# Algorithmique et complexité

Décembre 1998

S. Pinchinat,
V. Schmitt

# Algorithmique et complexité

Sophie Pinchinat - Vincent Schmitt[1]

1. **Document en cours d'élaboration**. Ces notes s'inspirent grandement des notes rédigées par Rumen Andonov, actuellement professeur à l'Université de Valenciennes.

# Table des matières

# Chapitre 1

# Introduction

## 1.1 Généralités

Définissons d'abord de manière informelle les algorithmes. Selon l'Encyclopédie Universelle, un algorithme est "la spécification d'un schéma de calcul, sous forme d'une suite finie d'opérations élémentaires obéissant a un enchaînement déterminé", ou encore "une méthode ou procédé permettant de résoudre un problème a l'aide d'un calculateur — c'est a dire, une séquence finie d'instructions exécutables dans un temps fini et avec une mémoire finie."
Donnons quelques exemples. En algèbre, l'algorithme d'Euclide pour calculer le p.g.c.d. de deux nombres. Dans le domaine du traitement de l'information, de nombreux algorithmes opèrent sur des données non numériques. Pour exemples typiques, citons les algorithmes de tri, de recherche d'une chaîne de caractères dans un texte, d' ordonnancement (agencement de différentes tâches pour mener a bien un projet), de compression de données, de codage/décodage, de "parsing" en compilation, etc... De manière générale, un programme destiné a être exécuté par un ordinateur est très souvent la description d'un algorithme dans un langage accepté par la machine. La notion d'algorithme est donc fondamentale en informatique.

Il existe une notion formelle d'algorithme. Celle-ci est liée a l'idée de machines capables d'exécuter des opérations arithmétiques élémentaires. En 1930 (avant les premiers ordinateurs!) plusieurs mathématiciens (Gödel, Church, Turing...) ont formalisé la notion d'algorithme et de fonction calculable. Ces fonctions sont l'objet d'étude de la théorie de la récursivité. Plusieurs définitions abstraites de fonctions calculables ont été proposées (fonctions récursives, machines de Turing, lambda-calcul). Il est prouvé que tous ces modèles sont équivalents.

La théorie de la récursivité traite de la résolution par algorithme de problèmes. Un problème pouvant être résolu par algorithme est dit *décidable*. Certains problèmes sont indécidables : il est démontré qu'il n'existe pas d'algorithme général pour décider de la satisfiabilité d'une proposition quelconque de l'arithmétique. Par ailleurs, il

existe des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ non calculables. L'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ n'est pas dénombrable, alors que celui des fonctions calculables l'est. En effet, a chaque fonction calculable peut être associé un algorithme qui est un texte de longueur finie sur un alphabet de longueur finie. Un exemple fondamental de problème non décidable est le problème "de l'arrêt" : Il n'existe pas d'algorithme général qui, pour tout programme $P$ et toute donnée $D$, répondrait par oui ou non a la question "$P$ termine-t-il pour $D$?".

La théorie de la calculabilité ne prend pas en compte les ressources nécessaires a un calcul sur machine. Il y a plusieurs sortes de ressources :
- deux sont théoriquement illimitées : le temps et l'espace;
- d'autres sont disponibles en un (petit nombre) : les registres, les piles, les processeurs, ...

La complexité d'un algorithme est l'ensemble des ressources nécessaires a son exécution. L'algorithmique étudie les complexités des algorithmes.

## 1.2 Domaines de l'algorithmique

- (1) Algorithmique théorique : c'est l'étude fondamentale de la complexité.
- (2) Algorithmique pratique : sa finalité est la conception d'algorithmes efficaces.

Attention : les algorithmes usuels sont généralement codés dans des bibliothèques! Il s'agit entre autre de :
- la validation des algorithmes — preuves de correction, vérification...
- l'analyse des algorithmes (estimation de leur complexité). On distingue deux grands types de complexité : "dans le pire des cas" et "en moyenne". Des approches expérimentales sont également mises en œuvre pour tester les algorithmes. Par exemple, la définition de jeux d'essais consiste a choisir des entrées représentatives afin de tester les programmes (debugging) ou évaluer leurs complexités (temps et mémoire nécessaire a leurs exécutions).

## 1.3 But du cours

- (1) Quelques notions théoriques (NP-complétude);
- (2) Techniques de conception d'algorithmes et analyses de ceux-ci :
  - Diviser pour régner;
  - Essais successifs;
  - Séparation et évaluation progressive;
  - Programmation dynamique;
  - Algorithmes gloutons.

Ces techniques seront illustrées par des exemples concrets de problèmes.
  - pb. des tours de Hanoi;
  - les tris;

- pb. du voyageur de commerce;
- pb. du sac a dos;
- pb. dans les graphes.

# Chapitre 2

# Notions de complexité

## 2.1 Calculabilité

La théorie de la récursivité définit les modèles fondamentaux et robustes d'algorithmes, et de problèmes.

### 2.1.1 Les modèles de calcul

*Les fonctions récursives*

Une fonction (partielle) *numérique* est une fonction $\mathbb{N}^k \to \mathbb{N}$. Les fonctions (partielles) *récursives* sont les éléments du plus petit ensemble de fonctions numériques contenant :
- la constante $0 : \mathbb{N} \longrightarrow \mathbb{N}$;
- le successeur $s : \mathbb{N} \longrightarrow \mathbb{N}$;
- les projections $p_i^k : \mathbb{N}^k \longrightarrow \mathbb{N}$, $1 \leq i \leq k$ (où $p_i^k(x_1,...,x_i,...,x_k) = x_i$);

et clos par composition, récursion primitive et minimisation.
- schéma de composition :

$$\frac{h_1,...,h_p : \mathbb{N}^n \longrightarrow \mathbb{N},\ g : \mathbb{N}^p \longrightarrow \mathbb{N}}{(x_1,...,x_n) \mapsto g(h_1(x_1,...,x_n),...,h_p(x_1,...,x_n)) : \mathbb{N}^n \longrightarrow \mathbb{N}}$$

- schéma de récursion :

$$\frac{g : \mathbb{N}^{n-1} \longrightarrow \mathbb{N},\ h : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}}{f : \mathbb{N}^n \longrightarrow \mathbb{N} \,/\, \left\{ \begin{array}{l} f(0,x_2,...,x_n) = g(x_2,...,x_n) \\ f(x_1+1,x_2,...,x_n) = h(f(x_1,x_2,...,x_n),x_1,x_2,...,x_n) \end{array} \right.}$$

- schéma de minimisation :

$$\frac{g : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}}{\qquad\qquad}$$

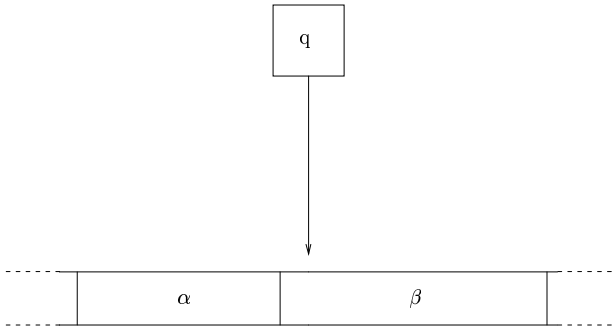
$(x_1,...,x_n) \mapsto min\{y \mid g(y,x_1,...,x_n) = 0\}$

*Les machines de Turing*

Une *machine de Turing* (MT) $(\Gamma,Q,R,q_0)$ consiste en trois ensembles finis $\Gamma,Q,R$ :
- $\Gamma$ : l'alphabet de bande avec $\square \notin \Gamma$;
- $Q$ : l'ensemble des états;
- $R$ : l'ensemble des règles, qui sont des quintuplets $(q,s,q',s',d)$ avec $q,q' \in Q$, $s,s' \in \Gamma \cup \{\square\}$ et $d \in \{-,+\}$;
- $q_0 \in Q$ est l'état initial.

Une telle MT décrit une machine constituée :
- d'une bande infinie, composée de cases pouvant chacune recevoir un symbole de $\Gamma$ ou $\square$,
- d'une tête de lecture/écriture sur la bande, se déplaçant à chaque pas d'au plus une case,
- d'une unité de commande contenant le programme $R$ et à chaque pas se trouvant dans un état de $Q$.

Ainsi une configuration d'une telle MT est un triplet $(\alpha,q,\beta)$ où $\alpha$ et $\beta$ sont des mots sur l'alphabet $\Gamma \cup \{\square\}$, $\alpha$ (respectivement $\beta$) ne commençant pas (respectivement ne terminant pas) par $\square$ — i.e. le contenu de la bande est le mot $\alpha\beta$, l'état de la machine est $q$ et la tête pointe sur le premier caractère de $\beta$.




Une transition $C \xrightarrow{\rho} C'$ pour cette MT est un triplet $(C,\rho,C')$ où :
- $C$ et $C'$ sont des configurations,
- $\rho = (q,s,q',s',d) \in R$,
- dans $C$, l'état est $q$ et la tête pointe sur $s$;
- dans $C'$, l'état est $q'$, la tête a écrit $s'$ à la place de $s$ et s'est déplacée d'une case à gauche si $d = -$ ou d'une case à droite si $d = +$.

Une configuration d'arrêt est une configuration en laquelle aucune règle ne s'applique. Un calcul de longueur $l$ est une suite de transitions du type $(C_i \xrightarrow{\rho_{i+1}} C_{i+1})_{0 \leq i < l}$ soit infinie (auquel cas $l = \omega$), soit se terminant par une configuration d'arrêt. Une MT est *déterministe* (MTD) lorsque pour tout $q \in Q$ et tout $s \in \Gamma \cup \{\square\}$, il y a au plus une règle du type $(q,s,...)$ dans $R$. Soit une $f : \Gamma^* \longrightarrow \Gamma^*$. On dit que la MTD $(\Gamma,Q,R,q_0)$ calcule $f$ lorsque pour tout $\alpha \in \Gamma^*$, $f(\alpha)$ est défini si et seulement

si le calcul issu de la configuration $(q_0,\epsilon,\alpha)$ termine, et si c'est le cas la configuration d'arrêt est du type $(u,q,v)$ avec $uv = f(\alpha)$. Une fonction $f : \Gamma^* \longrightarrow \Gamma^*$ est *Turing-calculable* s'il existe une MTD la calculant. Cette notion de calculabité s'applique aux fonctions sur des mots et peut être transposée aux fonctions numériques en adoptant des *codages* pour les entiers.

*Rappel :* On entend par codage d'un ensemble $S$ dans un alphabet fini $\Gamma$, une injection de $S$ dans $\Gamma^*$, l'alphabet $\Gamma$ contenant au moins deux lettres. Considérons par exemple, le codage des entiers sur l'alphabet $\{0,1\}$, $0 \mapsto 0$, $1 \mapsto 1$, $2 \mapsto 11$, $3 \mapsto 100$, ... On peut de même coder les caractères (codes ASCII), les couples, les fonctions, les graphes, les programmes,...

Une fonction numérique $f : \mathbb{N}^p \longrightarrow \mathbb{N}$ est dite *Turing-calculable* si et seulement si pour des codages "standards", $\gamma_1 : \mathbb{N}^p \longrightarrow \Gamma^*$ et $\gamma_2 : \mathbb{N} \longrightarrow \Gamma^*$, il existe une fonction Turing-calculable $\tilde{f} : \Gamma^* \longrightarrow \Gamma^*$, telle que $\tilde{f} \cdot \gamma_1 = \gamma_2 \cdot f$.

$$\begin{array}{ccc} \mathbb{N}^p & \xrightarrow{f} & \mathbb{N} \\ {\scriptstyle \gamma_1}\downarrow & & \downarrow{\scriptstyle \gamma_2} \\ \Gamma^* & \xrightarrow[\tilde{f}]{} & \Gamma^* \end{array}$$

Turing a montré que toutes les fonctions récursives sont Turing-calculables. Inversement les fonctions Turing-calculables sont récursives. En outre, c'est un fait d'expérience toutes les fonctions que l'on sait calculer sont récursives.

**Thèse de Church**
• Toute fonction "calculable" est récursive.
• Si une fonction $f$ est calculable par un "programme", alors celui-ci est "effectivement" transformable en une machine de Turing calculant $f$.

### 2.1.2 Algorithmes, problèmes

La thèse de Church justifie des descriptions informelles de haut niveau d'algorithme, de problème et de décidabilité. En pratique, ces abstractions sont indispensables. Ainsi, on se contente des définitions suivantes.
- Un *algorithme* est un traitement sur des données finies (éventuellement aucune); il spécifie un enchaînement d'un nombre fini d'opérations, lesquelles sont effectives — c'est à dire réalisables par une machine (par exemple les opérations mathématiques ont des arguments finis); quelque soit la donnée d'entrée, un algorithme doit toujours terminer après un nombre fini d'opérations et fournir un résultat.
- Un *problème (de décision)* est un prédicat sur un ensemble d'instances, un *problème de recherche* est une relation entre un ensemble d'instances et un ensemble de solutions, les instances et les solutions sont des données finies.

*Exemples*

- Le problème de décision $CLIQUE$ :
*Instance : Un graphe $G = (V,E)$ et un entier positif $k \leq |V|$.*
*Question : $G$ contient-il une clique de taille $k$ ?*
- Le problème de recherche de "la plus grande clique" $PG - CLIQUE$ :
*Instance : Un graphe.*
*Requête : trouver une plus grande clique de ce graphe.*

- Un problème de décision $P$ est *décidable* s'il existe un algorithme le *résolvant* : c'est à dire qui pour tout instance $x$ en entrée répond 1 si $x \in P$ et 0 sinon. Un algorithme résout un problème de recherche $P$, s'il calcule la fonction qui à toute instance $x$ associe l'ensemble des solutions lui correspondant via $P$.

En se référant à la thèse de Church les notions d'algorithme, de problème, et de décidabilité peuvent être définies à partir des langages. Un algorithme s'apparente à une MTD dont tous les calculs terminent. Un problème $P$ est associé par codage au langage $L_P$ constitués des codes des instances satisfaisant $P$. Introduisons un peu de terminologie. Un algorithme opérant sur un langage *accepte* (respectivement *rejette*) un mot $x \in \Gamma^*$ si pour l'entrée $x$, l'algorithme répond 1 (respectivement 0). On note $L_A$ le langage accepté par l'algorithme $A$. Un langage $L$ est *décidé* par un algorithme $A$ si $L = L_A$. Une *réduction* d'un langage $L_1 \subseteq \Gamma_1^*$ en un langage $L_2 \subseteq \Gamma_2^*$ est une application $f : \Gamma_1^* \longrightarrow \Gamma_2^*$ telle que $x \in L_1 \Leftrightarrow f(x) \in L_2$ i.e., en notant $\chi_L$ la fonction caractéristique de $L$, $f$ fait commuter le diagramme ci-dessous :

$$\begin{array}{ccc} \Gamma_1^* & \xrightarrow{\chi_{L_1}} & \{0,1\} \\ {\scriptstyle f}\downarrow & & \| \\ \Gamma_2^* & \xrightarrow[\chi_{L_2}]{} & \{0,1\} \end{array}$$

D'après la thèse de Church un problème est décidable s'il peut être associé par codage à un langage décidable.

## 2.2 Complexité

L'analyse de la complexité des algorithmes a pour but d'établir des résultats généraux permettant d'estimer l'efficacité intrinsèque des algorithmes, indépendamment de la machine, du langage de programmation, du compilateur, ou de tout autre détail d'implantation. On souhaite établir des énoncés du type : "Sur toute machine, et quelque soit le langage de programmation, l'algorithme 1 est meilleur que l'algorithme 2 pour des données de grande taille", ou encore "L'algorithme A est optimal en nombre d'opérations d'un type donné, pour résoudre le problème P".
Tout est à préciser, bien entendu ("meilleur", "taille d'une donnée", "optimalité"...).

### 2.2.1 Fondements

Il convient de disposer d'une mesure robuste et fondamentale de complexité, c'est à dire définie uniformément pour les fonctions récursives. Parmi les modèles de calcul celui des machines de Turing déterministes (MTD) est le mieux adapté.

**Définition 2.2.1** *Par la suite, $T$ désigne une fonction de $\mathbb{N} \to \mathbb{R}^+$.*
*(i) Soit $M$ une MTD dont tous les calculs terminent. On dit que "$M$ est de complexité temporelle $T$" (ou "inférieure à $T$" ou encore "en $O(T)$") lorsque l'ordre de grandeur de $T$ majore celui de la fonction $C_M : \mathbb{N} \longrightarrow \mathbb{N}$ où $C_M(n)$ est la longueur maximum d'un calcul de $M$ issu d'un mot de longueur au plus $n$.*
*(ii) Soit $f : \Gamma^* \to \Gamma^*$ une fonction récursive totale, on dit que "$f$ est de complexité $T$" s'il existe une MTD de complexité $T$ calculant $f$.*
*(iii) Soit $L$ un langage sur un alphabet fini, la complexité du problème $x \in L$? est en $T$ s'il existe un algorithme de complexité $T$ décidant de $L$.*

### 2.2.2 En pratique

Dans la pratique, l'évaluation de la complexité temporelle d'un algorithme $A$ résolvant un problème $P$ est définie comme suit.
(1) L'ensemble $D$ des instances de $P$ est muni d'une fonction de taille $|\text{-}| : D \longrightarrow \mathbb{N}$, $D_i \subseteq D$ désignant l'ensemble des instances de longueur majorée par $i$;
*Exemples de fonctions de taille*
- tri : nombres d'éléments manipulés,
- multiplications d'entiers : nombres de chiffres des nombres,
- produit des matrices : les dimensions des matrices,
- parcours d'arbres ou de graphes : les nombres de nœuds et/ou d'arêtes.

(2) La complexité correspond à l'ordre de grandeur de la fonction $C_A$ où $C_A(n)$ est le "temps d'exécution maximal" de l'algorithme pour les données $x \in D_n$. Ce temps est calculé en associant à chaque instruction de l'algorithme un temps d'exécution et en simulant son exécution.
*Exemples.*
- Le calcul du maximum de $n$ entiers nécessite $n-1$ comparaisons, sa complexité est donc "en $O(n)$" ou "de l'ordre de $n$" - voir 2.2.5.
- La multiplication de deux matrices : on évalue le coût d'un calcul par le nombre de multiplications et d'additions à effectuer.

Pour justifier d'une telle pratique dans le cadre des MTD, il faut coder les instances du problème dans un alphabet et transcrire l'algorithme en une MTD. Ceci est très éloigné de la programmation usuelle. En pratique on recourt à des modèles de machines de plus haut niveau. Par exemple les machines (RAM) sont à accès direct, les données numériques ne sont généralement pas codées. Les RAM simulent de façon réaliste le fonctionnement d'un ordinateur (on adopte l'hypothèse du coût logarithmique, le temps d'exécution de "INST n" est proportionnel au $log$ de $n$).

Les résultats des calculs de complexité dans chacun des modèles MTD et RAM sont différents mais généralement reliés.

### 2.2.3 Remarques

- Seule la notion de complexité temporelle a jusqu'ici été abordée. La complexité spatiale est définie de façon analogue pour les MTD, les fonctions et les problèmes..., en prenant en compte le nombre de cases utilisées par les calculs.
- En pratique, on s'intéresse également à la complexité "en moyenne". Pour un algorithme $A$, celle-ci est définie à partir de la fonction qui envoie tout entier $n$ sur

$$\Sigma_{d \in D_n} p(d).l(d)$$

où $D_n$ est l'ensemble des instances de taille inférieure à $n$, $l(d)$ désigne la longueur du calcul issu de $d$ et $p(d)$ est la probabilité que $d$ soit en entrée d'algorithme ($\Sigma_{d \in D_n} p(d) = 1$). La complexité définie en 2.2.2 est en pratique également appelée complexité "dans le pire des cas".
- Les complexités dans le pire des cas ne peuvent être confondues : comme nous le verrons dans ce cours, ce n'est pas parce qu'un algorithme est meilleur qu'un autre dans le pire des cas qu'il l'est en moyenne.
- La complexité en moyenne est généralement beaucoup plus difficile à estimer que la complexité dans le pire des cas. La notion de complexité en moyenne dépend d'une fonction de distribution sur les instances de problèmes. La probabilité d'occurrence des instances intervient dans le calcul de l'estimation des temps moyens d'exécution. De fait, ces calculs s'avèrent souvent très compliqués.

### 2.2.4 Comportements asymptotiques

La complexité est définie à partir du coût des calculs qui est fonction de la taille $n$ des données. Il est important de connaître le comportement de cette fonction lorsque $n$ tend vers l'infini. C'est ce comportement asymptotique qui indique les limites d'un algorithme à traiter des problèmes de grandes tailles. Une question systématique à se poser est : "Que devient le temps de calcul d'un algorithme lorsque la taille des données est multipliée par 2?".

### 2.2.5 Rappels sur les ordres de grandeur

On note $\mathbb{R}^+$, l'ensemble des réels positifs ou nuls. Soit $f : \mathbb{N} \to \mathbb{R}^+$. On définit un préordre $\sqsubseteq$ sur les fonctions de $\mathbb{N}$ vers $\mathbb{R}^+$ par :
$f \sqsubseteq g \Leftrightarrow (\exists p \in \mathbb{N}, c \in \mathbb{R}^+, n \geq p \Rightarrow f(n) \leq c.g(n))$ — i.e. au voisinage de l'infini $f$ est dominée par $c.g$. L'*ordre (de grandeur)* $O(f)$ d'une fonction $f : \mathbb{N} \to \mathbb{R}^+$ est sa classe pour la relation d'équivalence $\sim$ définie par $x \sim y \Leftrightarrow (x \sqsubseteq y \wedge y \sqsubseteq x)$. Les classes $O(-)$ sont ordonnées par l'ordre quotient de $\sqsubseteq$ — i.e. $O(f) \leq O(g)$ si et seulement si $f \sqsubseteq g$ (alors $O(f) \leq O(g)$ si et seulement si $\exists f' \in O(f), \exists g' \in O(g), f' \sqsubseteq g'$

si et seulement si $\forall f' \in O(f), \forall g' \in O(g), f' \sqsubseteq g')$.

*Exemples.*
(1) $O(2n) < O(n^2)$ — $2n \sqsubseteq n^2$, voir pour $c = 1$ et $p = 2$, et $n^2 \not\sqsubseteq 2n$;
(2) $O(2^{n+1}) = O(2^n)$ — $c = 2$ et $p = 1$;
(3) $O((n+1)!) > O(n!)$.

**Propriétés 2.2.2**
- $O(f(n) + g(n)) = O(max(f(n), g(n)))$;
- $\forall c \geq 0$, $O(c \cdot f(n)) \in O(f(n))$;
- $lim_{n \to +\infty} f(n)/g(n) \in \mathbb{R}^{+*} \Rightarrow O(f(n)) = O(g(n))$.
- $lim_{n \to +\infty} f(n)/g(n) = +\infty \Rightarrow O(f(n)) > O(g(n))$.
- $lim_{n \to +\infty} f(n)/g(n) = 0 \Rightarrow O(f(n)) < O(g(n))$.

Quelques classes pour $O$ :
$O(1) < O(log(n)) < O(\sqrt{n}) < O(n) < O(n \cdot log(n)) < O(n^2) < O(n^a)$ (où $a > 2$)$< O(c^n)$ (où $c \in \mathbb{R}^+$)$< O(n^n)$.

*Exemple :* Calculer la somme $f(n)$ des $n$ premiers entiers.

algorithme A1 : $s := 0$;
pour $i := 1$ jusqu'à $n$ faire $s := s + i$.

$f(n)$ s'obtient en faisant $n$ additions donc $C_{A1}(n) \in O(n)$.

algorithme A2 : $s := n \times (n+1)/2$.

$f(n)$ s'obtient en trois opérations donc $C_{A2}(n) \in 0(1)$.

### 2.2.6 Terminologie

Les classes usuelles de complexités d'algorithmes sont les suivantes.

**Définition 2.2.3** *La complexité d'un algorithme est :*
- constante *ssi elle est en* $O(1)$;
- linéaire *ssi elle est en* $O(n)$;
- polynomiale *ssi elle est en* $O(n^k)$ *pour un* $k > 0$;
- superpolynomiale *ssi elle n'est en* $O(n^k)$ *pour aucun entier* $k$;
- exponentielle *ssi elle est en* $O(a^n)$ *pour un* $a > 1$.

Le fait important est

**Fait 2.2.4** *Tous les codages "raisonnables" connus sont "équivalents", ce au sens suivant: si $\gamma_1$ et $\gamma_2$ sont deux codages d'un même problème alors il existe une* traduction *de $\gamma_1$ vers $\gamma_2$ — c'est à dire une fonction $f$ telle que $f \circ \gamma_1 = \gamma_2$ — et celle-ci est* récursive de complexité polynomiale.

## 2.3 La théorie de la NP-complétude

### 2.3.1 Introduction

Rappelons qu'un algorithme est *en temps polynomial* s'il est en $O(n^k)$. Les problèmes que l'on peut résoudre avec un ordinateur en temps polynomial sont dits *traitables*. Ce sont ceux de la classe **P**. Les autres problèmes sont dits *intraitables*.

Ce chapitre présente une classe intéressante de problèmes : celle des problèmes NP-complets. Leur statut est mal connu. Aucun algorithme en temps polynomial n'est connu pour un problème NP-complet, et aucune borne inférieure superpolynomiale n'est connu pour sa complexité.

Par la suite sont définies :
- la classe **P** des problèmes de décision résolubles en temps polynomial;
- la classe **NP** des problèmes de décision dont les solutions peuvent être validées en temps polynomial.

Les relations entre problèmes via les réductions sont présentées. Enfin, la NP-complétude est définie, et quelques exemples très connus de problèmes NP-complets sont donnés.

### 2.3.2 La classe de complexité P (Cobham 1964)

D'après l'équivalence des codages mentionnées précédemment (Fait 2.2.4), on peut assimiler un problème à n'importe lequel des langages qui lui est associé par codage et formuler ainsi la définition de la la classe **P** (Cobham, 1964) :

> *Un problème de décision appartient à la classe* **P** *s'il existe un algorithme le résolvant en temps polynomial.*

C'est à dire qu'un problème de décision $P$ appartient à la classe **P**, s'il en existe un codage de $P$ et un algorithme décidant du langage associé en temps polynomial. La classe **P** se définit plus aisément sur les langages par :

**P** *est la classe des langages décidés par un algorithme en temps polynomial.*

### 2.3.3 La classe de complexité NP (Edmonds 1965)

Un algorithme de validation est un algorithme $A$ à deux arguments $x$ et $y$ (qui sont des mots), $y$ est appelé *certificat*. Un tel algorithme *valide* le mot $x$ s'il existe

un certificat $y$ tel que $A(x,y) = 1$. Le langage validé par un algorithme de validation opérant sur les mots de $\Gamma^*$ est $\{x \in \Gamma^* \mid \exists y \in \Gamma^*, A(x,y) = 1\}$

On définit la classe **NP** de langages :

*Un langage appartient à* **NP** *s'il est validé en temps polynomial.*

Ceci signifie que $L \in \mathbf{NP}$ s'il existe un algorithme polynomial à deux entrées validant $L$ et une constante $c > 0$ tels que pour tout $x$ de $L$, il existe un certificat de longueur $l(x) \in O(|x|^c)$. Un problème de décision $P$ appartient à la classe **NP** lorsque pour un quelconque de ces codages, le langage associé $L_P$ est tel que $L_P \in \mathbf{NP}$.

**Remarque 2.3.1** *Si $L \subseteq \Gamma^*$ est dans* **NP**, *la complexité du problème $x \in L$? est en $O(n^k.|\Gamma|^{n^c})$, pour des entiers positifs $k$ et $c$, où $n = |x|$. En effet la complexité de la recherche exhaustive pour les $x$ des certificats $y$ est en $O(|\Gamma|^{n^c})$ et la complexité de la validation des $(x,y)$ est en $O(n^k)$ .*

*Exemple :* CIRC-HAM, le problème du circuit Hamiltonien.
*étant donné un graphe non orienté, simple et sans boucles, existe-t-il dans celui-ci un circuit Hamiltonien ?*

Étant donné une séquence $C$ de sommets correspondant à un circuit Hamiltonien d'un graphe $G$, il existe un algorithme vérifiant en temps polynomial en la taille de $G$ que $C$ décrit un cycle Hamiltonien.

Évidemment, $\mathbf{P} \subseteq \mathbf{NP}$.

### 2.3.4 L'état de l'art

On ne sait pas si $\mathbf{P} = \mathbf{NP}$ mais on soupçonne que non. On ne sait pas si $\mathbf{NP} = \mathbf{co-NP}$ c'est à dire si **NP** est fermé par complémentaire : $L \in \mathbf{NP} \Rightarrow \bar{L} \in \mathbf{NP}$? On sait juste que $\mathbf{P} = \mathbf{co-P}$ (évident). Donc $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{co-NP}$. Par contre on ne sait pas si $\mathbf{P} = \mathbf{NP} \cap \mathbf{co-NP}$. En résumé, notre compréhension de la relation entre **P** et **NP** est lamentablement incomplète. Néanmoins en explorant la théorie de la NP-complétude, nous allons voir que notre incapacité à montrer qu'un problème est théoriquement intraitable est surmontée en pratique.

### 2.3.5 NP-complétude et réductibilité

Les problèmes NP-complets constituent une sous-classe de **NP**. Cette classe possède la propriété suivante : si un seul des problèmes NP-complets peut être résolu en temps polynomial, alors tous les problèmes **NP** peuvent être également résolus en temps polynomial, c'est à dire alors $\mathbf{P} = \mathbf{NP}$. *Le fait est qu'aucun algorithme polynomial résolvant un problème NP-complet n'est connu aujourd'hui.*

**Exemple**. Le problème CIRC-HAM est NP-complet.

Nous allons voir maintenant que les problèmes NP-complets sont d'une certaine façon les plus "difficiles" de la classe **NP**.

**Définition 2.3.2** *Un langage $L_1$ est* réductible (en temps polynomial) *en un langage $L_2$ ce que l'on note $L_1 \leq L_2$, s'il existe une réduction en temps polynomiale de $L_1$ vers $L_2$.*

Remarquons que la relation de réduction est un préordre sur **NP**.

**Fait 2.3.3** *Si $L_1 \leq L_2$ alors $L_2 \in \mathbf{P} \Rightarrow L_1 \in \mathbf{P}$.*

**Définition 2.3.4 (NP-complétude, Cook 1971)** *Un langage $L \subseteq \{0,1\}^*$ est* **NP**$-$*complet si :*
*- $L \in NP$;*
*- $L$ est* **NP***-difficile : pour tout langage $L' \in \mathbf{NP}$, $L' \leq L$.*

**NPC** désigne la classe des langages NP-complets.

On soupçonne $\mathbf{P} \cap \mathbf{NPC} = \emptyset$.

**Le problème** *SAT*
Soit $X$ un ensemble de variables booléennes. Une valuation $\delta$ de $X$ est une application de $X$ vers $\{0,1\}$. Un littéral est un terme de la forme $x$ ou $\neg x$ pour $x \in X$, une clause sur $X$ est ensemble de littéraux. Toute valuation $\delta$ sur $X$ peut être étendue aux clauses sur $X$ en posant :
$\delta(\neg x) = \neg\delta(x)$, pour tout $x \in X$, et
$\delta(c) = \delta(l_1) \vee \delta(l_2) \vee ... \vee \delta(l_n)$ pour toute clause $c$ composée des littéraux $l_1$, ...,$l_n$.
Enfin, un ensemble $C$ de clauses est *satisfait* par une valuation $\delta$, lorsque pour toute $c \in C$, $\delta(c) = 1$. $C$ est dit satisfiable lorsqu'il existe une telle valuation. Le problème *SAT* est le suivant :

*étant donné un ensemble de clauses, est-il satisfiable?*

Cook a montré par des preuves directes :
- *SAT* $\in$ **NP**;
- *SAT* est **NP**-difficile.

Le premier point est facile à prouver. Le second point est prouvé en montrant que pour une MTD $(\Gamma,Q,R,q_0)$ validant $L$ en temps polynomial, pour tout $x \in \Gamma^*$, peut être calculé en temps polynomial un ensemble de clauses $\varphi_x$ telle que : $x \in L \Leftrightarrow \varphi_x$ *est satisfiable.*

*SAT* étant NP-complet, la relation de réduction a donc des éléments maximaux. Une fois que l'on connaît un problème NP-complet, on peut montrer qu'il y en a

d'autres. Comment faire une preuve de NP-complétude d'un problème $P$ :
- (1) : Prouver $P \in \mathbf{NP}$ en trouvant un algorithme;
- (2) : Choisir $P' \in \mathbf{NPC}$ connu et trouver une réduction polynomiale de $P'$ dans $P$.

### 2.3.6 Exemples très connus de problèmes NP-complets

On sait déjà $CIRC - HAM$,$SAT \in \mathbf{NPC}$. Mentionnons :

$3 - SAT$
*Instance : un ensemble $E$ de clauses, chacune comportant au plus trois littéraux.*
*Question : $E$ est-il satisfiable?*

$CLIQUE$ (vu en 2.1.2)
*Instance : Un graphe $G = (V,E)$ et un entier positif $k \leq |V|$.*
*Question : $G$ contient-il une clique de taille $k$?*

$PARTITION$
*Instance : Un ensemble fini et une fonction de taille $s : A \longrightarrow I\!N$.*
*Question : Existe-t-il $A' \subseteq A$ tel que : $\Sigma_{a \in A'} s(a) = \Sigma_{a \in A \setminus A'} s(a)$?*

### 2.3.7 Problèmes de recherche

La théorie de la NP-complétude ne prend en compte que les problèmes de décision. Les résultats de cette théorie peuvent souvent être étendus aux problèmes de recherche en leur associant des problèmes de décision "plus facile".

*Exemple :* Considérons PG-CLIQUE, le problème de "la plus grande clique" introduit en 2.1.2. PG-CLIQUE est associé au problème de décision CLIQUE. CLIQUE qui est NP-complet se réduit trivialement (donc polynomialement!) à PG-CLIQUE. PG-CLIQUE est donc au moins aussi difficile que CLIQUE, il est dit NP-dur...

**Références**

• M.R Carey, D.S Johnson, *Computers and Intractability, A guide to the theory of NP-completness* - W.H. Freeman and Company-79.
• R. Lalement, *Logique, Réduction, Résolution* - Masson-90.
• T.Cormen, C.Leiserson, R.Rivest, *Introduction à l'algorithmique* - Dunod 94.

# Chapitre 3

# Diviser pour régner

## 3.1 Principe

La technique "diviser pour régner" s'applique à des problèmes pour lesquels :
- pour $n$ assez grand, la solution pour la donnée $x$ de taille $n$ s'exprime en fonction des solutions pour des données $x'_1$,...,$x'_k$ de taille strictement inférieure à n;
- on sait résoudre directement ou par une autre méthode le problème pour des données de tailles inférieures à un seuil.

La technique de résolution DR appliquée à un tel problème $P$ pour la donnée $x$ de taille $n$, consiste à :
- (1) réduire la donnée $x$ en les données $x'_1$,...,$x'_k$;
- (2) résoudre $P$ pour les données $x'_1$,...,$x'_k$;
puis
- (3) composer ces solutions pour calculer la solution de $P$ pour $x$.

## 3.2 Avantages de la méthode

La méthode DR donne lieu à des algorithmes présentant des propriétés intéressantes :
- Il est facile de prouver leurs corrections par induction;
- Leurs complexités sont souvent intéressantes et surtout celles-ci peuvent être évaluées par une résolution d'équations récurrentes;
- Leur programmation est aisée grâce au mécanisme de récursivité.

## 3.3 Exemples

- 1. La recherche dichotomique est un algorithme pouvant être programmé sans appels récursifs, ni piles. Il rentre néanmoins dans la catégorie DR.

- 2. L'algorithme de tri par fusion.

- 3. Les tours de Hanoi.
On dispose de 3 bâtons $g$, $m$, $d$ et de $n$ disques de taille différentes. Dans la configuration initiale, les disques sont tous empilés dans l'ordre de leur taille croissante sur le bâton $g$. Le but est de parvenir à la configuration où tous les disques sont empilés dans l'ordre de leur taille décroissante sur le bâton $d$, en respectant les règles de déplacement des disques suivantes :
- On déplace un seul disque à la fois;
- Un disque ne peut pas être empilé sur un disque taille inférieure.

Analyse du problème : On numérote les disques de 1 à $n$ par ordre de taille croissante.
• $n = 1$ : une solution est de déplacer le seul disque de $g$ à $d$.
• $n > 1$ : Si on sait résoudre le problème pour $n - 1$, une solution consiste en les opérations suivantes :
(1) on déplace les $n - 1$ disques 1,...,$n - 1$ de $g$ à $m$,
(2) on déplace le disque $n$ de $g$ à $d$,
(3) on déplace les $n - 1$ disques 1,...,$n - 1$ de $g$ à $d$.

On sait donc résoudre le problème pour $n = 1$, et on sait exprimer la solution du problème pour $n$ en fonction des solutions pour $n - 1$ et 1.
L'algorithme en pseudo-code est le suivant :

```
Hanoi(n,g,m,d)
    debut
        si n = 1 alors deplacer(g,d)
        sinon
                    Hanoi(n-1,g,d,m)
                    deplacer(g,d)
                    Hanoi(n-1,m,g,d)
        fsi
    fin
```

évaluons maintenant la complexité d'un tel algorithme. On choisit comme unité de mesure le déplacement d'un disque. Soit $C(n)$ le temps d'exécution de l'algorithme pour un nombre de disques égal à $n$. Alors

$$C(1) = 1$$

et pour tout $n > 1$,

$$\begin{aligned} C(n) &= 2 \times C(n-1) + 1 \\ &= (2^{n-1} + ... + 2 + 1) \\ &= 2^n - 1. \end{aligned}$$

## 3.4 Calculs des complexités

Soit un algorithme $A$ résolvant le problème $P$ et conçu selon la méthode DR. On note $T(n)$ le temps maximal d'éxécution de $A$ pour des données de taille $n$. On suppose que la solution de $P$ pour la donnée $x$ de taille $n$ s'exprime en fonction de $a$ solutions de $P$ pour des données $x'$ de taille $n' < n$. On suppose que $A$ est tel que le temps maximal des calculs :
- des $x'$ en fonction de $x$ et,
- de la solution pour $x$ en fonction des solutions pour les $x'$,

est polynomiale en la taille de $x$, de la forme $c.n^k$, pour $c \in \mathbb{R}^+$ et $k \in \mathbb{N}$. On a

$$\begin{cases} T(n) = a.T(n') + c.n^k \ \ si \ \ n > 1 \\ T(1) = d \end{cases}$$

où $d$ est une constante positive.

Les réductions des données $x$ en les ${x'}_1$, ..., ${x'}_a$ sont dites :
- *simples* lorsque $n' = n - 1$, (Cf. les "tours de Hanoi");
- *logarithmiques* lorsque $n' \leq n/b$ avec $b > 1$.

Nous allons étudier la complexité de l'algorithme $A$ pour ces deux cas de réduction.

### 3.4.1 Réductions simples

$$\begin{cases} T(n) = a.T(n-1) + c.n^k \ \ pour \ \ n > 1 \\ T(1) = d \end{cases}$$

Pour $n > 1$,

$$\begin{aligned} T(n) &= a.T(n-1) + c.n^k \\ a.T(n-1) &= a^2.T(n-2) + a.c.(n-1)^k \\ &... \\ a^{n-2}.T(2) &= a^{n-1}.T(1) + a^{n-2}.c.2^k \\ a^{n-1}.T(1) &= a^{n-1}.d \end{aligned}$$

donc

$$T(n) = d.a^{n-1} + c.\Sigma_{i=0,...,n-2} a^i.(n-i)^k$$

Mentionnons les cas particuliers suivants :
- $k = 0$, $a = 1$ : $T(n) = d + c.(n-1) \in O(n)$;
- $k = 0$, $a > 1$ : $T(n) = d.a^{n-1} + c.(a^{n-1} - 1)/(a-1) \in O(a^n)$;
- $k = 1$, $a = 1$ : $T(n) = d + c.(n-1)(n+2)/2 \in O(n^2)$.

### 3.4.2 Réductions logarithmiques

$$\begin{cases} T(n) = a.T(n') + c.n^k \ \ avec \ \ n' \leq n/b \ \ pour \ \ n > 1 \\ T(1) = d \end{cases}$$

On peut définir une suite d'entiers $(n_{(i)})_{1\leq i\leq m}$ telle que :
- $n_{(0)} = n$,
- $n_{(i)} < n_{(i-1)}/b$, pour $1 \leq i \leq m$,
- $n_{(m)} = 1$, et

$$\begin{array}{lll} T(n) & \leq a.T(n_{(1)}) + c.n^k & (I_1) \\ T(n_{(1)}) & \leq a.T(n_{(2)}) + c{n_{(1)}}^k & (I_2) \\ ... & & \\ T(n_{(m-1)}) & \leq a.T(n_{(m)}) + c.{n_{(m-1)}}^k & (I_m). \end{array}$$

En combinant ces équations $(\Sigma_{1\leq i\leq m} a^{i-1} I_i)$, on obtient :

$$T(n) \leq a^m.d + c.(n^k + a.{n_{(1)}}^k + ... + a^{m-1}.{n_{(m-1)}}^k)$$

Si $l$ est la partie entière de $log_b(n)$ augmentée d'1 alors $m \leq l$, $n \leq b^l$ et $n_{(i)} \leq b^{(l-i)}$ pour tout $i \in \{0,...,m\}$. Alors,

$$\begin{array}{rl} T(n) & \leq a^l.d + c.(b^{lk} + a.b^{(l-1)k} + ... + a^{l-1}.b^k) \\ & = a^l.d + c.a^l(({b^k}/{a})^l + ({b^k}/{a})^{l-1} + ... + ({b^k}/{a})) \end{array}$$

En posant $q = b^k/a$,

$$T(n) \in O(a^l.\Sigma_{i=0,...,l} q^i).$$

Puisque $a^l \leq a.a^{log_b(n)} = a.n^{log_b a} \in O(n^{log_b a})$
- si $q < 1$ : $T(n) \in O(a^l.(1-q^{l+1})/(1-q)) = O(a^l) = O(n^{log_b a})$;
- si $q = 1$ : $T(n) \in O(l.a^l) = O(log_b n.n^{log_b a}) = O(log(n).n^k)$;
- si $q > 1$ : $T(n) \in O(a^l.q^l) = O(n^k)$.

En résumé :

$$\begin{array}{l} \text{Si } a > b^k,\ T(n) \in O(n^{log_b a}), \\ \text{si } a = b^k,\ T(n) \in O(n^k.log(n)), \\ \text{si } a < b^k,\ T(n) \in O(n^k). \end{array}$$

**Références**

• T.Cormen, C.Leiserson, R.Rivest, *Introduction à l'algorithmique* - Dunod 94.

# Chapitre 4

# Les tris

## 4.1 Le problème du tri

Les méthodes de tri sont fondamentales pour l'informatique pratique. Le tri est également un bon exemple de problème pour lequel de nombreux algorithmes existent. L'analyse fine des performances respectives de ces algorithmes est donc intéressante. A noter qu'il existe des algorithmes d'efficaté optimale.

**Spécification du tri.** La donnée est une liste de $n$ éléments. A chaque élément est associé une clé qui appartient à un ensemble totalement ordonné. Le résultat est une liste avec les mêmes éléments mais dont les clés vont en ordre croissant lors d'un parcours séquentiel de la liste. Plus simplement : étant donné une suite finie d'objets $(a_i)_{1\leq i\leq n}$, trier cette suite consiste à déterminer une permutation $\sigma$ sur $\{1,...,n\}$ telle que la suite $(a_{\sigma(i)})_{1\leq i\leq n}$ soit croissante.

Dans la suite du cours on supposera que les algorithmes s'exécutent sur machine RAM. Les listes à trier étant rangées dans des tableaux (à accès directs). Pour simplifier, ces listes seront constituées d'entiers distincts.

## 4.2 Une borne inférieure pour la complexité

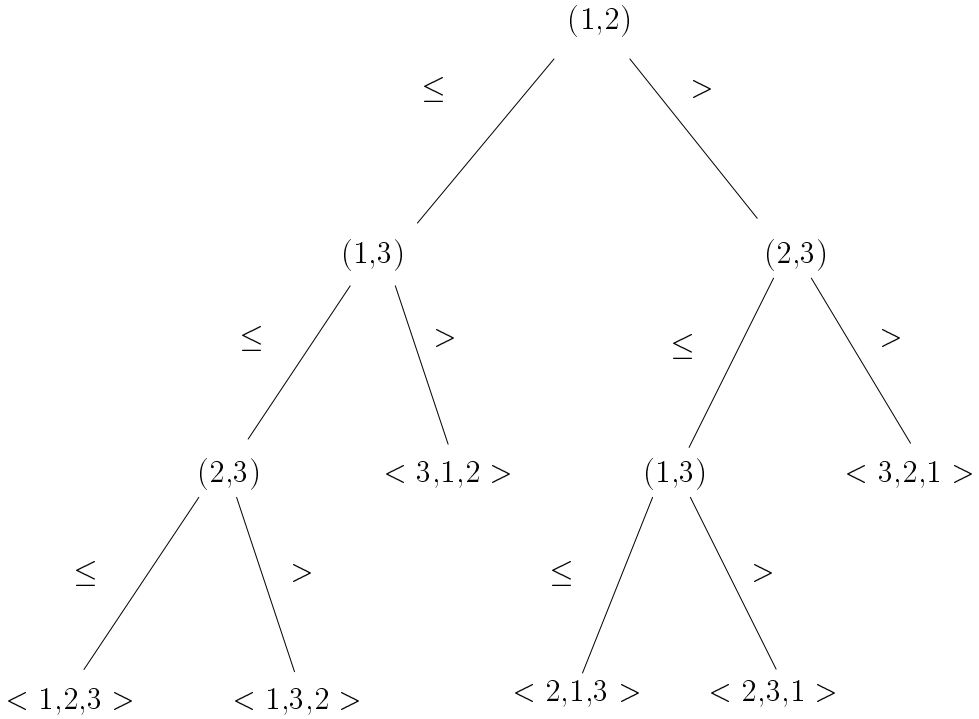
Un arbre de décision d'un tri représente les diverses exécutions du tri pour différentes données en entrée.

Dans un arbre de décision d'un tri pour des données de taille $n$ :
- les noeuds internes ont exactement deux fils et sont étiquetés par des couples $(i,j)$ où $1 \leq i < j \leq n$;
- chaque feuille de l'arbre est étiquetée par une permutation $\sigma$.

Une exécution possible de l'algorithme correspond exactement à un chemin de la racine à une feuille :

- La succession des noeuds internes décrit la succession des comparaisons d'éléments effectués : le couple $(i,j)$ correspond à la comparaison des éléments $a_i$ et $a_j$ de la liste;
- La permutation en feuille correspond au tri de la liste en fin d'exécution.

**Exemple** L'arbre de décision du tri par sélection ordinaire pour une liste de 3 éléments est représenté ci-après.




**Remarques 4.2.1** - *Dans un arbre de décision, la longueur d'un chemin est le nombre de comparaisons effectuées lors de l'exécution lui correspondant.*
- *Un arbre de décision pour des données de taille $n$ doit contenir autant de feuilles que de permutations possibles de $n$ éléments soit $n!$.*

**Theorem 4.2.2** *Tout arbre de décision d'un tri pour $n$ éléments a une hauteur d'ordre supérieur à $O(n.log_2 n)$*

*preuve :* Un arbre binaire de hauteur $h$ ne comporte pas plus de $2^h$ feuilles. De ce fait si $h$ est la hauteur d'un arbre de décision d'un tri pour $n$ éléments,

$$n! \leq 2^h$$

donc

$$h \geq log_2(n!).$$

Le résultat s'ensuit alors du fait connu $ln(n!) \in O(n.ln(n))$ (montré ci-après). ∎

**Fait 4.2.3** $ln(n!) \in O(n.ln(n))$.

*preuve :* Pour $n \leq 1$,

$$ln(n!) = \Sigma_{i=1}^{n} ln(n)$$

et

$$\int_1^{n-1} ln(x)dx \leq \Sigma_{i=1}^{n} ln(n) \leq \int_2^{n} ln(x)$$

En intégrant par partie :

$$\begin{aligned} \int_{x_1}^{x_2} ln(x)dx &= [x.ln(x)]_{x_1}^{x_2} - \int_{x_1}^{x_2} 1dx \\ &= [x.ln(x) - x]_{x_1}^{x_2} \end{aligned}$$

donc $\forall a > 0$, $\int_a^n ln(x)dx \in O(n.ln(n))$ ∎

## 4.3 Les tris par sélection

### 4.3.1 Principe

Soit à trier la liste $L$. Le minimum de $L$ est recherché puis placé en début de liste. Soit $L'$ la liste ainsi obtenue. On trie alors la liste commencant à partir du second élément de $L'$.

Le code de l'algorithme de tri par sélection opérant sur une liste $L$ de $n$ entiers est le suivant.

```
pour i ← 1 jqa n − 1 faire
        min ← L(i)
        pour j ← i + 1 jqa n faire
                si L(j) < min alors
                        min ← L(j)
                        m ← j
                fsi
        fait
        L(m) ← L(i)
        L(i) ← min
fait
```

### 4.3.2 Complexités

Quelquesoit la liste de longueur $n$ à trier l'algorithme s'exécute en $O(n^2)$. $O(n^2)$ est donc l'ordre de grandeur des complexités au pire et en moyenne.

## 4.4 Les tris par insertion

### 4.4.1 Principe

Soit $L$ la liste à trier, les $i-1$ premiers éléments de $L$ étant déjà triés. En déplaçant le $i^{eme}$ de $L$, on trie la liste constituée des $i$ premiers élements de $L$. Soit $L'$ la liste ainsi obtenue. On trie alors $L'$ dont les $i$ premiers éléments sont déjà triés.

Le code de l'algorithme de tri par insertion opérant sur une liste $L$ de $n$ entiers est le suivant.

```
1.      pour i ← 2 jqa n faire
2.          x ← L(i)
3.          j ← i − 1
4.          tant que (j > 0 et L(j) > x) faire
5.              L(j + 1) ← L(j)
6.              j ← j − 1
7.          fait
8.          L(j) ← x
9.      fait
```

### 4.4.2 Complexité

Pour une valeur de $i$ fixée, le temps d'exécution pour la boucle $4-7$ est maximal lorsque $x = L(i)$ est comparé à tous les éléments $L(j)$ pour $j \in \{i-1,...,1\}$. Ce cas se produit lorsque $L(i) = x < L(j)$ pour tous ces $j$. Cette situation se produit pour toutes les valeurs de l'indice $i$ lorsque la liste $L$ est triée par ordre décroissant en début d'algorithme. Ceci correspond au pire des cas pour le temps d'exécution de l'algorithme. La complexité (au pire) est donc en $O(n^2)$, $n$ étant la longueur de la liste à trier.

## 4.5 Le tri fusion

L'algorithme de tri fusion a une complexité au pire, optimale. Néanmoins il n'est pas utilisé en pratique parce que trop coûteux en mémoire. En effet, les divisions successives des tableaux nécessitent systématiquement de dupliquer les données.

### 4.5.1 Principe

Soit à trier un tableau de $n$ entiers $L[1...n]$. $L$ est séparé en deux tableaux $L'$ et $L''$ dont les tailles sont, soient égales si $n$ pair, sinon diffèrent de 1. $L'$ et $L''$ sont ensuite triés séparément par tri fusion. Enfin, $L'$ et $L''$ ainsi triés, sont fusionnés en temps linéaire en un unique tableau trié.

Le code de la procédure l'algorithme de tri par fusion opérant sur une liste $L$ de $n$ entiers est le suivant.

```
    procedure Tri − fusion(L[1...n])
debut           L' ← L[1...⌊n/2⌋]
                L'' ← L[⌊n/2⌋ + 1,n]
                Tri − fusion(L[1...⌊n/2⌋])
                Tri − fusion(L[⌊n/2⌋ + 1,n])
                Fusion(L,L',L'')
fin
```

La procédure $Fusion$, de fusion, en temps lineaire étant :

```
    procedure Fusion(L,L',L'')
debut
    i ← 1
    j ← 1
    k ← 1
    faire
        si L'(j) < L''(k) alors
            L(i) ← L'(j)
            j ← j + 1
        sinon
            L(i) ← L''(k)
            k ← k + 1
        fsi
    tant que i ≤ |L| ou j ≤ |L'| ou k ≤ |L''|
    si j > |L'| alors
        pour l ← k jqa |L''| faire
            L(i) ← L''(l)
            i ← i + 1
        fait
    sinon
        pour l ← j jqa |L'| faire
            L(i) ← L'(l)
            i ← i + 1
        fait
    fsi
fin
```

### 4.5.2 Complexité

Posons $T(n)$ le temps d'exécution au pire cas de l'algorithme de tri fusion pour un tableau de taille $n$. La phase de division du tableau $L$ en les tableaux $L'$ et $L''$

étant en temps linéaire, la phase de fusion de $L'$ et $L''$ étant également linéaire, pour tout $n$ :

$$T(n) \leq 2.T(n/2) + c.n.$$

où c est une constante positive. D'après l'étude des méthodes "diviser pour régner" — voir Chap.3, on obtient $T(n) \in O(n.log_2(n))$ (cas de réduction logarithmique avec $a = 2$, $b = 2$ et $k = 1$).

## 4.6 Le tri rapide "Quicksort" - Hoare 1962

Cet algorithme est basé sur la méthode "diviser pour régner". Sa complexité en moyenne est en $O(n.logn)$. Il est très utilisé parce que très efficace en pratique.

### 4.6.1 Principe

Soit à trier le tableau $L[g,...,d]$, avec $d > g$. *Quicksort* appliqué à $L[g,...,d]$ procède comme suit. Un élément $v$, le *pivot*, est choisi au hasard dans le tableau. Le tableau est ensuite réorganisé de manière à regrouper les éléments strictement inférieurs à $v$ en début de tableau et positionner $v$ à sa place définitive, la fin du tableau regroupant alors les éléments strictement supérieurs à $v$. Si $m$ correspond à la place définitive $v$, *Quicksort* est appelé pour trier les sous-tableaux

$$\begin{cases} L[g,...,m-1], \textit{ si } g < m-1, \\ L[m+1,...,d], \textit{ si } m+1 < d. \end{cases}$$

Dans le schéma de programme ci-dessous, la partition du tableau incombe à la fonction *Partition(L,g,d)* qui modifie $L$ et retourne la position finale du pivot dans le tableau $L[g,...,d]$.

```
Procedure Quicksort(L,g,d)
        debut
                si g < d alors
                        m ← Partition(L,g,d)
                        Quicksort(L,g,m − 1)
                        Quicksort(L,m + 1,d)
                fsi
        fin

Fonction Partition(L,g,d) — note : L est modifié par Partition
        debut
                l ← g + 1
                m ← d
                v ← L(g) — note : le pivot est pris en L(g)
                tant que l ≤ m faire
                        tant que L(m) > v faire m ← m − 1 fait
                        tant que L(l) ≤ v faire l ← l + 1 fait
```

```
                    si l < m alors
                        Echanger(L,l,m)
                        l ← l + 1
                        m ← m − 1
                    fsi
                fait
                Echanger(L,g,m)
                Partition ← m
        fin
```

**Observation 4.6.1** *La fonction Partition s'exécute en $O(n)$ pour un tableau de taille $n$ en entrée.*

### 4.6.2 Complexité au pire

La complexité au pire de *Quicksort* est en $O(n^2)$. Il existe certaines listes telles que *Quicksort* appliquées à celles-ci s'exécute de facon que les pivots soient systématiquement positionnés par *Partition* en extrémité de tableaux. Si $T(n)$ désigne le temps d'exécution de *Quicksort* pour de telles données de taille $n$, on obtient

$$\begin{cases} T(n) \geq T(n-1) + P(n) \\ T(1) = c \end{cases}$$

pour un certain polynôme du $1^{er}$ degré $P(n)$. D'après l'étude des méthodes DR on conclut que l'ordre de $T$ est $O(n^2)$.

On montre maintenant que la complexité au pire de *Quicksort* est en $O(n^2)$. Posons $T(n)$ le temps d'exécution dans le pire des cas pour une entrée de taille $n$. D'après l'Observation 4.6.1, pour tout $n \geq 2$,

$$T(n) \leq max_{1 \leq q \leq n-1}(T(q) + T(n-q)) + P(n)$$

où $P(n)$ est un polynôme de degré 1, positif en tout $n \in \mathbb{N}$. Montrons que pour une certaine constante $c$, $\forall n \in \mathbb{N}, T(n) \leq c.n^2$. Soit $n \geq 2$. Supposons l'hypothèse vérifiée pour tout $k < n$. Alors

$$T(n) \leq max_{1 \leq q \leq n-1}(c.q^2 + c.(n-q)^2) + P(n).$$

L'expression $q^2 + (n-q)^2$ atteint son maximun pour des valeurs dans l'intervalle $[1,...,n-1]$ pour $q = 1$ et $q = n-1$. Donc $max_{1 \leq q \leq n-1}(1 + (n-1)^2) = n^2 - 2.(n-1)$. Donc $T(n) \leq c.n^2 - 2.c.(n-1) + P(n)$.
En choisissant $c$ de telle manière que :

$$\begin{cases} 2.c.(n-1) \geq P(n) \quad \textit{pour tout } n \geq 2 \\ c \geq T(1) \end{cases}$$

on a $T(n) \leq c.n^2$.

### 4.6.3 Complexité en moyenne

Analysons l'exécution de *Quicksort* pour un tableau $L$ de longueur $n$. On suppose distincts les éléments du tableau et on fait l'hypothèse que les places définitives du pivot sont équiprobables, i.e. pour tout $m$ avec $g \leq m \leq d$, le pivot a une chance $1/n$ d'être placé en $m$ après l'appel de *Partition*. On observe que la fonction *Partition* s'exécute en $O(n)$ pour un tableau de taille $n$ en entrée. On note $P(n) = P_1.n + P_0$ où $P_1$,$P_0 \in \mathbb{R}^+$, un majorant du temps d'exécution de *Partition* pour un tableau de taille $n$. Si $T(n)$ désigne le temps moyen d'exécution de *Quicksort* pour des tableaux de longueur $n$ alors

pour $n \geq 2$,

$$\begin{aligned} T(n) \leq\ & P(n) + (1/n).(T(n-1) \\ & +(T(1) + T(n-2)) \\ & +...+ \\ & +(T(m) + T(n-m-1)) \\ & +...+ \\ & +(T(n-2) + T(1)) \\ & +T(n-1)) \\ =\ & P(n) + (2/n).\Sigma_{m=1,...,n-1} T(m) \end{aligned}$$

et

$$T(1) \leq c$$

pour une constante positive $c$.
On montre maintenant qu'il existe des constantes positives $a$ et $b$ telles que pour tout $n \geq 1$

$$T(n) \leq a.n.ln(n) + b.$$

Choisissons $b \geq T(1)$ (alors $a.1.ln(1) + b \geq T(1)$).
Soit $n > 1$. Supposons prouvée $T(k) \leq a.k.ln(k) + b$ pour $k < n$ alors

$$\begin{aligned} T(n) &\leq (2/n).\Sigma_{k=1,...,n-1} T(k) + P(n) \\ &\leq (2/n).\Sigma_{k=1,...,n-1}(a.k.ln(k) + b) + P(n) \\ &= (2a/n).\Sigma_{k=1,...,n-1} k.ln(k) + (2b/n)(n-1) + P(n). \end{aligned}$$

En utilisant la majoration (voir 4.6.2)

$$\Sigma_{k=1,...,n-1} k.ln(k) \leq (1/2).n^2.ln(n) - (1/4).n^2$$

$$\begin{aligned} T(n) &\leq (2a/n).((1/2).n^2.ln(n) - (1/4)n^2) + (2b/n).(n-1) + P(n) \\ &\leq a.n.ln(n) - (a/2).n + 2.b + P(n) \\ &= a.n.ln(n) + b + (P(n) + b - (a/2).n). \end{aligned}$$

En choisissant $a$ correctement, i.e. tel que $(a/2).n \geq P(n) + b$ pour $n \geq 2$ on a $T(n) \leq a.n.ln(n) + b$, pour tout $n \leq 1$. ∎

**Fait 4.6.2** $\Sigma_{k=1}^{n-1} k.ln(k) \leq (1/2).n^2.ln(n) - (1/4)n^2$

*preuve:* $\Sigma_{k=1}^{n-1} k.ln(k) \leq \int_2^n x.ln(x)dx$. En intégrant par partie, $((1/2).x^2.ln(x))' = x.ln(x)+1/2.x$, et $\int x.ln(x)dx = (1/2).x^2.ln(x) - \int (1/2).xdx = 1/2.x^2.ln(x) - 1/4x^2$. Finalement $\int_2^n x.ln(x)dx \leq 1/2.n^2.ln(n) - 1/4.n^2$. ∎

## 4.7 Le tri par tas (Williams 1964)

Comme pour le tri fusion, la complexité du tri par tas est en $O(n.log(n))$, mais le tri par tas présente l'avantage de trier "sur place". Le tri par tas utilise la structure de tas.

### 4.7.1 Les tas

Rappelons tout d'abord quelque terminologie à propos des arbres. La *profondeur* d'un noeud est la longueur du chemin de la racine à ce noeud. Un arbre est *équilibré* lorsque les profondeurs de deux quelconques de ces feuilles diffèrent au plus d'un. La *hauteur* d'un noeud dans un arbre est le nombre d'arcs maximal d'un chemin élémentaire allant du noeud à une feuille. La hauteur d'un arbre est la hauteur de sa racine.

Un *tas* consiste en :
- un ensemble d'éléments totalement ordonnés — pour simplifier des entiers distincts;
- un arbre binaire équilibré dont les noeuds sont les éléments;
tels que la propriété suivante soit satisfaite :

$$\text{si } i \text{ est le père de } j \text{ alors } i < j.$$

Tout arbre binaire équilibré de $n$ noeuds correspond à un tableau de $O(n)$ éléments $(T(i))_{1 \leq i \leq k}$ où les éléments du tableau sont les noeuds de l'arbre et les fils de $T(i)$ sont, s'ils existent, $T(2.i)$ et $T(2.i+1)$. Dans un tel tableau, si $i \neq 1$, le noeud $T(i)$ n'est pas la racine et son père est $T(\lfloor i/2 \rfloor)$.

**Observation 4.7.1** *Un arbre binaire équilibré de n noeuds à une hauteur en $O(log(n))$. Précisement, pour un arbre binaire équilibré de n éléments et de hauteur h,*

$$2^h \leq n < 2^{h+1}$$

Ci-après, deux procédures élémentaires sur les tas permettant de réaliser le tri par tas :
- *Entasser* de complexité $O(log(n))$ servant à maintenir la propriété du tas;
- *Construire* de complexité linéraire produisant un tas à partir d'un tableau.
La procédure de tri par tas *Trier* trie un tableau sur place en $O(n.log(n))$.

### 4.7.2 La procédure *Entasser*

*Entasser* prend en argument un tableau $L$ et un indice $i$ du tableau. Lorsque les arbres binaires de racines $T(2.i)$ et $T(2.i+1)$ satisfont la propriété du tas, *Entasser* réorganise l'arbre de racine $T(i)$ en un tas, en faisant "descendre" l'entier $T(i)$.

```
Procedure Entasser(L,i)
    debut
        gauche ← 2.i
        droit ← 2.i + 1
        si gauche ≤ taille(L) et L(gauche) > L(i)
        alors
                max ← gauche
        sinon
                max ← i
        fsi
        si droit ≤ taille(L) et L(droit) > L(i)
        alors
                max ← droit
        fsi
        si max ≠ i
        alors
                Echanger(L,i,max)
                Entasser(L,max)
        fsi
    fin
```

**Observation 4.7.2** *Si l'arbre $L$ est équilibré et l'arbre $L(i)$ contient $n$ noeuds, $L(2.i)$ et $L(2.i+1)$ contiennent chacun au plus $2.n/3$ noeuds.*

Si $T(n)$ désigne le nombre maximal d'opérations élémentaires à l'exécution de *Entasser* pour un tableau $L$ de taille $n$ (avec $i = 1$) alors

$$\begin{cases} T(n) \leq T(\lfloor 2.n/3 \rfloor) + c \\ T(1) = c' \end{cases}$$

pour des constantes positives $c$ et $c'$. On obtient donc que $T(n) \in O(log(n))$ d'après l'étude des méthodes DR.

### 4.7.3 La procédure *Construire*

On utilise *Entasser* pour réorganiser un tableau $L$ en un tas. Les éléments du tableau $L[\lfloor (n/2)+1 \rfloor,...,n]$ sont tous des feuilles de l'arbre de racine $L(1)$, le père du noeud $L(n)$ étant le noeud $L(\lfloor (n/2) \rfloor)$. *Construire* traverse les noeuds restants par hauteur croissante et exécute *Entasser* pour chacun d'eux. Ce parcours garantit

que lorsque *Entasser* est appelée pour un noeud, les sous arbres enracinés en ces fils sont des tas.

```
Procedure Construire(L)
        debut
                pour i ← ⌊taille(L)/2⌋ en descendant jqa 1 faire
                        Entasser(L,i)
                fait
        fin
```

Considérons un tas de $n$ éléments. Sa hauteur $H$ est au plus $\lfloor log_2(n) \rfloor$ et le nombre d'éléments de hauteur $h$ est majorée par $2^{H-h} \leq n/2^h$. Le temps maximal requis par l'exécution de *Entasser* en un noeud de hauteur $h$ étant en $O(h)$, le temps d'exécution de *Construire* pour un tableau de taille $n$ est majoré par

$$\Sigma_{h=0,...,\lfloor log_2 n \rfloor} n/2^h.O(h) = O(n.\Sigma_{h=0,...,\lfloor log_2 n \rfloor}(h/2^h)).$$

En appliquant la formule [1]

$$x/(1-x)^2 = \Sigma_{h=0}^{\infty} h.x^h,$$

pour $x = 1/2$, on a

$$\Sigma_{h=0,...,\infty} h/2^h = 2.$$

Le temps d'exécution de *Construire* est en

$$O(n.\Sigma_{h=0,...,\infty}(h/2^h)) = O(n).$$

### 4.7.4 La procédure *Trier*

La procédure *Trier* appliquée au tableau $L$ de taille $n$ commence par organiser $L$ en un tas par un appel de *Construire*. On trie ensuite le tas $L$ en lui appliquant le processus récursif suivant. L'élément minimum du tableau est $L[1]$. On le place ensuite à sa position finale correcte en permutant les contenus des cases 1 et $n$. Le tableau $L[1,...,n-1]$ constitué des cases de $L$ de 1 à $n-1$ correspond à un arbre binaire dont les sous-arbres enracinés en $L[1]$ sont des tas. *Entasser* appliquée à $L[1,...,n-1]$ réorganise $L[1,...,n-1]$ en un tas. Ce processus est ensuite répété successivement pour les tas $L[1,...,k]$, $k$ variant de $n-1$ à 2.

```
Procedure Trier(L)
        debut
```

1. Rappel : de $1/(1-x) = \Sigma_{h=0}^{\infty} x^h$, on obtient par dérivation $1/(1-x)^2 = \Sigma_{h=0}^{\infty} h.x^{h-1}$, en multipliant les deux membres de l'égalité par $x$, $x/(1-x)^2 = \Sigma_{h=0}^{\infty} h.x^h$

```
        Construire(L)
        pour k ← taille(L) jqa 2 faire
                    Echanger(L,1,k)
                    k ← k − 1
                    Entasser(L[1,...,k],1)
        fait
fin
```

La complexité de *Trier* est en $O(nlog(n))$ puisque l'appel à construire est en $O(n)$ et les $n-1$ appels à *Entasser* sont chacun en $O(log(n))$.

**Références**

• T.Cormen, C.Leiserson, R.Rivest, *Introduction à l'algorithmique* - Dunod 1994.

Le tri par tas :

• Williams J.W.J., *Algorithm 232 (heapsort)*, Communications of the ACM, vol.7, 1964, pp. 347-348.

Le tri rapide :

• Hoare C.A.R. *Quicksort*, Computer Journal, vol 5., n.1, 1962, pp. 10-15.

# Chapitre 5

# Méthode des essais successifs (E.S.)

## 5.1 Principe

Pour de nombreux problèmes il n'existe pas d'algorithme connu autre qu'une recherche parmi l'ensemble des solutions et d'un test afin de déterminer les bonnes solutions. Pour ces problèmes, on peut rechercher :
- une bonne solution;
- une "meilleure" solution selon un certain critère d'optimalité;
- toutes les bonnes solutions;
- toutes les meilleures solutions.

La méthode des essais successifs peut s'appliquer à ce type de problèmes lorsque l'ensemble des solutions est un produit fini. Le modèle de problème traitable par la méthode E.S. est le suivant :
- Les solutions se décomposent en vecteurs de la forme $(X_1,...,X_n)$, $X_i$ parcourant l'ensemble $S_i$ qui est fini. Il y a donc $card(S_1) \times ... \times card(S_n)$ solutions possibles à priori.
- Un critère d'optimalité ou de validité $P$ est défini. Celui-ci permet l'évaluation des solutions.

Le but de la méthode E.S. est d'éviter de construire toutes les solutions (la recherche exhaustive). Pour ce faire la méthode E.S. utilise des critères $P_i$ dits "critères partiels". L'idée est de détecter le plus tôt possible au cours de l'élaboration d'une solution si celle-ci ne peut s'avérer bonne (ou optimale).

**Essence de la technique E.S.**
(1) On construit les solutions coordonnées par coordonnées :
$X_1$, puis $X_2$, puis ... $X_i$, ...
(2) Quand on calcule la $i$ème coordonnée $X_i$, on teste un prédicat partiel $P_i$ lequel

satisfait les conditions suivantes.
- $P_i(X_1,...,X_i) = faux$ si et seulement si pour tout $(Y_{i+1},...,Y_n)$, $(X_1,...,X_i,Y_{i+1},...,Y_n)$ ne peut être une bonne solution (ou la meilleure solution suivant le critère d'optimalité choisi).
- Lorsque $P_n(X_1,...,X_n)$ est vrai alors $(X_1,...,X_n)$ est une bonne solution (ou une meilleure solution).

## 5.2 Exemple : Le problème des $n$ reines

Le problème consiste à placer $n$ reines sur un échiquier $n \times n$ sans situation de prise mutuelle. Il n'existe pas de solution analytique connue à ce problème.

Le critère $P$ définit les solutions acceptées :

$$P = P_{lig} \cap P_{col} \cap P_{diag};$$

où :

- $P_{lig}$ : il n'y a pas deux reines sur la même ligne;
- $P_{col}$ : ——————————————— colonne;
- $P_{diag}$ : ——————————————— diagonale.

**Espace des solutions possibles.** D'après $P_{lig}$, on peut se restreindre aux configurations avec une reine par ligne. Dans notre modélisation on convient que la $i^{eme}$ reine est placée sur la $i^{eme}$ ligne. Une solution $(X_1,X_2,X_3,...,X_n)$ correspond à la configuration dans laquelle la $i^{eme}$ reine occupe la colonne numéro $X_i$.
$P_{col}$ équivaut alors à :

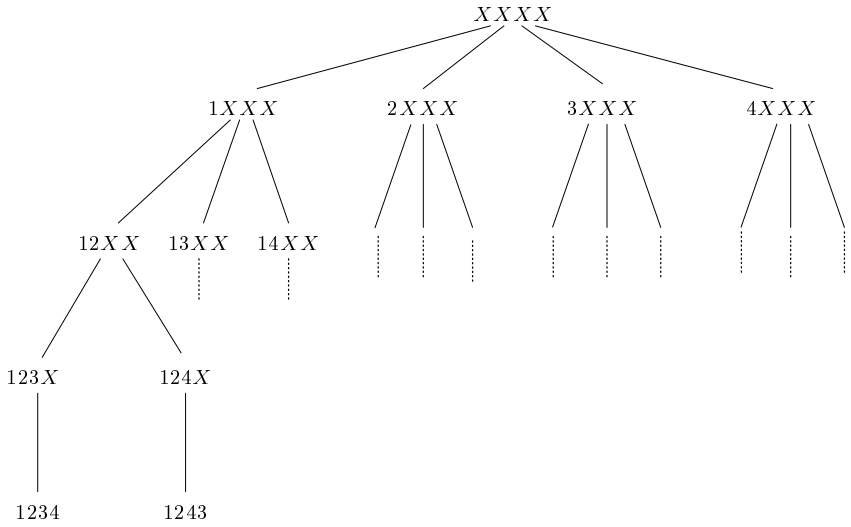
$$\forall i \neq j, X_i \neq X_j.$$

Toute solution est nécessairement une permutation sur les $n$ premiers entiers. L'espace des solutions est donc réduit à $n!$ (au lieu des $n^n$).

*Exemple pour* $n = 4$. $4^4 = 256$, $4! = 24$.

Les solutions peuvent être représentées de manière arborescente. La hauteur de l'arbre étant la taille des vecteurs solutions.

*Exemple pour* $n = 4$.




Le nombre de feuilles est $n!$ (ici $4! = 24$).

**Remarque.** Attention à la programmation! En effet, il convient d'apporter un soin tout particulier à la programmation comme le montre les exemples suivants. Soit à calculer l'ensemble des solutions candidates pour le problème des $n$ reines. Pour simplifier nous écrirons les algorithmes pour $n = 4$. Considérons d'abord l'algorithme suivant.

$S \leftarrow \emptyset$
pour $X_1 \leftarrow 1$ jqa 4 faire
  pour $X_2 \leftarrow 1$ jqa 4 faire
    pour $X_3 \leftarrow 1$ jqa 4 faire
      pour $X_4 \leftarrow 1$ jqa 4 faire
        si $((X_1 \neq X_2) \wedge (X_1 \neq X_3) \wedge (X_1 \neq X_4) \wedge$
        $(X_2 \neq X_3) \wedge (X_2 \neq X_4) \wedge (X_3 \neq X_4))$ alors
          $S \leftarrow S \cup \{(X_1,X_2,X_3,X_4)\}$
        fsi
      fait
    fait
  fait
fait

Ce type d'algorithme est de complexité $O(n^n)$ bien que l'espace des solutions soit de cardinalité $n!$. En effet, le test en boucle interne est effectué $n^n$ fois.

Considérons maintenant l'algorithme suivant.

$S \leftarrow \emptyset$
pour $X_1 \leftarrow 1$ jqa 4 faire

```
        pour X2 ← 1 jqa 4 faire
            si X1 ≠ X2 alors
                pour X3 ← 1 jqa 4 faire
                    si (X3 ≠ X1) ∧ (X3 ≠ X2) alors
                        pour X4 ← 1 jqa 4 faire
                            si (X4 ≠ X1) ∧ (X4 ≠ X2) ∧ (X4 ≠ X3)
                            alors S ← S ∪ {(X1,X2,X3,X4)} fsi
                        fait
                    fsi
                fait
            fsi
        fait
    fait
```

Ce second type d'algorithme est de complexité $O(n!)$. L'idée est de tester les solutions par rapport aux critères le plus tôt possible.

Déterminons maintenant les prédicats partiels pour le problème des $n$ reines.
- $P_{lig}$ est garanti par construction.
- Pour tout $i$, $1 \leq i \leq n$ :

$$P_i(X_1,...,X_i) = P^i_{col}(X_1,...,X_i) \wedge P^i_{diag}(X_1,...,X_i)$$

où :
- $P^i_{col}(X_1,...,X_i) = \bigwedge_{1\leq k\leq i}(X_i \neq X_k)$,
- $P^i_{diag}(X_1,...,X_i) = \bigwedge_{1\leq k\leq i}(i,X_i) \notin diag(k,X_k)$,

$diag(x,y)$ désignant les diagonales de $(x,y)$.

En appliquant le principe de tester au plus tôt les prédicats partiels, on obtient l'algorithme ci-dessous lequel calcule l'ensemble des bonnes solutions du problème (toujours pour $n = 4$).

```
    S ← ∅
    pour X1 ← 1 jqa 4 faire
        pour X2 ← 1 jqa 4 faire
            si P²col(X1,X2) ∧ P²diag(X1,X2) alors
                pour X3 ← 1 jqa 4 faire
                    si P³col(X1,X2,X3) ∧ P³diag(X1,X2,X3) alors
                        pour X4 ← 1 jqa 4 faire
                            si P⁴col(X1,X2,X3,X4) ∧ P⁴diag(X1,X2,X3,X4) alors
                                S ← {(X1,X2,X3,X4)}
                            fsi
                        fait
                    fsi
```

```
                fait
            fsi
        fait
    fait
```

## 5.3 Un modèle général d'algorithme - récursivité

**Utilisation de la récursivité.**
Les techniques récursives sont mises en oeuvre pour la recherche parmi les solutions. Ceci conduit à un procédé indépendant de la taille $n$ des vecteurs solutions $(X_i)_{1 \leq i \leq n}$. Ce qui n'était pas le cas avec les écritures précédentes. Pour certains problèmes, les tailles des vecteurs solutions ne sont pas connues à priori, voire ne sont pas constantes. La récursivité apporte donc plus de généralité et plus de souplesse à la méthode.

La procédure de recherche récursive $M$ est de la forme :

$procedure\ M(i)$

pour $X_i$ parcourant $S_i$ faire
    si $P_i(X_1,...,X_i)$ alors $M(i+1)$ fsi.
fait.

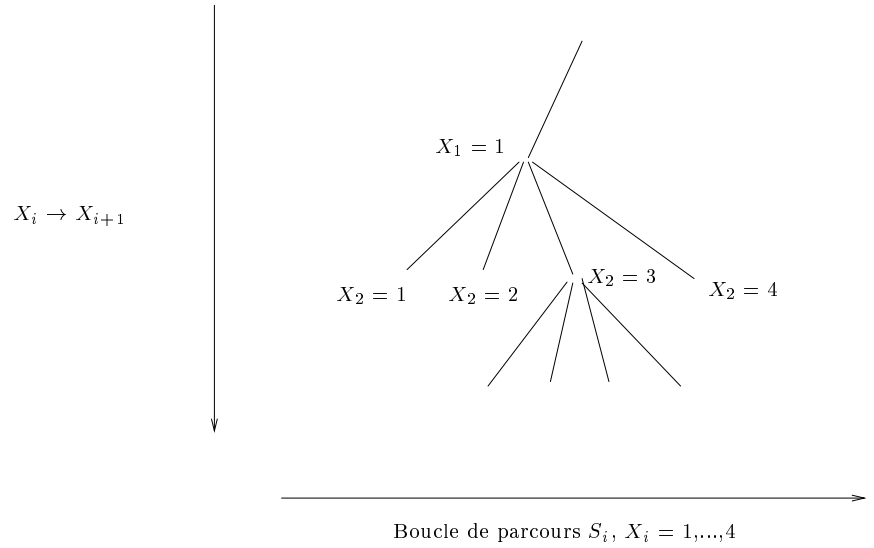
Le paramètre $i$ indiquant que $M$ scrute les $X_i \in S_i$ Un algorithme générique de recherche de toutes les solutions par E.S. est le suivant.

$procedure\ Recherche - solutions(i)$

pour $X_i$ parcourant $S_i$ faire
    si $P_i(X_1,...,X_i)$ alors
        - effectuer les modifications sur l'énoncé du problème
          qu'entraîne le choix de $X_i$,
        - si $i < n$ alors $Recherche - solutions(i+1)$
        sinon $(X_1,...,X_n)$ est une solution fsi
        - Revenir à l'énoncé du problème antérieur
          au choix de $X_i$.
    fsi
fait

Afin de calculer toutes les solutions satisfaisant le critère $P$, on appelle $recherche - solutions(1)$.

La récursivité construit de manière incrémentale la solution, l'itération en $M(i)$ fournit les valeurs successives de $X_i$ à essayer.




Il est nécessaire d'effectuer les modifications qu'entraînent les choix des $X_i$ sur les énoncés : il s'agit de gérer correctement la mémoire en fonction de l'état du vecteur solution partielle courant ($(X_1,...,X_i)$ pour $M(i)$). Une gestion récursive de la mémoire par pile convient généralement.

### 5.3.1 Application au problème des $n$ reines

**Représentation des données.**
Une modélisation triviale est d'utiliser une matrice $n \times n$ booléenne. Cette solution nécessite une taille mémoire en $O(n^2)$ et implique des calculs intermédiaires lourds pour les prédicats partiels $P_i$. On propose une modélisation plus adaptée au problème. Les objets intéressants sont :
- les lignes;
- les colonnes;
- les diagonales.

*Les lignes.* Elles sont prises en compte dans la représentation de la solution : la reine $i$ occupe la colonne $i$.

*Les colonnes.* On utilise un tableau $X$ de taille $n$ d'entiers appartenant à $\{1,...,n\}$. Le $i^{eme}$ élément du tableau, $X(i)$, désignera la colonne de la reine $i$. Pour simplifier on utilisera également un tableau $a$ de booléens de taille $n$ tel que $a(i)$ sera vrai si et seulement s'il n'y a pas de reine en colonne $i$.

*Les diagonales.* Soit $(i,j) \in \{1,...,n\}^2$. $(x,y)$ appartient à la même diagonale que $(i,j)$ si et seulement si $x - i = y - j$ ou $x - i = j - y$. On numérote les diagonales

comme suit.
- Pour $l \in \{1-n,...,n-1\}$, $(x,y)$ appartient à la $l^{ieme}$ diagonale *positive* si et seulement si $l = y - x$.
- Pour $l \in \{2,...,2n\}$, $(x,y)$ appartient à la $l^{ieme}$ diagonale *négative* si et seulement si $l = y + x$.
On gère deux tableaux de booléens $dp$ et $dn$.
- $dp$ a pour indices $\{1-n,...,n-1\}$ et $dp(l)$ est vrai si et seulement s'il n'y a pas de reines placées sur la $l^{ieme}$ diagonale positive.
- $dn$ a pour indices $\{2,...,2n\}$ et $dn(l)$ est vrai si et seulement s'il n'y a pas de reines placées sur la $l^{ieme}$ diagonale négative.

On adapte l'algorithme générique précédent en posant pour tout $i$,

$$P_i(X_1,...,X_i) = a(X_i) \wedge dp(X_i - i) \wedge dn(X_i + i).$$

# Chapitre 6

# Heuristiques

*Rappel :* Les problèmes d'optimisation sont les problèmes de recherche parmi un ensemble de solutions correctes d'une meilleure solution selon un critère d'optimalité.

Comme on l'a vu lors de l'étude des méthodes "par essais successifs", les algorithmes résolvant les problèmes de recherche ou d'optimisation élaborent souvent les solutions de manière incrémentale. Lors de leurs exécutions ces algorithmes parcourent une série d'étapes au cours desquelles ils choisissent parmi un ensemble d'options la forme de la solution définitive. Avec la technique des essais successifs (ES) ces choix ne sont pas gérés : l'algorithme teste les options les unes après les autres sans critères de préférence. Ces choix peuvent aussi être réalisés par des fonctions de sélection — appelées *heuristiques*. En général et comme pour les méthodes par ES, afin d'obtenir une solution exacte ou optimale avec une méthode heuristique, l'algorithme doit pouvoir remettre en cause les choix qu'il effectue.

## 6.1 Les algorithmes gloutons

Les algorithmes gloutons sont des algorithmes fondés sur les heuristiques. Leur particularité est de ne jamais invalider les choix qu'ils effectuent – d' où la dénomination de "glouton". à cause de ces choix définitifs, la correction d'un algorithme glouton n'est assurée que si la succession des choix dus à son heuristique mène à une bonne solution, ou à une solution optimale. L'avantage des algorithmes gloutons est leur faible complexité. Ainsi pour un problème d'optimisation, il est parfois préférable d'obtenir une solution approchée par un algorithme glouton qu'une solution optimale par un algorithme trop coûteux.

### 6.1.1 Le problème du choix d'activités

Le problème consiste à répartir une ressource parmi plusieurs activités concurrentes. Soit un ensemble $S = \{1,...,n\}$ de $n$ activités concurrentes souhaitant utiliser

une même ressource laquelle ne peut supporter qu'une activité à la fois. Chaque activité a un horaire de début $d_i$ et un horaire de fin $f_i$ avec $d_i \leq f_i$. Si elle est choisie l'activité $i$ a lieu pendant l'intervalle de temps $[d_i,f_i[$. Les activités $i$ et $j$ sont compatibles lorsque $d_i \geq f_j$ ou $d_j \geq f_i$ Le problème consiste à choisir le plus grand nombre d'activités compatibles entre elles.

Un algorithme glouton résolvant ce problème est indiqué par le pseudo-code suivant. On suppose les activités triées de manière à ce que

$$f_1 \leq f_2 \leq ... \leq f_n.$$

Si tel n'est pas le cas cette liste de $n$ activités peut être triée en $O(n.log(n))$.

procedure $Choix - activites - glouton$
debut
$A \leftarrow 1$
$j \leftarrow 1$
pour $i \leftarrow 2$ jqa $n$ faire
si $d_i \geq f_j$ alors
$A \leftarrow A \cup \{i\}$
$j \leftarrow i$
fait
fin

L'algorithme opère comme suit. L'ensemble $A$ collecte les activités sélectionnées. La variable $j$ indique l'ajout le plus récent à A. D'après le tri initial des activités, $f_j = max\{f_k \mid k \in A\}$. L'activité 1 est la première sélectionnée, puis l'algorithme ajoute systématiquement à $A$, la première activité $i$ compatible avec celles déjà présentes dans $A$.

*Remarque :* $Choix - activites - glouton$ est de complexité linéaire.

**Fait 6.1.1** *$Choix-activites-glouton$ donne les solutions optimales pour le problème du choix d'activités.*

La correction de $Choix - activites - glouton$ résulte du lemme suivant.

**Lemme 6.1.2** *Soit une instance $I = \{(d_i,f_i) \mid i \in S\}$, $m$ l'indice d'activité d'horaire de fin minimal dans $I$, l'instance $I' = \{(d_i,f_i) \mid i \in S \wedge d_i \geq f_m\}$, et $A'$ est une solution optimale du problème pour l'instance $I'$. $A = A' \cup \{m\}$ est une solution optimale du problème pour $I$.*

*preuve :* $A$ est une solution correcte pour $I$, puisque composée d'activités compatibles de $I$. Considérons une solution $B$ optimale pour $I$ et montrons que $card(B) \leq card(A)$ — de ce fait $A$ est optimale pour $I$. Soit $k$ l'indice de l'activité de $B$ d'horaire

de fin minimale. Alors $f_k \geq f_m$ et $B' = B \backslash \{k\}$ est composé d'indices d'activités compatibles de $I'$. Par optimalité de $A'$, $card(B') \leq card(A')$, donc $card(B) \leq card(A)$. ∎
En reprenant les notations du Lemme 6.1.2, une solution optimale $A$ pour un ensemble d'instance $I$ peut donc être construite en déterminant une solution optimale pour l'instance $I'$ et en lui adjoignant l'indice $m$. C'est exactement le principe de $Choix - activites - glouton$.

### 6.1.2 Le problème du sac à dos (KP)

Le problème du sac à dos ("KP" pour "Knapsach problem") dans sa version "tout ou rien" est le suivant. Il s'agit de déterminer

$$Z = max\{\Sigma_{i=1}^{n} p_i.x_i \mid \Sigma_{i=1}^{n} w_i.x_i \leq c, x_i \in \{0,1\}\}$$

pour des valeurs entières positives $p_i$, $w_i$ et $c$ données. L'image est celui d'un sac à dos que l'on souhaite remplir d'objets numérotés de 1 à n. L'objet numéro $i$ ayant la valeur $p_i$ et le poids $w_i$. On recherche le chargement du sac de plus grande valeur et dont le poids n'excède pas $c$.

Le critère de sélection — ou heuristique, est le suivant. Les indices $i$ sont choisis préférentiellement selon le rapport $p_i/w_i$ : si $p_j/w_j \geq p_i/w_i$ alors j est considéré meilleur que $i$.

Donnons le pseudo-code de l'algorithme glouton correspondant à cette heuristique. On suppose les indices triés de 1 à $n$ de telle façon que :

$$p_1/w_1 \geq p_2/w_2 \geq ... \geq p_n/w_n.$$

L'algorithme utilise les variables suivantes.
- $C$, un entier mémorisant le poids libre restant dans le sac à dos,
- des variables "booléennes" $X_i \in \{0,1\}$ où $1 \leq i \leq n$, telles que $X_i = 1$ si seulement si l'objet $i$ est mis dans le sac.

*Note :* La fonction ci-dessous retourne le résultat de l'heuristique, elle sera utilisée ultérieurement pour coder d'autres algorithmes (Cf. 6.2.2).

```
fonction greedy − KP
debut
        C ← c
        pour i ← 1 jqa n faire
                si w_i ≤ C alors
                        C ← C − w_i
                        X_i ← 1
```

```
                    sinon X_i ← 0
            fait
            greedy − KP ← C
fin
```

**Exemple.** Trouver le maximum de

$$20.x_1 + 16.x_2 + 11.x_3 + 9.x_4 + 7.x_5 + x_6$$

avec

$$9.x_1 + 8.x_2 + 6.x_3 + 5.x_4 + 4.x_5 + x_6 \leq 12.$$

En appliquant $greedy - KP$, on obtient successivement : $i = 1$, $c = 12$, $x_1 = 1$, $c = 3$, puis $x_2 = x_3 = x_4 = x_5 = 0$, $i = 6$, $c = 3$, $x_6 = 1$. à la fin de l'algorithme $c = 2$ (le sac n'est pas plein) $Z = 21$. Une solution optimale est en fait (0,1,0,0,1,0) avec $Z = 23$ et le sac est plein.

On étudie maintenant le problème KP dans sa version fractionnaire. Le problème diffère du précédent simplement du fait que les objets sont supposés "divisibles". C'est à dire que l'on veut estimer

$$Z = max\{\Sigma_{i=1}^{n} p_i.x_i \mid \Sigma_{i=1}^{n} w_i.x_i \leq c, 0 \leq x_i \leq 1\}$$

où les valeurs des $x_i$ sont rationnelles.

L'algorithme glouton $greedy - KP - frac$ pour ce problème est défini en pseudo-code par :

```
procedure greedy − KP − frac
debut
        C ← c
        pour i ← 1 jqa n faire
                si w_i ≤ C alors
                        X_i ← 1
                        C ← C − w_i
                sinon
                        X_i ← C/w_i
                        C ← 0
                fsi
        fait
fin
```

**Fait 6.1.3** *Si une instance de $KP$ est donnée par les valeurs $c$ et $p_i$, $w_i$ avec $1 \leq i \leq n$ telles que $p_1/w_1 \geq p_2/w_2 \geq ... \geq p_n/w_n$ alors $greedy - KP - frac$ produit une solution $(x_i)_{1 \leq i \leq n}$, optimale dans $\mathbb{R}^n$ pour cette instance.*

### 6.1.3 Algorithmes gloutons classiques

- concernant les graphes : Kruskal, Prim : calcul d'arbre de recouvrement (de poids minimal), Dijkstra : calculs du plus court chemin ou du chemin de coût minimal à partir d'une source unique.
- pour la compression de données : les codages de Huffman.

### 6.1.4 Applicabilité de la méthode gloutonne

Il existe deux caratéristiques qu'un problème d'optimisation peut présenter et qui se prêtent à la stratégie gloutonne.

**La propriété du choix gloutons**

On peut construire une solution optimale en effectuant des choix localement optimaux.

**L'existence de sous-structure optimale**

Un problème a la propriété de sous-structure optimale lorsqu'une solution optimale contient des solutions optimales de ses sous-problèmes. Reprenons par exemple la démonstration de la correction de l'algorithme $Choix - activites - gloutons$, on a vu qu'une solution optimale $A$ pour une instance $I$ contenait une solution optimale $A' = A\backslash\{m\}$ pour l'instance $I' = \{(d_i,f_i) \in I \mid d_i \geq f_m\}$.

## 6.2 Séparation et évaluation progressive (SEP)

### 6.2.1 Principes

Ces méthodes s'appliquent aux problèmes d'optimisation. Ce sont des améliorations des méthodes type ES. Elles utilisent des heuristiques afin de limiter au mieux l'exploration de l'espace des solutions. Pour déterminer les solutions optimales, la méthode SEP reprend le principe d'exploration de la méthode ES. Une évaluation des solutions partielles permet certaines améliorations. Un premier point est de limiter l'exploration aux options pouvant mener à de meilleures solutions que celles déjà calculées — c'est le principe d'*élagage*. Un second est d'explorer préférentiellement les options estimées les meilleures (au vu de leurs évaluations).

Le nom "séparation et évaluation progressive" est justifié par les remarques suivantes :

- (1) Si le noeud sélectionné n'est pas solution on génère ses successeurs. C'est le principe de *séparation*.
- (2) L'heuristique permet un parcours intelligent de l'arbre des solutions en sélectionnant les noeuds estimés les meilleurs, c'est le principe d'*évaluation*.

Par la suite, on considère un problème d'optimisation générique, et on donne un algorithme de résolution type SEP pour ce problème.

Pour une instance du problème, on pose :

- $SA$ = l'ensemble des solutions,
- $SC$ = l'ensemble des solutions correctes,
- $SO$ = l'ensemble des solutions optimales.

Bien sûr

$$SO \subseteq SC \subseteq SA.$$

Comme pour l'étude des méthodes ES on suppose que :
- Les solutions s'expriment sous forme de vecteurs, i.e.

$$SA = \{(X_1,...,X_n) \mid X_i \in S(X_1,...,X_{i-1})\}.$$

- On sait calculer des prédicats de correction partielles $PP$ tels que

$$\exists Y_{i+1},...,Y_n,(X_1,...,X_i,Y_{i+1},...,Y_n) \in SC \Rightarrow PP(X_1,...,X_i) = vrai.$$

On suppose de plus que le critère d'optimalité est spécifié par une fonction $f$ définie sur l'ensemble des solutions partielles, à valeurs réelles. Le critère consiste à minimiser $f$, c'est à dire

$$SO = \{X^* \in SC \mid f(X^*) \leq f(X), \forall X \in SC\}.$$

**Exemple.** Pour l'instance générique de KP définie au 6.1.2 :
- $SA = \{(X_1,...,X_n) \mid X_i \in \{0,1\}\}$,
- $SC = \{(X_1,...,X_n) \in SA \mid \Sigma_{i=1}^n w_i.X_i \leq c\}$,
- $SO = \{(X_1^*,...,X_n^*) \in SC \mid \Sigma_{i=1}^n p_i.X_i^* \geq \Sigma_{i=1}^n p_i.X_i, \forall (X_1,...,X_n) \in SC\}$,

i.e. $f(X_1,...,X_n) = -\Sigma_{i=1}^n p_i.X_i$.

Le pseudo-code d'un algorithme type SEP résolvant un tel problème d'optimisation est donné ci-après. Il utilise les variables suivantes.
- $g^{opt}$, la valeur de la meilleure solution trouvée jusqu'alors. Elle est initialisée à $+\infty$ et ne peut que décroître. La valeur de $g^{opt}$ peut parfois être initialisée plus habilement, à l'aide d'un algorithme glouton par exemple (voir ci-après 6.2.2)
- $Z = (X_1^*,...,X_n^*)$ correspond au vecteur de la meilleure solution trouvée jusqu'alors : $f(Z) = g^{opt}$.
- $X = (X_1,...,X_i)$ est le vecteur de la solution partielle en cours d'étude.
- $L$ est la liste des solutions partielles restant à développer.
- Enfin, $solution - trouvee$ est une variable booléenne indiquant si une solution a été trouvée.

Une heuristique $g((X_1,...,X_i))$ définit un minorant des valeurs des solutions que l'on peut construire à partir de $(X_1,...,X_i)$. C'est à dire,

$$\forall (Y_{i+1},...,Y_n),((X_1,...,X_i,Y_{i+1},...,Y_n) \in SC \Rightarrow g((X_1,...,X_i)) \leq f(X_1,...,X_i,Y_{i+1},...,Y_n)).$$

$PP((X_1,...,X_i))$ retourne le prédicat de correction partielle en $(X_1,...,X_i)$.
Pour une solution partielle $X$, $Succ(X)$ retourne l'ensemble des options possibles au

noeud $X$. Ces options sont les solutions partielles, "fils" de $X$, c'est à de la forme $(X_1,...,X_i,X')$ où $X = (X_1,...,X_i)$ et $X'$ est une nouvelle coordonnée.

```
procedure meilleure − sol − SEP
debut
        — initialisations
        solution − trouvee ← faux
        g^opt ← +∞
        X ← () — liste vide
        L ← ({X}) —
        faire
                si X est solution alors
                        si g(X) < g^opt alors
                        — X est la meilleure solution trouvée jusqu'alors
                                Z ← X
                                g^opt ← g(X)
                                solution − trouvee ← vrai
                        sinon si PP(X) et g(X) < g^opt alors
                        — X est une solution partielle correcte pouvant
                        — mener à une meilleure solution que celle
                        — déjà trouvée.
                        Retirer X de L — séparation
                        Pour tout Y de Succ(X) faire — séparation
                                calculer g(Y) — évaluation
                                et inserer Y dans L
                                suivant les valeurs de g croissantes;
                        fait
                fsi
                X ← premier element de L
                L ← L moins son premier element
        jusqu'à ce que L soit vide.
```

**Remarque.** Contrairement à la méthode ES, il n'est généralement pas possible d'implémenter le parcours de l'arbre des solutions partielles par des procédures récursives. SEP est également coûteuse en mémoire puisque le parcours de l'arbre des solutions nécessite de stocker simultanément plusieurs vecteurs de solutions en cours d'étude.

### 6.2.2 SEP appliquée à KP

Reprenons l'instance générique de $KP$ du 6.1.2 :

$$\begin{cases} \text{minimiser } -\Sigma_{1\leq i\leq n} p_i.x_i \\ \text{avec } \Sigma_{1\leq i\leq n} w_i.x_i \leq c \end{cases}$$

L'heuristique $g$ peut être définie de la facon suivante. $(X_1,...,X_i)$ désignant une solution partielle, l'heuristique en $X = (X_1,...,X_i)$ est $f(X) + g^*(X)$ où :
- $f(X_1,...,X_i) = -\Sigma_{k=1}^{i} p_k.X^k$ est la valeur de la solution partielle $(X_1,...,X_i)$,
- $g^*((X_1,...,X_i))$ est la partie entière de la valeur de la solution obtenue par l'application de $greedy - KP - frac$ à l'instance :

$$\begin{cases} \text{minimiser } -\Sigma_{k=i+1}^{n} p_k.x_k \\ \text{avec } \Sigma_{k=i+1}^{n} w_k.x^k \leq c - W((X_1,...,X_i)), \end{cases}$$

où $W((X_1,...,X_i)) = \Sigma_{k=1}^{i} w_k.X_k$ est le poids de la solution partielle $(X_1,...,X_i)$.
Ceci est un exemple d'heuristique calculée à partir d'un algorithme glouton. $g^{opt}$, la valeur de la meilleure solution jusqu'alors trouvée peut être initialisée par la valeur retournée par la fonction $greedy - KP$ (voir 6.1.2).

Le pseudo-code de l'algorithme résolvant $KP$ selon la méthode SEP est le suivant.

```
procedure meilleure - sol - KP
debut
        — initialisations
        solution - trouvee ← faux
        g^opt ← greedy - KP
        X ← () — liste vide
        L ← ({X})
        faire
                si W(X) ≤ c et longueur(X) = n
                — n est le nombre d'objets de l'instance
                alors
                        si f(X) < g^opt alors
                        — X est la meilleure solution trouvée jusqu'alors
                                Z ← X
                                g^opt ← f(X)
                                solution - trouvee ← vrai
                        sinon si W(X) ≤ c et
                        f(X) + ⌊greedy - KP - frac(Y,c)⌋ < g^opt alors
                        — X est une solution partielle correcte pouvant
                        — mener à une meilleure solution que celle déjà
                        — trouvée.
                        — Retirer X de L.
                        Pour tout x de {0,1} faire
                                Y ← (X_1,...,X_i,x) — où X = (X_1,...,X_i)
                                g(Y) ← f(Y) + ⌊greedy - KP - frac(Y,c)⌋ et
                                inserer Y dans L suivant les valeurs
                                de g croissantes;
                        fait
```

fsi
$X \leftarrow$ *premier element de* $L$
$L \leftarrow$ $L$ *moins son premier element*
jusqu'à ce que $L$ soit vide.

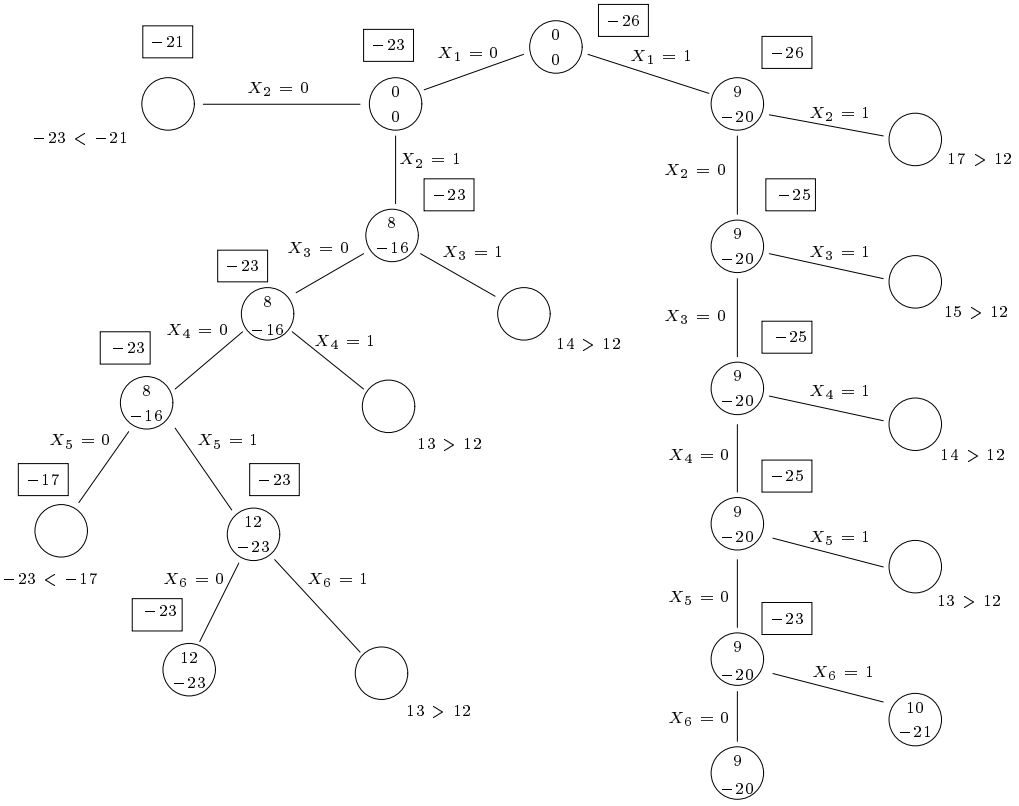
fonction $greedy - KP - frac(Y,c)$
debut
$z \leftarrow 0$
$C \leftarrow c - W(Y)$
pour $j \leftarrow longueur(Y) + 1$ jusqu'à $n$ faire
si $w_j \leq C$ alors
$z \leftarrow z + p_j$
$C \leftarrow C - w_j$
sinon
$z \leftarrow z + p_j * C/w_j$
$C \leftarrow 0$ — car $C \leftarrow C - w_j * C/w_j$
fsi
fait
$greedy - KP - frac \leftarrow z$
fin

La figure suivante indique l'exécution de l'algorithme précédent pour l'instance :

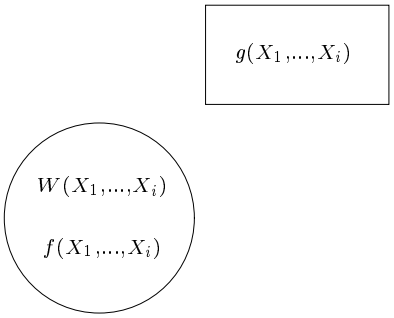
$$\begin{cases} \text{minimiser } -20.x_1 - 16.x_2 - 11.x_3 - 9.x_4 - 7.x_5 - x_6 \\ \text{avec } 9.x_1 + 8.x_2 + 6.x_3 + 5.x_4 + 4.x_5 + x_6 \leq 12 \end{cases}$$

à l'initialisation $g^{opt} = 21$ qui est la valeur de la solution (1,0,0,0,0,1). Pendant le parcours de l'arbre des solutions, les noeuds sont évalués selon les valeurs de

$g = f + g^*$ croissantes.




Les étiquettes aux noeuds indiquent:



# Chapitre 7

# Programmation dynamique

## 7.1 Introduction

Le principe de programmation dynamique s'applique à des problèmes d'optimisation possédant certaines propriétés de décomposition. Comme pour la méthode DR, une solution à une instance d'un problème est construite à partir des solutions pour des instances de moindre taille. Le principe de la programmation dynamique diffère de celui des méthodes DR sur le point suivant. Lors de l'élaboration d'une solution, certaines instances du problème peuvent apparaître plusieurs fois. La gestion récursive du type DR nécessite le calcul d'une solution pour chaque occurrence d'instance rencontrée. Une solution pour une même instance peut donc être calculée plusieurs fois. Le principe de programmation dynamique évite cette répétition de calculs : les solutions optimales sont calculées pour des instances de tailles croissantes et ces solutions sont systématiquement mémorisées.

## 7.2 La programmation dynamique pour KP

Considérons l'instance générique de $KP$ :

$$\begin{cases} \text{minimiser } -\Sigma_{1\leq i\leq n} p_i.x_i \\ \text{avec } \Sigma_{1\leq i\leq n} w_i.x_i \leq c \\ \text{et pour tout } i,\ x_i \in \{0,1\}. \end{cases}$$

Par la suite $KP(n,c)$ désigne cette instance et généralement $KP(j,y)$ est l'instance

$$\begin{cases} \text{minimiser } -\Sigma_{1\leq i\leq j} p_i.x_i \\ \text{avec } \Sigma_{1\leq i\leq j} w_i.x_i \leq y \\ \text{et pour tout } i,\ x_i \in \{0,1\}. \end{cases}$$

On note encore $f(j,y) = max\{\Sigma_{1\leq i\leq j} p_i.x_i \mid \Sigma_{1\leq i\leq j} w_i.x_i \leq y \wedge x_i \in \{0,1\}\}$.

L'idée est d'abord d'établir l'équation de récurrence définissant les solutions optimales de KP. On verra que d'après cette équation, le calcul de la solution optimale

pour $KP(n,c)$ nécessite le calcul et la mise en mémoire des solutions pour les instances $KP(k,y)$ avec $k \leq n$ et $y \leq c$.

Considérons une solution optimale $(X_1,...,X_j) \in \{0,1\}^j$ pour $KP(j,y)$, (i.e. $\Sigma_{1\leq i\leq j} p_i.X_i = f(j,y)$). Alors $(X_1,...,X_{j-1})$ est une solution optimale pour $KP(j-1, y - X_j.w_j)$.
*preuve:* Par l'absurde. Supposons qu'il existe une autre séquence $(Y_1,...,Y_{j-1}) \in \{0,1\}^{j-1}$ telle que

$$\begin{cases} \Sigma_{1\leq i\leq j-1} p_i.Y_i > \Sigma_{1\leq i\leq j-1} p_i.X_i \\ \text{et} \\ \Sigma_{1\leq i\leq j-1} w_i.Y_i \leq y - X_j.w_j. \end{cases}$$

Alors,

$$\begin{cases} \Sigma_{1\leq i\leq j-1} p_i.Y_i + p_j.X_j > \Sigma_{1\leq i\leq j} p_i.X_i \\ \text{et} \\ \Sigma_{1\leq i\leq j-1} w_i.Y_i + w_j.X_j \leq y, \end{cases}$$

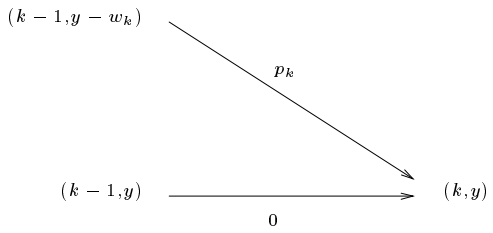
contredisant l'optimalité de $(X_1,...,X_j)$. ∎
De ce fait, pour tout $j$ tel que $2 \leq j \leq n$ et tout $y \in \mathbb{N}$:

$$f(j,y) = max\{f(j-1,y), f(j-1, y-w_j) + p_j)\}$$

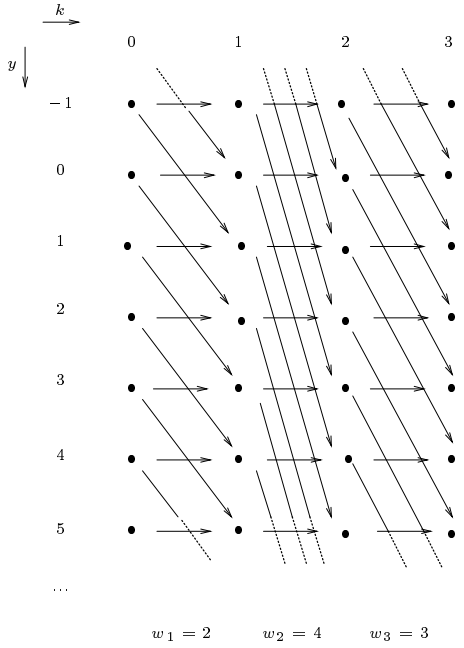
et

$$f(0,y) = 0.$$

étendons tout d'abord $f$ en posant $f(k,y) = -\infty$ pour $y < 0$. Considérons le graphe (simple) valué suivant. L'ensemble de ses sommets est $\{0,...,n\} \times \{y \in \mathbb{Z} \mid y \leq c\}$. Les arcs sont les couples de la forme $((k-1,y-w_k),(k,y))$ et $((k-1,y),(k,y))$. Les arcs $((k-1,y-w_k),(k,y))$ ont pour valeur $p_k$ et les arcs $((k-1,y),(k,y))$ ont pour valeur 0 - voir la figure ci-après.



Ci-dessous est représenté le graphe correspondant au cas : $n = 3$, $w_1 = 2$, $w_2 = 4$, $w_3 = 3$.




Le graphe précédent indique le fait que $f(k,y)$ est calculable en prenant le maximum des valeurs $f(k-1,y)$ et $f(k-1,y-w_k)+p_k$. Ainsi tout chemin d'origine en colonne 0, c'est à dire de la forme $(0,y)$ avec $0 \leq y \leq c$, et d'extrémité en colonne $n$, c'est à dire de la forme $(n,y)$ avec $0 \leq y \leq c$, correspond à une solution pour l'instance $KP(n,c)$. De plus, la valeur de ce chemin (i.e. la somme des coûts de ses arcs) est le coût de la solution associée. Une solution optimale pour $KP(n,c)$ correspond donc à un chemin de valeur maximale de la colonne 0 à la colonne $n$.

Au vu de l'équation de récurrence précédente, les valeurs $f(k,y)_{1 \leq y \leq c}$ peuvent être évaluées selon des indices $k$ croissants.
Afin de calculer une solution optimale de $KP(n,c)$, on calcule aussi et simultanément aux $f(k,y)$, les indices $u(k,y)$. Ceux-ci sont définis par induction :

$$\begin{cases} u(1,y) = 0 \text{ si } f(1,y) = 0, \\ u(1,y) = 1 \text{ sinon} \end{cases}$$

et si $k > 1$

$$\begin{cases} u(k,y) = k \text{ si } f(k-1,y) + p_k > f(k-1,y), \\ u(k,y) = u(k,y-1) \text{ sinon} \end{cases}$$

D'après cette définition, pour $(k,y) \in \{1,...,n\} \times \mathbb{Z}$, $u(k,y)$ est plus grand indice $j \leq k$ tel que $f(k,y) = f(j,y-w_j)+p_j$. Une solution optimale $(X_i)_{1 \leq i \leq k}$ à l'instance $KP(k,y)$ peut être déterminée au vu des valeurs de $u$ par récurrence :

$$\begin{cases} X_j = 0 \text{ si } u < j \leq k \\ X_{u(k,y)} = 1 \text{ si } u \leq 1 \\ (X_1,...,X_{u(k,y)-1}) \text{ est une solution optimale pour } KP(u(k,y)-1, y - w_u(k,y)). \end{cases}$$

L'algorithme découlant de la méthode résout donc $KP(n,c)$ en $O(nc)$.

Appliquons maintenant cette méthode de résolution à l'instance

$$\begin{cases} \max 20.x_1 + 16.x_2 + 11.x_3 + 9.x_4 + 7.x_5 + x_6 \\ \text{avec } 9.x_1 + 8.x_2 + 6.x_3 + 5.x_4 + 4.x_5 + x_6 \leq 12 \\ \text{et } \forall i, x_i \in \{0,1\}. \end{cases}$$

Les valeurs de $f$ et de $u$ sont indiquées dans le tableau suivant. Le contenu de la case colonne $k$, ligne $y$ étant $f(k,y)$, $u(k,y)$.

| $y$ \ $k$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 0 | 0,0 | 0,0 | 0,0 | 0,0 | 0,0 | 0,0 |
| 1 | 0,0 | 0,0 | 0,0 | 0,0 | 0,0 | 1,6 |
| 2 | 0,0 | 0,0 | 0,0 | 0,0 | 0,0 | 1,6 |
| 3 | 0,0 | 0,0 | 0,0 | 0,0 | 0,0 | 1,6 |
| 4 | 0,0 | 0,0 | 0,0 | 0,0 | 7,5 | 7,5 |
| 5 | 0,0 | 0,0 | 0,0 | 9,4 | 9,4 | 9,4 |
| 6 | 0,0 | 0,0 | 11,3 | 11,3 | 11,3 | 11,3 |
| 7 | 0,0 | 0,0 | 11,3 | 11,3 | 11,3 | 12,6 |
| 8 | 0,0 | 16,2 | 16,2 | 16,2 | 16,2 | 16,2 |
| 9 | 20,1 | 20,1 | 20,1 | 20,1 | 20,1 | 20,1 |
| 10 | 20,1 | 20,1 | 20,1 | 20,1 | 20,1 | 21,6 |
| 11 | 20,1 | 20,1 | 20,1 | 20,1 | 20,1 | 21,6 |
| 12 | 20,1 | 20,1 | 20,1 | 20,1 | 23,5 | 23,5 |

Une solution optimale est (0,1,0,0,1,0) pour un coût de 23 et un poids de 12. Le chemin correspondant est indiqué en pointillés dans le tableau.