y$ . Opérations élémentaires :

- ▶  $\text{Del}(a)$  : suppression<sup>1</sup> d'une lettre de  $X$  à une position donnée.
- ▶  $\text{Ins}(a)$  : insertion d'une lettre de  $Y$  dans  $X$  à une position donnée.
- ▶  $\text{Sub}(a, b)$  substitution d'une lettre de  $X$  à une position donnée par une lettre de  $Y$ .
  
- ▶  $\text{Del}(a) = \text{Ins}(a) = 1$
- ▶  $\text{Sub}(a, b) = \delta_{[a \neq b]}$ <sup>2</sup>

---

1. En biologie et en bioinformatique on parle plutôt de délétion qui dénote la perte d'un fragment de chromosome.

2.  $\delta_{[b]}$  désigne une variante du delta de Kronecker et s'évalue à 1 si  $b$  est vrai, 0 sinon.

## Exemple

"plusieurs" → "malheur"

$X$	p		u	s	i	e	u	r	s
$Y$	m	a		h		e	u	r	
Opération	S	I	S	S	D	D	S	S	S
Coût	1	1	0	1	1	1	0	0	0

$S = Sub, I = Ins, D = Del$

## Schéma récursif

- ▶ Alphabet :  $\Sigma$
- ▶  $\Sigma^*$  : toutes les suites possibles formées avec les symboles de  $\Sigma$
- ▶ Suite vide :  $\varepsilon \in \Sigma^*$ .
- ▶  $a \in \Sigma$ ,  $b \in \Sigma$ ,  $U \in \Sigma^*$  et  $V \in \Sigma^*$
- ▶  $\text{Lev}(U, V)$  : distance minimale d'édition de  $U$  et  $V$ .

$$\text{Lev}(Ua, \varepsilon) = \text{Lev}(U, \varepsilon) + \text{Del}(a),$$

$$\text{Lev}(\varepsilon, Vb) = \text{Lev}(\varepsilon, V) + \text{Ins}(b),$$

$$\text{Lev}(Ua, Vb) = \min \begin{cases} \text{Lev}(U, V) + \text{Sub}(a, b), \\ \text{Lev}(U, Vb) + \text{Del}(a), \\ \text{Lev}(Ua, V) + \text{Ins}(b). \end{cases}$$

$$d_{ij} = \begin{cases} 0 & \text{si } i = 0 \text{ et } j = 0, \\ d_{i-1,0} + \text{Del}(x_i) & \text{si } i > 0 \text{ et } j = 0, \\ d_{0,j-1} + \text{Ins}(y_j) & \text{si } i = 0 \text{ et } j > 0, \\ \min \begin{cases} d_{i-1,j-1} + \text{Sub}(x_i, y_j), \\ d_{i-1,j} + \text{Del}(x_i), \\ d_{i,j-1} + \text{Ins}(y_j). \end{cases} & \text{sinon.} \end{cases}$$

**fonction** distanceEdition (  $x, y$  ) **retourne** entier

entrée :

$x$  : suite de  $m$  symboles

$y$  : suite de  $n$  symboles

**début**

```
1    $D[0, 0] \leftarrow 0$ 
2   pour  $i \leftarrow 1$  haut  $m$  faire
3        $D[i, 0] \leftarrow D[i - 1, 0] + Del(x_i)$ 
4   fin pour
5   pour  $j \leftarrow 1$  haut  $n$  faire
6        $D[0, j] \leftarrow D[0, j - 1] + Ins(y_j)$ 
7   pour  $i \leftarrow 1$  haut  $m$  faire
8        $D[i, j] \leftarrow \min(D[i - 1, j - 1] + Sub(x_i, y_j),$ 
9            $D[i - 1, j] + Del(x_i), D[i, j - 1] + Ins(y_j))$ 
10  fin pour
11  fin pour
12  retourner  $D[m, n]$ 
13 fin distanceEdition
```

**Analyse :**

## Exemple

"plusieurs" → "malheur"

$X$	p		u	s	i	e	u	r	s
$Y$	m	a		h		e	u	r	
Opération	S	I	T	S	D	D	T	T	T
Coût	1	1	0	1	1	1	0	0	0

- ▶  $S = Sub(a, b)$  avec  $a \neq b$  (coût 1)
- ▶  $T = Sub(a, a)$  (coût 0)
- ▶  $I = Ins$
- ▶  $D = Del$

**fonction** distanceEdition (  $x, y$  ) **retourne** entier

entrée :

$x$  : suite de  $m$  symboles

$y$  : suite de  $n$  symboles

**début**

```
1       $D[0, 0] \leftarrow 0$ 
2      pour  $i \leftarrow 1$  haut  $m$  faire
3           $D[i, 0] \leftarrow D[i - 1, 0] + Del(x_i)$ 
4      fin pour
5      pour  $j \leftarrow 1$  haut  $n$  faire
6           $D[0, j] \leftarrow D[0, j - 1] + Ins(y_j)$ 
7          pour  $i \leftarrow 1$  haut  $m$  faire
8               $D[i, j] \leftarrow \min(D[i - 1, j - 1] + Sub(x_i, y_j),$ 
                   $D[i - 1, j] + Del(x_i), D[i, j - 1] + Ins(y_j))$ 
                  V[i, j] = S, T, D ou I selon le minimum
9      fin pour
10     fin pour
11     retourner  $D[m, n]$ 
fin distanceEdition
```

Tableau  $D$ 

$X$	$Y$	$m$	$a$	$l$	$h$	$e$	$u$	$r$	
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
$p$	1	1	1	2	3	4	5	6	7
$l$	2	2	2	2	3	4	5	6	
$u$	3	3	3	3	3	4	4	5	
$s$	4	4	4	4	4	4	5	5	
$i$	5	5	5	5	5	5	5	6	
$e$	6	6	6	6	6	5	6	6	
$u$	7	7	7	7	7	6	5	6	
$r$	8	8	8	8	8	7	6	5	
$s$	9	9	9	9	9	8	7	6	

Tableau V

$Y$	$m$	$a$	$l$	$h$	$e$	$u$	$r$
$X$	1	2	3	4	5	6	7
$p$	S	I	I	I	I	I	I
$l$	D	S	T	I	I	I	I
$u$	D	D	D	S	I	T	I
$s$	D	D	D	D	S	I	S
$i$	D	D	D	D	D	S	I
$e$	D	D	D	D	T	I	S
$u$	D	D	D	D	D	T	I
$r$	D	D	D	D	D	D	T
$s$	D	D	D	D	D	D	D

Tableau V

 $I \leftarrow D \uparrow S, T \nwarrow$ 

$Y$	$m$	$a$	$l$	$h$	$e$	$u$	$r$
$X$	1	2	3	4	5	6	7
$p$	S	I	I	I	I	I	I
$l$	D	S	T	I	I	I	I
$u$	D	D	D	S	I	T	I
$s$	D	D	D	D	S	I	S
$i$	D	D	D	D	D	S	I
$e$	D	D	D	D	T	I	S
$u$	D	D	D	D	D	T	I
$r$	D	D	D	D	D	D	T
$s$	D	D	D	D	D	D	D

## Voyageur de commerce

**Problème :** Soit  $N = \{1, 2, \dots, n\}$ , un ensemble de villes. Trouver un chemin de la ville 1, passant par toutes les autres villes une seule fois avant de revenir à la ville 1 et ce, de façon optimale (chemin de coût minimal).

On pose :

- ▶  $L_{i,j} = \text{coût de l'arc}(i,j)$ ,
- ▶  $L_{i,j} \geq 0$ ,
- ▶  $L_{i,j} = +\infty$  s'il n'y a pas d'arc entre le sommet  $i$  et le sommet  $j$ .

## **Algorithme 1**

Générer tous les chemins possibles et prendre celui qui a le coût minimal.

**Analyse :** Il y a  $(n - 1)!$  chemins possibles. Pour chacun, on calcule le coût. Le calcul nécessite  $\Theta(n)$  opérations. Au total, la complexité temporelle de l'algorithme est dans  $\Theta(n!)$

## Algorithme 2

Utilisation de la programmation dynamique.

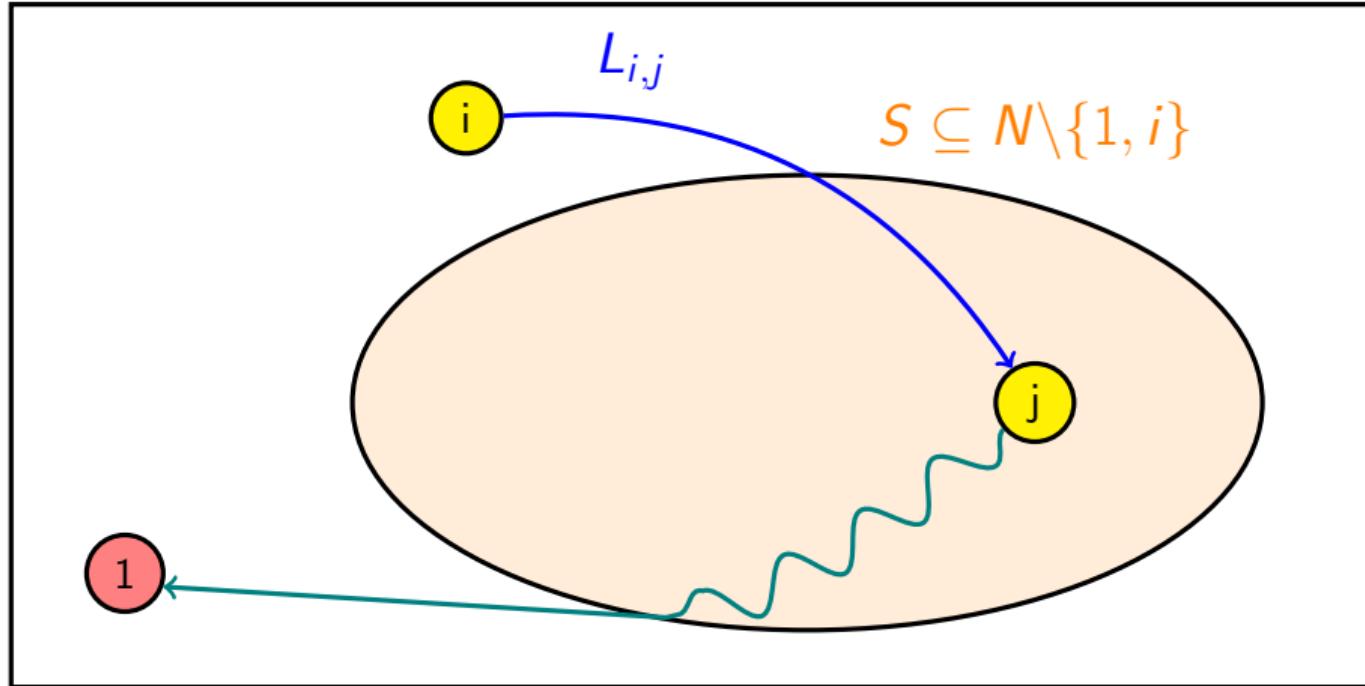
- ▶  $G(i, S) = \text{coût du chemin le moins coûteux de la ville } i \text{ à la ville 1 en passant une et une seule fois par toutes les villes de } S \text{ où } S \subseteq N \setminus \{1, i\}$
- ▶  $G(i, \emptyset) = L_{i,1}$ .
- ▶ Solution :  $G(1, N \setminus \{1\})$  où

$$G(1, N \setminus \{1\}) = \min_{2 \leq j \leq n} (L_{1,j} + G(j, N \setminus \{1, j\})) \quad (1)$$

$$G(i, S) = \min_{j \in S} (L_{i,j} + G(j, S \setminus \{j\})) \quad (2)$$

$$G(i, S) = \min_{j \in S} (L_{i,j} + G(j, S \setminus \{j\}))$$

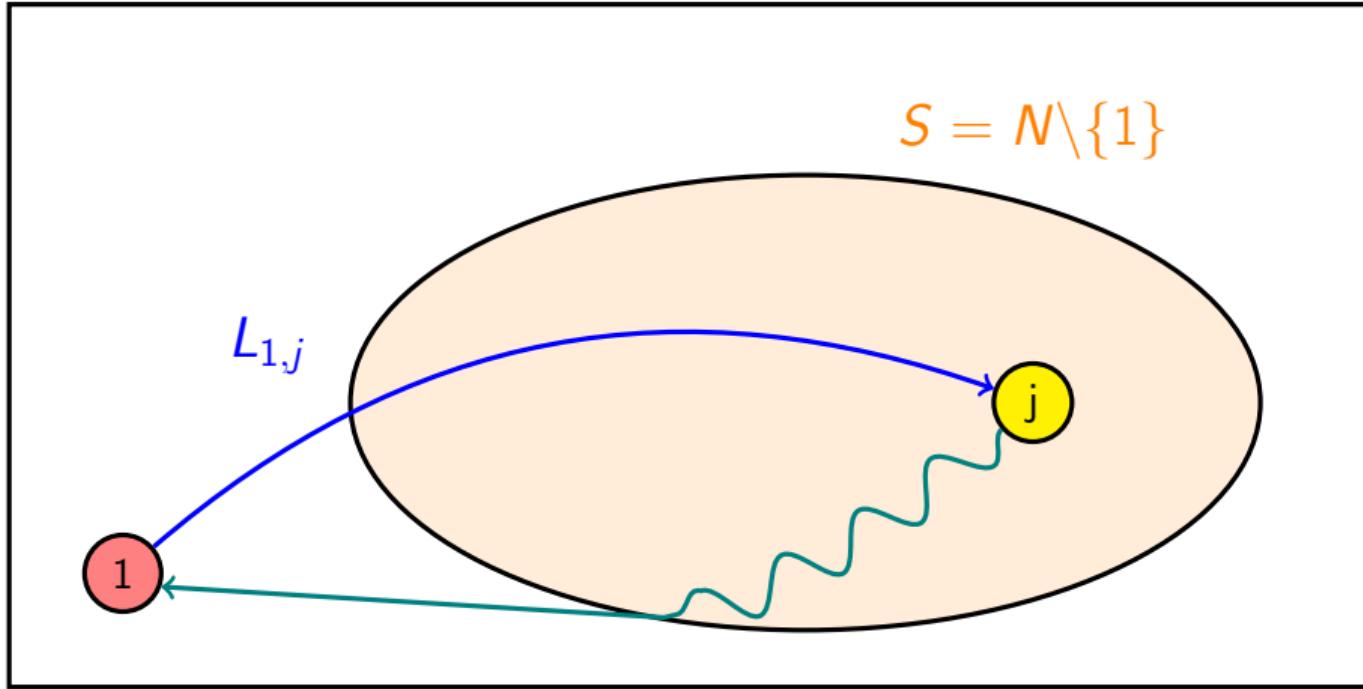
N



$$G(1, N \setminus \{1\}) = \min_{2 \leq j \leq n} (L_{1,j} + G(j, N \setminus \{1, j\}))$$

N

$$S = N \setminus \{1\}$$



- ▶ On génère tous les sous-ensembles possibles  $S$  de l'ensemble  $\{2, 3, \dots, n\}$
- ▶ Pour les sous-ensembles  $S$  de cardinalité inférieure à  $n - 1$ , on utilise l'équation (2)
- ▶ Pour le seul sous-ensemble  $S = \{2, 3, \dots, n\}$  de cardinalité  $n - 1$ , on utilise l'équation (1).

**procédure** trouverChemin (  $L, n, G, J$  )

entrée

$L$  : matrice des coûts  $n \times n$

$n$  : nombre entier positif correspondant au nombre de villes

sortie

$G$  : matrice des longueurs  $n \times 2^{n-1}$

$J$  : matrice permettant de trouver les chemins optimaux

antécédent : -

conséquent :  $G$  contiendra les longueurs de chemins optimaux.

$G[1, \{2, 3, \dots, n\}]$  donnera la solution optimale pour tout le graphe.

$J$  contiendra l'information pour trouver les chemins optimaux.

$J[1, \{2, 3, \dots, n\}]$  contiendra le premier sommet à atteindre

**début**

à partir du sommet 1.

Calcul des  $G[i, S]$  pour  $|S| = 0$

1      **pour**  $i \leftarrow 2$  **haut**  $n$  **faire**

2           $G[i, \emptyset] \leftarrow L[i, 1]$

3      **fin pour**

Calcul des  $G[i, S]$ ,  $0 < |S| < n - 1$ , en ordre croissant de cardinalité de  $S$

```
4  pour card ← 1 haut n - 2 faire
  5    pour tous les sous-ensembles  $S$  de  $\{2, 3, \dots, n\}$  tels que  $|S| = card$  faire
  6      pour chaque  $i \in \{2, 3, \dots, n\} \setminus S$  faire
  7        min ←  $+\infty$ 
  8        pour chaque  $j \in S$  faire
  9          temp ←  $L[i, j] + G[j, S \setminus \{j\}]$ 
 10         si temp < min alors
 11           min ← temp
 12           jMin ← j
 13         fin si
 14       fin pour
 15        $G[i, S] \leftarrow min$ 
 16        $J[i, S] \leftarrow jMin$ 
 17     fin pour
 18   fin pour
 19 fin pour
```

## Calcul des $G[1, S]$ pour $|S| = n - 1$

```
20   min ← +∞
21   pour  $j \leftarrow 2$  haut  $n$  faire
22     temp ←  $L[1, j] + G[j, \{2, 3, \dots, n\} \setminus \{j\}]$ 
23     si temp < min alors
24       min ← temp
25       jMin ← j
26     fin si
27   fin pour
28    $G[1, \{2, 3, \dots, n\}] \leftarrow min$ 
29    $J[1, \{2, 3, \dots, n\}] \leftarrow jMin$ 
fin trouverChemin
```

# Analyse

## Complexité temporelle

- ▶ Nombre de fois où la ligne 3 est exécutée :  $n - 1$
- ▶ Nombre de sous-ensembles de taille  $card$  d'un ensemble de taille  $n - 1$  :  $\binom{n-1}{card}$
- ▶ Nombre de valeurs de  $i$  à la ligne 6 :  $n - 1 - card$
- ▶ Nombre de valeurs de  $j$  à la ligne 8 :  $card$
- ▶ Nombre de fois où la ligne 10 sera exécutée :  $card(n - 1 - card)\binom{n-1}{card} =$
- ▶ Nombre de fois où la ligne 23 est exécutée :  $n - 1$
- ▶ Au total :  $2(n - 1) + \sum_{card=1}^{n-2} card(n - 1)\binom{n-2}{card} =$

## Complexité spatiale

- ▶ Espace nécessaire pour stocker  $G$  et  $J$  : deux tableaux de nombres de  $n$  lignes et  $2^{n-1}$  colonnes.

## Remarques

- ▶ Il faut ajouter le temps requis pour aller chercher la valeur  $G[j, S]$  puisque  $S$ , le numéro de colonne, n'est pas un entier.
- ▶ Ce temps va dépendre de la représentation de  $S$ .
- ▶ On peut représenter un ensemble avec une suite de bits (0 = élément absent, 1 = élément présent)

## Exemple

$L$	1	2	3	4
1	$\infty$	10	15	20
2	5	$\infty$	9	10
3	6	13	$\infty$	12
4	8	8	9	$\infty$

L'ensemble des villes  $N = \{1, 2, 3, 4\}$ ,  $S \subseteq \{2, 3, 4\}$ .

$$|S| = 0 \quad S = \emptyset$$

$$G[2, \emptyset] = L[2, 1] = 5, \quad J[2, \emptyset] = 1$$

$$G[3, \emptyset] = L[3, 1] = 6, \quad J[3, \emptyset] = 1$$

$$G[4, \emptyset] = L[4, 1] = 8, \quad J[4, \emptyset] = 1$$

$|S| = 1$   $S = \{2\}$ ,  $S = \{3\}$ ,  $S = \{4\}$

$$G[3, \{2\}] = L[3, 2] + G[2, \emptyset] = 13 + 5 = 18, J[3, \{2\}] = 2$$

$$G[4, \{2\}] = L[4, 2] + G[2, \emptyset] = 8 + 5 = 13, J[4, \{2\}] = 2$$

$$G[2, \{3\}] = L[2, 3] + G[3, \emptyset] = 9 + 6 = 15, J[2, \{3\}] = 3$$

$$G[4, \{3\}] = L[4, 3] + G[3, \emptyset] = 9 + 6 = 15, J[4, \{3\}] = 3$$

$$G[2, \{4\}] = L[2, 4] + G[4, \emptyset] = 10 + 8 = 18, J[2, \{4\}] = 4$$

$$G[3, \{4\}] = L[3, 4] + G[4, \emptyset] = 12 + 8 = 20, J[3, \{4\}] = 4$$

$|S| = 2$   $S = \{2, 3\}$ ,  $S = \{2, 4\}$ ,  $S = \{3, 4\}$

$$G[4, \{2, 3\}] = \min(L[4, 2] + G[2, \{3\}], L[4, 3] + G[3, \{2\}]) = \\ \min(8 + 15, 9 + 18) = 23, J[4, \{2, 3\}] = 2$$

$$G[3, \{2, 4\}] = \min(L[3, 2] + G[2, \{4\}], L[3, 4] + G[4, \{2\}]) = \\ \min(13 + 18, 12 + 13) = 25, J[3, \{2, 4\}] = 4$$

$$G[2, \{3, 4\}] = \min(L[2, 3] + G[3, \{4\}], L[2, 4] + G[4, \{3\}]) = \\ \min(9 + 20, 10 + 15) = 25, J[2, \{3, 4\}] = 4$$

$$|S| = 3 \quad S = \{2, 3, 4\}$$

$$G[1, \{2, 3, 4\}] = \min(L[1, 2] + G[2, \{3, 4\}], L[1, 3] + G[3, \{2, 4\}], L[1, 4] + G[4, \{2, 3\}]) = \min(10 + 25, 15 + 25, 20 + 23) = 35, \textcolor{orange}{J[1, \{2, 3, 4\}] = 2}$$

$G$	$\emptyset$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2, 3\}$	$\{2, 4\}$	$\{3, 4\}$	$\{2, 3, 4\}$
1								35
2	5		15	18			25	
3	6	18		20		25		
4	8	13	15		23			

$J$	$\emptyset$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2, 3\}$	$\{2, 4\}$	$\{3, 4\}$	$\{2, 3, 4\}$
1								2
2	1		3	4			4	
3	1	2		4		4		
4	1	2	3		2			

Pour trouver le chemin optimal :

$$J(1, \{2, 3, 4\}) = 2$$

$$J(2, \{3, 4\}) = 4$$

$$J(4, \{3\}) = 3$$

$$J(3, \emptyset) = 1$$

Le chemin optimal est  $1 - 2 - 4 - 3 - 1$  et il est de longueur  $G(1, \{2, 3, 4\}) = 35$ .

# 7. Algorithmes gloutons

Principaux éléments d'un algorithme utilisant la stratégie gloutonne (vorace) :

- ▶ Un **ensemble de candidats** dans lequel sera puisée la solution.
- ▶ Une fonction de **sélection** qui permet de choisir le meilleur candidat à être ajouté à la solution.
- ▶ Une fonction de **faisabilité** qui vérifie si un candidat peut faire partie d'une solution.
- ▶ Une fonction **objective** qui associe une valeur à une solution totale ou partielle.
- ▶ Une fonction **solution** qui nous permet de déterminer lorsque nous sommes en présence d'une solution complète.

Caractéristiques d'un algorithme utilisant l'approche vorace :

- ▶ À chaque étape, il choisit l'élément **le plus prometteur**.
- ▶ Il **ne revient jamais en arrière**.
- ▶ S'il **rejette** un candidat, il **ne le considérera plus** par la suite.
- ▶ Il **ne donne pas toujours de solution**.
- ▶ S'il donne une solution, celle-ci n'est **pas nécessairement optimale**.

# Algorithme générique

**fonction** vorace ( *C* ) **retourne** ensemble

entrée :

*C* : Ensemble de candidats

**début**

1       $S \leftarrow \emptyset$

2      **tant que** pas de solution **et**  $C \neq \emptyset$  **faire**

3           $x \leftarrow$  élément de *C* maximisant sélect (*x*)      **sélect** (*x*) : valeur du choix de *x*

4           $C \leftarrow C - \{x\}$

5          **si** réalisable ( $S \cup \{x\}$ ) **alors**

            réalisable (*E*) : vrai si l'ensemble *E* peut faire partie d'une solution

6             $S \leftarrow S \cup \{x\}$

7            **fin si**

8          **fin tant que**

9          **si** solution **alors**

10             $res \leftarrow S$

11          **sinon**

12             $res \leftarrow \emptyset$

13          **fin si**

14          **retourner** *res*

**fin**

## 7.1. Exemples

## Petite monnaie

**Problème :** Faire la monnaie d'un montant d'argent avec un nombre minimal de pièces.

- ▶ Valeur des pièces :  $p_1 < p_2 < \dots < p_n$ ,  $p_i > 0$ ,  $1 \leq i \leq n$ .
- ▶  $S$  : Montant à remettre.  $S \geq 0$

Trouver les valeurs  $x_i \geq 0$ ,  $1 \leq i \leq n$  qui minimisent  $\sum_{i=1}^n x_i$  sous la contrainte  $\sum_{i=1}^n x_i p_i = S$ .

### Exemple

Pièces : 1 ¢, 5 ¢, 10 ¢, 25 ¢, 1 \$ et 2 \$.

Montant : 1,47 \$

## Stratégie gloutonne

- ▶ Utiliser la valeur maximale pour  $x_n$
- ▶ Utiliser la valeur maximale pour  $x_{n-1}$
- ▶ ...

## Schéma récursif

$$\text{Monnaie}(S, p, n) = \begin{cases} \emptyset & \text{si } S = 0, \\ \text{Monnaie}(S \bmod p_n, p, n - 1) \cup \{(n, \lfloor S/p_n \rfloor)\} & \text{sinon.} \end{cases}$$

## Version itérative

**fonction** Monnaie (  $S, p, n$  ) **retourne** ensemble de couples

entrée :

$S$  : Montant à remettre

$p$  : Tableau de la valeur des pièces

$n$  : Nombre de pièces.

antécédent :

$p_1 < p_2 < \dots < p_n$

**début**

```
1    $E \leftarrow \emptyset$ 
2    $i \leftarrow n$ 
3   tant que  $i > 0$  faire
4        $x \leftarrow \lfloor S/p_i \rfloor$ 
5        $S \leftarrow S \bmod p_i$ 
6        $E \leftarrow E \cup \{(i, x)\}$ 
7        $i \leftarrow i - 1$ 
8   fin tant que
9   si  $S \neq 0$  alors
10       $E \leftarrow \emptyset$ 
11   fin si
12   retourner  $E$ 
fin
```

$\Theta(n)$

## Exemples

$p$	$S$	Solution gloutonne
(1, 5, 10, 25, 100, 200)	147	(6,0), (5,1), (4,1), (3,2), (2,0), (1,2) : 6
(1, 5, 11, 25)	15	(4,0), (3,1), (2,0), (1,4) : 5 Solution optimale : (2,3) : 3
(2, 4, 5)	16	(3,3), (2,0), (1,0) Pas de solution trouvée. Solution : (2,4)

## Sac alpin (Havresac)

Soit les entiers non-négatifs, pour  $1 \leq i \leq n$

- ▶  $p_i$  : poids de l'objet  $i$
- ▶  $v_i$  : valeur de l'objet  $i$  (exemple : valeur nutritive)
- ▶  $x_i$  : nombre d'objets  $i$

**Problème :** Maximiser  $\sum_{i=1}^n v_i x_i$  sous la contrainte  $\sum_{i=1}^n p_i x_i \leq P_{max}$   
On numérote les éléments de façon telle que

$$\frac{v_1}{p_1} \leq \frac{v_2}{p_2} \leq \dots \leq \frac{v_n}{p_n}$$

## Stratégie gloutonne

- ▶ Choisir le plus grand nombre d'objets de type  $n$ , i.e. la plus grande valeur de  $x_n$  qui satisfait la contrainte.
- ▶ Choisir le plus grand nombre d'objets de type  $n - 1$ , i.e. la plus grande valeur de  $x_{n-1}$  qui satisfait la contrainte.
- ▶ etc.

```
1 pour  $i \leftarrow n$  bas 1 faire
2    $x_i \leftarrow \lfloor \frac{P_{max}}{p_i} \rfloor$ 
3    $P_{max} \leftarrow P_{max} - x_i p_i$  (ou  $P_{max} \leftarrow P_{max} \bmod p_i$ )
4 fin pour            $\Theta(n)$  (Si on inclut le tri :  $\Theta(n \log n)$ )
```

## Remarques

- ▶ L'algorithme fournit toujours une solution car la contrainte est  $\leq P_{max}$
- ▶ On peut montrer que l'algorithme ne fournit pas toujours la solution optimale mais s'en approche.

## Choix d'activités

- ▶ Une ressource
- ▶  $n$  activités (tâches)
- ▶  $d_i$  : début de la tâche  $i$ ,  $f_i$  : fin de la tâche  $i$ ,  $1 \leq i \leq n$

**Problème :** Trouver le maximum de tâches pouvant être traitées par la ressource.

- ▶ La ressource ne peut traiter qu'une tâche à la fois.
- ▶ Une tâche ne peut être interrompue.

## Définition

Les tâches  $i$  et  $j$  sont **compatibles**ssi

$$[d_i, f_i] \cap [d_j, f_j] = \emptyset$$

sinon, elles sont **incompatibles**.

## Définition

**Ensemble stable** : Sous-ensemble de tâches ne contenant pas de tâches incompatibles.

**Problème** : Trouver un ensemble stable de cardinalité maximale.

## Algorithme 1

- ▶ Générer tous les sous-ensembles possibles de l'ensemble  $\{1, 2, \dots, n\}$ .
- ▶ Pour chacun, vérifier s'il est stable.
- ▶ Retourner celui de cardinalité maximale.
- ▶ Complexité temporelle :  $\Omega(2^n)$

## **Algorithme 2**

Utilisation de la stratégie gloutonne : Parmi les tâches qui peuvent être ajoutées (compatibles) choisir celle qui se termine le plus tôt.

**fonction** choixActivites ( *d, f* ) **retourne** ensemble

entrée :

$d$  : tableau des débuts d'activités, indicé de 1 à  $n$

$f$  : tableau des fins d'activités, indicé de 1 à  $n$

antécédent :

$f$  est trié en ordre croissant ;  $d[i]$  et  $f[i]$  sont le début et la fin de la tâche  $i$ .

conséquent :

retourne un ensemble stable de cardinalité maximale

début

$$A \leftarrow \{1\}$$

*finTacheCourante*  $\leftarrow f[1]$

**pour**  $i \leftarrow 2$  à  $n$  faire

**si**  $d[i] > finTacheCourante$  **alors**

tâche  $i$  compatible avec celle de  $A$  ?

$$A \leftarrow A \cup \{i\}$$

*finTacheCourante*  $\leftarrow f[i]$

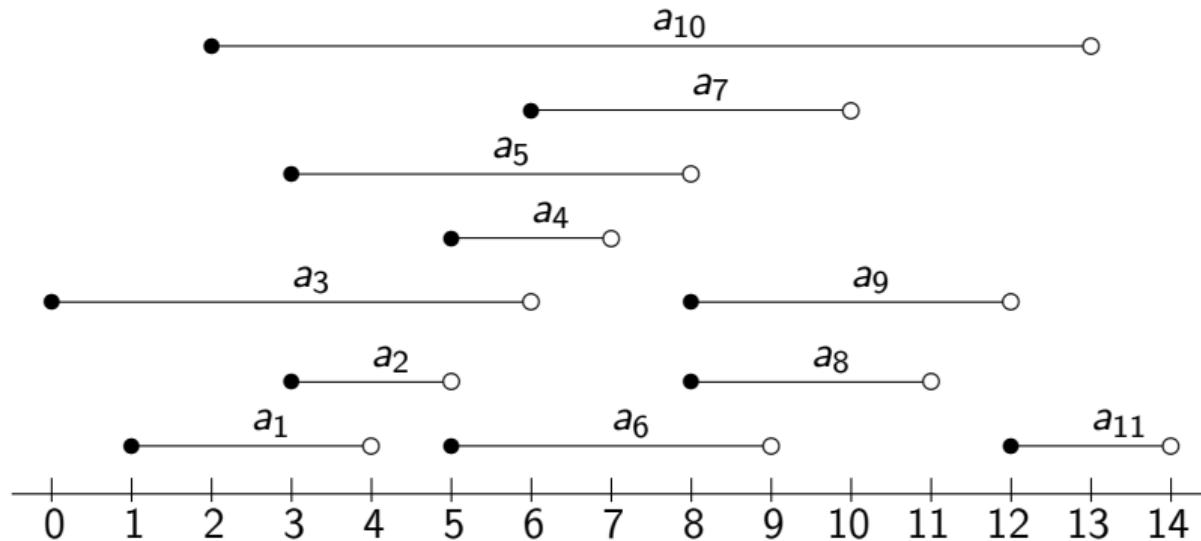
fin si

**fin pour**

**retourner A**

fin

## Exemple



# Codes de Huffman

## Codage de caractères (symboles)

- ▶ Codes de longueur fixe : ASCII (8 bits), Unicode (16 bits ou plus), etc<sup>3</sup>.
- ▶ Codes de longueur variable : Code morse, codes préfixes.

Dans un code préfixe, le code d'un caractère n'est jamais le préfixe d'un autre caractère.

**Problème :** Trouver un codage qui minimise la longueur moyenne des codes pour un ensemble de  $n$  symboles compte tenu de la fréquence d'apparition des symboles :

Minimiser  $\mu = \sum_{i=1}^n \ell_i f_i$  où  $f_i$  est la fréquence et  $\ell_i$ , la longueur du code du symbole  $i$ .

---

3. voir <https://namok.be/blog/?post/2009/11/30/unicode-UTF8-UTF16-UTF32-et-tutti-quant>

## Exemples

	$a$	$b$	$c$	$d$	$e$	$f$	$\mu$
$f_i$	0,45	0,13	0,12	0,16	0,09	0,05	
code 1	000	001	010	011	100	101	3
code 2	0	11	10	100	111	01	1,8 X
code 3	0	101	100	111	1101	1100	2,24 ✓

## Coder

	<i>cadefe</i>	longueur
code 1	010 000 011 100 101	15
code 2	10 0 100 111 01	11
code 3	100 0 111 1101 1100	15

## Décoder

	texte codé	texte décodé
code 1	011100010101	011 <b>100010101</b>
code 2	1001111001	100 <b>1111001</b> ou 100 <b>1111001</b>
code 3	11111011001100	111 <b>11011001100</b>

## Algorithme - Codage de Huffman

**fonction** huffmann ( *symboles* ) **retourne** nœud

entrée :

*symboles* : série de  $n$  noeuds feuilles contenant un couple (symbole, fréquence)

conséquent :

retourne le nœud racine de l'arbre de Huffman

**début**

```
1   filePrio.construire(symboles)
2   pour i ← 1 à n – 1 faire
3       c1 ← filePrio.extraireMin()
4       c2 ← filePrio.extraireMin()
5       r ← nouveau nœud (c1.symbole & c2.symbole, c1.freq + c2.freq)
6       r.gauche ← c1
7       r.droite ← c2
8       filePrio.inserer(r)
9   fin pour
10  retourner filePrio.extraireMin()
fin
```

## Exemple

## Analyse

Si on utilise un monceau (*heap*) min pour la file de priorité :

- ▶ *filePrio.construire(symboles)* :  $O(n)$
- ▶ *filePrio.extraireMin()* :  $O(\log n)$
- ▶ Au total :  $O(n \log n)$

## Ordonnancement de tâches avec pénalités et échéances

- ▶  $E = \{1, 2, \dots, n\}$  un ensemble de tâches unitaires (durée = 1)
- ▶  $d_i$  : échéance de la tâche  $i$
- ▶  $w_i$  : pénalité si la tâche  $i$  se termine après l'échéance
- ▶ Une tâche est **en retard** si elle se termine après l'échéance, sinon, elle est **en avance**.

**Problème** : Dans quel ordre effectuer les tâches pour minimiser la somme des pénalités des tâches en **retard** (ou maximiser la somme des pénalités des tâches en **avance**).

## Algorithme

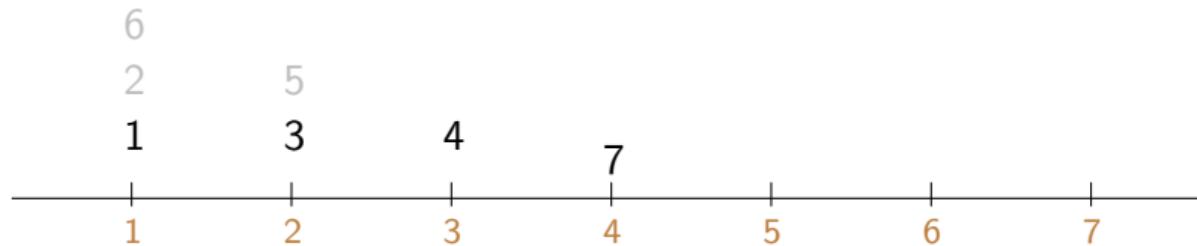
- ▶  $F \subseteq E$  : ensemble de tâches
- ▶  $N_t(F) = \text{card}\{i | i \in F \text{ et } d_i \leq t\}$  : Nombre de tâches de  $F$  dont l'échéance est  $\leq t$  (tâches en avance)
- ▶  $w(F) = \sum_{i \in F} w_i$  : le poids de  $F$
- ▶  $F$  est indépendant ssi  $N_t(F) \leq t$ ,  $1 \leq t \leq n$

## Exemples

Exemple 1 –  $E = \{1, 2, 3, 4, 5, 6, 7\}$

$i$	1	2	3	4	5	6	7
$d_i$	1	1	2	3	2	1	4
$N_i(E)$	3	5	6	7	7	7	7

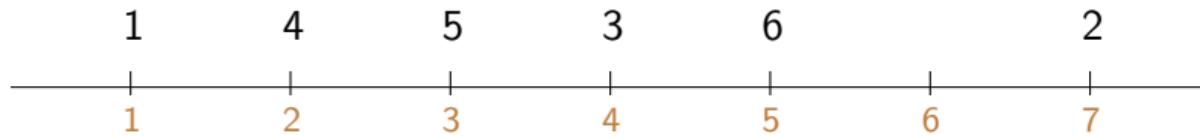
$E$  n'est pas indépendant car, entre autres,  $N_1(E) = 3 \not\leq 1$



Exemple 2 –  $E = \{1, 2, 3, 4, 5, 6\}$

$i$	1	2	3	4	5	6
$d_i$	1	7	4	2	3	5
$N_i(E)$	1	2	3	4	5	5

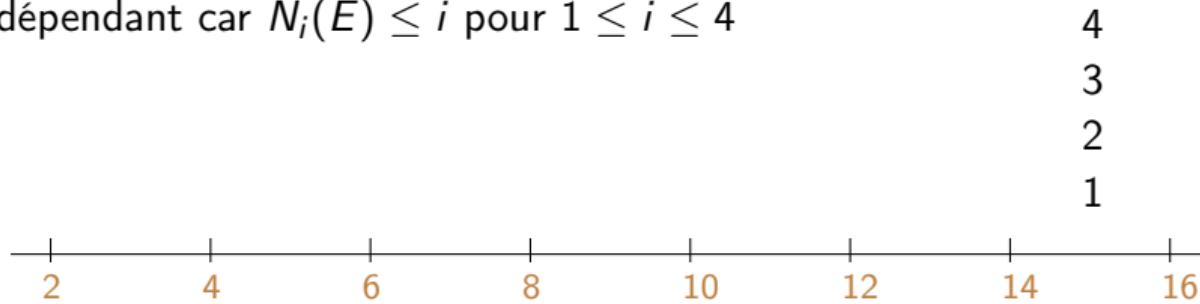
$E$  est indépendant car  $N_i(E) \leq i$  pour  $1 \leq i \leq 6$



Exemple 3 –  $E = \{1, 2, 3, 4\}$

$i$	1	2	3	4
$d_i$	15	15	15	15
$N_i(E)$	0	0	0	0

$E$  est indépendant car  $N_i(E) \leq i$  pour  $1 \leq i \leq 4$



## Propriétés

- ▶ Si  $E$  est indépendant alors  $F \subseteq E$  l'est aussi
- ▶  $F$  est indépendant si et seulement s'il existe un ordonnancement de  $F$  tel que toutes ses tâches sont en avance.
- ▶ Trouver un ordonnancement optimal pour  $E$  : Trouver un sous-ensemble indépendant de  $E$  de poids maximal.

## **Algorithme 1**

Générer les  $2^n$  sous ensembles de  $F$  et prendre l'un de ceux, de cardinalité maximale et de poids maximal, qui sont indépendants.

## Algorithme 2 – Utilise la stratégie gloutonne

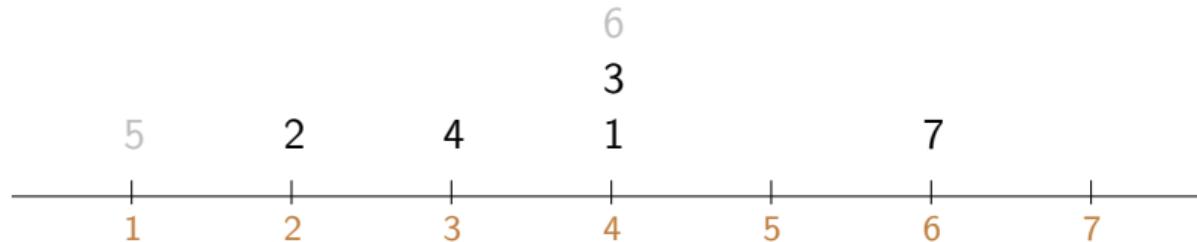
- ▶ **Étape 1** – Trouver un sous-ensemble indépendant de poids maximal  
**début**

```
1    trier (numéroter) les tâches en ordre de pénalités décroissantes
2     $F \leftarrow \emptyset$ 
3    pour  $i \leftarrow 1$  à  $n$  faire
4        si estIndépendant ( $F \cup \{i\}$ ) alors
5             $F \leftarrow F \cup \{i\}$ 
6        fin si
7    fin pour
8    retourner  $F$ 
fin
```

- ▶ **Étape 2 – Trouver l'ordonnancement**
  - ▶ Ordonner les tâches de  $F$  par ordre croissant d'échéance.
  - ▶ Placer à la suite les éléments de  $E - F$  dans n'importe quel ordre

## Exemple 1

$i$	1	2	3	4	5	6	7
$d_i$	4	2	4	3	1	4	6
$w_i$	70	60	50	40	30	20	10



Ordonnancement : 2, 4, 1, 3, 7, 5, 6 de pénalité 50

## Exemple 2

$i$	1	2	3	4	5	6	7
$d_i$	1	1	2	3	2	1	4
$w_i$	70	60	50	40	30	20	10



Ordonnancement : 1, 3, 4, 7, 2, 5, 6 de pénalité 110

## Exemple 3

$i$	1	2	3	4	5	6	7
$d_i$	1	7	4	2	3	5	7
$w_i$	70	60	50	40	30	20	10



Ordonnancement : 1, 4, 5, 3, 6, 2, 7 de pénalité 0

## Arbre de recouvrement minimal ARM

- ▶  $G = (S, E)$ , non orienté, pondéré, connexe
- ▶ Fonction de poids  $w : S \times S \rightarrow \mathbb{R}$
- ▶ Soit  $G' = (S, E')$ , connexe, sans cycle (i.e. un arbre) où  $E' \subseteq E$
- ▶  $G'$  est un ARM si  $\sum_{(u,v) \in E'} w_{u,v}$  est minimale
- ▶ Remarque  $|E'| = |S| - 1 = n - 1$  (car  $G'$  est un arbre)

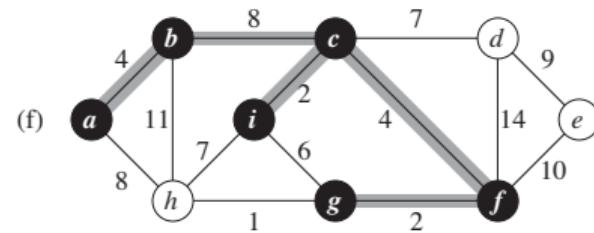
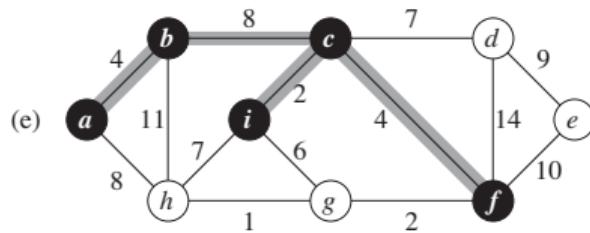
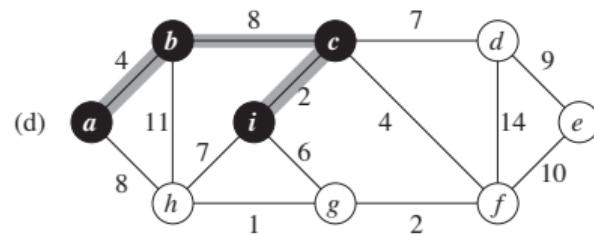
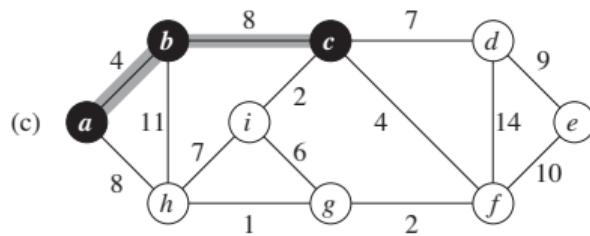
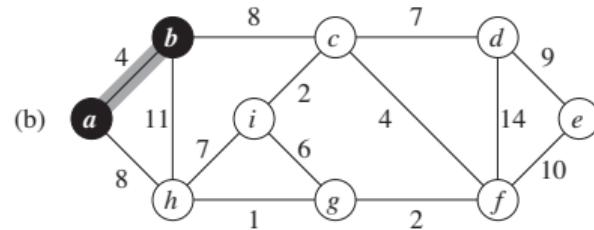
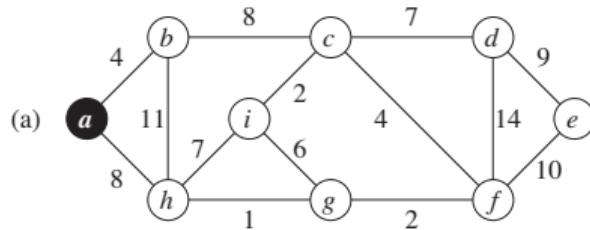
### **Algorithme 1 - naïf**

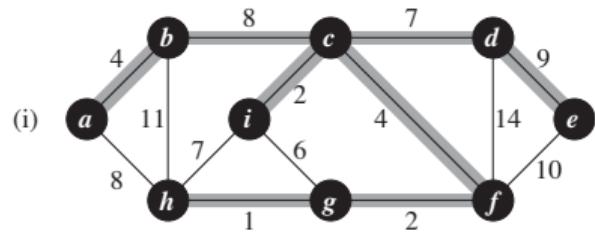
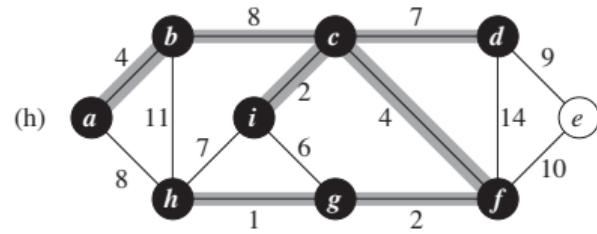
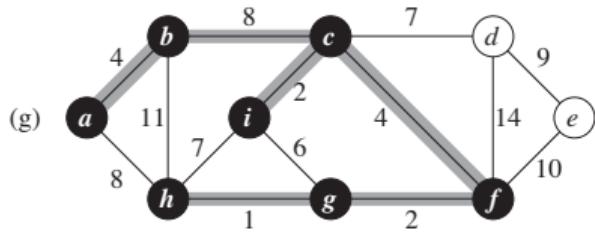
- ▶ Pour tous les ensembles  $E' \subseteq E$  de cardinalité  $n - 1$  vérifier si  $G' = (S, E')$  est connexe et sans cycle. Prendre celui de poids minimal.

**Analyse :**

- ▶ Nombre d'ensembles  $E' \subseteq E$  de cardinalité  $n - 1$  :  $\binom{|E|}{n-1}$
- ▶  $n - 1 \leq |E| \leq \binom{n}{2}$
- ▶ Pire qu'exponentiel dans le pire cas

# Algorithme de Prim





Source de l'image : CLRS - Introduction to algorithms - 3e ed, page 635

## Utilisation d'une file de priorité

insérer ( $F, x$ ) ou  $F \leftarrow F \cup \{x\}$  : insertion de l'élément  $x$  dans la file  $F$ .

minimum ( $F$ ) : retourne l'élément ayant la plus petite clé dans  $F$ .

extraireMin ( $F$ ) : enlève et retourne l'élément ayant la plus petite clé dans  $F$ .

modifierClé ( $F, x, k$ ) : modifier la clé de  $x$  par la nouvelle valeur  $k$ .

**fonction** Prim (  $G, w, r$  ) **retourne** ensemble d'arêtes

entrée :

$G$  : graphe connexe non-orienté

$w$  : fonction de poids sur les arêtes

$r$  : sommet de départ

**début**

$A \leftarrow \emptyset$

**pour** chaque  $u \in G.sommets$  **faire**

$u.cle \leftarrow \infty$ ;  $u.parent \leftarrow null$

**fin pour**

$r.cle \leftarrow 0$

$file \leftarrow G.sommets$

**tant que**  $file \neq \emptyset$  **faire**

$u \leftarrow \text{extraireMin} (file)$

$A \leftarrow A \cup \{(u, u.parent)\}$

**pour** chaque  $v \in u.adjacents$  **faire**

**si**  $v \in file$  **et**  $w(u, v) < v.cle$  **alors**

$v.parent \leftarrow u$ ;  $v.cle \leftarrow w(u, v)$

**fin si**

**fin pour**

**fin tant que**

**retourner**  $A - \{(r, r.parent)\}$

**fin** Prim

$\Theta(|S|)$

$|S|$  fois

$O(\log |S|)$

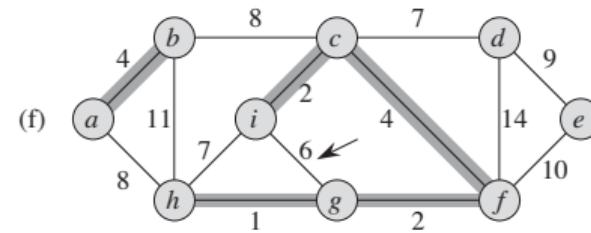
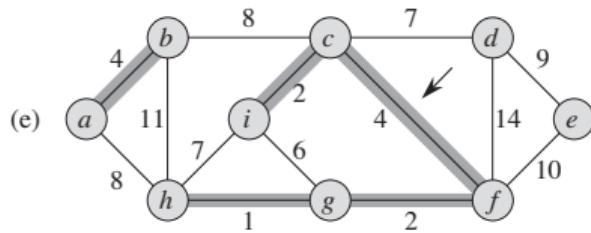
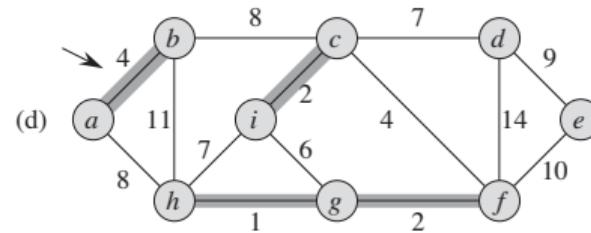
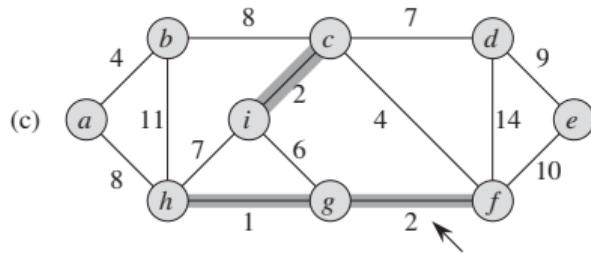
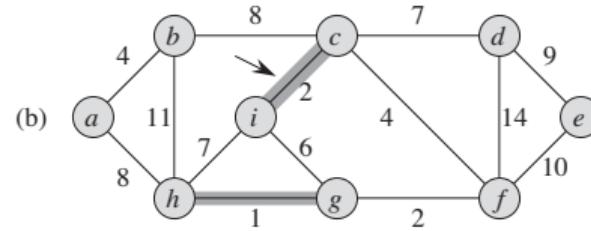
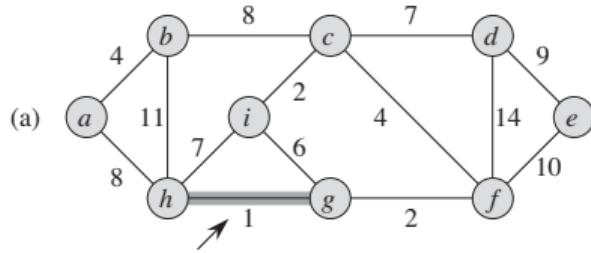
$O(|E|)$  (au total)

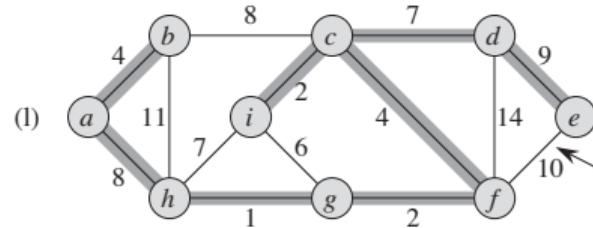
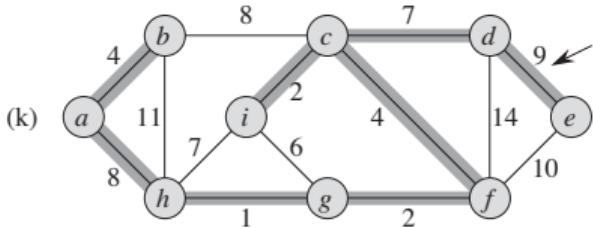
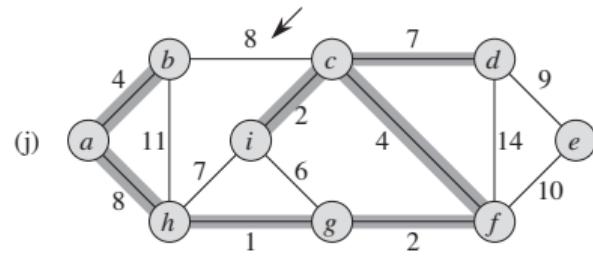
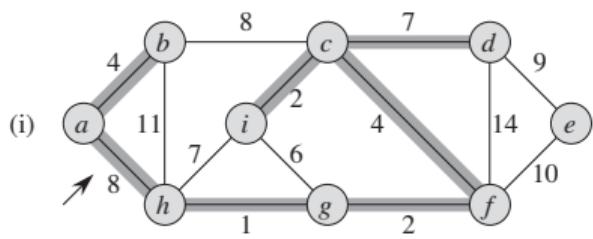
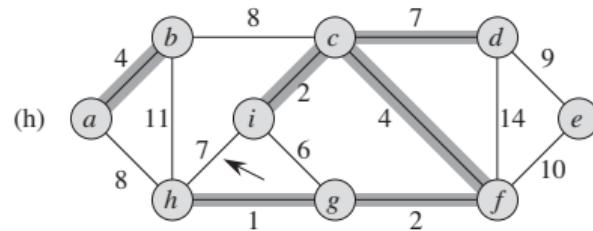
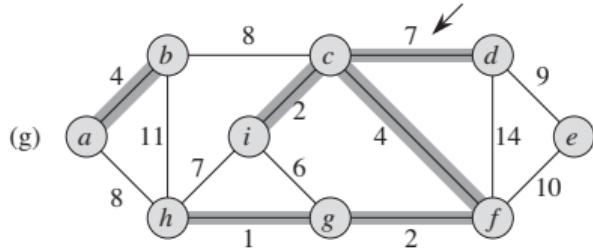
$O(1)$  (si bit d'appartenance)

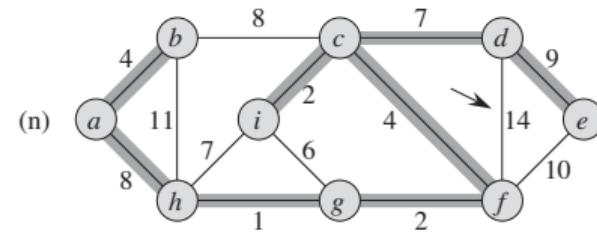
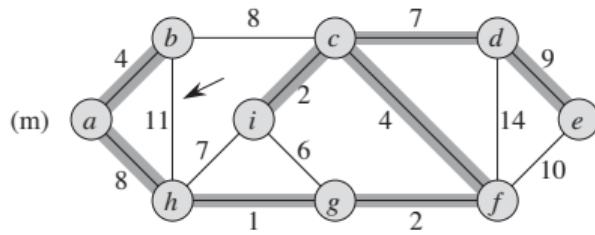
$O(\log |S|)$  (Diminuer Clé)

$O(|E| \log |S|)$

# Algorithme de Kruskal







*Source de l'image : CLRS - Introduction to algorithms - 3e ed, page 632-633*

- ▶ On maintient une forêt d'arbres sous-ensembles d'un ARM
- ▶ Utilisation de la structure ensemble disjoints

`créerEnsemble( x )` : crée un ensemble ne contenant que  $x$  (qui ne doit pas faire partie d'un autre ensemble.)

`union( x, y )` : retourne l'ensemble résultant de l'union des deux ensembles contenant  $x$  et  $y$ .

`trouverEnsemble( x )` : retourne l'ensemble contenant  $x$ .

### Exemple d'implémentation :

- ▶ Éléments :  $\{1, 2, \dots, n\}$
- ▶ Ensemble de sous-ensembles de  $\{1, 2, \dots, n\}$
- ▶ On choisit un représentant par ensemble, exemple, le minimum.
- ▶ Tableau  $t$  tel que  $t[i] =$  représentant de  $i$

**fonction** union (  $i, j$  )

$i \leftarrow \text{trouverEnsemble} ( i )$

$j \leftarrow \text{trouverEnsemble} ( j )$

**si**  $i < j$  **alors**

$t[j] \leftarrow i$

**sinon**

$t[i] \leftarrow j$

**fin si**

**fonction** trouverEnsemble (  $i$  )

**tant que**  $t[i] \neq i$  **faire**

$i \leftarrow t[i]$

**fin tant que**

**retourner**  $i$

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

**fonction** Kruskal (  $G, w$  ) **retourne** ensemble d'arêtes

entrée :

$G$  : graphe connexe non-orienté

$w$  : fonction de poids sur les arêtes

**début**

$A \leftarrow \emptyset$

**pour** chaque  $v \in G.sommets$  **faire**  
    créerEnsemble(  $v$  )

**fin pour**

trier les arêtes de  $G.arestes$  par ordre croissant de poids

**pour** chaque  $(u, v) \in G.arestes$  en ordre croissant de poids **faire**  
    **si**  $u$  et  $v$  n'appartiennent pas au même ensemble **alors**

$A \leftarrow A \cup \{(u, v)\}$   
        union(  $u, v$  )

**fin si**

**fin pour**

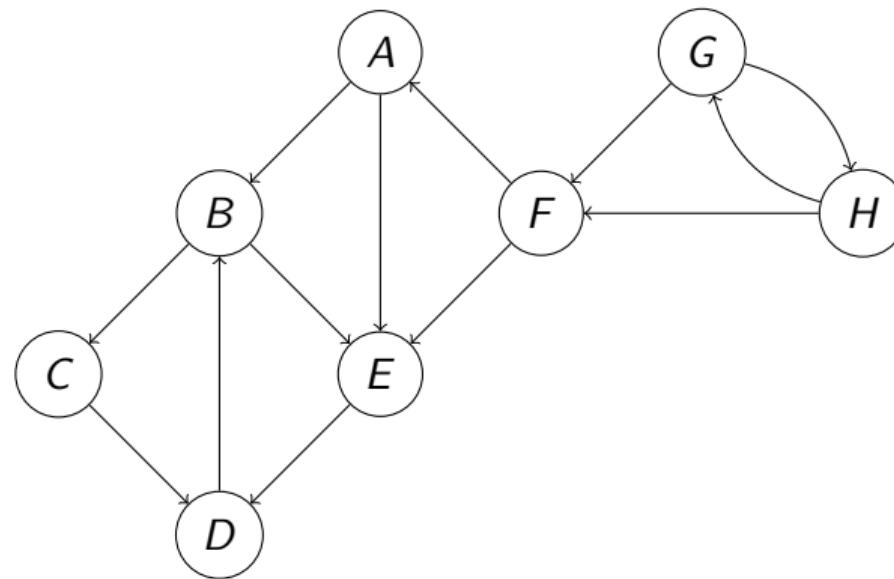
**fin Kruskal**

# 8. Algorithmes sur les graphes

## Definition

Un graphe  $G = (S, A)$  est constitué d'un ensemble  $S$  de sommets reliés entre eux par des arcs (graphe orienté,  $A \subseteq S \times S$ ) ou des arêtes (graphe non-orienté  $A \subseteq \{\{a, b\} | a \in S, b \in S\}$ )

## Exemple



## 8.1. Parcours de graphes

## Parcours de graphes

- ▶  $g.sommets$  : ensemble des sommets de  $g$
- ▶  $g.successeurs(u)$  : ensemble des sommets accessibles de  $u$  par un arc ou une arête
- ▶ Chaque  $u \in g.sommets$  possède les attributs suivants :
  - ▶  $marqué$  : un booléen
  - ▶  $parent$  : lors d'une visite, le sommet visité avant  $u$
  - ▶  $distance$  : lors d'une visite à partir d'un sommet  $v$ , le nombre d'arcs (arêtes) rencontré(e)s avant d'arriver à  $u$

► Parcours en profondeur

**procedure** parcoursProf ( *g* )

entrée :

*g* : graphe à visiter

**début**

```
1   pour chaque u ∈ g.sommets faire
2       u.marqué ← faux
3       u.parent ← null
4       u.distance ← 0
5   fin pour
6   pour chaque u ∈ g.sommets faire
7       si non u.marqué alors
8           visiterProf ( g, u )
9       fin si
10  fin pour
fin parcoursProf
```

**procedure** visiterProf ( *g, u* )

entrée

*g* : graphe à visiter

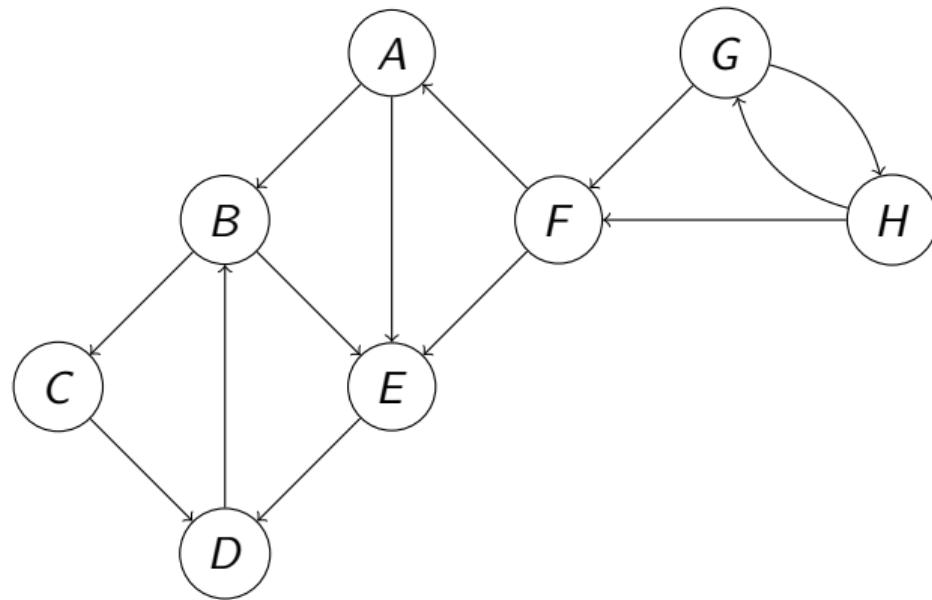
*u* : sommet de départ

**début**

- 1     *u.marqué*  $\leftarrow$  vrai
- 2     **pour chaque** *v*  $\in g.\text{successeurs}(u)$  **faire**
- 3         **si non** *v.marqué* **alors**
- 4             *v.parent*  $\leftarrow u$
- 5             *v.distance*  $\leftarrow u.distance + 1$
- 6             visiterProf ( *g, v* )
- 7         **fin si**
- 8     **fin pour**

**fin** visiterProf

## Exemple



	A	B	C	D	E	F	G	H
parent								
dist.								

► Parcours en largeur

**procedure** parcoursLarg ( *g* )

entrée :

*g* : graphe à visiter

**début**

```
1   pour chaque u ∈ g.sommets faire
2       u.marqué ← faux
3       u.parent ← null
4       u.distance ← 0
5   fin pour
6   pour chaque u ∈ g.sommets faire
7       si non u.marqué alors
8           visiterLarg ( g, u )
9       fin si
10  fin pour
fin parcoursLarg
```

**procedure** visiterLarg ( *g, u* )

entrée

*g* : graphe à visiter

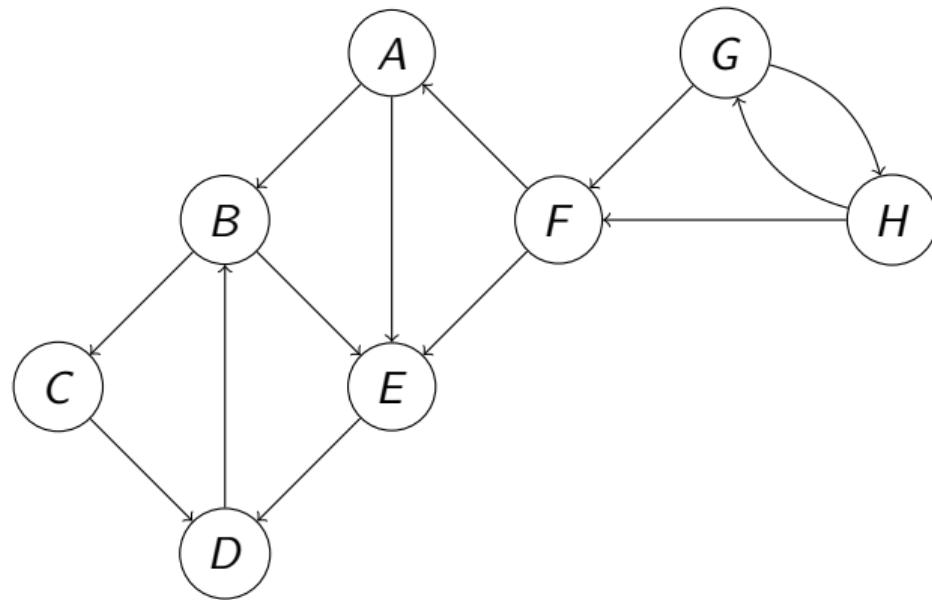
*u* : sommet de départ

**début**

- 1     *u.marqué*  $\leftarrow$  **vrai**
- 2     *file.enfiler*(*u*)
- 3     **tant que** *file* non vide **faire**
- 4         *u*  $\leftarrow$  *file.défiler*()
- 5         **pour chaque** *v*  $\in$  *g.successeurs*(*u*) **faire**
- 6             **si non** *v.marqué* **alors**
- 7                 *v.parent*  $\leftarrow$  *u*
- 8                 *v.distance*  $\leftarrow$  *u.distance* + 1
- 9                 *v.marqué*  $\leftarrow$  **vrai**
- 10                 *file.enfiler*(*v*)
- 11             **fin si**
- 12         **fin pour**
- 13     **fin tant que**

**fin** visiterLarg

## Exemple



	A	B	C	D	E	F	G	H
parent								
dist.								

## 8.2. Flot dans un réseau

## Flot dans un réseau

- ▶ Réseau  $R = (V, A, s, t, c)$
- ▶  $V$  : ensemble des **sommets**
- ▶  $A \subseteq V \times V$  : ensemble des **arcs**
- ▶  $s \in V$  : sommet **source**
- ▶  $t \in V$  : sommet **puits**
- ▶  $c$  : matrice des **capacités** des arcs où  $c_{u,v}$  est la capacité de l'arc  $(u, v)$ ,  $u \in V$ ,  $v \in V$
- ▶  $c_{u,v} > 0$  pour  $(u, v) \in A$ ,  $c_{u,v} = 0$  pour  $(u, v) \notin A$
- ▶ Un **flot** dans le réseau  $R$  est une fonction  $f$  telle que
  - ▶  $f : V \times V \rightarrow \mathbb{R}$
  - ▶  $f(u, v)$  : flot du sommet  $u$  au sommet  $v$
  - ▶ **Valeur d'un flot** :

$$|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t),$$

## Propriétés

- ▶ **Contrainte de capacité** : Pour tous les sommets  $u$  et  $v$ ,  $f(u, v) \leq c(u, v)$ .
- ▶ **Symétrie inverse** : Pour tous les sommets  $u$  et  $v$ ,  $f(u, v) = -f(v, u)$ .
- ▶ **Conservation du flot** : Pour sommets  $u \in V - \{s, t\}$ ,  $\sum_{v \in V} f(u, v) = 0$
- ▶ Pour chaque sommet  $v \in V$ ,

$$\sum_{(u,v) \in A} f(u, v) = \sum_{(v,w) \in A} f(v, w).$$

**Problème** : Trouver le flot maximal dans  $R$ , c'est-à-dire, maximiser  $|f|$ .

- ▶ **Algorithme 1 – Algorithme glouton**
  - ▶ Trouver un chemin de la source au puits
  - ▶ Ajuster les capacités
  - ▶ Répéter jusqu'à l'absence de chemin dans le réseau résiduel (réseau dont les capacités sont modifiées pour tenir compte des chemins déjà trouvés)

Cet algorithme ne donne pas toujours une solution optimale.

## ► Algorithme 2 – Algorithme de Ford-Fulkerson

Réseau résiduel :  $R_f = (V, A_f, s, t, c)$ . Intuitivement, le réseau résiduel contient les arcs qui peuvent accepter plus de flot.

Flot additionnel : Quantité (capacité résiduelle) que l'on peut ajouter sans excéder la capacité maximale.

Ensemble  $A_f$  : Si  $(u, v) \in A$  alors  $(u, v) \in A_f$  ou  $(v, u) \in A_f$ . Ici, le **ou** n'est pas un **ou** exclusif.

## Construction du réseau résiduel $R_f$

si  $(u, v) \in A$  alors

Combien il manque pour atteindre la capacité maximale

si  $f(u, v) < c(u, v)$  alors

$$c_f(u, v) = c(u, v) - f(u, v)$$

fin si

Combien peut-on retirer

si  $f(u, v) > 0$  alors

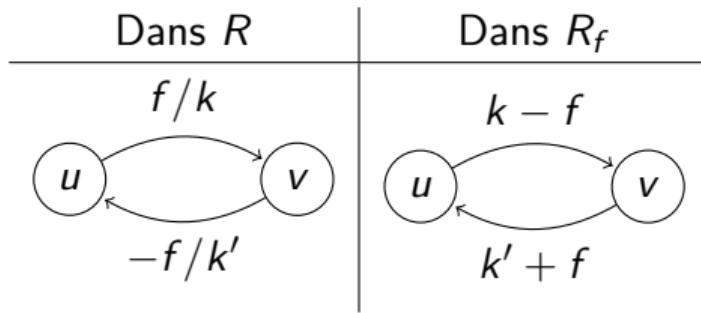
$$c_f(v, u) = c(v, u) - f(v, u) = c(v, u) + f(u, v)$$

fin si

fin si

Un chemin de  $s$  à  $t$  dans  $R_f$  est appelé **chemin d'augmentation** par rapport à  $f$ .

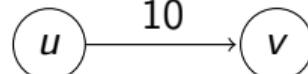
Si  $f = f(u, v)$ ,  $k = c(u, v)$  et  $k' = c(v, u)$ , on a



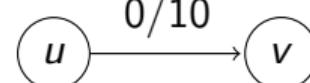
## Exemple

	Dans $R$	Dans $R_f$
$k = 13, k' = 0, f = 8$		
$k = 10, k' = 4, f = -1$		
$k = 10, k' = 0, f = 0$		

Remarque



est équivalent à



- ▶ Pour tous les sommets  $u$  et  $v$ , poser  $f(u, v) = 0$ .
- ▶ Augmenter le flot dans le réseau en trouvant un chemin d'augmentation (un chemin de  $s$  à  $t$  dans le réseau résiduel dans lequel on peut « envoyer » plus de flot). Répéter cette étape jusqu'à ce qu'il n'y ait plus de chemin d'augmentation.
- ▶ Version Edmonds-Karp : on utilise une fouille en largeur pour trouver le chemin d'augmentation

**début**

Initialiser le flot au flot nul

Construire le réseau résiduel ( $R_f \leftarrow R$ )

Effectuer une fouille en largeur dans le réseau résiduel  $R_f$

**tant que** la fouille en largeur a trouvé un chemin de  $s$  à  $t$  dans  $R_f$  **faire**

Modifier le flot en lui ajoutant le chemin d'augmentation minimal

Mettre à jour les réseaux  $R$  et  $R_f$

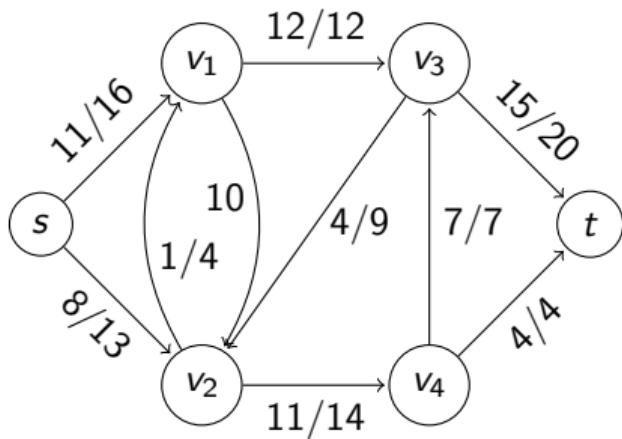
Effectuer une fouille en largeur dans le réseau résiduel  $R_f$

**fin tant que**

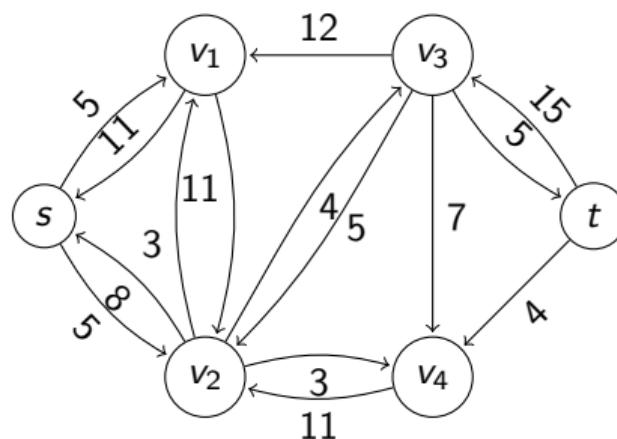
**fin**

## Exemple

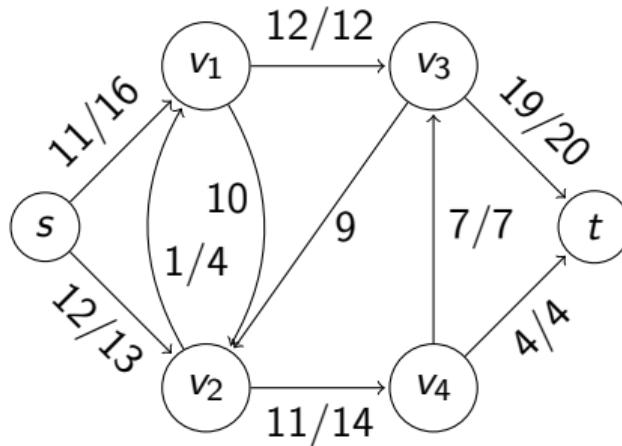
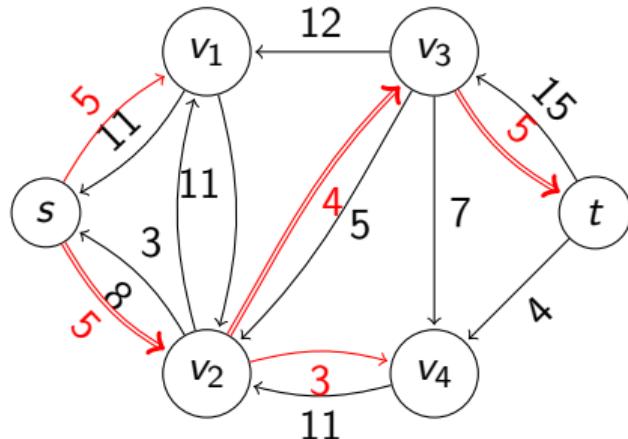
Réseau  $R$



Réseau résiduel  $R_f$



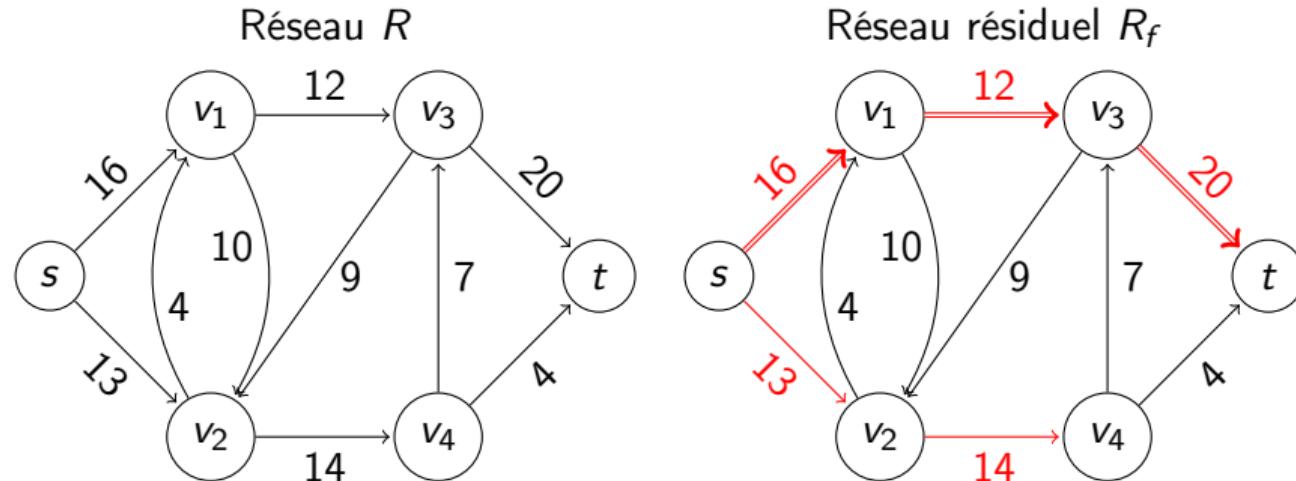
## Parcours en largeur dans le réseau résiduel



sommet	$s$	$v_1$	$v_2$	$v_3$	$v_4$	$t$
prédécesseur	0	$s$	$s$	$v_2$	$v_2$	$v_3$

Chemin d'augmentation :  $s - v_2 - v_3 - t$  de coût 4

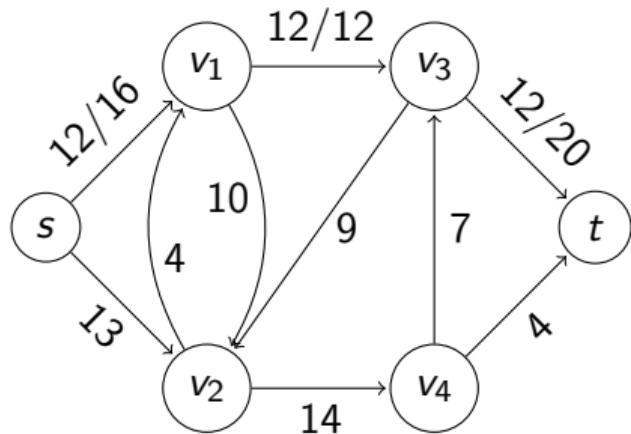
## Exemple complet



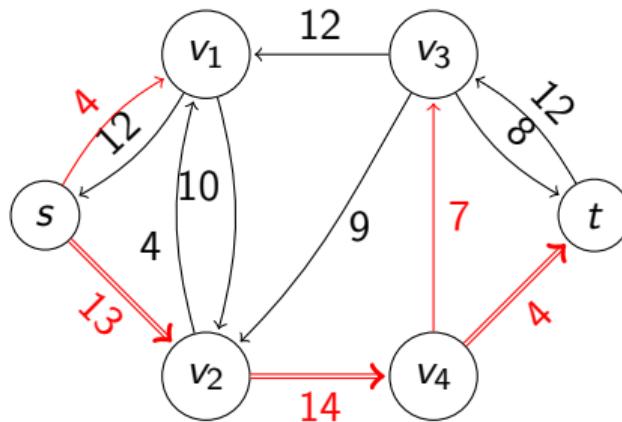
sommet	$s$	$v_1$	$v_2$	$v_3$	$v_4$	$t$
prédécesseur	0	$s$	$s$	$v_1$	$v_2$	$v_3$

$s - v_1 - v_3 - t : 12$

Réseau augmenté  $R$



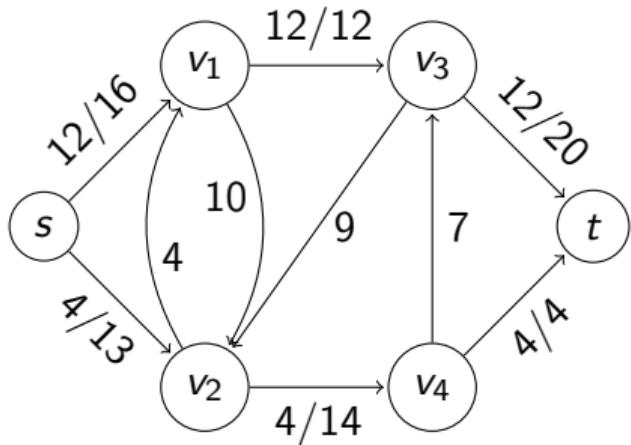
Réseau résiduel  $R_f$



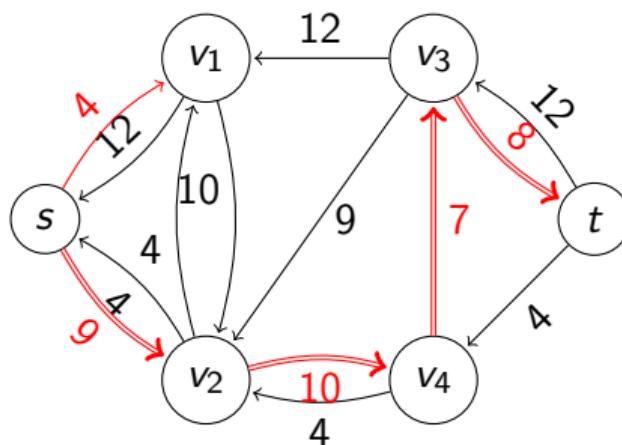
sommet	$s$	$v_1$	$v_2$	$v_3$	$v_4$	$t$
prédécesseur	0	$s$	$s$	$v_4$	$v_2$	$v_4$

$$s - v_2 - v_4 - t : 4$$

Réseau augmenté  $R$



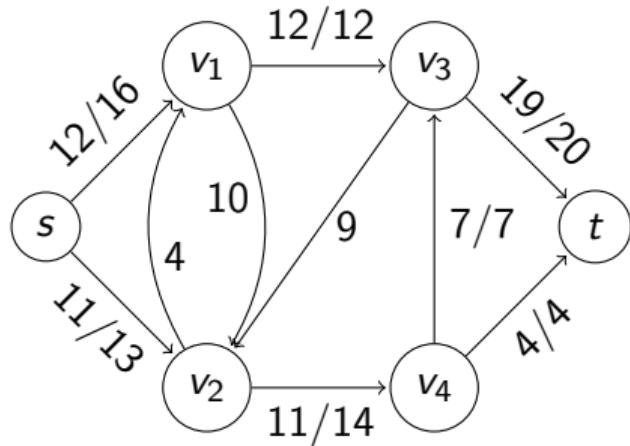
Réseau résiduel  $R_f$



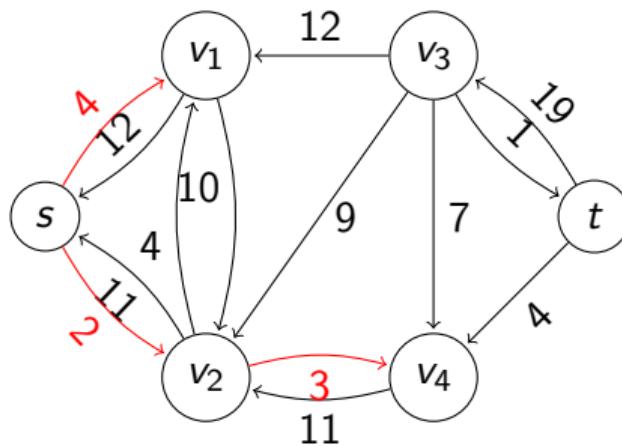
sommet	$s$	$v_1$	$v_2$	$v_3$	$v_4$	$t$
prédécesseur	0	$s$	$s$	$v_4$	$v_2$	$v_3$

$$s - v_2 - v_4 - v_3 - t : 7$$

Réseau augmenté  $R$



Réseau résiduel  $R_f$



sommet	$s$	$v_1$	$v_2$	$v_3$	$v_4$	$t$
prédécesseur	0	$s$	$s$	0	$v_2$	0

Aucun chemin.

Flot maximal =

# Analyse

- ▶ Ford Fulkerson :  $O(|A||f_{max}|)$
- ▶ Version Edmonds-Karp :  $O(|V||A|^2)$

# 9. Analyse amortie

## 9.1. Introduction

# Introduction

- ▶ **Analyse traditionnelle**, basée sur une exécution d'un algorithme :
  - ▶ Meilleur cas, cas moyen, pire cas
  - ▶ Surévalue dans certains cas la complexité temporelle, en particulier pour le pire cas.
- ▶ **Analyse amortie** :
  - ▶ Analyse d'une séquence de  $n$  opérations sur une même structure de données<sup>4</sup>.
  - ▶ Une opération particulière peut être très coûteuse mais n'être effectuée qu'une seule fois dans la séquence, de là la surestimation du temps d'exécution.
  - ▶ Temps total requis pour effectuer la séquence de longueur  $n \leq$  temps dans le pire cas de cette opération, multiplié par  $n$
  - ▶ On amortit le coût élevé de l'opération sur toute la séquence.

---

4. Exemple provenant de l'API de Java pour la classe `ArrayList`. *The add operation runs in amortized constant time, that is, adding  $n$  elements requires  $O(n)$  time. All of the other operations run in linear time (roughly speaking).* qu'on peut trouver à l'adresse suivante : <http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html>

Méthodes pouvant être utilisées pour l'analyse amortie d'une structure de données :

- ▶ analyse de l'agrégat
- ▶ méthode comptable
- ▶ méthode du potentiel

## Exemple 1 : Structure de données **Pile** munie des opérations :

- $\text{estVide}( P )$  : retourne **vrai** si  $P$  est vide, **faux** sinon.
- $\text{taille}( P )$  : retourne le nombre d'éléments dans  $P$ .
- $\text{empiler}( P, x )$  : met  $x$  au sommet de  $P$ .
- $\text{dépiler}( P )$  : enlève et retourne l'élément au sommet de  $P$ . Si  $P$  est vide, une erreur sera détectée.
- $\text{multiDépiler}( P, k )$  : Dépile  $\min( k, \text{taille}( P ) )$  éléments.

- ▶  $\text{empiler}, \text{dépiler}, \text{estVide}, \text{taille} \in O(1)$
- ▶  $\text{multiDépiler} \in O(\min(p, k))$  où  $p = \text{taille}(P)$
- ▶ Suite de  $n$  opérations  $\text{empiler}, \text{dépiler}, \text{multiDépiler}$ .  $p \leq n$
- ▶ une opération quelconque  $\in O(n)$
- ▶ Suite de  $n$  opérations  $\in O(n^2)$

## Exemple 2 : Incrémentation d'un compteur binaire

- ▶ A tableau indicé de 0 à  $k - 1$  de bits. Bit de poids le plus faible =  $A[0]$
- ▶ incrémenter(  $A$  ) : Augmente de 1 la suite binaire contenue dans  $A$

```
1   i ← 0
2   tant que  $i < A.longueur$  et  $A[i] = 1$  faire
3        $A[i] \leftarrow 0$ 
4        $i \leftarrow i + 1$ 
5   fin tant que
6   si  $i < A.longueur$  alors
7        $A[i] \leftarrow 1$ 
8   fin si
```

$$\begin{array}{r} 0100111010111010\textcolor{orange}{11111} \\ + 1 \\ \hline 0100111010111011\textcolor{orange}{00000} \end{array}$$

- ▶ incrémenter(  $A$  )  $\in O(k)$ .
- ▶ Suite de  $n$  incrémenter  $\in O(nk)$

## 9.2. Analyse de l'agrégat

## Analyse de l'agrégat

- ▶ Calcul du temps total  $T(n)$  d'une suite de  $n$  opérations dans le pire cas
- ▶ Coût amorti pour une opération (toutes opérations confondues) :  $T(n)/n$

### Analyse de $n$ opérations sur une Pile

- ▶ un élément ne peut être déplié plus de fois que son nombre d'empilements
- ▶ **#op** : nombre de fois où l'opération **op** est effectuée
- ▶ **#dépiler**  $\leq$  **#empiler**  $\leq n$  (incluant les dépilements effectués par l'opération **multiDépiler**)
- ▶ n'importe quelle suite de  $n$  opérations prendra  $O(n)$ .
- ▶ coût amorti de  $O(1)$  par opération.

## Analyse de $n$ incrémentations d'un compteur binaire

		Coût total			Coût total
0	00000000	0	9	00001001	16
1	00000001	1	10	00001010	18
2	00000010	3	11	00001011	19
3	00000011	4	12	00001100	22
4	00000100	7	13	00001101	23
5	00000101	8	14	00001110	25
6	00000110	10	15	00001111	26
7	00000111	11	16	00010000	31
8	00001000	15	...		

- ▶  $A[0]$  change à chaque appel de incrémenter
- ▶  $A[1]$  change une fois sur deux (2)
- ▶  $A[2]$  change une fois sur quatre (4)
- ▶  $A[i]$  change une fois sur  $2^i$

Sur une séquence de  $n$  appels de incrémenter

- ▶  $A[0]$  change  $n$  fois
- ▶  $A[1]$  change  $\lfloor \frac{n}{2} \rfloor$  fois
- ▶  $A[2]$  change  $\lfloor \frac{n}{4} \rfloor$  fois
- ▶  $A[i]$  change  $\lfloor \frac{n}{2^i} \rfloor$  fois
- ▶ Nombre total de changements :

$$\sum_{i=0}^{k-1} (\text{Nb chang. de } A[i]) = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{k-1} \frac{n}{2^i} < \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n \in O(n)$$

- ▶ Coût moyen de chaque opération :  $O(n)/n = O(1)$

### 9.3. Méthode comptable

## Méthode comptable

- ▶ À chaque opération  $i$ ,  $1 \leq i \leq n$ , on associe
  - ▶  $cr(i)$  : coût réel
  - ▶  $ca(i)$  : coût amorti
- ▶ coût amorti : coût « facturé »
- ▶ Si  $ca(i) > cr(i)$ , on affecte  $ca(i) - cr(i)$  en crédit
- ▶ Servira plus tard pour des opérations telles que  $ca(j) < cr(j)$  pour  $j > i$
- ▶ Chaque opération peut avoir son propre coût amorti.
- ▶  $\sum_{i=1}^n ca(i) \geq \sum_{i=1}^n cr(i)$
- ▶ Coût total stocké dans la structure de données est  $\sum_{i=1}^n ca(i) - \sum_{i=1}^n cr(i) \geq 0$

## Analyse de $n$ opérations sur une Pile

Opération	coût réel (payé)	coût amorti (facturé)
empiler	1	2
dépiler	1	0
multiDépiler	$\min(k, p)$	0

- ▶ empiler : On paye une unité et on empile une unité avec l'élément (servira lors de son dépilement)
- ▶ dépiler : On se sert de l'unité empilée avec l'élément lors de son empilement pour payer l'opération
- ▶ multiDépiler : Comme dépiler
- ▶ Pour  $n$  opérations :
  - ▶ Coût réel total  $\leq$  coût amorti total  $\leq 2n \in O(n)$
  - ▶ Coût moyen de chaque opération :  $O(n)/n = O(1)$

## Analyse de $n$ incrémentations d'un compteur binaire

Opération	coût réel (payé)	coût amorti (facturé)
$0 \rightarrow 1$	1	2
$1 \rightarrow 0$	1	0

- ▶  $0 \rightarrow 1$  : On paye 1 unité, on empile 1 unité avec le bit changé pour 1
- ▶  $1 \rightarrow 0$  : On se sert de l'unité empilée avec le bit lors de son changement pour 1 pour payer l'opération
- ▶ Pour  $n$  opérations :
  - ▶ Coût réel total  $\leq$  coût amorti total  $\leq 2n \in O(n)$
  - ▶ Coût moyen de chaque opération :  $O(n)/n = O(1)$

## 9.4. Méthode du potentiel

## Méthode du potentiel

- ▶ Crédit de la méthode comptable vu comme du potentiel.
- ▶ Le potentiel est associé à la structure de données et non à des éléments spécifiques.
- ▶  $D_i$  : structure de données après  $i$  opérations,  $0 \leq i \leq n$  où la  $i$ ème opération est appliquée sur  $D_{i-1}$ .
- ▶  $\Phi(D_i)$  est un nombre réel correspondant au potentiel de  $D_i$ .
- ▶  $ca(i) = cr(i) + \Phi(D_i) - \Phi(D_{i-1})$ .
- ▶ Coût amorti total :  
$$\sum_{i=1}^n ca(i) = \sum_{i=1}^n (cr(i) + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^n cr(i) + \Phi(D_n) - \Phi(D_0).$$
- ▶ Trouver  $\Phi$  telle que  $\Phi(D_n) \geq \Phi(D_0)$  pour que le coût amorti total soit une borne supérieure pour le coût réel total.

## Analyse de $n$ opérations sur une Pile

- ▶  $\Phi(P)$  : nombre d'éléments dans la pile  $P$
- ▶  $\Phi(D_0) = 0$ ,  $\Phi(D_i) \geq 0$  et  $\Phi(D_n) \geq \Phi(D_0)$
- ▶ Différence de potentiel pour la  $i$ ème opération :
  - ▶ **empiler** :  $\Phi(D_i) - \Phi(D_{i-1}) = (p + 1) - p = 1$   
 $ca(i) = cr(i) + \Phi(D_i) - \Phi(D_{i-1}) = 2$
  - ▶ **dépiler** :  $\Phi(D_i) - \Phi(D_{i-1}) = (p - 1) - p = -1$   
 $ca(i) = cr(i) + \Phi(D_i) - \Phi(D_{i-1}) = 0$
  - ▶ **multiDépiler** : nombre réel d'éléments dépliés :  $\ell = \min(p, k)$  donc  
 $\Phi(D_i) - \Phi(D_{i-1}) = -\ell$   
 $ca(i) = cr(i) + \Phi(D_i) - \Phi(D_{i-1}) = 0$
- ▶ Coût amorti pour  $n$  opérations  $\leq 2n$
- ▶ Coût amorti des trois opérations est  $O(1)$

## Analyse de $n$ incrémentations d'un compteur binaire

- ▶  $\Phi(A)$  :  $b_i$ , le nombre de bits 1 dans le compteur après la  $i$ ème opération
- ▶  $t_i$  : nombre de bits mis à 0 après  $i$ ème incrémenter

	$cr(i)$	$b_i$	$t_i$		$cr(i)$	$b_i$	$t_i$	
0	00000000	0	0	9	00001001	1	2	0
1	00000001	1	1	10	00001010	2	2	1
2	00000010	2	1	11	00001011	1	3	0
3	00000011	1	2	12	00001100	3	2	2
4	00000100	3	1	13	00001101	1	3	0
5	00000101	1	2	14	00001110	2	3	1
6	00000110	2	2	15	00001111	1	4	0
7	00000111	1	3	16	00010000	5	1	4
8	00001000	4	1					

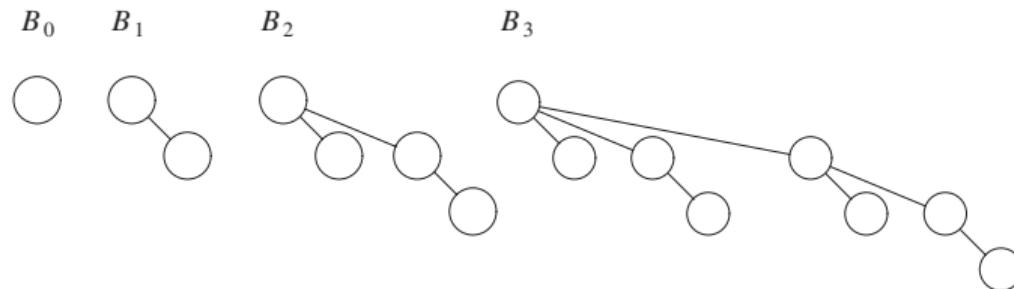
- $b_i$ , le nombre de bits 1 dans le compteur après la  $i$ ème opération
- $t_i$  : nombre de bits mis à 0 après  $i$ ème incrémenter

- ▶  $cr(i) \leq t_i + 1$
- ▶ Si  $b_i = 0$  alors  $b_{i-1} = t_i = k$                      $111111 + 1 \rightarrow 000000$
- ▶ Si  $b_i > 0$  alors  $b_i = b_{i-1} - t_i + 1$              $\dots 011 + 1 \rightarrow \dots 100$
- ▶  $b_i \leq b_{i-1} - t_i + 1$
- ▶  $\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1} \leq b_{i-1} - t_i + 1 - b_{i-1} = 1 - t_i$
- ▶ Coût amorti :  $ca(i) = cr(i) + \Phi(D_i) - \Phi(D_{i-1}) = (t_i + 1) + (1 - t_i) = 2 \in O(1)$

## 9.5. Files binomiales

## Files binomiales – implémentation pour une file de priorité

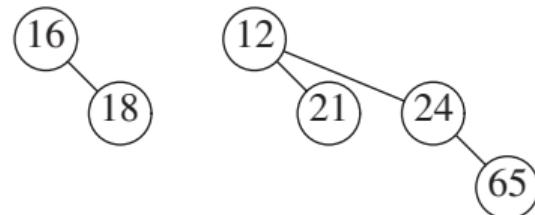
- ▶ Arbre binomial  $B_k$  : la racine a  $k$  enfants, la hauteur est  $k$
- ▶ Rang d'un nœud = nombre d'enfants du nœud
- ▶ Dans  $B_k$ , le nombre de nœuds au niveau  $d = \binom{k}{d}$
- ▶  $|B_k| = \sum_{d=0}^k \binom{k}{d} = 2^k$
- ▶  $B_{k+1}$  est formé en prenant un  $B_k$  auquel on ajoute une branche à la racine vers un autre  $B_k$



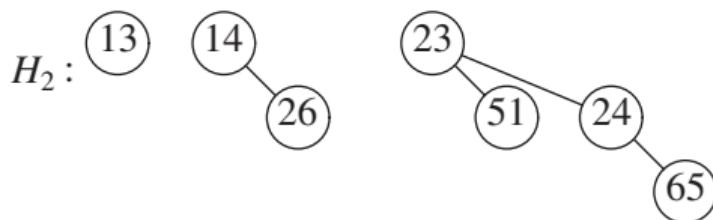
- ▶ File binomiale : forêt d'arbres binomiaux où  $B_k$  apparaît au plus une fois,  $k \geq 0$
- ▶ Chaque  $B_i$  dans la forêt est un monceau
- ▶ Soit  $b$ , la taille de la file exprimée en base deux alors  $B_i$  est présent ssi  $b_i = 1$
- ▶ Le nombre d'arbres binomiaux  $\leq \lfloor \log n \rfloor + 1$

Exemples :

$H_1$ :



$H_2$ :

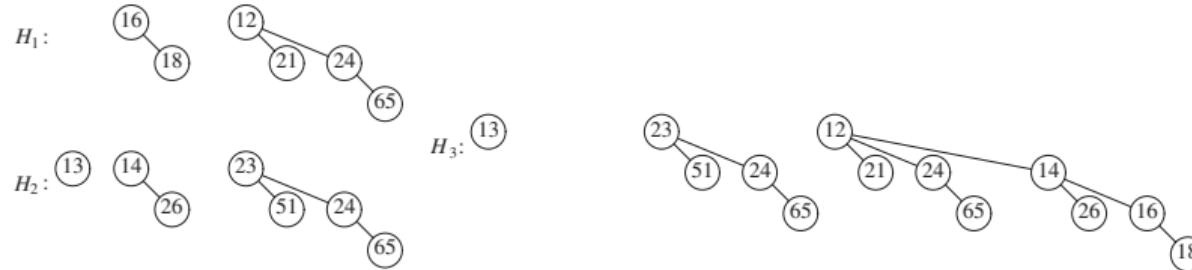


$$|H_1| = 6, 6_{10} = 110_2 \rightarrow B_1 \cup B_2$$

$$|H_2| = 7, 7_{10} = 111_2 \rightarrow B_0 \cup B_1 \cup B_2$$

# Opérations I

- ▶ **Trouver le minimum (maximum)** : Parcourir les racines des arbres  $\in O(\log n)$ 
  - ▶ Amélioration : maintenir le minimum (maximum) et faire la mise à jour au fur et à mesure ce qui va donner  $O(1)$
- ▶ **Fusion de deux files binomiales**



$\in O(\log n)$  (Fusion de deux  $B_k \in O(1)$ , au plus  $O(\log n)$  arbres)

- ▶ **Insertion** : Faire un  $B_0$  avec l'élément à insérer suivi d'une fusion avec la file  $\in O(\log n)$ .  
En moyenne :

- ▶ Soit  $B_i$ , le plus petit arbre absent, alors l'insertion va prendre  $i + 1$  étapes.

## Opérations II

- ▶ Chaque arbre  $B_i$  est présent avec une probabilité de  $\frac{1}{2}$ .
- ▶ Pour avoir  $i$  étapes : ...0111...1. Le bit 0 est en position  $i$ .
- ▶  $\rightarrow \Pr[i \text{ étapes}] = (1/2)^{i+1}$ .
- ▶  $E[\text{nombre d'étapes}] = \sum_{i \geq 0} i \times \Pr[i \text{ étapes}] = 1$
- ▶ **Extraire le minimum (maximum) :**
  - ▶ Chercher le  $B_k$  qui contient le minimum à la racine  $\in O(\log n)$
  - ▶ En retirer la racine donne la suite  $B_0, B_1, \dots, B_{k-1}$  à fusionner avec les autres  $\in O(\log n)$
- ▶ **Faire une file de  $n$  éléments** : insérer  $n$  fois  $\in O(n \log n)$
- ▶ On peut montrer qu'en fait, c'est  $O(n)$

## Nombre total d'étapes pour fabriquer une file de $n$ éléments

00...0 → 00...01 → 00...010 → 00...011 → 00...0100 → ...

Nombre d'étapes	finissent par	%
0	0, 2, 4, 6, ...	0
1	1, 5, 9, 13, ...	01
2	3, 11, 19, 27, ...	011
...		
$i$	$2^i - 1, \dots$	$\underbrace{011\dots1}_i$
		$(1/2)^{i+1}$

On compte une unité de temps pour chaque insertion et une unité de temps pour chaque étape (chaînage).

Coût total pour  $n$  insertions :  $n + \sum_{i=1}^n i \times n(1/2)^{i+1} \in O(n)$

## Autre approche

- ▶ Soit  $b$ , une suite de bits de la forme  $b_g 0 b_d$  où
  - ▶  $b_g$  : suite quelconque de bits
  - ▶  $b_d$  : suite de bits 1
  - ▶  $b_g$  et  $b_d$  peuvent être vides
  - ▶ Exemple :  $11010100001011111 \rightarrow \underbrace{11010100001}_{{b_g}} 0 \underbrace{11111}_{{b_d}}$
- ▶ Soit  $N_1(x) = \text{nombre de bits 1 dans } x$ .
- ▶ Soit  $c_i$  le coût et  $T_i$  le nombre d'arbres après la  $i$ e insertion.
- ▶ Soit  $b$  la suite de bits représentant  $i - 1$
- ▶  $c_i = N_1(b_d) + 1$ ,  $T_{i-1} = N_1(b_g) + N_1(b_d)$  et  $T_i = N_1(b_g) + 1$
- ▶  $T_i - T_{i-1} = 1 - N_1(b_d)$
- ▶  $c_i = 2 - (T_i - T_{i-1})$
- ▶  $\sum_{i=1}^n c_i = \sum_{i=1}^n (2 - (T_i - T_{i-1})) = 2n - (T_n - T_0) \leq 2n \in O(n)$

## Analyse amortie

- ▶ Fonction potentiel  $\Phi(F)$  = nombre d'arbres binomiaux dans la file  $F$
- ▶  $F_i$  : file après  $i$  opérations
- ▶  $T_i - T_{i-1}$  représente la différence de potentiel
- ▶  $T_{\text{actuel}} + \Delta\text{potentiel} = T_{\text{amorti}}$
- ▶ D'opération en opération,  $T_{\text{actuel}}$  varie,  $T_{\text{amorti}}$  est stable.

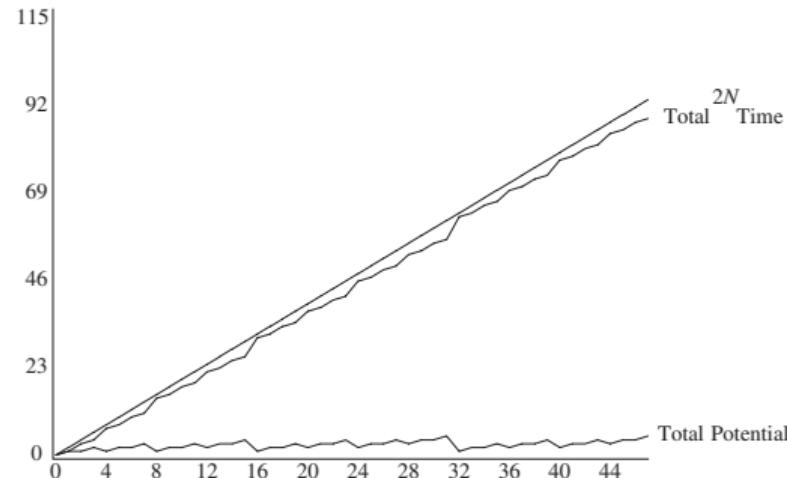
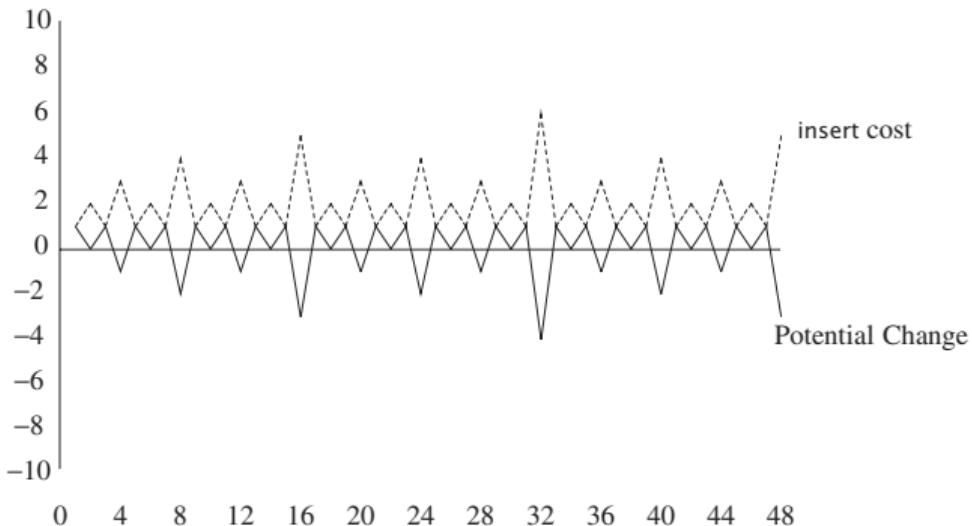


Figure 11.4 A sequence of  $N$  inserts

## En général :

- ▶ Choisir une bonne fonction de potentiel (qui donne de bonnes bornes) n'est pas trivial.
- ▶ Il n'y a pas vraiment de méthode pour la choisir.
- ▶ En général, plusieurs fonctions potentiel sont essayées avant de trouver la bonne.
- ▶ La fonction potentiel doit :
  - ▶ toujours être à son minimum au début de la séquence. Par exemple, choisir le potentiel initial égal à zéro et toujours non négatif;
  - ▶ éliminer un terme dans le temps actuel (ici, si le coût était  $c$ , alors le potentiel était  $2 - c$ )



**Figure 11.5** The insertion cost and potential change for each operation in a sequence

Pour les files binomiales, les temps amortis sont :

- ▶ insertion :  $O(1)$
- ▶ extraire minimum :  $O(\log n)$
- ▶ fusion :  $O(\log n)$

---

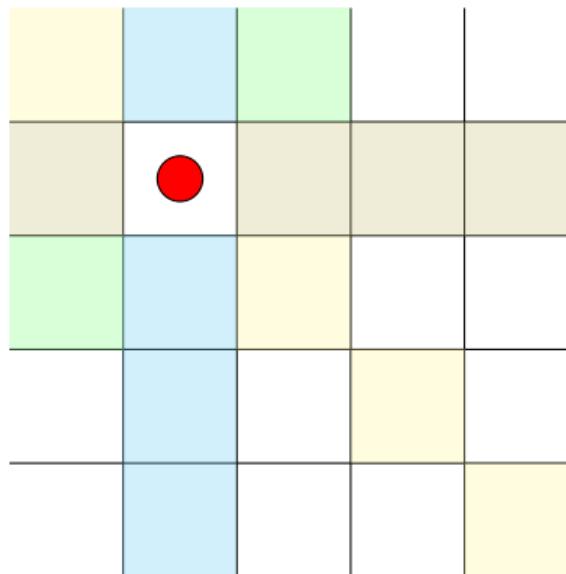
*Sources utilisées pour ce chapitre :*

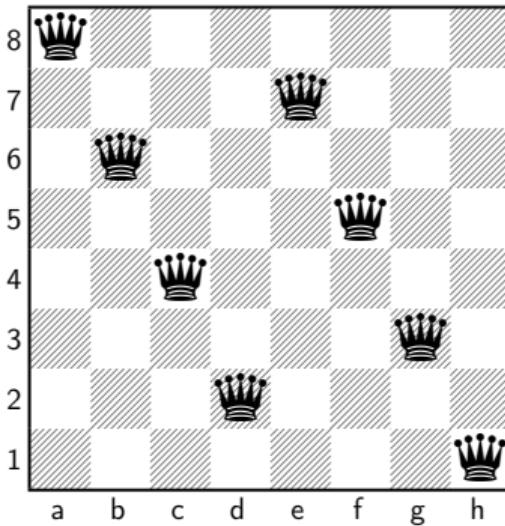
- Cormen, T., Leiserson, C., Rivest, R., Stein, C., *Introduction à l'algorithmique*, 2ème édition, Dunod, 2004.
- Weiss, M.A., *Data Structures and Algorithms Analysis in C++* 3ème édition, Addison-Wesley 2006.

# 10. Algorithmes de retour en arrière

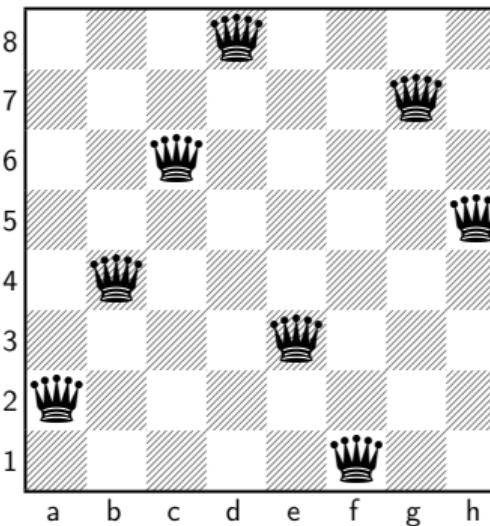
## 10.1. Introduction

- ▶ On utilise la technique de **retour en arrière** (*backtracking*) pour des problèmes où une **séquence** d'objets est choisie parmi un **ensemble** défini de sorte que la séquence satisfasse un certain **critère**.
- ▶ Exemple classique : le problème des  $n$  reines aux échecs
  - ▶ une reine menace une autre pièce si les deux sont dans la même ligne, colonne ou diagonale





a8 et h1 se menacent



aucune menace

Il y a 92 solutions possibles, en fait 12 distinctes si on tient compte des rotations et des réflexions<sup>5</sup>

---

5. voir [https://fr.wikipedia.org/wiki/Problème\\_des\\_huit\\_dames](https://fr.wikipedia.org/wiki/Problème_des_huit_dames)

## 10.2. Exemple - 4 reines

Placer 4 reines sur un échiquier  $4 \times 4$  sans qu'aucune ne se menace

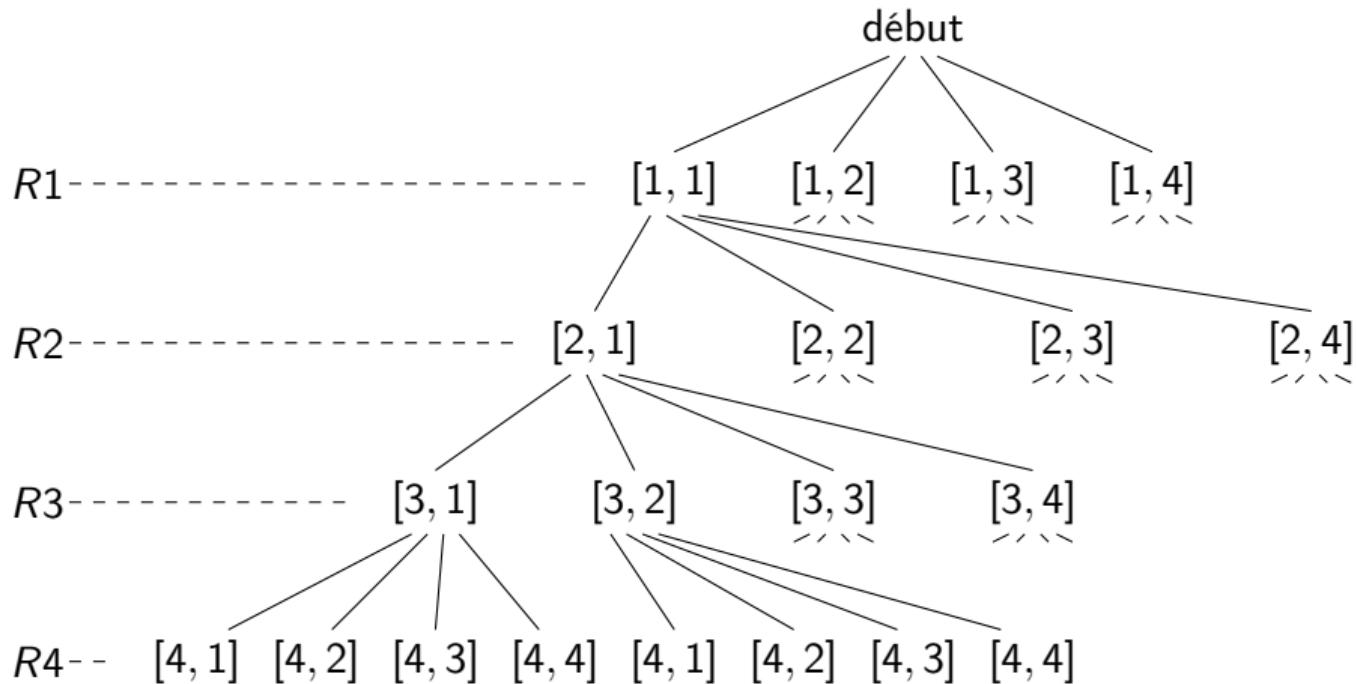
**Algorithme naïf** – examiner toutes les possibilités :

- ▶ Nombre de possibilités à vérifier :  $\binom{16}{4} = 43680$
- ▶ En général :  $\binom{n^2}{n}$

**Algorithme naïf amélioré** – éliminer des possibilités :

- ▶ Aucune reine ne peut être sur la même ligne
- ▶ Nombre de possibilités à vérifier :  $4 \times 4 \times 4 \times 4 = 256$
- ▶ En général :  $n^n$

## Arbre de l'espace d'état (arbre des possibilités)



On fait un parcours préfixe de l'arbre afin d'examiner chaque possibilité pour  $R1$ ,  $R2$ ,  $R3$  et  $R4$  :

1 (1,1), (2,1), (3,1), (4,1) -> pas possible

2 (1,1), (2,1), (3,1), (4,2) -> pas possible

3 (1,1), (2,1), (3,1), (4,3) -> pas possible

4 (1,1), (2,1), (3,1), (4,4) -> pas possible

5 (1,1), (2,1), (3,2), (4,1) -> pas possible

...

114 (1,2), (2,4), (3,1), (4,2) -> pas possible

115 (1,2), (2,4), (3,1), (4,3) -> ok

...

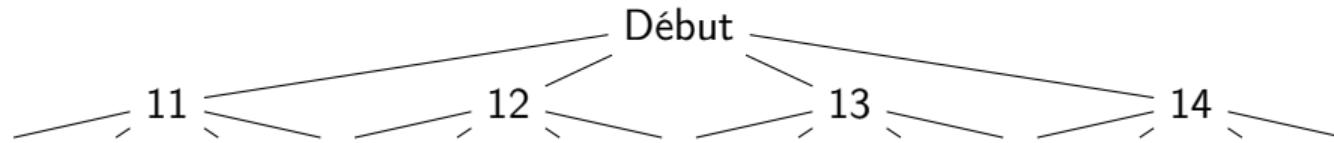
256 (1,4), (2,4), (3,4), (4,4) -> pas possible

## Algorithme naïf encore amélioré – éliminer encore des possibilités :

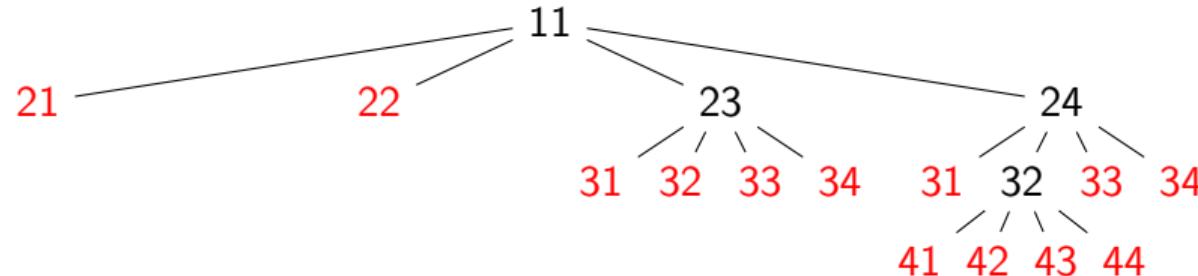
- ▶ Aucune reine ne peut être sur une même colonne
- ▶ On peut élaguer (*pruning*) des branches de l'arbre
- ▶ Le sous-arbre [2,1] dont le parent est [1,1] peut être éliminé car  $R_1$  et  $R_2$  ne peuvent être sur la même colonne
- ▶ Le sous-arbre [2,2] dont le parent est [1,1] peut être éliminé car  $R_1$  et  $R_2$  ne peuvent être sur la même diagonale
- ▶ C'est le principe du retour en arrière

## Arbre de l'espace d'état élagué

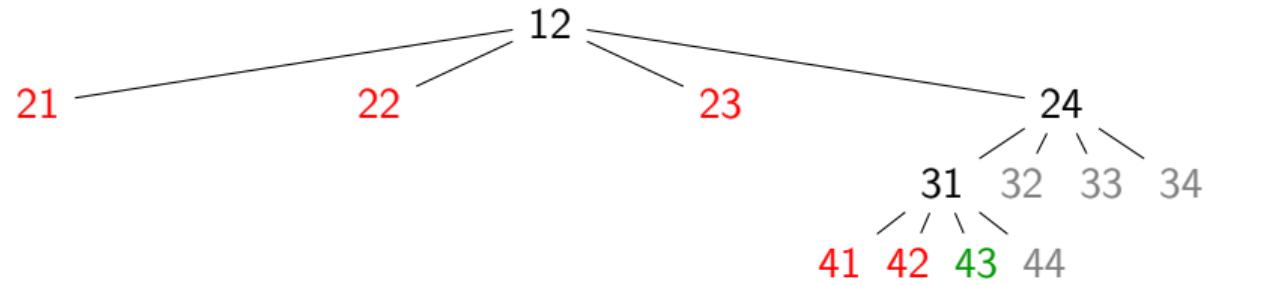
$[\ell, c] \rightarrow \ell c$



Détail du premier sous-arbre de début :



Détail du deuxième sous-arbre de début :



Il y a deux solutions : 12-24-31-43 (2,4,1,3) et 13-21-34-42 (3,1,4,2)

- ▶ Nœud non-prometteur : si on voit qu'il ne mènera pas à une solution

Algorithme général de l'approche du retour en arrière :

**procédure** verifierNoeud ( *v* )

    entrée : *v* : noeud

**début**

- 1     **si** estPrometteur ( *v* ) **alors**
- 2         **si** on a une solution avec *v* **alors**
- 3             écrire la solution
- 4         **sinon**
- 5             **pour** chaque enfant *u* de *v* **faire**
- 6                 verifierNoeud ( *u* )
- 7             **fin pour**
- 8         **fin si**
- 9     **fin si**

**fin** verifierNoeud

Variante de l'algorithme général de l'approche du retour en arrière :

**procédure** verifierLesEnfants ( *v* )

entrée : *v* : noeud

**début**

1      **pour** chaque enfant *u* de *v* **faire**

2          **si** estPrometteur ( *u* ) **alors**

3            **si** on a une solution avec *u* **alors**

4                écrire la solution

5            **sinon**

6                verifierLesEnfants ( *u* )

7                **fin si**

8                **fin si**

9            **fin pour**

**fin** verifierLesEnfants

### 10.3. Exemple - $n$ reines

- ▶ Soit  $Ri$  et  $Rk$ ,  $1 \leq i \leq n$ ,  $1 \leq k \leq n$ ,  $i \neq k$
- ▶ Soit  $C[i] =$  numéro de colonne de la reine  $Ri$  sur la ligne  $i$ ,  $1 \leq i \leq n$ .
- ▶ Afin que  $Ri$  et  $Rk$  ne se menacent pas, il faut qu'elles ne soient pas
  - ▶ dans la même ligne :  $i \neq j$
  - ▶ dans la même colonne :  $C[i] \neq C[k]$
  - ▶ dans la même diagonale :  $|C[i] - C[k]| \neq |i - k|$

**procédure** reines ( *i* )

entrée : *i* : numéro de la reine

**début**

- 1     **si** estPrometteur ( *i* ) **alors**
- 2         **si** *i* = *n* **alors**
- 3             écrire la solution *C*[1], ..., *C*[*n*]
- 4         **sinon**
- 5             **pour** *j*  $\leftarrow$  1 à *n* **faire**
- 6                 *C*[*i* + 1]  $\leftarrow$  *j*
- 7                 reines ( *i* + 1 )
- fin pour**
- fin si**
- fin si**
- fin** reines

**fonction** estPrometteur ( *i* ) **retourne** booléen

entrée : *i* : numéro de la reine

**début**

- 1       $k \leftarrow 1$
- 2      *menacee*  $\leftarrow$  **faux**
- 3      **tant que**  $k < i$  et non *menacee* faire
- 4        **si**  $C[i] = C[k]$  ou  $|C[i] - C[k]| = i - k$  **alors**
- 5            *menacee*  $\leftarrow$  **vrai**
- 6        **fin si**
- 7       $k \leftarrow k + 1$

**fin tant que**

**retourner** non *menacee*

**fin** estPrometteur

## Analyse

- ▶ Borne supérieure : nombre de nœuds dans l'arbre

$$1 + n + n^2 + \cdots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

▶

$n$	1	2	3	4	5	6	7	8
	1	7	40	341	3906	55987	960800	19173961

- ▶ Ça ne tient pas compte de l'élagage.
- ▶ C'est une borne supérieure pour les nœuds prometteurs.
- ▶ Deux reines ne peuvent être dans la même colonne.
- ▶  $1 + n + n \times (n - 1) + n \times (n - 1) \times (n - 2) + \cdots + n!$  nœuds.

▶

$n$	1	2	3	4	5	6	7	8
	1	5	16	65	326	1957	13700	109601

## Analyse empirique

$n$	4	8	12	14
1 Nb total nœuds	341	19 173 961	$9,73 \times 10^{12}$	$1,20 \times 10^{16}$
2 Nb permut. de $[n]$	24	40 320	$4,79 \times 10^8$	$8,72 \times 10^{10}$
3 Nb nœuds vérif (page 13) <i>(ligne 1 reines)</i>	61	15 721	$1,01 \times 10^7$	$3,78 \times 10^8$
Nb nœuds prometteurs : <i>(estPrometteur vrai, ligne 2)</i>	17	2057	$8,56 \times 10^5$	$2,74 \times 10^7$

- ▶ Algorithme 1 : Tous les nœuds de l'arbre des possibilités sont visités (aucun élagage).
- ▶ Algorithme 2 : Génération des  $n!$  possibilités où chaque reine est dans une ligne et une colonne différente des autres.
- ▶ Algorithme 3 : Retour en arrière avec élagage.
- ▶ Nb nœuds vérif par RA : Nombre total de nœuds de l'arbre de la page 9.
- ▶ Nb nœuds prometteurs : Nombre de nœuds internes + nombre de feuilles solution de l'arbre de la page 8.
- ▶ On veut minimiser le nombre de nœuds à vérifier.
- ▶ Il faut avoir une fonction `estPrometteur` efficace.

## Sources

- ▶ Neapolitan, Naimipour, *Foundations of Algorithms Using Java Pseudocode*, Addison Wesley 2007
- ▶ Cormen, Leiserson, Rivest, Stein, *Algorithmique 3e ed.*, Dunod 2010
- ▶ Wikipedia

# 11. NP-complétude

## 11.1. Introduction

# Introduction

Algorithme polynomial :  $T(n) = O(n^k)$  pour  $k \geq 0$ .

## ► Problèmes non résolvables

- Problème de l'arrêt d'une machine de Turing : "*Il n'existe pas de programme permettant de tester n'importe quel programme informatique d'un langage suffisamment puissant, tels tous ceux qui sont utilisés en pratique, afin de conclure dans tous les cas s'il s'arrêtera en un temps fini ou bouclera à jamais.*"<sup>6</sup>

## ► Problèmes résolvables mais plus que polynomial, i.e. non $O(n^k)$

- Ces problèmes sont qualifiés d'intractables ou **difficiles**
- Exemples : Commis voyageur, sac à dos, tours de Hanoï, etc.

## ► Problèmes résolvables en temps polynomial, i.e. $O(n^k)$

- Ces problèmes sont qualifiés de traitables ou **faciles**
- Exemples : Multiplication chainée de matrices, tri fusion, etc.

---

6. <https://fr.wikipedia.org/wiki/D%C3%A9cidabilit%C3%A9>

## 11.2. Bornes inférieures

## Bornes inférieures

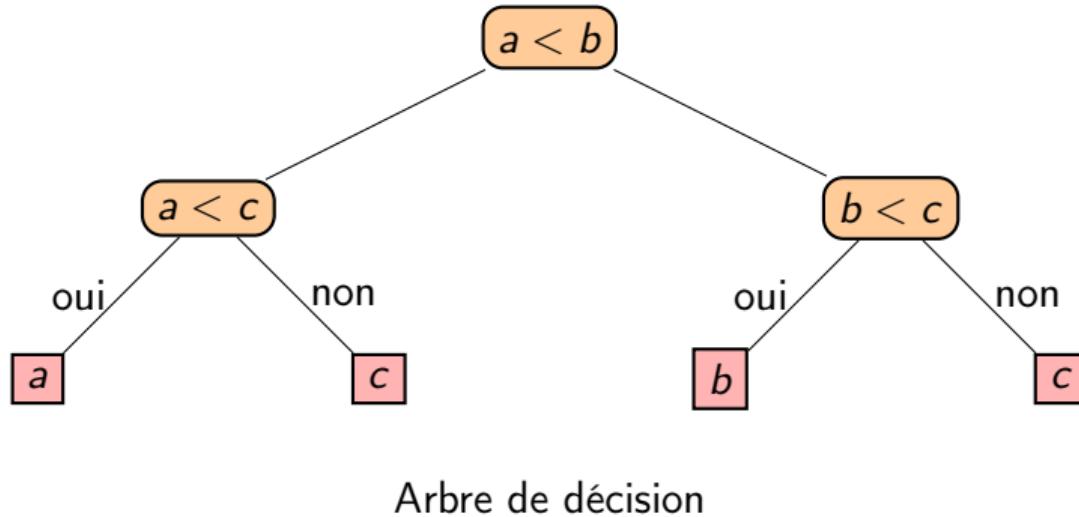
- ▶ Borne inférieure pour un problème : Travail minimal requis pour solutionner le problème peu importe l'algorithme utilisé.
- ▶ Borne inférieure **serrée** s'il existe un algorithme dont la complexité temporelle dans le pire cas est cette borne.
- ▶ Borne inférieure **triviale** : Combien d'items de l'entrée doivent être traités + combien d'items doivent être produits.

## Exemples

- Algorithme générant les  $n!$  permutations de  $n$  premiers entiers :  $\Omega(n!)$
- Fusionner deux listes triées de longueur  $n$  :  $\Omega(n)$
- Évaluation d'un polynôme  $p(n) = a_nx^n + a_{n-1}x^{n-1} + \dots + a_0$  :  $\Omega(n)$
- Nombre d'essais pour deviner un nombre entre 1 et  $n$  que quelqu'un a choisi :  $\Omega(\log n)$
- Multiplication de deux matrices carrées  $A \times B$  :  $\Omega(n^2)$ . On ne sait pas si borne serrée.
- Commis voyageur avec  $\frac{n(n-1)}{2}$  distances qui produit un circuit de  $n$  villes :  $\Omega(n^2)$ . On n'a rien trouvé de polynomial.

### 11.3. Arbre de décision

Trouver le minimum entre  $a$ ,  $b$  et  $c$



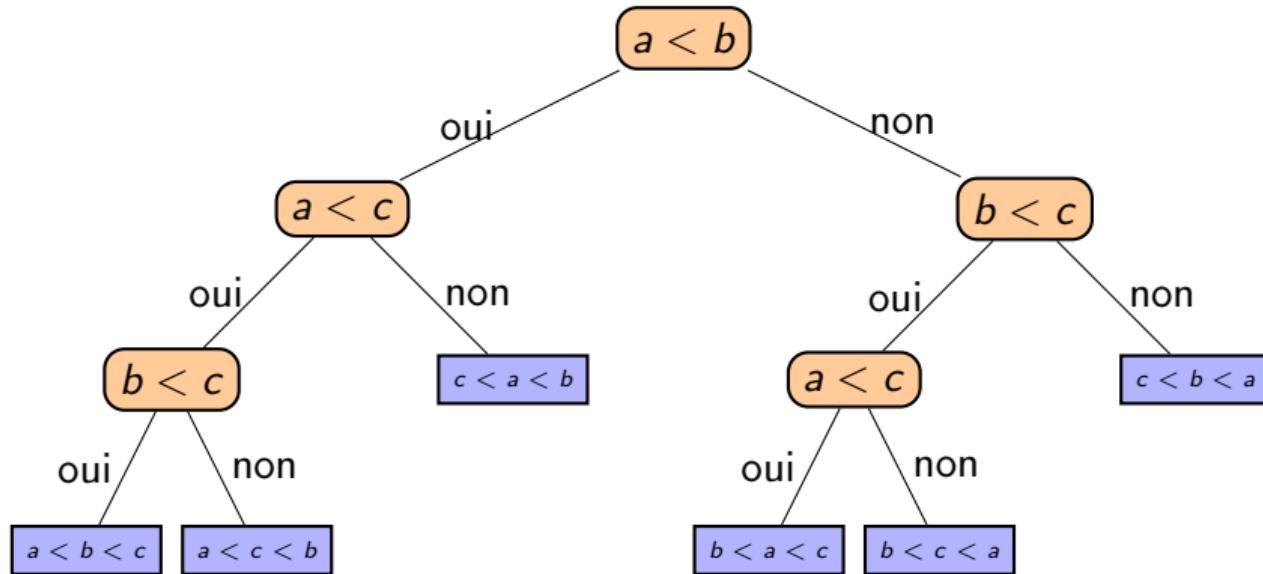
$f = \text{nombre de feuilles} \geq \text{nombre de réponses possibles}$

$h = \text{hauteur de l'arbre} = \text{nombre maximal de comparaisons}$

$$h \geq \lceil \log_2 f \rceil ; h + 1 \leq f \leq 2^h$$

## 11.4. Arbre de décision

Trier un tableau  $t$  où  $t_1 = a$ ,  $t_2 = b$  et  $t_3 = c$



Arbre de décision

## Algorithme de tri basé sur les comparaisons

- ▶ Tableau de taille  $n$ ,  $n!$  tableaux possibles.
- ▶  $f = \text{nombre de feuilles} \geq n!$
- ▶  $h = \text{hauteur de l'arbre} = \text{nombre maximal de comparaisons}$
- ▶  $h \geq \lceil \log_2 f \rceil \geq \lceil \log_2 n! \rceil$
- ▶ Formule de Stirling :  $n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$
- ▶  $\log_2 n! = \left(\log_2 \sqrt{2\pi} + \frac{1}{2} \log_2 n + n \log_2 n - n \log_2 e\right) \left(1 + \Theta\left(\frac{1}{n}\right)\right) = \Theta(n \log n)$
- ▶  $h = \Omega(n \log n)$

## 11.5. Réduction de problème

## Réduction de problème

Solutionner un problème en le réduisant à un problème dont on connaît la solution.

- ▶ Solutionner  $dx + e = 0$ . Utiliser la solution pour  $ax^2 + bx + c = 0$  en posant  $a = 0$ ,  $b = d$  et  $c = e$ .
- ▶ PPCM( $a, b$ ) : trouver la factorisation en nombres premiers de  $a$  et  $b$  et déduire PPCM( $a, b$ ). Plus efficace :  $\text{PPCM}(a, b) = \frac{ab}{\text{PGCD}(a, b)}$ . Utiliser l'algorithme d'Euclide pour trouver  $\text{PGCD}(a, b) = O(\log_{10} b)$  où  $b = \min(a, b)$
- ▶ Calcul du nombre de chemins de longueur  $k$  entre  $s_i$  et  $s_j$  dans un graphe :  $a_{i,j}^k$ . Utiliser l'exponentiation à la russe pour calculer  $A^k$ .
- ▶  $\min f(x) = - \max(-f(x))$

## Problèmes souvent utilisés pour la réduction de problème

Problème	Borne inférieure	Serrée ?
Trier	$\Omega(n \log n)$	oui (ex. : tri fusion)
Recherche dans tab. trié	$\Omega(\log n)$	oui (ex. : fouille binaire)
Prob. él. unique	$\Omega(n \log n)$	oui
Mult. entiers à $n$ chiffres	$\Omega(n)$	non ( $O(n^2)$ , $O(n^{1,585})$ )
Mult. matrices carrées	$\Omega(n^2)$	non ( $O(n^3)$ , $O(n^{2,808})$ )

## 11.6. Classes de problèmes

## Trois classes de problèmes

- ▶ **P** : la solution se trouve en temps polynomial
- ▶ **NP** : problèmes vérifiables en temps polynomial
  - ▶ Exemple : Circuit hamiltonien (passe une seule fois par tous les sommets d'un graphe). Trouver un circuit hamiltonien est plus que polynomial. Vérifier si un circuit donné est hamiltonien est polynomial.

On peut vérifier en temps polynomial si une solution donnée est correcte mais qu'en est-il de trouver la solution ?

- ▶ **NP-Complets**

## Classe des problèmes **NP-Complets** (statut inconnu)

Un problème NP-Complet :

- Appartient à **NP** et est aussi difficile que n'importe quel problème de **NP**
- Aucun algorithme polynomial trouvé à ce jour
- Aucune preuve de l'absence d'un algorithme polynomial
- Question appelée  $P \neq NP$  (posée en 1971)

S'il existe un problème NP-Complet résoluble en temps polynomial alors tous les problèmes NP-Complets seront aussi résolubles en temps polynomial.

### Problème polynomial

- Plus courts chemins à origine unique dans un graphe orienté (ex. : Algorithme de Dijkstra)  $O(|S||A|)$
- Tournée eulérienne (passe une seule fois par chaque arc)  $O(|A|)$
- Satisfaisabilité 2-FNC (2-SAT)

### Problème NP-Complet

- Plus longs chemins à origine unique dans un graphe orienté
- Existence d'un chemin de longueur au moins  $k$ .
- Circuit hamiltonien (passe une seule fois par chaque sommet)
- Satisfaisabilité 3-FNC (3-SAT)

- ▶ La NP-Complétude s'applique aux problèmes de décision mais pas directement aux problèmes d'optimisation.
- ▶ On peut transformer un problème d'optimisation en un de décision :

### Exemple

*Optimisation* : PLUS-COURT-CHEMIN Trouver le chemin utilisant le moins d'arêtes qui relie  $u$  à  $v$  dans le graphe  $G$ .

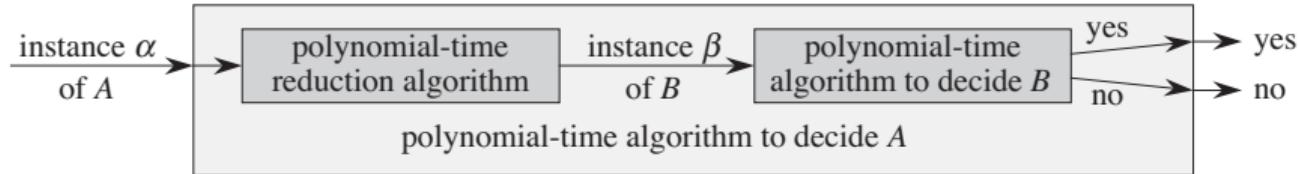
*Décision* : CHEMIN Existe-t-il un chemin de  $u$  à  $v$  dans le graphe  $G$  qui utilise au plus  $k$  arêtes.

- ▶ Un problème de décision est plus facile (pas plus difficile) que le problème d'optimisation correspondant.
- ▶ Fournir la preuve qu'un problème de décision est difficile prouve que le problème d'optimisation qui lui est associé est aussi difficile.
- ▶ S'applique aussi lorsqu'on utilise un problème de décision pour résoudre un autre problème de décision.

## Réductibilité

On peut ramener un problème  $A$  à un problème  $B$  si on peut transformer une instance  $\alpha$  du problème  $A$  en instance  $\beta$  équivalente du problème  $B$

- ▶ La transformation prend un temps polynomial
- ▶ Réponse de l'algorithme A exécuté sur  $\alpha \iff$  Réponse de l'algorithme B exécuté sur  $\beta$



Source de l'image : CLRS - Introduction to algorithms - 3e ed, page 1052

## Exemple

- ▶ Problème A : CHEMIN (Existe-t-il un chemin de  $u$  à  $v$  dans le graphe  $G$  qui utilise au plus  $k$  arêtes.)
- ▶ Problème B : autre problème de décision dans  $\mathbf{P}$  dont on connaît la solution

Instance de CHEMIN :  $\alpha = (G, u, v, k)$

Transformer  $\alpha$  pour en faire une instance  $\beta$  de B en temps polynomial.

## Le problème SAT

C'est un problème de décision : étant donné une formule de logique propositionnelle, déterminer s'il existe une assignation des variables propositionnelles qui rend la formule vraie.

- ▶ **Définition** : Un littéral est une expression booléenne qui est soit une formule atomique ( $V$ ,  $F$  ou une variable booléenne) ou la négation d'une formule atomique.
- ▶ **Définition** : Soit  $\phi$ , une expression booléenne.  $\phi$  est en forme normale conjonctive (FNC) si  $\phi$  est de la forme  $\bigwedge_{j=1}^m C_j$  où chaque  $C_j$  (clause de  $\phi$ ) est une disjonction de littéraux.

## Le problème k-SAT

Soit  $\phi$  une expression booléenne en FNC où chaque clause contient  $k \geq 2$  littéraux, (appelée aussi formule de Krom lorsque  $k = 2$ ). Existe-t-il une affectation aux variables qui rende  $\phi$  vraie ?

### Exemple 2-SAT

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

On remarque que  $a \vee b \leftrightarrow \neg a \rightarrow b$  ou bien  $\neg b \rightarrow a$ . On construit un graphe avec les arcs  $(\neg a, b)$  et  $(\neg b, a)$  pour chaque clause  $a \vee b$ . Une instance du problème est satisfaisable si aucune variable et sa négation n'appartiennent à la même composante fortement connexe<sup>7</sup>. Problème résolvable en temps linéaire, problème dans la classe **P**.

$k$ -SAT est NP-Complet pour  $k > 2$

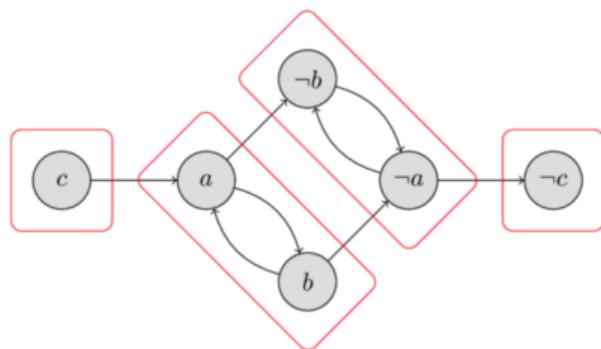
---

7. Composante fortement connexe : il existe un chemin entre chaque paire de sommets.

## Exemple de résolution de problème 2-SAT

$$\phi = (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c)$$

Graphe induit avec ses composantes connexes en rouge :



Aucune variable et sa négation ne se retrouvent dans la même composante fortement connexe. Il existe donc une assignation aux variables rendant  $\phi$  vraie.

Source de l'image : <https://cp-algorithms.com/graph/2SAT.html>

## 11.7. Problèmes abstraits

## Problèmes abstraits

**Définition :** Un **problème abstrait**  $Q$  est une relation binaire  $R \subseteq I \times S$  où

- ▶  $I$  : ensemble des instances de  $Q$
- ▶  $S$  : ensemble des solutions

Pour un problème de décision,  $S = \{0, 1\}$  où 1 = oui et 0 = non.

**Exemple :** Si  $i = (G, u, v, k)$  est une instance de CHEMIN alors  $\text{CHEMIN}(i) = 1$  s'il existe un chemin de longueur  $k$  entre  $u$  et  $v$  dans  $G$ , 0 sinon.

## 11.8. Encodages

# Encodages

**Définition :** Un **encodage** d'un ensemble  $S$  d'objets abstraits est une fonction  
 $e : S \rightarrow \{0, 1\}^*$

## Exemples

- ▶  $S = \mathbb{N}$ ,  $e(S) = \{0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, \dots\}$
- ▶  $S$  : caractères unicode,  $e(A) = 1000001$

Tout peut être représenté par des chaînes binaires : polygones, graphes, polynomes, fonctions, programmes, etc.

L'efficacité de la résolution d'un problème ne doit pas dépendre de l'encodage choisi.

- ▶ Algorithme qui résout un **problème de décision abstrait** : prend en entrée un encodage d'une instance du problème.
- ▶ **Problème concret** : l'ensemble de ses instances sont des chaînes binaires.
- ▶ Algorithme qui résout un **problème concret** en temps  $O(T(n))$  : Sur une instance  $i$  de longueur  $n = |i|$  il produit la solution en temps  $O(T(n))$ .
- ▶ **Problème concret** : Résoluble en temps polynomial s'il existe un algorithme de complexité temporelle  $O(n^k)$  pour une certaine constante  $k$ .
- ▶ Classe de complexité P : ensemble des problèmes de décision concrets résolubles en temps polynomial.

**Définition** : Une fonction  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  est **calculable en temps polynomial** s'il existe un algorithme  $A$  à temps polynomial qui produit  $f(x)$  où  $x \in \{0, 1\}^*$ , une entrée de  $A$ .

**Définition** : Pour un certain ensemble  $I$  d'instances, deux encodages  $e_1$  et  $e_2$  sont **reliés polynomialement** s'il existe deux fonctions calculables en temps polynomial  $f_{12}$  et  $f_{21}$  telles que  $\forall i \in I$ ,  $f_{12}(e_1(i)) = e_2(i)$  et  $f_{21}(e_2(i)) = e_1(i)$ .

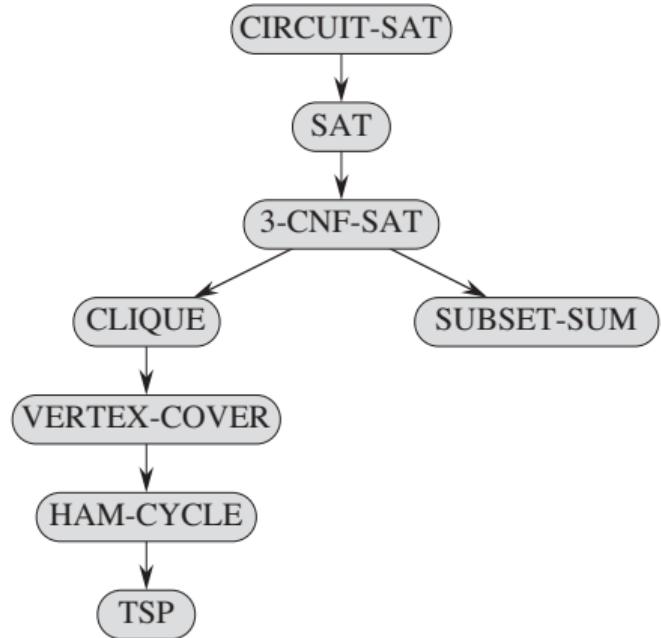
# Théorie des langages formels

- ▶ Alphabet  $\Sigma$  : ensemble fini de symboles (exemple  $\Sigma = \{0, 1\}$ )
- ▶  $\Sigma^*$  : Ensemble de toutes les chaînes possibles avec des symboles de  $\Sigma$  incluant la chaîne vide. (exemple :  $\{\varepsilon, 0, 1, 10, 11, 100, 101, 110, \dots\}$ )
- ▶ Langage  $L$  :  $L \subseteq \Sigma^*$
- ▶ L'ensemble des instances d'un problème de décision  $Q$  est l'ensemble  $\{0, 1\}^*$  et  $Q$  est un langage  $L$  tel que  $L = \{x \in \{0, 1\}^* : A(x) = 1\}$

- ▶ Un algorithme  $A$  **accepte** une chaîne  $x \in \{0, 1\}^*$  si l'algorithme produit le résultat  $A(x) = 1$
- ▶ Un algorithme  $A$  **rejette** une chaîne  $x \in \{0, 1\}^*$  si l'algorithme produit le résultat  $A(x) = 0$
- ▶ Langage accepté par  $A$  :  $L = \{x \in \{0, 1\}^* : Q(x) = 1\}$
- ▶ Le langage  $L$  est **décidé en temps polynomial** par un algorithme  $A$  s'il est accepté par  $A$  et que  $\forall x \in \{0, 1\}^*, |x| = n$ , l'algorithme décide correctement si  $x \in L$  en temps  $O(n^k)$  pour une constante  $k$ .
- ▶  $\mathbf{P} = \{L \subseteq \{0, 1\}^* : \exists A \text{ qui décide } L \text{ en temps polynomial}\}$
- ▶ Exemple : EST-PREMIER( $n$ ),  $L = \{10, 11, 101, 111, 1011, 1101, 10001, \dots\}$

## 11.9. Exemples de problèmes NP-Complets

# Problèmes NP-Complets

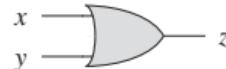
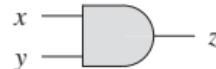
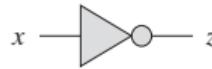


Source de l'image : CLRS - Introduction to algorithms - 3e ed, page 1087

- ▶ CIRCUIT-SAT : Existe-t-il une affectation aux variables qui rende le circuit satisfaisable ?
- ▶ SAT : Existe-t-il une affectation aux variables qui rende l'expression booléenne satisfaisable ?
- ▶ 3-FNC-SAT (3-CNF-SAT) : Existe-t-il une affectation aux variables qui rende l'expression booléenne sous forme normale conjonctive satisfaisable ?
- ▶ CLIQUE : Existe-t-il une clique de taille  $k$  dans le graphe  $G$  ?
- ▶ COUVERTURE-SOMMETS (VERTEX-COVER) : Existe-t-il une couverture de sommets de taille  $k$  dans  $G$  ?
- ▶ On peut montrer que CIRCUIT-SAT est NP-Complet.
- ▶ Il servira de base pour montrer que SAT, 3-FNC-SAT, CLIQUE et COUVERTURE-SOMMETS sont aussi NP-Complets en utilisant des algorithmes de réduction en temps polynomial.

# Problèmes NP-Complets

## ► CIRCUIT-SAT – Satisfaisabilité de circuit



$x$	$\neg x$
0	1
1	0

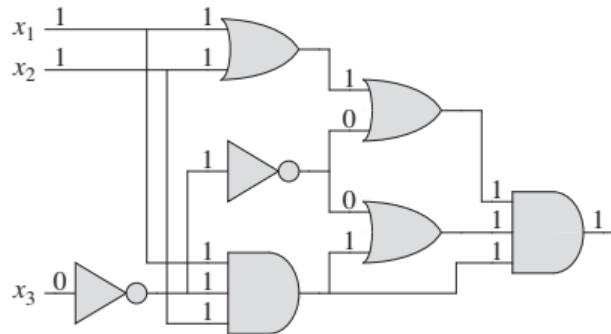
(a)

$x$	$y$	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

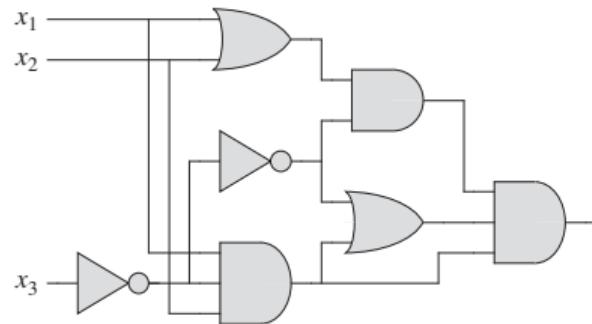
(b)

$x$	$y$	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

(c)



(a)



(b)

Source des images : CLRS - Introduction to algorithms - 3e ed, pages 1071-1072

Algorithme naïf en  $\Omega(2^k)$  : vérifier toutes les  $2^k$  possibilités pour les  $k$  variables.

On peut montrer que ce problème est NP-Complet.

► SAT – Satisfaisabilité booléenne

Déterminer si une formule booléenne est satisfaisable.

- ▶  $n$  variables booléennes
- ▶  $m$  connecteurs logiques
- ▶ des parenthèses

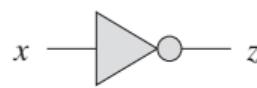
Exemple :  $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

$\phi$  donne 1 pour  $x_1 = 0$ ,  $x_2 = 0$ ,  $x_3 = 1$  et  $x_4 = 1$

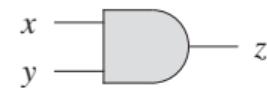
Algorithme naïf en  $\Omega(2^k)$  : vérifier toutes le  $2^k$  possibilités pour les  $k$  variables.

## Réduction de CIRCUIT-SAT vers SAT

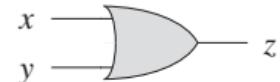
- ▶ Ajouter une variable au bout de chaque porte logique
- ▶ Pour chaque porte logique, écrire l'expression booléenne lui correspondant :



$$z \leftrightarrow \neg x$$

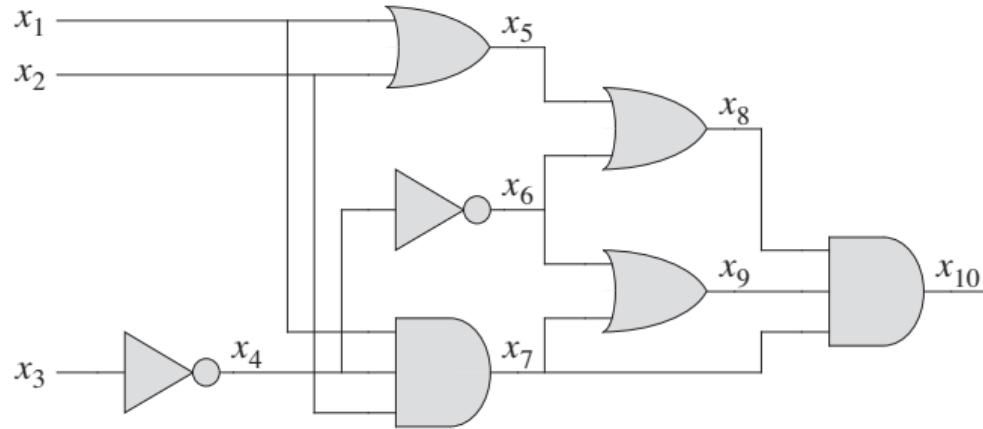


$$z \leftrightarrow x \wedge y$$



$$z \leftrightarrow x \vee y$$

- ▶ Relier toutes les expressions obtenues par des  $\wedge$  puis ajouter  $\wedge$  avec la variable de la dernière porte puisque nous voulons une assignation qui donne la valeur vrai à cette variable.



Source de l'image : CLRS - Introduction to algorithms - 3e ed, page 1081

$$\begin{aligned}\phi = & \quad x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \wedge (x_6 \leftrightarrow \neg x_4) \wedge \\ & (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \wedge \\ & (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))\end{aligned}$$

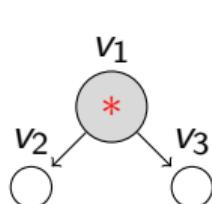
Puisque CIRCUIT-SAT est NP-Complet alors SAT l'est aussi.

## 3-FNC-SAT

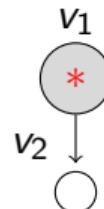
- ▶ 3-FNC-SAT – Satisfaisabilité booléenne en FNC  
Déterminer si une formule booléenne est satisfaisable.
  - ▶  $n$  variables booléennes
  - ▶  $m$  connecteurs logiques
  - ▶ des parenthèses

## Réduction de SAT vers 3-FNC-SAT

- ▶ Construire un arbre représentant l'expression logique  $\phi$ . Les feuilles seront les variables, les nœuds internes seront les connecteurs logiques.
- ▶ Ajouter une variable pour chaque nœud interne.
- ▶ Pour chaque nœud interne, écrire l'expression booléenne lui correspondant ( $*$   $\in \{\vee, \wedge, \leftrightarrow, \rightarrow, \neg\}$ ) :



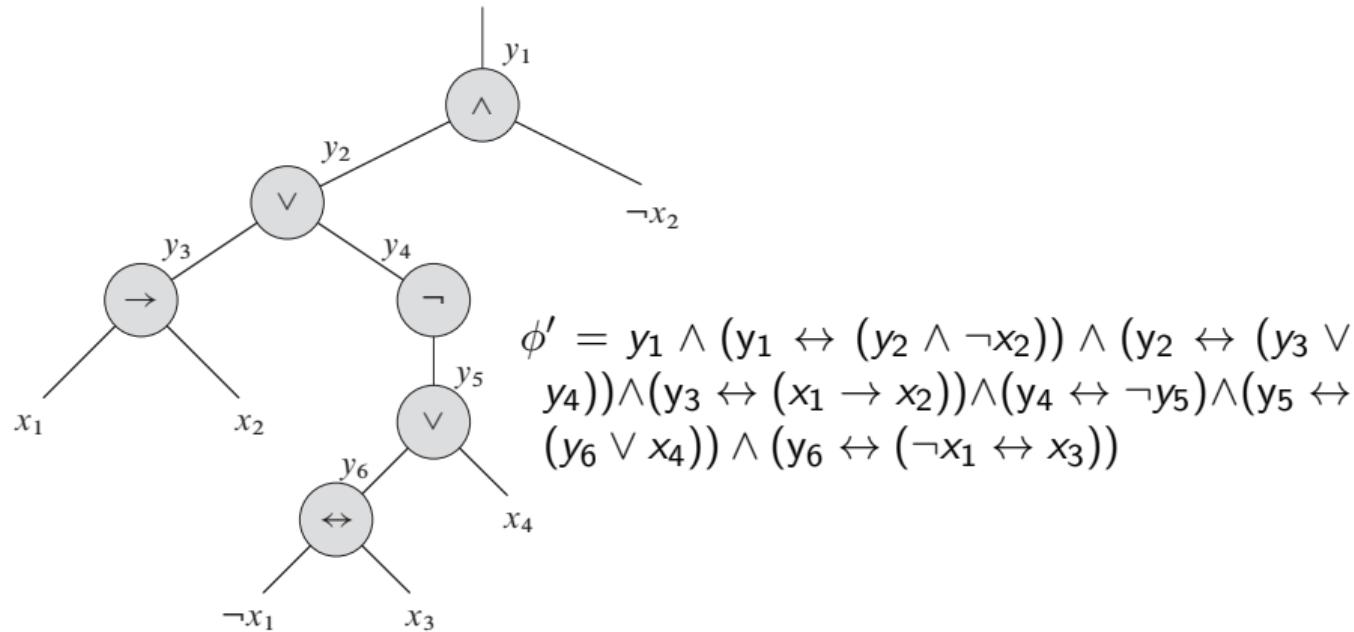
$$v_1 \leftrightarrow v_2 * v_3$$



$$v_1 \leftrightarrow *v_3$$

- ▶ Relier toutes les expressions obtenues par des  $\wedge$  puis ajouter  $\wedge$  avec la variable de la racine puisque nous voulons une assignation qui donne la valeur vrai à cette variable. Ceci donne l'expression  $\phi'$ .

Exemple :  $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$



- ▶ On pose  $\phi' = \bigwedge_{i=1}^k \phi'_i$  où chaque  $\phi'_i$  contient au plus 3 littéraux
- ▶ Il faut convertir chaque  $\phi'_i$  en une suite de OU de 3 littéraux
- ▶ Pour chaque  $\phi'_i$  qui n'est pas une suite de littéraux, faire une table de vérité qui contiendra au plus 8 lignes ( $2^3$ )
- ▶ Exemple :  $\phi'_1 = y_1 \leftrightarrow (y_2 \wedge \neg x_2)$

$y_1$	$y_2$	$x_2$	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

$$\neg\phi'_1 = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

$$\phi'_1 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

- ▶ Chaque clause contient au plus 3 littéraux.
- ▶ Il faut que chaque clause contienne exactement 3 littéraux.
  - ▶ Celles ayant 2 littéraux de la forme  $(a \vee b)$  : on remplace par  $(a \vee b \vee c) \wedge (a \vee b \vee \neg c)$
  - ▶ Celles ayant 1 littéral de la forme  $(a)$  : on remplace par  $(a \vee b \vee c) \wedge (a \vee b \vee \neg c) \wedge (a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg c)$
- ▶ L'instance de départ de SAT a été réduite à une instance de 3-FNC-SAT
- ▶ Puisque SAT est NP-Complet alors 3-FNC-SAT l'est aussi.

## CLIQUE

Une **clique** dans un graphe non-orienté  $G = (S, A)$  est un sous-ensemble  $S' \subseteq S$  de sommets dont chaque paire est reliée par une arête de  $A$ , i.e. c'est un sous-graphe complet de  $G$  (chaque sommet est connecté à tous les autres sommets).

Problème CLIQUE : Trouver la clique de taille maximale dans un graphe.

On peut réduire 3-FNC-SAT vers CLIQUE.

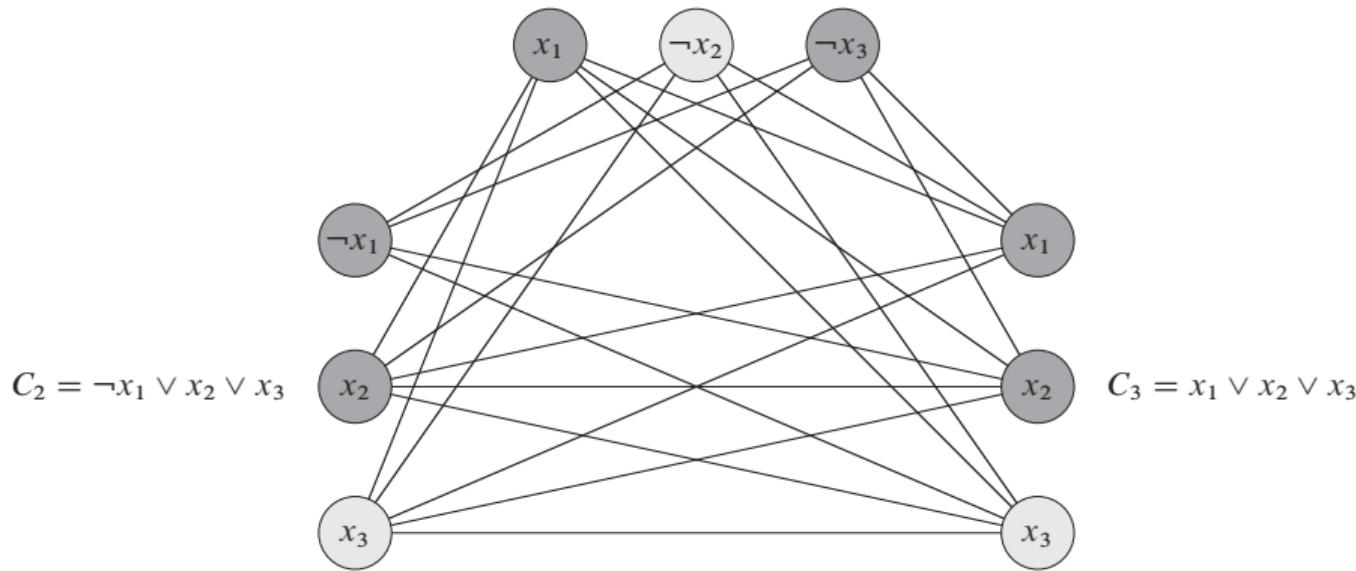
Puisque 3-FNC-SAT est NP-Complet alors CLIQUE l'est aussi.

## Réduction de 3-FNC-SAT vers CLIQUE

- ▶ Soit  $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$  une formule booléenne en FNC.
- ▶  $C_r = (\ell_1^r \vee \ell_2^r \vee \ell_3^r)$
- ▶ On construit  $G = (S, A)$  (se fait en temps polynomial) :
  - ▶ Pour chaque  $C_r$  on place 3 sommets  $v_1^r$ ,  $v_2^r$  et  $v_3^r$  dans  $S$
  - ▶ On met un arc entre  $v_i^r$  et  $v_j^s$  si :
    - ▶  $r \neq s$
    - ▶  $I_i^r$  n'est pas la négation de  $I_j^s$
- ▶  $\phi$  est satisfaisable si et seulement si  $G$  possède une clique de taille  $k$ .

**Exemple** :  $\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$

$$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$$



Une assignation qui satisfait  $\phi$  :  $x_1 = 0$  ou  $1$ ,  $x_2 = 0$  et  $x_3 = 1$ .

Clique : Sommets en gris clair

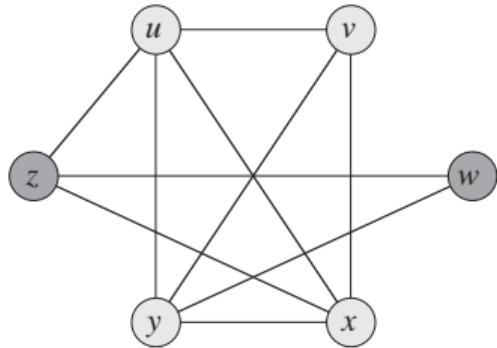
## COUVERTURE-SOMMETS

Une couverture de sommets d'un graphe non-orienté  $G = (S, A)$  est un sous-ensemble  $S' \subseteq S$  tel que si  $(u, v) \in A$  alors  $u \in S'$  ou (inclusif)  $v \in S'$  (chaque arête a au moins une de ses extrémités dans  $S'$ ).

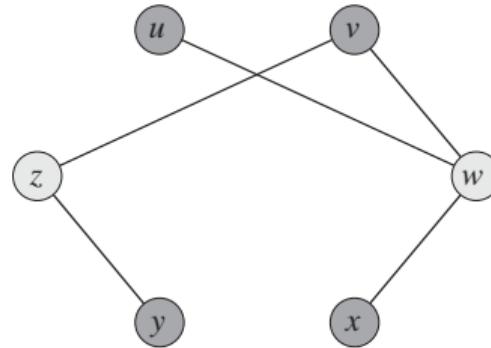
Problème COUVERTURE-SOMMETS : trouver l'ensemble  $S$  de cardinalité minimale.

## Réduction de CLIQUE vers COUVERTURE-SOMMETS

- ▶  $G = (S, A)$
- ▶ Complémentaire de  $G$  :  $\overline{G} = (S, \overline{A})$  où  
 $\overline{A} = \{(u, v) : u \in S, v \in S, u \neq v, (u, v) \notin A\}$



(a)



(b)

Dans (a) : clique  $= S' = \{u, v, x, y\}$ .

Dans (b) : Couverture de sommets  $= S - S' = \{w, z\}$

## Sources

- ▶ Anany Levitin, *The design & analysis of algorithms* 2e ed., Addison Wesley 2007
- ▶ Cormen, Leiserson, Rivest, Stein, *Algorithmique* 3e ed., Dunod 2010
- ▶ Wikipedia

# 12. Introduction à la théorie de l'information

## 12.1. Introduction

## Introduction

- ▶ Seconde moitié du 20ème siècle : développement de systèmes où **l'information transmise est codée sous forme numérique.**
- ▶ Transmission simultanée d'informations de nature très différentes : données, signaux audio, TV, etc.
- ▶ Fondements théoriques : Claude Shannon, articles sur les *Théories mathématiques des communications* (1948)  
Problèmes mathématiques :
  1. Comment convertir de façon optimale des signaux analogiques en signaux numériques ?
  2. Comment transmettre sans erreur un message numérique sur une voie de transmission bruyante ?

Source d'information -> Codage de source -> Codage de canal -> Canal de transmission -> Décodage de canal -> Décodage de source -> Destination de l'information

## Le codage de source

[Problème 1.] Comment convertir de façon optimale des signaux analogiques en signaux numériques ?

- ▶ On considère les ensembles [source + codage de source] et [décodage de source + destinataire].
- ▶ Ils sont reliés par une voie de transmission fictive non bruyante : l'ensemble [codage de canal + voie de transmission + décodage de canal].
- ▶ La source d'information engendre l'information sous forme numérique.

## Le codage de source – Cas d'une source numérique

- ▶ Établir une correspondance bijective entre les éléments de l'alphabet de source A et les mots-codes constitués à l'aide de l'alphabet de code B.
- ▶ Si B a seulement deux symboles (0,1) : codage binaire  
Si B a  $q$  éléments : codage  $q$ -aire.
- ▶ Exemple : le code morse utilise un alphabet ternaire  $B = \{ ., -, \text{ blanc} \}$

## Le codage de source – Cas d'une source numérique

- ▶ Le codage étant bijectif et le canal non bruité, le décodage se fait sans ambiguïté.
- ▶ Objectif dans le cas d'une source numérique : réduire au maximum la représentation de l'information
  - la réduction (compression) de données dépend du débit de symboles nécessaires à la représentation complète du signal de source.
- ▶ Ce débit est lié à un paramètre de la source : **l'entropie** (de Shannon).

## Le codage de source – Cas d'une source analogique

- ▶ Source de type analogique : le signal engendré ne peut pas être représenté par une suite finie de symboles d'un alphabet fini.
- ▶ Conversion analogique vers numérique : alphabet de source A infini *versus* alphabet de code infini  
⇒ perte d'information *i.e.* "distorsion" entre signal source et signal reconstruit.

## Le codage de canal

[Problème 2.] Comment transmettre sans erreur un message numérique sur une voie de transmission bruyante ?

- ▶ Opérations de codage et décodage de canal, la voie de transmission étant considérée comme bruyante.
- ▶ Ce codage/décodage a pour objectif d'annuler les effets du bruit présent sur la voie de transmission qui causerait des erreurs de décodage à la réception  $\Leftrightarrow$  réaliser une voie de transmission fictive non bruyante.
- ▶ Théorème faisant apparaître la **capacité** d'une voie de transmission (Shannon) : si le débit de symboles à l'entrée du codeur de voie de transmission est inférieur à la capacité de la voie, on peut trouver un codage permettant de reconstruire la séquence sans erreur.

## 12.2. L'information et sa mesure

## L'information et sa mesure

- ▶ Qu'est-ce qui fait que certains messages apportent plus d'information que d'autres ?
  
- **Quantifier l'information !**

# L'information et sa mesure

Propriétés :

1. La quantité d'information d'un événement est lié au degré d'incertitude de cet événement.
2. Plus l'événement est incertain, plus il contient d'informations.
3. Si deux événements sont indépendants, l'information qu'ils contiennent ensemble est égale à la somme des informations propres à chacun pris séparément.

## L'information et sa mesure

La mesure d'information d'un événement  $\alpha$  de probabilité d'occurrence  $p(\alpha)$  est notée :

$$I(\alpha) = -\log p(\alpha)$$

- ▶ En base 2, l'unité d'information est le *bit* (choix par défaut dans ces notes). On écrira souvent  $I(\alpha) = h(\alpha) = -\log_2 p(\alpha)$
  
- ▶ En base  $e$ , l'unité d'information est le *nat*.

**Exemple** : Un candidat a 50% de chance de réussir un test.  
Annoncer sa réussite apporte donc une information de 1 bit.

## Information relative à une variable aléatoire - Entropie

$X$ , variable aléatoire à valeurs dans  $\{a_1, \dots, a_k, \dots, a_m\}$  avec les probabilités  $\{p_X(a_1), \dots, p_X(a_k), \dots, p_X(a_m)\}$ .

Si  $X$  vaut  $x$ , la quantité d'information fournie est  $I(x) = -\log p(x)$

L'espérance mathématique de  $I(X)$  est appelée l'**entropie** de la variable aléatoire  $X$  :

$$H(X) = - \sum_{k=1}^m p_X(a_k) \log p_X(a_k)$$

Soit :

$$H(X) = - \sum_x p(x) \log p(x)$$

Rq :  $I(x)$  mesure l'*incertitude* a priori sur le fait que  $X$  puisse prendre la valeur  $x$ . Dès que cette valeur est réalisée, l'incertitude devient *information*.

## Information mutuelle

$(X, Y)$  couple de variables aléatoires **non indépendantes** à valeurs dans  $\{a_k, k = 1, 2, \dots\} \times \{b_j, j = 1, 2, \dots\}$  caractérisé par une distribution de probabilités  $p_{XY}(a_k, b_j)$ .

- ▶ L'incertitude initiale (a priori) de l'événement  $x = a_k$  (*i.e.* en l'absence de l'information  $y = b_j$ ) est  $-\log p_X(a_k)$ .
- ▶ L'incertitude finale (a posteriori, *i.e.* quand on sait que  $y = b_j$ ) de l'événement  $x = a_k$  est  $-\log p_{X|Y}(a_k|b_j)$ .

La différence d'incertitude entre les deux situations est donc :

$$I_{X|Y}(a_k, b_j) = -\log p_X(a_k) - (-\log p_{X|Y}(a_k|b_j)) = \log \frac{p_{X|Y}(a_k|b_j)}{p_X(a_k)}$$

## Information mutuelle

On a :

$$I_{Y|X}(b_j, a_k) = \log \frac{p_{Y|X}(b_j|a_k)}{p_Y(b_j)} = I_{X|Y}(a_k, b_j)$$

On appelle  $I_{Y|X}(b_j, a_k) = I_{X|Y}(a_k, b_j)$  **information mutuelle** entre les événements  $x = a_k$  et  $y = b_j$  que l'on note :

$$I(x, y) = \log \frac{p(x|y)}{p(x)}$$

L'espérance de  $I(x, y)$  sur toutes les valeurs possibles de  $x$  et  $y$  est (l'espérance de) l'information mutuelle entre  $X$  et  $Y$  :

$$I(X, Y) = \sum_x \sum_y I(x, y)p(x, y)$$

Rq :  $I(X, Y)$  caractérise conjointement le couple de variables aléatoires  $(X, Y)$  tandis que  $I(x, y)$  caractérise une réalisation particulière  $(x, y)$ .

## Information mutuelle et conditionnelle

On peut également définir l'**information propre conditionnelle** de l'événement  $x = a_k$  sachant que l'événement  $y = b_j$  est réalisé :

$$I_{X|Y}(a_k|b_j) = \log \frac{1}{p_{X|Y}(a_k|b_j)} \text{ que l'on note } I(x|y) = \log \frac{1}{p(x|y)}$$

→ information nécessaire à un observateur pour déterminer complètement l'événement  $x = a_k$  s'il sait déjà que  $y = b_j$ .

C'est donc l'incertitude propre à l'événement  $x = a_k$  sachant que  $y = b_j$ .

## Information mutuelle et conditionnelle

On peut alors calculer l'**entropie conditionnelle** :

$$H(X|Y) = \sum_{x,y} I(x|y)p(x,y)$$

On déduit de  $I(x,y) = I(x) - I(x|y)$  et de l'équation précédente que :

$$I(X, Y) = H(X) - H(X|Y)$$

Rq : L'espérance de l'information fournie sur  $X$  par la réalisation de  $Y$  (information mutuelle entre  $X$  et  $Y$ ) est égale à l'espérance de l'incertitude a priori sur  $X$  moins l'incertitude restante (a posteriori) sur  $X$  après observation de  $Y$ .

## 12.3. Codages de source classiques

# Source discrète sans mémoire

## Définition

Une source est dite **discrète et sans mémoire** si :

- ▶ elle utilise un alphabet fini ;
- ▶ les symboles de source sont engendrés selon une distribution de probabilités fixée ;
- ▶ les symboles successifs sont statistiquement indépendants.

Rq : On peut démontrer que la source discrète sans mémoire d'entropie  $H$  maximale est celle pour laquelle les  $m$  symboles de l'alphabet sont équiprobables.

## Définition du codage de source

Soit  $A$  un alphabet de source fini et  $B$  un alphabet de code également fini.

Le **codage de source** est une règle qui permet d'assigner à chaque symbole de source (de  $A$ ) un mot-code constitué à l'aide des symboles de l'alphabet de code  $B$ .

On peut donc définir un code de source comme l'ensemble des mots-codes  $\{K(a_1), \dots, K(a_m)\}$  où :

$A = \{a_1, \dots, a_m\}$  est l'alphabet de source ( $|A| = m$ ) ;

$K$  est la fonction faisant correspondre un mot-code à chaque symbole de source.

Rq : Si  $B$  est binaire, le codage est dit binaire.

## Code décodable sans ambiguïté, code séparable

Un code est *décodable sans ambiguïté* si, pour toute séquence de symboles de source de longueur finie, la séquence de symboles de code qui lui correspond est différente de toute autre séquence de symboles de code correspondant à toute autre séquence de source.

Cette propriété étant difficile à vérifier en pratique, on s'intéresse à une classe de code plus restrictive qui satisfait à la condition de non ambiguïté : la classe des codes **codes séparables**.

- ▶ Définition : Un code est dit **séparable** si aucun mot-code n'est le préfixe d'un autre mot-code.

Rq : Les codes séparables sont décodables sans ambiguïté et ont la propriété de permettre le décodage mot-code par mot-code, sans observer les mots-codes futurs (décodage en ligne ou instantané).

## Codage de Shannon-Fano

Le codage de Shannon-Fano fournit des codes séparables efficaces. La méthode nécessite de connaître la distribution de probabilités d'émission des symboles de source a priori.

Procédure d'encodage :

1. Classer les symboles de source dans l'ordre décroissant de leur probabilité d'émission ;
2. Diviser l'ensemble des symboles de source en deux sous-ensembles qui doivent avoir (presque) la même probabilité totale d'émission (somme sur tous les symbole du sous-ensemble). Attribuer un "0" comme premier bit de chaque mot-code des membres du premier sous-ensemble et un 1 comme premier bit de chaque mot-code des membres du second sous-ensemble ;
3. Itérer le processus sur chaque sous-ensemble jusqu'à ce que les deux sous-ensembles obtenus ne possèdent plus qu'un seul symbole.

## Codage de Shannon-Fano – Exemple

Soit une source définie par ses symboles et leur distribution de probabilités :

Symbole	1	2	3	4	5	6	7
Probabilité	0,4	0,1	0,1	0,1	0,1	0,1	0,1

Première division :

Symbole	1, 2	3,4,5,6,7
Probabilité	0,5	0,5
Code	0	1

Deuxième division :

Symbole	1	2
Probabilité	0,4	0,1
Code	00	01

Troisième division :

Symbole	3,4,5	6,7
Probabilité	0,3	0,2
Code	10	11

Quatrième division :

Symbole	3,4	5
Probabilité	0,2	0,1
Code	100	101

## Codage de Shannon-Fano – Exemple

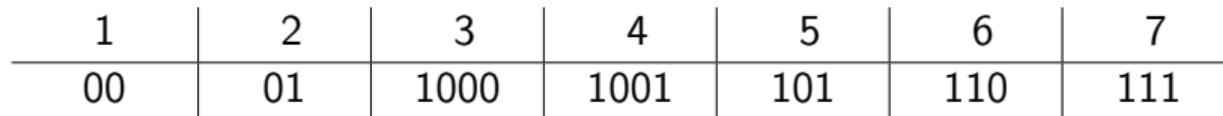
Cinquième division :

Symbol	3	4
Probabilité	0,1	0,1
Code	1000	1001

Sixième division :

Symbol	6	7
Probabilité	0,1	0,1
Code	110	111

Codage final (un des codes possibles) :



## Code optimal

Soit une source d'entropie  $H$  dont on code les mots émis avec  $K$  mots-codes de longueur  $\ell_k$ , chaque mot ayant la probabilité  $p(k)$ .

- ▶ La longueur moyenne (espérance mathématique) des mots-codes est :

$$\ell_{av} = \sum_{k=1}^K p(k)\ell_k$$

- ▶ Théorème fondamental du codage de source :

Pour toute source discrète sans mémoire d'entropie  $H$ , on peut trouver un code séparable  $q$ -aire décodable sans ambiguïté tel que

$$\frac{H}{\log q} \leq \ell_{av} \leq \frac{H}{\log q} + 1$$

## Code optimal – Efficacité d'un code

- ▶ L'efficacité d'un code est  $\frac{H}{\ell_{av} \log q}$
- ▶ Un code séparable est **optimal** pour toutes les distributions de probabilités si il n'existe aucun autre codage symbole par symbole qui possède une meilleure efficacité.
- ▶ Le codage binaire de Huffman fournit des codes séparables optimaux.

Dans l'exemple précédent (Shannon-Fano), on a  $\ell_{av} = 2,7$  bits et  $H = 2,5219$  bits. Le code de cet exemple a donc une efficacité de 93,4%.

Rq :  $\ell_{av} = 2 * 0,4 + (2 + 4 + 4 + 3 + 3 + 3) * 0,1$  et  $H = -(0,4 \log(0,4) + 6 * 0,1 * \log(0,1))$

## Codage de Huffman – Exemple

Soit une source définie par :

Symbol	A	E	B	C	D	F
Probabilité	0,4	0,2	0,1	0,1	0,1	0,1

Première réduction :

A	DF	E	B	C
0,4	0,2	0,2	0,1	0,1

Deuxième réduction :

A	BC	DF	E
0,4	0,2	0,2	0,2

Troisième réduction :

A	DEF	BC
0,4	0,4	0,2

Quatrième réduction :

ABC	DEF
0,6	0,4

## Codage de Huffman – Exemple

On obtient un alphabet à 2 symboles que l'on code 0 et 1.

On remonte ensuite vers l'alphabet original :

Symbol	ABC	DEF
Mot-code	0	1

Symbol	A	DEF	BC
Mot-code	00	1	01

Symbol	A	BC	DF	E
Mot-code	00	01	10	11

Symbol	A	DF	E	B	C
Mot-code	00	10	11	010	011

Symbol	A	E	B	C	D	F
Mot-code	00	11	010	011	100	101

Rq : Avec le codage de Huffman sur l'exemple précédent, on obtient  $\ell_{av} = 2,6$  et une efficacité de 97,4% !

## 12.4. Codage de canal

## Codage de canal

On suppose ici que la source fournit une séquence de symboles binaires.

- ▶ Le **codage de canal** désigne les traitements effectués sur le signal pour lui permettre d'être transmis sur un canal imparfait.
- ▶ Parmi les traitements, on s'intéresse au codage permettant **la détection et la correction d'erreurs** à la réception. Ces traitements impliquent l'ajout d'information – **redondance** – qui permettra de déterminer la présence d'erreurs de transmission et la correction de ces erreurs.
- ▶ Stratégies utilisées :
  - ▶ détection d'erreur et retransmission (Automatic Repeat Request ARQ)
  - ▶ détection et correction sans retransmission (Forward Error Correcting FEC)

Rq : Dans ce chapitre, on s'intéressera seulement à la stratégie FEC.

## Détection et correction d'erreurs

On distingue deux types de codes de détection et correction d'erreurs :

- ▶ Les **codes en blocs** ( $n, k$ ) : les mots de  $k$  symboles binaires (blocs) à coder sont remplacés par des mots-codes de  $n$  symboles binaires ( $n > k$ ) composés des  $k$  symboles binaires et de  $n - k$  symboles (*redondance*) calculés à partir de l'information du mot à coder. Le codage d'un bloc ne dépend pas des blocs précédents.
- ▶ Les **codes convolutifs** : des mots-codes de  $n$  symboles binaires sont associés aux mots de  $k$  symboles binaires mais les mots-codes dépendent du mot à coder **et** des mots qui le précédent.

Rq : un codeur convolutif est une machine à états finis.

## Distance de Hamming

- ▶ La **distance de Hamming** entre deux mots de  $n$  symboles binaires est le nombre de positions où ces mots ont des symboles différents.
- ▶ Si deux mots sont représentés par des vecteurs  $u$  et  $v$  en dimension  $n$ , alors :

$$d_H(u, v) = |\{i : u_i \neq v_i, 0 \leq i < n\}|$$

- ▶ La **distance minimale de Hamming** d'un code  $C$  est la distance minimale entre toutes les paires de mots-codes de  $C$  :

$$d_{min}(C) = \min_{(u,v) \in C^2, u \neq v} d_H(u, v)$$

Rq : La distance minimale de Hamming du code  $C$  est aussi appelée *distance du code C*.

## Exemples

Un code correcteur d'erreurs très simple est le code **binaire de répétition** de longueur 3.

- ▶ On répète 3 fois chaque bit d'information : 0 est codé 000 et 1 est codé 111.
- ▶ La distance de Hamming entre les mots-codes 000 et 111 est 3.
- ▶ La distance minimale de Hamming est aussi 3 (le code n'a que les deux mots-codes 000 et 111).

## Détection et correction d'erreurs - Les codes en blocs

- ▶ La loi de codage définit la correspondance bijective entre les  $2^k$  mots (de  $k$  symboles binaires) à coder et les  $2^k$  mots-codes (de  $n$  symboles binaires,  $n > k$ ) choisis parmi les  $2^n$  mots de  $n$  symboles possibles.  
On notera  $[n, k, d_{\min}]$  un code en bloc de longueur  $n$  qui code  $k$  bits (de dimension  $k$ ) et dont la distance est  $d_{\min}$ .
- ▶ Les  $2^n - 2^k$  autres mots de  $n$  symboles non retenus comme mots-codes ne sont **jamais** utilisés.
- ▶ Les  $2^k$  mots-codes retenus doivent être aussi différents les uns des autres que possible pour augmenter l'efficacité du code.
- ▶ Le *taux* ou *rendement* du code est  $\frac{k}{n}$ .

## Capacité de détection et de correction d'un code

- ▶ La capacité de détection d'un code  $[n, k, d_{min}]$  est :  $d_{min} - 1$
- ▶ La capacité de correction d'un code  $[n, k, d_{min}]$  est :  $\lfloor \frac{d_{min}-1}{2} \rfloor$
- ▶ Dans le cas code binaire de répétition de longueur 3, on a donc :
  - ▶ Le nombre d'erreurs détectables est :  $d_{min} - 1 = 2$
  - ▶ Le nombre d'erreurs pouvant être corrigées est :  $\lfloor \frac{d_{min}-1}{2} \rfloor = 1$ .

## Exemple de codage en blocs

Un récepteur reçoit le mot  $u = (u_1, \dots, u_n)$ .

Pour détecter la présence d'erreurs :

- ▶ On examine la redondance **calculée par le récepteur** sur les  $k$  premiers symboles,  $r = f(u_1, \dots, u_k)$ .
- ▶ Si  $r$  n'est pas égale à la redondance reçue  $(u_{k+1}, \dots, u_n)$ , le mot  $u$  n'appartient pas au code et on a donc **détecté une erreur**.
- ▶ On appelle *syndrome* :  $s = r - (u_{k+1}, \dots, u_n) \bmod 2$ .  
Si le syndrome est nul, on n'a pas d'erreur de transmission.

## Exemple de codage en blocs

Soit  $x = (x_1, x_2, x_3, x_4)$  un mot à transmettre auquel on ajoute 3 symboles  $y_1, y_2$  et  $y_3$  pour former un mot-code.

La fonction  $f$  suivante permet de calculer les symboles de contrôle (addition modulo 2) :

$$\begin{array}{rcl} y_1 & = & x_1 + x_2 + x_3 + x_4 \\ y_2 & = & x_1 + x_3 + x_4 \\ y_3 & = & x_1 + x_2 + x_4 \end{array}$$

D'où :

- ▶ l'information  $x = (x_1, x_2, x_3, x_4)$ .
- ▶ la redondance  $y = f(x) = (y_1, y_2, y_3)$ .
- ▶ le mot-code  $c = (x_1, x_2, x_3, x_4, y_1, y_2, y_3)$ .

**Rappel** : Si  $a$  et  $b$  sont des bits alors  $a + b \bmod 2 = a - b \bmod 2 = a \oplus b$

## Exemple de codage en blocs

1. Quel est l'ensemble des mots-codes ?
2. Déterminer la longueur, la dimension et la distance du code.
3. Calculer le taux, la capacité de détection et la capacité de correction du code.
4. On reçoit le mot 1000110. Une erreur de transmission s'est-elle produite ?
5. Quelle est la valeur du syndrome ?

# Exemple de codage en blocs

*Solution*

1.  $2^4$  mots-codes :

mot	mot-code	mot	mot-code	mot	mot-code	mot	mot-code
0000	0000 000	0001	0001 111	0010	0010 110	0011	0011 001
0100	0100 101	0101	0101 010	0110	0110 011	0111	0111 100
1000	1000 011	1001	1001 100	1010	1010 101	1011	1011 010
1100	1100 110	1101	1101 001	1110	1110 000	1111	1111 111

2. longueur : 7, dimension : 4, distance : 3
3. taux :  $\frac{4}{7}$ , capacité de détection : 2, capacité de correction : 1
4. une erreur de transmission s'est produite car on devrait avoir 1000 011.
5. syndrome :  $011 - 110 = 101$

## Codes linéaires

Un code  $C[n, k, d_{min}]$  est appelé **code linéaire** s'il existe une matrice  $\mathcal{M}$  de dimension  $k \times n$  permettant d'obtenir les mots-codes par produit matriciel à partir des mots de source  $x$ , i.e. :

$$C = \{y, y = x \cdot \mathcal{M}, x \in \{0, 1\}^k\}$$

- ▶ Le code de Hamming [7,4,3] précédent est un code linéaire.
- ▶ On montrera qu'une matrice de ce code est :

$$\mathcal{M} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

## Ressources

Les ressources suivantes ont été utilisées pour la préparation de ce chapitre :

- ▶ *Introduction à la théorie de l'information* – P. Verlinde, 2006
- ▶ *Codes correcteurs d'erreurs* – M. Chaumont, 2008