Algorithmique

Prof. Amin Shokrollahi

Semestre d'Automne 2007-2008

Table des matières

0 Introduction

0.1 QU'EST CE QU'UN ALGORITHME ?

Informellement, on peut définir un algorithme comme suit :

> Un *algorithme* est une suite d'étapes dont le but est de résoudre un problème ou d'accomplir une tâche.

Les algorithmes sont utilisés partout. Par exemple, une recette pour votre plat préféré peut être un algorithme. Nous raffinerons le concept d'un algorithme en étudiant des exemples.

0.2 EXEMPLES D'ALGORITHMES

0.2.1 Somme d'entiers positifs

Problème: **Calculer la somme des n premiers entiers positifs**
Input: $n \in \mathbb{N} := \{1, 2, 3, \ldots\}$
Output: $S(n) := 1 + 2 + 3 + \ldots + n = \sum_{k=1}^{n} k$

Par exemple, si $n = 10$, l'algorithme retourne 55.

Une approche naïve serait de définir une variable s initialisée à zéro, puis d'y ajouter un par un les nombres de 1 à n. Le code correspondant s'écrit :

Algorithme 1 Méthode naïve de sommation

Input: Un entier positif $n \in \mathbb{N}$.
Output: La somme $s := \sum_{k=1}^{n} k$.

```
s ← 0
for i = 1 to n do
    s ← s + i
end for
return s
```

Une meilleure méthode a été découverte par le mathématicien allemand Carl Friedrich Gauß à l'âge de 5 ans. Son instituteur, pour le punir, lui demanda de calculer la somme des entiers de 1 à 100. Le jeune Gauß annonca le résultat très peu de temps après, car il avait réalisé qu'on pouvait ajouter les termes deux par deux en partant des côtés opposés de la liste, et que les termes intermédiaires ainsi obtenus étaient tous identiques : $1+100 = 101, 2+99 = 101, \ldots, 50+51 = 101$. On obtient ainsi 50 termes, le résultat est donc $50 \cdot 101 = 5050$.

Cette méthode peut être généralisée à tout n pair. On a :

$$\frac{n}{2} \text{ termes} \begin{cases} 1 & + & n & = & n+1 \\ 2 & + & n-1 & = & n+1 \\ 3 & + & n-2 & = & n+1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{n}{2}-1 & + & \frac{n}{2}+2 & = & n+1 \\ \frac{n}{2} & + & \frac{n}{2}+1 & = & n+1 \end{cases}$$

Donc,

$$S(n) = \frac{n(n+1)}{2}.$$

L'implémentation de la méthode de Gauß est triviale ; Il suffit d'écrire la formule ci-dessus :

Algorithme 2 Méthode de sommation de Gauß

Input: Un entier positif $n \in \mathbb{N}$.
Output: La somme $s := \sum_{k=1}^{n} k$.

```
return n(n+1)/2
```

Nous avons déclaré que la méthode de Gauß était meilleure que notre méthode originale. Mais que veut dire "meilleure" ? Nous pouvons donner un sens précis a ce terme : nous voulions dire qu'en utilisant la méthode de Gauß, la somme pouvait être calculée de façon plus efficace. Ceci peut être montré formellement en comparant le nombre d'opérations arithmétiques dont a besoin chaque algorithme pour parvenir au résultat.

La première méthode utilise n additions du compteur s. (Nous ignorons les additions de i). La deuxième méthode utilise 3 opérations arithmétiques (une addition, une multiplication et une division). La méthode de Gauß est donc plus performante pour $n > 3$.

Il nous manque toujours :
- Une démonstration que la formule de Gauß est juste. (Récurrence !)
- Une notion de performance et des méthodes pour analyser nos algorithmes.
- Une spécification formelle du problème.

0.2.2 Problème de Localisation

Problème: Problème de localisation
Input: Des entiers $a_0 < a_1 < \cdots < a_{n-1}$, et $x \in \mathbb{Z}$
Output: Trouver un indice i tel que $a_i = x$, si un tel i existe

Exemple: Supposons que les entiers soient $1, 3, 8, 11, 39, 41$; Si $x = 39$, l'output de l'algorithme serait 4, alors que si $x = 26$ il n'y aurait pas de réponse.

Voici une première solution au problème :

Algorithme 3 Recherche linéaire

```
i ← 0
while a_i ≠ x and i < n do
  i ← i + 1
end while
if i = n then
  return ERROR
else
  return i
end if
```

Quelle est l'efficacité de cette solution ? Comme avant, nous comptons le nombre d'opérations arithmétiques que doit réaliser l'algorithme. Nous ignorons de nouveau les opérations qui ne concernent que le compteur i. Les opérations restantes sont donc les comparaisons $a_i \neq x$. Cet algorithme effectue n comparaisons dans le pire des cas (ici le pire des cas se produit quand x n'est pas dans la liste ou quand x est le dernier élément, $x = a_{n-1}$).

Peut-on faire mieux ? Oui, car on peut utiliser le fait que les a_i sont ordonnés. C'est ce que fait l'algorithme suivant, appelé *Recherche Binaire* (*Binary Search*) :

Algorithme 4 Recherche binaire

```
i ← 0, j ← n
while j − i > 1 do
  ℓ ← ⌊(j−i)/2⌋ + i
  if x < a_ℓ then
    j ← ℓ
  else
    i ← ℓ
  end if
end while
if x ≠ a_i then
  return ERROR
else
  return i
end if
```

Supposons que k soit la solution recherchée, i.e., $a_k = x$. Par convention, nous posons $a_n = \infty$. Au début de chaque itération de la boucle **while**, on a

$$a_i \leq a_k < a_j. \tag{1}$$

En effet, avant la première itération, (1) est clairement vraie. Dans toutes les itérations qui suivent, nous modifions i et j de telle façon à ce qu'elle reste vraie. Puisque les a sont ordonnés, on déduit de (1) que

$$i \leq k < j.$$

A chaque itération l'intervalle $[i, j[$ est coupé en deux, puisque ℓ est choisi tel qu'il soit au milieu de i et j. On finira donc forcément par trouver k.

La recherche binaire utilise au plus $\lceil \log_2(n) \rceil$ comparaisons (pourquoi ?), et cet algorithme est donc beaucoup plus efficace que le premier algorithme. (Remarque : $\log_2(n)$ est *beaucoup plus petit* que n quand n est grand. Par exemple, si $n = 100$, $\log_2(n)$ vaut environ 6.64, et si $n = 1024$, $\log_2(n)$ vaut 10).

Il nous manque toujours :
- Outils de construction de l'algorithme
- Outils d'analyse de l'algorithme
- Structures de données (*Data structures*) pour réaliser l'algorithme

0.2.3 Monnaie Optimale

Problème: Monnaie optimale
Input: Une collection $C = \{1, 2, 5, 10, 20, 50, 100\}$ de valeurs de pièces de monnaie, et un entier x
Output: Une suite finie de pièces, contenant aussi peu de pièces que possible, dont la somme des valeurs vaut x

Exemple: Si $x = 325$, alors

$$c_0 = c_1 = c_2 = 100, c_3 = 20, c_4 = 5.$$

Nous pouvons donc obtenir le résultat demandé en utilisant 5 pièces.

Voici l'algorithme le plus simple qui vienne à l'esprit, appelé *Algorithme Glouton* (*Greedy Algorithm*) :

Algorithme 5 Algorithme Glouton

$v \leftarrow 0$ et $t \leftarrow 0$
while $v < x$ **do**
  Trouver le plus grand $c \in C$ tel que $c + v \leq x$
  $c_t \leftarrow c$
  $v \leftarrow v + c$
  $t \leftarrow t + 1$
end while
return $(c_0, c_1, \ldots, c_{t-1})$

Cet algorithme retourne-t-il une suite de pièces dont la somme des valeurs vaut x ? Si oui, le nombre de pièces est-il minimal ?

Il nous manque toujours :
- Une preuve que l'algorithme est juste
  - Spécification du problème
  - Analyse
  - Solution optimale
- Structures de données (*Data structures*)
- Principes de construction (*greedy algorithms*)

Le premier pas vers ces objectifs est d'avoir une manière rigoureuse de décrire ce que l'algorithme doit accomplir, c'est-à-dire une *spécification formelle* du problème. Ceci nécessite

quelques concepts mathématiques (ensembles, relations, fonctions, ...) que nous décrirons dans les prochaines sections.

0.3 THÉORIE DES ENSEMBLES

Nous supposons que la notion d'ensemble est connue. En général, nous noterons les ensembles par des lettres majuscules A, B, etc. Les mots "famille" et "collection" seront pour nous des synonymes de "ensemble".

0.3.1 Rappels de notations de logique

Rappelons les symboles suivants :

$\forall$	pour tout	$\exists$	il existe
$\wedge$	et	$\vee$	ou
$\Longrightarrow$	implique	$\in$	appartient à

Exemple: L'expression

$$\forall x \in A : x \neq y \implies x > y \vee x < y,$$

est lue comme suit :

> Pour tout x appartenant à A, si x est distinct de y, alors x est strictement plus grand que y ou x est strictement plus petit que y.

Rappelons aussi que les versions barrées $\nexists, \notin$, etc... dénotent les opposés des symboles correspondants (c'est-à-dire, "$x \notin A$" veut dire que "x n'appartient pas à A").

Remarque:
- Le symbole $\forall$ est un A tourné ; en anglais on dit "for **A**ll", en allemand "für **A**lle".
- Le symbole $\exists$ est un E tourné ; on dit "il **E**xiste".
- Le symbole $\in$ rappelle à un E, comme **É**lément ; $x \in A$ peut aussi être lu "x élément de A".

0.3.2 Rappels de notations de théorie des ensembles

Un ensemble A peut être spécifié de nombreuses façons ; si A est fini, contenant (exactement) les éléments $a_1, a_2, \ldots, a_n$, il suffit de les énumérer, i.e., d'écrire

$$A = \{a_1, \ldots, a_n\}.$$

Si A est le sous-ensemble d'un certain ensemble B, et si A est formé de tous les membres $x \in B$ vérifiant une certaine propriété $P(x)$, nous pouvons écrire

$$A = \{x \in B \mid P(x)\}.$$

Exemple: Si B est l'ensemble de tous les nombres premiers, alors nous pouvons écrire

$$\{2, 3, 5, 7, 11, 13, 17, 19\} = \{x \in B \mid x < 20\},$$

où l'égalité "=" ci-dessus signifie que l'ensemble à gauche est formé d'exactement les mêmes éléments que celui à droite. (Voir ci-dessous pour une définition formelle d'égalité entre ensembles.)

Rappelons les notations suivantes :

$A \subseteq B$	A est sous-ensemble de B.
$A \subset B$	A est sous-ensemble strict de B.
$A = B$	A et B sont égaux.

Comme avant, les versions barrées de ces symboles signifient leur opposé (i.e. $A \neq B$: A et B sont différents). Nous pouvons utiliser les notations de logique ci-dessus pour définir précisément ces symboles :

Définition. Pour des ensembles A et B, nous définissons $\subseteq, =, \subset$ entre ensembles comme suit :

- $A \subseteq B$ si et seulement si $\forall x : x \in A \implies x \in B$.
- $A = B$ si et seulement si $A \subseteq B$ et $B \subseteq A$.
- $A \subset B$ si et seulement si $A \subseteq B$ et $A \neq B$.

Nous disposons de quelques opérateurs sur les ensembles :

$A \cup B$	dénote l'union de A et B.
$A \cap B$	dénote l'intersection de A et B.
$A \setminus B$	dénote la différence de A et B.
$A \times B$	dénote le produit cartésien de A et de B.

Définition. L'union, l'intersection, la différence et le produit cartésien d'ensembles peuvent être définis comme suit :

- L'*union* de deux ensembles A et B est l'ensemble

$$A \cup B := \{x \mid x \in A \vee x \in B\}.$$

Plus généralement, si F est un ensemble de sous-ensembles d'un ensemble X, l'union de tous les membres de F est définie comme suit :

$$\bigcup_{A \in F} A := \{x \in X \mid \exists A \in F : x \in A\}$$

– L'*intersection* de deux ensembles A et B est l'ensemble

$$A \cap B := \{x \mid x \in A \wedge x \in B\}.$$

De manière plus générale, pour une famille F de sous-ensembles de X, l'intersection de tous les membres de X est l'ensemble

$$\bigcap_{A \in F} A := \{x \in X \mid \forall A \in F : x \in A\}.$$

– La *différence* de A et B est l'ensemble

$$A \setminus B := \{x \in A \mid x \notin B\}.$$

– La *différence symmétrique* de A et B est l'ensemble

$$A \Delta B = (A \setminus B) \cup (B \setminus A).$$

– Le *produit cartésien* $A \times B$ est l'ensemble de tous les couples (a, b), où $a \in A$ et $b \in B$. (Deux éléments $(a, b), (a', b') \in A \times B$ sont égaux si et seulement si $a = a'$ et $b = b'$). Pour des ensembles $A_1, \ldots, A_n$, le produit cartésien $A_1 \times \cdots \times A_n$ est l'ensemble de n-tuples

$$(a_1, \ldots, a_n) \text{ où } a_1 \in A_1, \ldots, a_n \in A_n.$$

Le n-ème produit cartésien de A est l'ensemble

$$A^n := \underbrace{A \times \cdots \times A}_{n \text{ fois}}.$$

Remarque: Les opérateurs $\cup$ et $\vee$, respectivement $\cap$ et $\wedge$, ne sont pas seulement superficiellement similaires. En fait, la définition de $\cup$ fait appel à $\vee$ et celle de $\cap$ à $\wedge$.

Définition. A tout ensemble A nous pouvons associer une quantité appelée sa *cardinalité* (ou simplement *taille*). Nous dénotons $|A|$ la cardinalité de A, définie comme étant

$$|A| := \begin{cases} n & \text{si } A \text{ contient exactement } n \text{ éléments,} \\ \infty & \text{si le nombre d'éléments de } A \text{ n'est pas fini.} \end{cases}$$

Pour dire que A est fini, nous écrivons souvent $|A| < \infty$. Il n'existe qu'un seul ensemble de cardinalité nulle, qui est l'*ensemble vide* $\emptyset$, caractérisé par le fait qu'aucun élément n'est contenu dans $\emptyset$.

Définition. Les ensembles A et B sont *disjoints* si

$$A \cap B = \emptyset,$$

i.e., si A et B n'ont aucun élément en commun.

Soit X un ensemble et F une famille de sous-ensembles disjoints de X. La *réunion disjointe* des membres de F est notée

$$\bigsqcup_{A \in F} A.$$

Il ne s'agit de rien autre que de l'union usuelle. Mais parfois, on parle de réunion disjointe et on écrit $\sqcup$ au lieu se $\cup$ pour souligner le fait que les ensembles sont disjoints. (Par exemple, de manière générale, on a

$$\left| \bigsqcup_{A \in F} A \right| = \sum_{A \in F} |A|,$$

ce qui n'est pas toujours vrai pour l'union usuelle.)

0.3.3 Ensembles des parties, Ensembles des mots

Définition. L'*ensemble des parties* d'un ensemble X est défini comme étant l'ensemble de tous les sous-ensembles de X. L'ensemble des parties de X est noté $\mathrm{Pot}(X)$ ou encore 2^X.

L'*ensemble des parties finies* de X est défini comme étant l'ensemble de tous les sous-ensembles finis de X. Il est noté $\mathrm{Pot}^*(X)$.

On voit que si X est fini, alors on a $\mathrm{Pot}(X) = \mathrm{Pot}^*(X)$.

Remarque: L'écriture 2^X est due au fait que si X est fini, on a $|2^X| = 2^{|X|}$. La notation $\mathrm{Pot}(X)$ vient de l'allemand : l'ensemble des parties s'appelle "die Potenzmenge" en allemand.

Exemple: Si $A = \{\mathrm{Ding}, 3, \mathrm{Brr}\}$, alors

$$\mathrm{Pot}(A) = \{\emptyset, \{\mathrm{Ding}\}, \{3\}, \{\mathrm{Brr}\}, \{\mathrm{Ding}, 3\}, \{\mathrm{Ding}, \mathrm{Brr}\}, \{3, \mathrm{Brr}\}, \{\mathrm{Ding}, 3, \mathrm{Brr}\}\}.$$

Définition. Soit A un ensemble. L'*ensemble des mots* sur A est l'ensemble

$$A^* := \bigcup_{n \geq 0} A^n.$$

L'*ensemble des mots non-vides* sur A est l'ensemble

$$A^+ := \bigcup_{n \geq 1} A^n.$$

On a donc $A^* = A^+ \cup \{\emptyset\}$.

Exemple: Si $A = \{a, b, c, d, \ldots, z\}$, alors A^* contient, entre autres, les mots

$$(t, o, u, r),\ (m, a, i, s, o, n),\ (p, k, z, r, f, a),\ \emptyset.$$

Tous ces mots sauf $\emptyset$ appartiennent aussi à A^+.

0.3.4 Ensembles standards

Les ensembles suivants seront souvent utilisés par la suite :

$\mathbb{N}$	les nombres naturels
$\mathbb{N}_0$	les entiers non négatifs
$\mathbb{Z}$	les entiers
$\mathbb{Q}$	les nombres rationnels
$\mathbb{R}$	les nombres réels
$\mathbb{R}_{\geq 0}$	les nombres réels non négatifs
$\mathbb{R}_{>0}$	les nombres réels positifs
$\mathbb{C}$	les nombres complexes

Rappelons que

$$\begin{array}{lcl} \mathbb{N} & = & \{1, 2, 3, \ldots.\}. \\ \mathbb{N}_0 & = & \{0, 1, 2, 3, \ldots.\}. \\ \mathbb{Z} & = & \{\ldots, -2, -1, 0, 1, 2, \ldots\}. \\ \mathbb{Q} & = & \{\frac{a}{b} \mid a, b \in \mathbb{Z}, b \neq 0\}. \end{array}$$

On a $\mathbb{N} \subset \mathbb{N}_0 \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$.

0.4 RELATIONS ET FONCTIONS

Dans toute cette partie, A et B seront des ensembles.

Définition. Une *relation entre A et B* est un sous ensemble $R \subseteq A \times B$.

Si $R \subseteq A \times A$, on dit que R est une *relation sur A* au lieu de "une relation entre A et A".

Exemple: $A = B = \mathbb{N}$,

$$R := \{(m, n) \mid m, n \in \mathbb{N},\ m \text{ divise } n\}$$

est la *relation de divisibilié*. On a donc $(3, 12), (4, 4) \in R$, mais $(5, 9) \notin R$.

Nous avons l'habitude de travailler avec des relations, mais en utilisant une notation différente. Par exemple "$\leq$" est une relation sur $\mathbb{Z}$, mais il est plus pratique d'écrire "$a \leq b$" que "$(a, b) \in R_{\leq}$", où $R_{\leq} := \{(m, n) \mid m, n \in \mathbb{Z}, m \leq n\}$.

Définition. La *relation inverse* de R est définie par

$$R^{-1} := \{(b, a) \in B \times A \mid (a, b) \in R\}.$$

Exemple: La relation inverse de "$\leq$" ($R_{\leq}$ dans l'exemple ci-dessus) est la relation *supérieur ou égal à* "$\geq$".

Définition. Une relation R est dite *symétrique* si $R = R^{-1}$.

Exemple: Si P est un ensemble d'hommes, alors "être le frère de" est une relation symétrique sur P : Si David est le frère de Georges, alors Georges est aussi le frère de David.

Définition. Une *fonction partielle* est une relation $F \subseteq A \times B$ qui vérifie

$$\forall a \in A: \quad |\{b \in B \mid (a, b) \in F\}| \leq 1. \tag{2}$$

Une *fonction* est une fonction partielle pour laquelle on a égalité dans (2).

Si F est une fonction et $(a, b) \in F$, alors on écrit $b = F(a)$, et on dit que F *envoie* a *sur* b, ou aussi que b est l'*image* de a par F.

On utilise parfois la notation $F\colon A \to B$ pour dire que F est une fonction de A vers B. L'ensemble de toutes les fonctions de A vers B est noté B^A.

On remarque que si F est une fonction $F : A \to B$, alors pour chaque $a \in A$, il existe exactement un $b \in B$ tel que $b = F(a)$. Nous avons l'habitude de travailler avec des fonctions, mais sans les considérer comme des relations. On voit une fonction plutôt comme une transformation à laquelle on donne un élément de A et qui nous retourne un élément de B.

Exemple: La relation

$$F = \{(a, b) \mid a, b \in \mathbb{R},\ b = a^2\}$$

est une fonction. La relation

$$R = \{(a, b) \mid a, b \in \mathbb{R},\ b^2 = a\}$$

n'est pas une fonction, car si $a < 0$, il n'y a pas de $b \in \mathbb{R}$ tel que $(a, b) \in R$. En fait, R n'est même pas une fonction partielle, car si $a > 0$, il y a *deux* valeurs de $b \in \mathbb{R}$ pour lesquelles $(a, b) \in R$. Cependant, la même relation définie entre $\mathbb{R}_{\geq 0}$ et $\mathbb{R}_{\geq 0}$ est une fonction : il est facile de vérifier que

$$R' = \{(a, b) \mid a, b \in \mathbb{R}_{\geq 0},\ b^2 = a\}$$

a pour chaque $a \in \mathbb{R}_{\geq 0}$ exactement une image $b \in \mathbb{R}_{\geq 0}$. R' est la fonction racine carrée.

Définition. Une fonction $F\colon A \to B$ est dite *surjective* si pour tout $b \in B$, il existe $a \in A$ tel que $b = F(a)$.

Exemple: La fonction $F = \{(a, b) \mid a, b \in \mathbb{R}, b = 2a + 1\}$ est surjective, car pour tout $b \in \mathbb{R}$, il existe un $a \in \mathbb{R}$ dont l'image par F est b, à savoir $a = (b - 1)/2$. Par contre, $G = \{(a, b) \mid a, b \in \mathbb{R}, b = a^4 + 1\}$ n'est pas surjective. En effet, pour $b = -1$ il n'y a pas de $a \in \mathbb{R}$ tel que $F(a) = b$.

Définition. Une fonction F est dite *injective* si $F(a) = F(a')$ implique que $a = a'$.

On remarque qu'une définition équivalente de l'injectivité de F est $a \neq a' \implies F(a) \neq F(a')$.

Exemple: La fonction $F = \{(a, b) \mid a, b \in \mathbb{R},\ b = a^3\}$ est injective. En effet, si $a^3 = b^3$ alors on a $a = b$. Par contre, $G = \{(a, b) \mid a, b \in \mathbb{R}, b = a^2\}$ n'est pas injective car $G(1) = 1 = G(-1)$.

Si $A = \{a_1, \ldots, a_m\}$ est fini, alors il peut être pratique d'exprimer $F : A \to B$ sous la forme d'un vecteur. On écrit

$$(F(a_1), \ldots, F(a_m)).$$

0.5 SPÉCIFICATION D'UN PROBLÈME

Définition. Un *problème computationnel* est spécifié de façon abstraite par un triplet d'ensembles

$$P = (I, O, R)$$

où I et O sont des ensembles non vides, et $R \subseteq I \times O$ est une relation. On appelle I l'*ensemble des inputs*, O l'*ensemble des outputs*, et R la *dépendance relationnelle* (*relational dependency*).

Exemple: Notre premier exemple, le calcul de la somme des n premiers entiers positifs, peut être spécifié comme suit :

$$\begin{aligned} I &= \mathbb{N} \\ O &= \mathbb{N} \\ R &= \left\{ (n, m) \;\middle|\; m = \sum_{k=1}^{n} k \right\} \subseteq I \times O. \end{aligned}$$

Exemple: Le problème de Monnaie Optimale a la spécification suivante :

$$\begin{aligned} I &= \mathrm{Pot}^*(\mathbb{N}) \times \mathbb{N} \\ O &= \mathbb{N}^+ \\ R &= \Bigg\{ ((a_1, \ldots, a_n; x), (c_1, \ldots, c_m)) \in I \times O \;\Bigg|\; \\ &\qquad \forall\, i\colon\ c_i \in \{a_1, \ldots, a_n\} \wedge \sum_{i=1}^{m} c_i = x \\ &\qquad \wedge \forall (d_1, \ldots, d_{m'}) \in \{a_1, \ldots, a_n\}^+ : \\ &\qquad\qquad \sum_{i=1}^{m'} d_i = x \implies m' \geq m \Bigg\} \end{aligned}$$

Rappelons que $\mathrm{Pot}^*(\mathbb{N})$ est l'ensemble des sous-ensembles finis de $\mathbb{N}$, et $\mathbb{N}^+$ est l'ensemble des mots non vides en $\mathbb{N}$. (Voir 0.3)

L'input est constitué d'un ensemble de valeurs de pièces de monnaie $A = \{a_1, \ldots, a_n\} \in \mathrm{Pot}^*(\mathbb{N})$, et d'un nombre naturel $x \in \mathbb{N}$ pour lequel nous voulons obtenir la monnaie optimale.

L'output est une suite $(c_1, \ldots, c_m) \in A^m$ de valeurs de pièces dont la somme vaut x, et telle que m est minimal parmi toutes les suites qui ont cette propriété.

Une spécification exacte d'un problème est déjà un pas majeur vers la construction et l'analyse d'un algorithme pour ce problème. En pratique, une spécification exacte d'un problème est souvent plus difficile à trouver qu'elle ne le paraît.

0.6 DU PROBLÈME À L'ALGORITHME ET SON IMPLÉMENTATION

Les étapes typiques dans la construction d'algorithmes et dans leur implémentation pratique sont les suivantes :

1. **Spécification formelle du problème**. Souvent, le problème nous est donné en prose (français, anglais, allemand, ...) et la tâche est de trouver une spécification exacte. Quel est l'ensemble des inputs possibles, quel est l'ensemble des outputs possibles, et quelle est la dépendance relationnelle entre les deux ?
2. **Séparation du problème en morceaux plus petits et plus gérables.** Souvent le problème est trop grand pour être attaqué d'un seul coup. Il doit être séparé en sous-problèmes plus petits, qui doivent être eux-mêmes spécifiés. De plus, un méchanisme doit être trouvé pour "recoller" les solutions des sous-problèmes afin d'obtenir une solution du problème original.
3. **Construction d'algorithmes.** Pour chaque sous-problème, un algorithme doit être construit pour le résoudre. Souvent en pratique l'une des nombreuses techniques de construction d'algorithmes peut être utilisée à cette fin. Cependant, dans certains cas, de nouveaux algorithmes doivent être constitués. La construction doit être motivée par des contraintes d'efficacité, et donc des méthodes d'analyse d'algorithmes doivent être utilisées. De plus, tous les algorithmes doivent être vérifiés et prouvés.
4. **Construction de structures de données.** Les structures de données sont le lien entre les algorithmes théoriques et leur implémentation pratique. Elles sont cruciales pour la performance d'un algorithme dans l'implémentation.
5. **Implémentation.** Au bout de la route on trouve l'implémentation. Celle-ci se classe dans le domaine du génie logiciel, qui ne nous concernera pas dans ce cours.

1
Induction

1.1 CE QUE C'EST ET À QUOI ÇA SERT

L'induction mathématique est une méthode pour prouver des affirmations concernant des ensembles dénombrables. Dans la plupart des cas, cet ensemble sera l'ensemble des nombres naturels $\mathbb{N}$.

Soit $A(n)$ une affirmation pour l'entier n.

Exemple: Si $A(n)$ déclare que "$n^2 \geq 2^n$", alors $A(3)$ dit par exemple que "$9 \geq 8$", ce qui est vrai, donc $A(3)$ est vraie. D'autre part, on peut trivialement vérifier que $A(5)$ n'est pas vraie. D'autres exemples d'affirmations $A(n)$ sont "n est pair", ou "n est un produit de nombres premiers".

Si $A(n)$ est vraie pour tout $n \in \mathbb{N}$, alors l'induction mathématique (aussi appelée *récurrence*) est souvent une façon élégante de le prouver.

1.2 TECHNIQUES D'INDUCTION

1.2.1 Induction simple

Théorème 1.1 *Une affirmation $A(n)$ est vraie pour tout $n \in \mathbb{N}$ si et seulement si les deux conditions suivantes sont vraies :*

(1) $A(1)$

(2) Pour tout $n \geq 1$ *:* $A(n) \implies A(n+1)$.

Définition. La technique d'*induction simple* consiste à montrer que $A(n)$ est vraie pour tout $n \in \mathbb{N}$ en utilisant le théorème précédent. Le point (1) s'appelle la *base d'induction* et le point (2) le *pas d'induction*. Dans les preuves, nous marquerons les étapes correspondantes par [**Base**] et [**Pas**] respectivement.

Pour effectuer le pas d'induction, nous supposons que $A(n)$ est vraie et montrons alors que $A(n+1)$ est aussi vraie. L'hypothèse $A(n)$ s'appelle alors l'*hypothèse d'induction*.

Preuve du théorème 1.1 : La partie "seulement si" est claire. Maintenant, si (1) et (2) sont les deux vraies, alors pour tout $n \in \mathbb{N}$, on a

$$\begin{array}{rcll} A(1), & & & \\ A(1) & \Longrightarrow & A(2), & \text{donc, } A(2) \text{ est vraie,} \\ A(2) & \Longrightarrow & A(3), & \text{donc, } A(3) \text{ est vraie,} \\ \vdots & & \vdots & \vdots \\ A(n-1) & \Longrightarrow & A(n), & \text{donc, } A(n) \text{ est vraie.} \end{array}$$

■

Remarque: Il n'est pas nécessaire de commencer l'induction avec $n = 1$. En utilisant la même technique, on peut prouver que $A(n)$ est vraie pour tout $n \geq k$ (où $k \in \mathbb{Z}$) en démontrant $A(k)$ au lieu de $A(1)$ comme base.

Pour illustrer l'utilité de l'induction simple, nous montrons maintenant que la formule de sommation de Gauß est correcte.

Exemple: Soit $S(n) = \sum_{k=1}^{n} k$. Montrer que $S(n) = \frac{1}{2}n(n+1)$.

Soit $A(n)$ l'assertion "$S(n) = \frac{1}{2}n(n+1)$". Nous allons montrer pas récurrence que $A(n)$ est vraie pour tout $n \in \mathbb{N}$.

[**Base**] $S(1) = 1$, donc $A(1)$ est vraie. ✓

[**Pas**] Supposons que $A(n)$ est vraie pour un certain n, c'est à dire supposons qu'on a $S(n) = \frac{1}{2}n(n+1)$. Nous voulons montrer que $A(n+1)$ est vraie :

$$\begin{array}{rcl} S(n+1) & = & S(n) + (n+1) \\ & \overset{A(n)}{=} & \dfrac{1}{2}n(n+1) + (n+1) \\ & = & \dfrac{1}{2}(n+1)(n+2). \end{array}$$

on a donc $A(n) \implies A(n+1)$, et donc par induction, la formule de Gauß est vraie pour tout $n \in \mathbb{N}$.

La base d'induction est généralement assez facile à établir ; le calcul trivial dans l'exemple ci-dessus est relativement typique.

Remarquons aussi comment le pas d'induction a été prouvé : nous écrivons d'abord $S(n+1)$ en fonction de $S(n)$ et n, et utilisons ensuite l'hypothèse d'induction $S(n)$. L'endroit où est utilisée l'hypothèse d'induction est indiqué par la notation $\overset{A(n)}{=}$.

Le procédé par récurrence est souvent utilisé pour montrer des assertions concernant le comportement de suites. Nous arrivons parfois à deviner une formule fermée sans disposer d'une preuve directe de celle-ci. Dans de tels cas, il est parfois possible de prouver la formule par récurrence. L'exemple suivant illustre cette technique.

Informellement, une *formule fermée* pour une suite $(u_0, u_1, \ldots)$ est une formule qui nous donne u_n en fonction de n sans utiliser de récurrence sur n ni de termes à indices qui dépendent de n (comme $\sum_{i=0}^{n}$ ou $\prod_{i=0}^{n}$).

Par exemple $u_n = 2^n$ et $u_n = n + \log n$ sont des formules fermées, alors que $u_n = \sum_{i=1}^{n} i^2$ ou $u_n = u_{n-1} + u_{n-2}$ ne sont pas des formules fermées.

Exemple: Trouver une formule fermée pour la suite $B(n) = \sum_{k=0}^{n} \binom{n}{k}$.

Rappel : Le coefficient binomial $\binom{n}{k}$ est défini comme suit :

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}.$$

Nous commençons par deviner une formule fermée de $B(n)$. Pour ce faire, nous regardons ce qui se passe quand n est petit :

					1						$B(0) = 1$
				1		1					$B(1) = 2$
			1		2		1				$B(2) = 4$
		1		3		3		1			$B(3) = 8$
	1		4		6		4		1		$B(4) = 16$
1		5		10		10		5		1	$B(5) = 32$

Nous devinons donc que $B(n) = 2^n$. Il faut le prouver, et pour ce faire, nous utilisons l'induction. Dans la preuve, nous aurons besoin de l'identité suivante :

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

Affirmation : Soit $A(n)$ l'affirmation

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n.$$

Nous aimerions montrer que $A(n)$ est vraie pour tout $n \geq 1$.

[**Base**] $n = 1 : \sum_{k=0}^{1} \binom{n}{k} = 1 + 1 = 2 = 2^1$.

[**Pas**]

$$\begin{array}{rcccccccc}
\binom{n+1}{0} & = & \binom{n}{0} & & & & & & \\
+ & & + & & & & & & \\
\binom{n+1}{1} & = & \binom{n}{0} & + & \binom{n}{1} & & & & \\
+ & & & & + & & & & \\
\binom{n+1}{2} & = & & & \binom{n}{1} & + & \binom{n}{2} & & \\
+ & & & & & & + & & \\
\vdots & & & & & & \vdots & & \\
+ & & & & & & & + & \\
\binom{n+1}{n} & = & & & & & & \binom{n}{n-1} & + \quad \binom{n}{n} \\
+ & & & & & & & & + \\
\binom{n+1}{n+1} & = & & & & & & & \binom{n}{n} \\
\hline
B(n+1) & = & & & 2 \cdot B(n) & & & &
\end{array}$$

Preuve en langage mathématique :

$$\begin{aligned}
B(n+1) &= \sum_{k=0}^{n+1} \binom{n+1}{k} \\
&= \underbrace{\binom{n+1}{0}}_{=1=\binom{n}{n}} + \sum_{k=1}^{n} \binom{n+1}{k} + \underbrace{\binom{n+1}{n+1}}_{=1=\binom{n}{0}} \\
&= \binom{n}{n} + \sum_{k=1}^{n} \left(\binom{n}{k-1} + \binom{n}{k} \right) + \binom{n}{0} \\
&= \binom{n}{n} + \sum_{k=0}^{n-1} \binom{n}{k} + \sum_{k=1}^{n} \binom{n}{k} + \binom{n}{0} \\
&= \underbrace{\sum_{k=0}^{n} \binom{n}{k}}_{=B(n)} + \underbrace{\sum_{k=0}^{n} \binom{n}{k}}_{=B(n)} \\
&= 2 \cdot B(n) \\
&\overset{A(n)}{=} 2^{n+1}.
\end{aligned}$$

■

1.2.2 Induction descendante

Une autre variante d'induction est l'*induction descendante*, décrite dans le théorème 1.2. Nous l'utiliserons plus tard pour prouver l'inégalité entre moyenne arithmétique et moyenne géométrique (*arithmetic-geometric-mean, AGM*).

Théorème 1.2 *Une affirmation $A(n)$ est vraie pour tout $n \in \mathbb{N}$ si et seulement si*

1. *$A(n)$ est vraie pour une infinité de n, et*
2. *On a $A(n) \implies A(n-1)$ pour tout $n \geq 2$.*

Intuitivement, ce theorème est vrai par le raisonnement suivant : Fixons un n pour lequel nous aimerions établir $A(n)$. Choisissons ensuite un $u \geq n$ tel que $A(u)$ soit vraie (ce qui est toujours possible, car nous savons que A est vraie sur un ensemble infini, par (1)). Nous itérons ensuite en utilisant (2) : $A(u) \implies A(u-1), A(u-1) \implies A(u-2), \ldots, A(n+1) \implies A(n)$. Nous prouvons maintenant ce théorème formellement :

Preuve. La partie "seulement si" est claire. Nous regardons donc la partie "si". Supposons que les points (1) et (2) soient vrais, et montrons que $A(n)$ est vraie pour tout $n \in \mathbb{N}$. Écrivons $U = \{u_1, u_2, \ldots\}$ (avec $u_1 < u_2 < \cdots$) l'ensemble infini sur lequel A est vraie. Nous allons montrer les deux affirmations suivantes :

(a) Pour tout $u \in U$ et tout $0 \leq k < u$, $A(u-k)$ est vraie.

(b) Pour tout $n \in \mathbb{N}$, il existe $u \in U$ et $k \in \mathbb{N}_0$ tel que $n = u - k$.

Ces deux affirmations réunies impliquent que $A(n)$ est vraie pour tout n. Nous utilisons l'induction simple pour montrer (a) et (b).

Preuve de (a). On fixe $u \in U$ et on procède par induction sur k.
[**Base**] $k = 0$. $A(u-k) = A(u)$, ce qui est vrai. ✓

[**Pas**] $A(u-k) \overset{(2)}{\implies} A(u-(k+1))$.

Preuve de (b)
[**Base**] $n = 1$: Posons $u = u_1$, $k = u_1 - 1$:

$$n = 1 = u - k = u_1 - (u_1 - 1). \quad \checkmark$$

[**Pas**] Hypothèse d'induction : $n = u_m - k$ pour un $m \geq 1$ et un $0 \leq k < u_m$.

Alors

$$n + 1 = \underbrace{u_{m+1}}_{\in U} - \underbrace{(u_{m+1} - u_m - 1 + k)}_{\geq 0 \text{ et } < u_{m+1}}.$$

■

Exemple : Moyenne arithmétique et moyenne géométrique (AGM) :

Théorème 1.3 *Pour des réels arbitraires positifs $x_1, \ldots, x_n$ on a*

$$A(n): \quad \underbrace{(x_1 \cdots x_n)^{1/n}}_{\textit{moy. géométrique}} \leq \underbrace{\frac{x_1 + \cdots + x_n}{n}}_{\textit{moy. arithmétique}}$$

Preuve. (Due à Cauchy, utilisant l'induction descendante)

Nous démontrons d'abord que $A(2^k)$ est vraie pour tout $k \geq 1$, ce qui montre que $A(n)$ est vérifiée pour une infinité de n. Ensuite, nous prouverons que si $A(n+1)$ est vraie, alors $A(n)$ l'est aussi.

Fait 1 : $A(2^k)$ est vraie pour tout $k \geq 1$

Preuve : On procède par induction sur k.

[**Base**] $k = 1$,

$$\begin{array}{rcl} & A(2) & \\ & \Updownarrow & \\ \sqrt{x_1 x_2} & \leq & \dfrac{x_1 + x_2}{2} \\ & \Updownarrow & \\ 4x_1x_2 & \leq & x_1^2 + 2x_1x_2 + x_2^2 \\ & \Updownarrow & \\ 0 & \leq & (x_1 - x_2)^2. \end{array}$$

La dernière ligne est clairement vraie, donc comme toutes les inégalites ci-dessus sont équivalentes, on en déduit que $A(2)$ doit être vraie.

[**Pas**] Soit $n = 2^k$. Supposons que $A(n)$ est vraie. Nous devons montrer qu'alors $A(2^{k+1})$ est vraie, i.e., que $A(2n)$ est vraie :

$$\begin{array}{rcl} (x_1 \cdots x_{2n})^{\frac{1}{2n}} & = & \sqrt{(x_1 \cdots x_n)^{\frac{1}{n}} \cdot (x_{n+1} \cdots x_{2n})^{\frac{1}{n}}} \\ & \overset{A(2)}{\leq} & \dfrac{(x_1 \cdots x_n)^{\frac{1}{n}} + (x_{n+1} \cdots x_{2n})^{\frac{1}{n}}}{2} \\ & \overset{A(n)}{\leq} & \dfrac{\dfrac{x_1 + \cdots + x_n}{n} + \dfrac{x_{n+1} + \cdots + x_{2n}}{n}}{2} \\ & = & \dfrac{x_1 + \cdots + x_{2n}}{2n}. \end{array}$$

Donc par induction, $A(2^k)$ est vraie pour tout $k \geq 1$.

Fait 2 : $A(n+1) \implies A(n)$

Preuve : Soit $x_1, \ldots x_n \in \mathbb{R}_{>0}$ donné. Posons

$$z \quad := \quad \frac{x_1 + \ldots + x_n}{n}. \tag{1.1}$$

Alors $z \in \mathbb{R}_{>0}$, et :

$$\begin{array}{rcl} (x_1 \cdots x_n \cdot z)^{\frac{1}{n+1}} & \overset{A(n+1)}{\leq} & \dfrac{x_1 + \cdots + x_n + z}{n+1} \\ & \overset{(1.1)}{=} & \dfrac{n \cdot z + z}{n+1} \;=\; z. \\ & \Updownarrow & \\ x_1 \cdots x_n \cdot z & \leq & z^{n+1} \\ & \Updownarrow & \\ (x_1 \cdots x_n)^{1/n} & \leq & z \;=\; \dfrac{x_1 + \cdots + x_n}{n}. \end{array}$$

Le fait que $A(n)$ est vraie pour tout $n \in \mathbb{N}$ suit des faits 1 et 2 par induction descendante (théorème 1.2). ■

1.2.3 Induction Forte

Rappelons que pour l'induction simple, le pas d'induction consistait à montrer que

$$A(n) \implies A(n+1).$$

Nous verrons maintenant qu'en fait, l'induction se fait toujours si l'on utilise l'hypothèse d'induction plus forte $A(1) \wedge \cdots \wedge A(n)$ pour montrer $A(n+1)$. Cette variante d'induction s'appelle l'*induction forte*.

Nous illustrerons l'utilisation de l'induction forte dans le paragraphe 1.3.1 en montrant la formule d'Euler pour les arbres.

Théorème 1.4 *L'affirmation $A(n)$ est vraie pour tout $n \in \mathbb{N}$ si et seulement si*
(1) $A(1)$ est vraie,
(2) pour tout $n \in \mathbb{N}$, $A(1) \wedge \cdots \wedge A(n) \implies A(n+1)$.

La preuve est très similaire à celle du théorème 1.1. Supposons que (1) et (2) sont les deux vraies, alors pour n'importe quel $n \in \mathbb{N}$ on a

$$
\begin{array}{rcll}
A(1) & & & \text{par hypothèse.} \\
A(1) & \Longrightarrow & A(2) & \text{Donc, } A(2) \text{ est vraie, et ainsi aussi } A(1) \wedge A(2). \\
A(1) \wedge A(2) & \Longrightarrow & A(3) & \text{Donc, } A(3) \text{ est vraie, et ainsi aussi } A(1) \wedge A(2) \wedge A(3). \\
\vdots & & \vdots & \\
A(1) \wedge \cdots \wedge A(n-1) & \Longrightarrow & A(n) &
\end{array}
$$

Passons maintenant aux exemples.

1.3 EXEMPLES

1.3.1 La formule d'Euler pour les arbres

Nous commençons par introduire quelques notions de base de théorie des graphes.

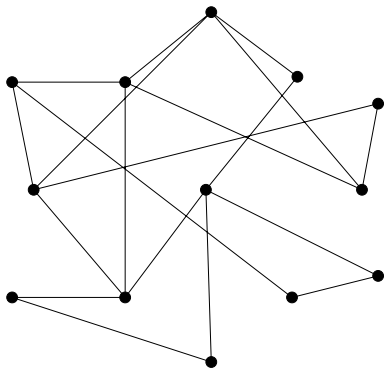
Définition. Un *graphe* est un ensemble fini V de *sommets* (*vertices*) muni d'une relation $E \subseteq V \times V$, appelée l'ensemble d'*arêtes* (*edges*).

Nous représentons les éléments de V par des nœuds, et les arêtes par des segments orienté. On a donc un segment orienté entre a et b si et seulement si $(a, b) \in E$. Dans ce cas nous dirons que a est *relié* (*joined*) à b.

Définition. Le graphe est *non-orienté* (*undirected*) si E est symmétrique.

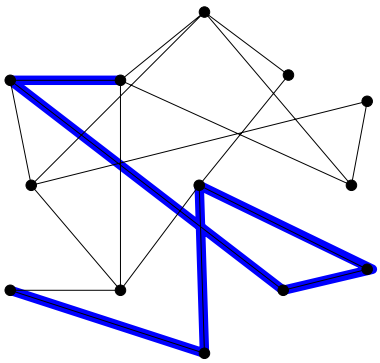
Dans le cas d'un graphe non-orienté, un sommet a est lié à un sommet b si et seulement si b est lié à a. Il est donc commun de représenter un tel graph avec un unique segment (sans direction) liant a et b, sans les doubles flèches liant a à b et b à a.

Exemple:

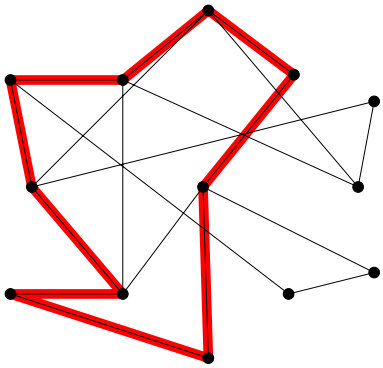


Un graphe non-orienté

Définition. Une *chaîne* (*path*) dans un graphe est une suite de sommets distincts qui sont reliés par des arêtes. Un *cycle* dans le graphe est une chaîne fermée.

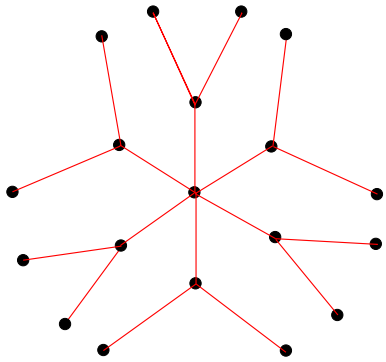


Une chaîne



Un cycle

Définition. Un graphe est dit *acyclique* (*acyclic*) s'il ne continent pas de cycles. On dit que le graphe est *connexe* (*connected*) si tout paire de sommets est reliée par une chaîne.



Un graphe acyclique et connexe

Définition. Un graphe acyclique et connexe s'appelle un *arbre* (*tree*).

Théorème 1.5 (La formule d'Euler pour les arbres) *Si T est un arbre de n sommets, alors T possède $n-1$ arêtes.*

Preuve. Nous procedons par induction forte sur le nombre de sommets n. Soit $A(n)$ l'affirmation "si T est un arbre sur n sommets, alors T a $n-1$ arêtes".

[**Base**] $n = 1$: Si l'arbre T n'a qu'un seul sommet alors il ne peut avoir aucune arête. Donc $A(1)$ est vraie. ✓

[**Pas**] Supposons $A(k)$ vraie pour tout $1 \leq k \leq n$. Soit T un arbre sur $n+1$ sommets. Choisissons une arête e dans T et enlevons-la. Comme T est connexe et acyclique, enlever e transforme T en deux sous-arbres de n_1 et n_2 sommets respectivement, avec les $n_i < n+1$ et $n_1 + n_2 = n+1$.

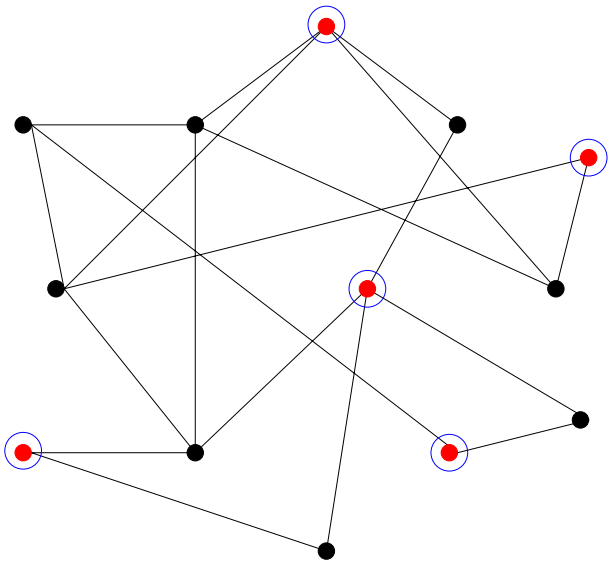
L'hypothèse d'induction permet maintenant de conclure que ces sous-arbres ont respectivement $n_1 - 1$ et $n_2 - 1$ arêtes. Et le nombre total d'arêtes de T est donc

$$\overbrace{n_1 - 1}^{1^{\text{er}} \text{ sous-arbre}} + \overbrace{n_2 - 1}^{2^{\text{ème}} \text{ sous-arbre}} + \overbrace{1}^{\text{Arête enlevée } e} = n.$$

Donc $A(n+1)$ est vraie. Le résultat suit par induction forte. ■

1.3.2 Ensemble Indépendant de l'Hypercube

Soit G un graphe non-orienté d'ensemble de sommets V et d'ensemble d'arêtes $E \subseteq V \times V$. Un *ensemble indépendant* de G est un ensemble de sommets tel qu'aucune paire de sommets n'est reliée.

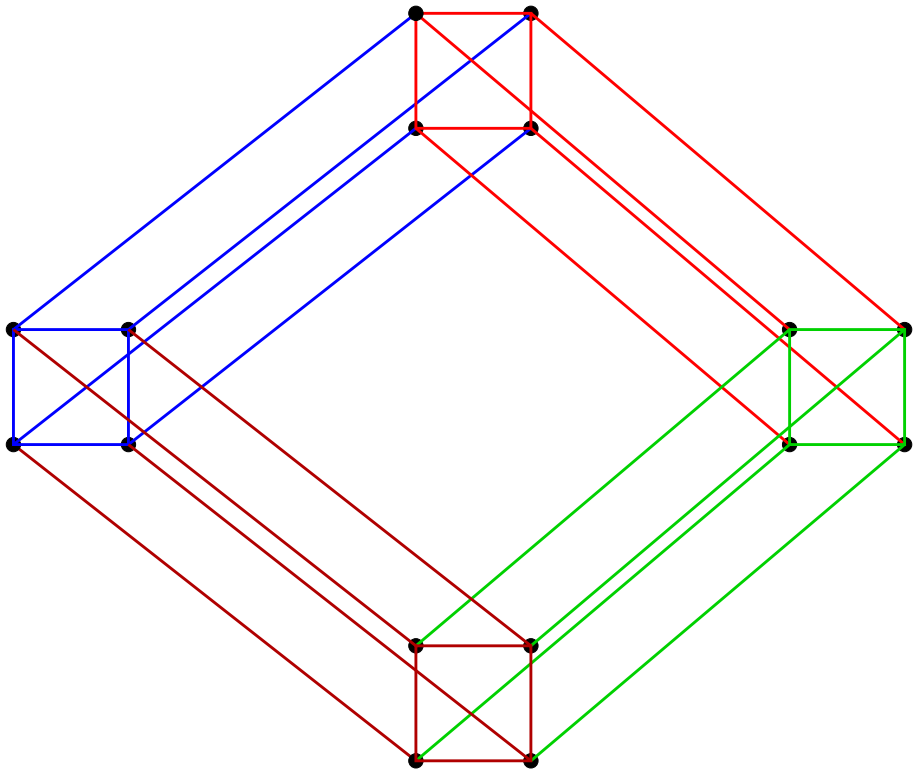


Un ensemble indépendant

Le n-hypercube est le graphe d'ensemble de sommets $\{0,1\}^n$ dans lequel deux sommets sont reliés si et seulement si les vecteurs correspondants diffèrent en exactement une coordonnée. (p.e. $(0,0,1)$ est relié à $(0,0,0)$ mais pas à $(1,1,1)$).

Exemple: Soit $n = 2$. L'ensemble de sommets est $V = \{0,1\}^2 = \{(0,0),(0,1),(1,0),(1,1)\}$. L'ensemble d'arêtes est $E = \{[(0,0),(0,1)],[(0,1),(1,1)],[(1,1),(1,0)],[(1,0),(0,0)]\}$. Si nous dessinons ces sommets dans un espace euclidien de dimension deux, nous voyons que le 2-hypercube est un carré.

Similairement, le 3-hypercube est le cube de dimension trois, et le n-hypercube nous dit à quoi ressemblerait un cube si nous vivions dans un espace de dimension n.



Une représentation du 4-hypercube

Donc, un ensemble indépendant dans l'hypercube est un sous-ensemble $S \subseteq \{0,1\}^n$ pour lequel deux vecteur distincts $x, x' \in S$ diffèrent en au moins deux positions.

Nous pouvons encore utiliser l'induction simple pour prouver le fait suivant concernant les n-hypercubes :

Proposition 1 *Le n-hypercube possède un ensemble indépendant de taille 2^{n-1}.*

Preuve. Nous prouverons même une affirmation plus forte : Le n-hypercube possède un ensemble indépendant de taille 2^{n-1} dont le complément dans $\{0,1\}^n$ est aussi indépendant.

[**Base**] $n = 2$. $V = \{(0,0),(0,1),(1,0),(1,1)\}$.

L'ensemble $\{(0,0),(1,1)\}$ est un ensemble indépendant de taille $2 = 2^{2-1}$ et son complément $\{(0,1),(1,0)\}$ en est aussi un. ✓

[**Pas**] Soit $V \subset \{0,1\}^n$ un ensemble indépendant de taille 2^{n-1}, et soit W son complément (aussi de taille 2^{n-1}.)

Soit $U = \{(x, 0) \mid x \in V\} \sqcup \{(y, 1) \mid y \in W\} \subset \{0, 1\}^{n+1}$. (Rappel : $\sqcup$ dénote l'union disjointe.)

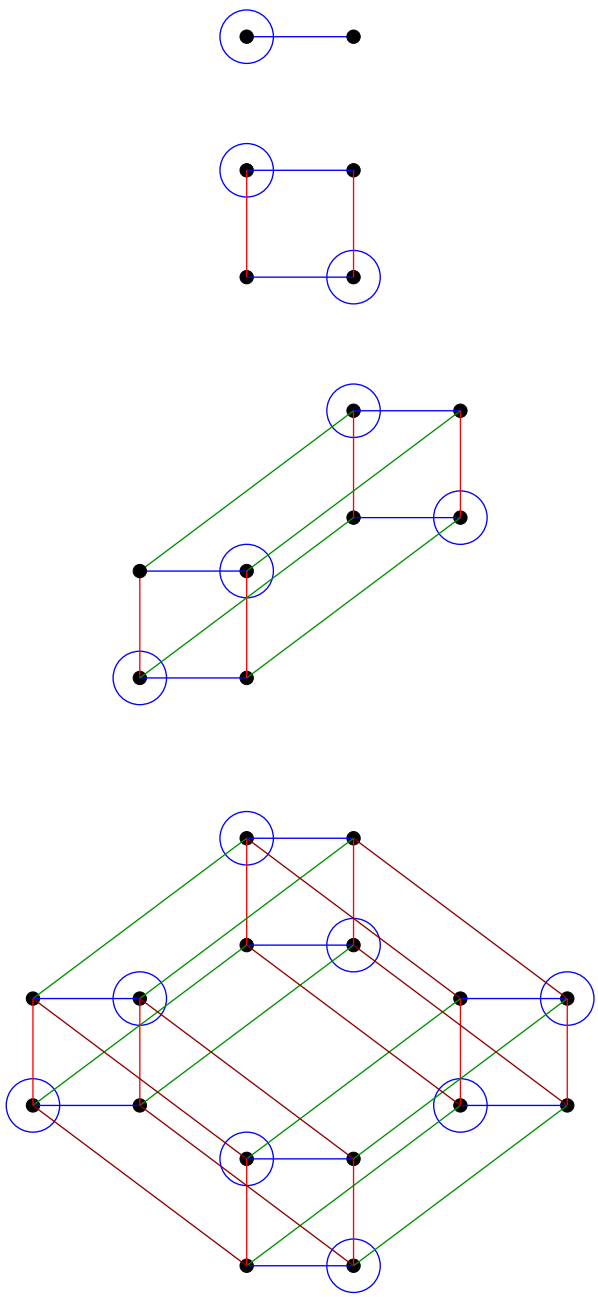
Nous montrons que U est un ensemble convenable. Remarquons d'abord que clairement $|U| = |V| + |W| = 2^n$. Il faut maintenant montrer que deux points distincts de U diffèrent en au moins deux coordonées. Pour deux points de la forme $(x, 0)$ et $(y, 1)$, il suffit de voir que $x \neq y$. Ceci est vrai, parce que $x \in V$ et $y \in W$ et que V et W sont disjoints. Les points de la forme $(x, 0)$ et $(x', 0)$ diffèrent en au moins deux coordonnées par hypothèse d'induction, comme $x, x' \in V$, où V est un ensemble indépendant. Il en va de même pour $(y, 1)$ et $(y', 1)$.

Finalement, pour completer le pas d'induction nous devons montrer que le complément de U est aussi un ensemble indépendant. Puisque $U = \{(x, 0) \mid x \in V\} \sqcup \{(y, 1) \mid y \in W\}$, son complément va consister en

- les elements de la forme $(x, 0)$ où $x \in \{0, 1\}^n$ et $x \notin V$, c'est à dire $(x, 0)$ où $x \in W$ puisque W est le complément de V dans $\{0, 1\}^n$.
- les elements de la forme $(y, 0)$ où $y \in \{0, 1\}^n$ et $x \notin W$, c'est à dire $(y, 0)$ où $y \in V$ puisque V est le complément de W dans $\{0, 1\}^n$.
  Le complément de U dans $\{0, 1\}^n$ est donc égal à $\overline{U}\{(x, 0) \mid x \in W\} \sqcup \{(y, 1) \mid y \in V\}$. On peut donc utiliser la même méthode qu'au dessus pour montrer que $\overline{U}$ est indépendant.

■

Voici quelques exemples d'ensembles indépendants du n-hypercube pour $n = 1, 2, 3, 4$:



2
Analyse d'Algorithmes

2.1 POURQUOI ANALYSER DES ALGORITHMES ?

L'analyse d'algorithmes est essentielle dans la comparaison de différents algorithmes pour des problèmes computationnels ; elle nous fournit un outil pour déterminer le meilleur algorithme pour nos besoins.

Dans l'analyse d'un algorithme nous essayons de calculer les ressources dont il aurait besoin. De telles ressources peuvent être, par exemple, l'utilisation CPU, mémoire, temps d'accès, etc. Comme le nombre de facteurs qui interviennent est souvent très grand, il est en général très difficile de calculer exactement les ressources utilisées. On se contente souvent de les estimer.

Dans ce chapitre, nous apprendrons des techniques pour calculer ou estimer la quantité de ressources utilisées par un algorithme.

2.2 EXEMPLE : MULTIPLICATION DE POLYNÔMES

La multiplication de polynômes est l'un des problèmes calculatoires principaux de l'algèbre computationelle (*computational algebra*) et du traitement du signal.

On rappelle qu'un polynôme sur $\mathbb{R}$ de degré d est une expression de la forme

$$p(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_d \cdot x^d$$

avec $a_0, \ldots, a_d \in \mathbb{R}$ et $a_d \neq 0$.

Les valeurs $a_0, a_1, \ldots, a_d$ s'appellent les *coefficients* du polynôme. Un polynôme peut être défini entièrement par ses coefficients. Si $p_1(x)$ et $p_2(x)$ sont des polynômes dont les degrés sont respectivement d_1 et d_2, alors leur produit est un polynôme de degré $d_1 + d_2$.

Problème: Multiplication de polynômes
Input: $(a_0, a_1, \ldots, a_{n-1}), (b_0, b_1, \ldots, b_{n-1}) \in \mathbb{R}^n$
Output: $(c_0, c_1, \ldots, c_{2n-2}) \in \mathbb{R}^{2n-2}$ tel que

$$\sum_{i=0}^{2n-2} c_i x^i = \sum_{k=0}^{n-1} a_k x^k \cdot \sum_{j=0}^{n-1} b_j x^j.$$

Les valeurs $a_0, a_1, \ldots, a_{n-1}$ et $b_0, b_1, \ldots, b_{n-1}$ sont les coefficients des polynômes d'entrée, et les valeurs $c_0, c_1, \ldots, c_{2n-2}$ sont les coefficients de leur produit. Remarquons qu'en multipliant deux polynômes de degré $n-1$, on obtient un polynôme de degré $2n-2$.

Si nous voulons multiplier deux polynômes, nous pouvons simplement multiplier chaque terme du premier polynôme par chaque terme du deuxième polynôme et combiner les termes de même exposant.

Nous voulons compter le *nombre d'opérations arithmétiques* (c'est-à-dire d'additions et de multiplications de nombres réels) dont a besoin l'algorithme naïf, qui effectue le calcul

$$c_i = \sum_{j=\max(0,i+1-n)}^{\min(i,n-1)} a_j \cdot b_{i-j}$$

pour tout $i = 0, 1, \ldots, 2n-2$.

Comptons le nombre d'additions et de multiplications à chaque étape.

$$\begin{array}{rcll}
c_0 & = & a_0 \times b_0 & 1+0 \\
c_1 & = & a_0 \times b_1 + a_1 \times b_0 & 2+1 \\
\vdots & & \vdots & \vdots \\
c_{n-2} & = & a_0 \times b_{n-2} + \cdots + a_{n-2} \times b_0 & n-1+n-2 \\
c_{n-1} & = & a_0 \times b_{n-1} + \cdots\cdots + a_{n-1} \times b_0 & n+n-1 \\
c_n & = & a_1 \times b_{n-1} + \cdots + a_{n-1} \times b_1 & n-1+n-2 \\
\vdots & & \vdots & \vdots \\
c_{2n-3} & = & a_{n-1} \times b_{n-2} + a_{n-2} \times b_{n-1} & 2+1 \\
c_{2n-2} & = & a_{n-1} \times b_{n-1} & 1+0
\end{array}$$

L'algorithme utilise

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

multiplications, et

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

additions.

En additionnant ces nombres, nous obtenons le nombre total d'opérations arithmétiques, qui vaut $n^2 + (n-1)^2$.

Nous disons que l'algorithme utilise *de l'ordre de* n^2 opérations.

La notation d'*ordre* supprime les constantes multiplicatives et donne une borne supérieure au nombre d'opérations. Nous étudierons cette notation plus en détail dans ce chapitre.

2.3 EXEMPLE : MULTIPLICATION DE MATRICES

Problème: Multiplication de matrices
Input: $A = (a_{ij}) \in \mathbb{R}^{n\times n}, B = (b_{ij}) \in \mathbb{R}^{n\times n}$
Output: $C = (c_{ij}) \in \mathbb{R}^{n\times n}$, où $C = A \cdot B$

La multiplication de matrices est un des problèmes principaux en algèbre linéaire. On peut montrer que de nombreux problèmes calculatoires en algèbre linéaire (résoudre des systèmes d'équations, inversion de matrices, calcul de déterminants, etc.) se réduisent à la multiplication de matrices.

Si $A = (a_{ij})$ et $B = (b_{ij})$ sont les deux des matrices $n \times n$, alors le produit matriciel $C := A \cdot B$ est une matrice $n \times n$, avec les composantes

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

pour tout $1 \leq i, j \leq n$.

Nous aimerions analyser l'algorithme naïf en comptant le nombre d'additions et de

multiplications (de nombres réels) à chaque pas.

$$\begin{array}{rcl} c_{11} & = & a_{11}\times b_{11}+a_{12}\times b_{21}+\cdots+a_{1n}\times b_{n1} \\ c_{12} & = & a_{11}\times b_{12}+a_{12}\times b_{22}+\cdots+a_{1n}\times b_{n2} \\ \vdots & & \vdots \\ c_{ij} & = & a_{i1}\times b_{1j}+a_{i2}\times b_{2j}+\cdots+a_{in}\times b_{nj} \\ \vdots & & \vdots \\ c_{nn} & = & a_{n1}\times b_{1n}+a_{n2}\times b_{2n}+\cdots a_{nn}\times b_{nn} \end{array}$$

L'algorithme utilise

$$n^2 \cdot n$$

multiplications, et

$$n^2 \cdot (n-1)$$

additions.

Au total, cet algorithme effectue donc $n^3 + n^2 \cdot (n-1)$ opérations. Nous disons que l'algorithme utilise *de l'ordre de* n^3 opérations arithmétiques.

La notation d'ordre supprime les constantes et donne la croissance approximative. L'idée est de ne considérer que les termes principaux (dans ce cas n^3) et de supprimer le reste, qui devient insignifiant quand n est grand.

n	$1000\log_2(n)$	$200\sqrt{n}$	$50n$	$10n^2$	$2n^3$	$\frac{1}{32}2^n$
2	1000	282.84	100	40	16	1/8
4	2000	400	200	160	128	1/2
8	3000	565.69	400	640	1024	8
16	4000	800	800	2560	8192	2048
32	5000	1131.37	1600	10240	65536	134217728
64	6000	1600	3200	40960	524288	Grand
128	7000	2262.74	6400	163840	4194304	GRAND
256	8000	3200	12800	655360	33554432	énorme
512	9000	4500.25	25600	2621440	268435456	ÉNORME
1024	10000	6400	51200	10485760	GRAND	$\cdots$

2.4 LA NOTATION "O"

Nous aimerions mesurer des fonctions de manière approximative, selon leur croissance. Ceci nous permettra d'avoir une première impression du temps de parcours d'un algorithme. Nous obtiendrons ainsi une caractérisation simple de son efficacité, permettant de comparer différents types d'algorithmes.

Dans ce qui suit, toutes les fonctions sont de $\mathbb{N}$ vers $\mathbb{R}_{>0}$.

Définition. Pour deux fonctions f et g nous dirons que $f = O(g)$ si

$$\exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : \quad f(n) \leq cg(n).$$

Alors *"f croît au plus aussi vite que g"*. O se prononce "grand O".

Exemple: Toute fonction est de l'ordre d'elle-même, on a par exemple $n = O(n)$ et de manière générale $f(n) = O(f(n))$ pour toute fonction $f(n)$. En effet, on peut prendre $c = 1$ et $n_0 = 1$ dans la définition ci-dessus puisque $\forall n \geq 1$ on a $f(n) \leq 1 \cdot f(n)$.

Exemple: Les constantes multiplicatives ne jouent pas de rôle : $3n = O(10n)$, et de manière générale, pour toute fonction $f(n)$ et pour tout $a, b \in \mathbb{R}_{>0}$ on a :

$$a \cdot f(n) = O(b \cdot f(n))$$

On peut obtenir ce résultat en prenant $c = \frac{a}{b}$ et $n_0 = 1$ dans la définition ci-dessus, puisque $\forall n \geq 1$ on a $a \cdot f(n) \leq \frac{a}{b} \cdot b \cdot f(n)$.

Exemple: On a $n^2 = O(n^3)$. En effet, en prenant $c = 1$ et $n_0 = 1$ ci-dessus, on voit que $\forall n \geq 1$ on a $n^2 \leq 1 \cdot n^3$.

Exemple: On a $1000n = O(n^2)$. En effet, en prenant $c = 1000$ et $n_0 = 1$ ci-dessus, on voit que $\forall n \geq 1$ on a $1000n \leq 1000 \cdot n^2$.

Exemple: On a $n+100 = O(n^2)$. On prend $c = 1$ et $n_0 = 11$ ci-dessus, on voit que $\forall n \geq 11$ on a $n + 100 \leq 1 \cdot n^2$.

Exemple: On a :

- $n = O(1000n)$
- $1000n = O(n)$
- $\log n = O(\log n)$
- $\log n = O((\log n)^2)$
- $n^2 = O(n^3)$
- $an^2 = O(bn^3)$ pour tous $a, b \in \mathbb{R}_{>0}$
- $n = O(100n^2 - 14n + 2)$
- $1000 \cdot n^{1000} = O(2^n)$
- $100 \cdot \log(n) = O(n)$
- $\log(n) + 3 + n^2 = O(n^2)$
- $1 = O(n)$
- $2003 = O(1)$

- $\frac{1}{n} = O(1)$
- Si $p(n)$ est un polynôme de degré $\leq d$, alors $p(n) = O(n^d)$
- Si $p(n)$ est un polynôme, et $a > 1$ alors $p(n) = O(a^n)$

Définition. Pour des fonctions f et g nous écrivons $f = o(g)$ si

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Alors, *"f croît plus lentement que g"*. On peut aussi dire que *"f devient insignifiante par rapport à g lorsque n tend vers l'infini"*. o so prononce "petit o".

Exemple: $n = o(n^2)$, en effet on a

$$\lim_{n\to\infty} \frac{n}{n^2} = \lim_{n\to\infty} \frac{1}{n} = 0.$$

Remarquons que si $f = o(g)$, alors on a aussi $f = O(g)$. $n = o(n^2)$ dans l'exemple ci-dessus, donc on a aussi $n = O(n^2)$.

Définition. Pour des fonctions f et g nous disons que
- $f = \Omega(g)$ si $g = O(f)$, et
- $f = \theta(g)$ si $f = O(g)$ et $f = \Omega(g)$.

Donc, si $f = \theta(g)$ alors *"f et g ont le même ordre de magnitude"*.

Exemple: On a :
- $n^7 = \Omega(n^6)$
- $100n^3 = \theta(10n^3)$
- $1 + n + 10n^2 = o(n^3)$
- $1 + n + 10n^2 \neq o(n^2)$
- $1 + n + 10n^2 = O(n^2)$
- $1 + n + 10n^2 = \theta(n^2)$
- $n^{1001} = o(1.001^n)$
- Si $p(n)$ est un polynôme de degré $< d$ alors $p(n) = o(n^d)$.
- Si $p(n)$ est un polynôme de degré d alors $p(n) \neq o(n^d)$.
- Si $p(n)$ est un polynôme de degré d alors $p(n) = \theta(n^d)$.

Dans ce langage nous disons que la multiplication naïve de polynômes est un algorithme $O(n^2)$, ou encore que c'est un algorithme avec un temps de parcours $O(n^2)$. La multiplication naïve de matrices est un algorithme $O(n^3)$, ou un algorithme avec temps de parcours $O(n^3)$.

Peut-on faire mieux ?

Oui. Nous allons introduire puis analyser des algorithmes plus rapides pour ces deux problèmes.

2.5 L'ALGORITHME DE KARATSUBA

Supposons que nous aimerions multiplier $f(x) = \sum_{i=0}^{2n-1} a_i x^i$ et $g(x) = \sum_{i=0}^{2n-1} b_i x^i$ aussi rapidement que possible. Nous pouvons écrire $f(x)$ et $g(x)$ comme suit.

$$\begin{aligned} f(x) &= \underbrace{\left(a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}\right)}_{=:f_0(x)} + x^n \cdot \underbrace{\left(a_n + a_{n+1}x + \cdots + a_{2n-1}x^{n-1}\right)}_{=:f_1(x)} \\ g(x) &= \underbrace{\left(b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}\right)}_{=:g_0(x)} + x^n \cdot \underbrace{\left(b_n + b_{n+1}x + \cdots + b_{2n-1}x^{n-1}\right)}_{=:g_1(x)}. \end{aligned}$$

Remarquons que les polynômes $f_0(x)$ et $f_1(x)$ respectivement $g_0(x)$ et $g_1(x)$ n'ont que la moitié de taille (degré) de $f(x)$ respectivement $g(x)$. L'idée sous-jacente à l'algorithme de Karatsuba est d'opérer sur les polynômes plus petits $f_i(x)$ et $g_i(x)$, où les calculs peuvent être effectués plus rapidement.

Considérons l'algorithme ci-dessous pour calculer $f(x) \cdot g(x)$.

Algorithme 6 L'Algorithme de Karatsuba

1: $h_0(x) \leftarrow f_0(x) \cdot g_0(x)$.
2: $h_2(x) \leftarrow f_1(x) \cdot g_1(x)$.
3: $u(x) \leftarrow f_0(x) + f_1(x)$
4: $v(x) \leftarrow g_0(x) + g_1(x)$.
5: $A(x) \leftarrow u(x) \cdot v(x)$.
6: $h_1(x) \leftarrow A(x) - h_0(x) - h_2(x)$.
7: $h(x) \leftarrow h_0(x) + x^n h_1(x) + x^{2n} h_2(x)$.
8: **return** $h(x)$

Remarquons que les opérations dans les pas 1, 2 et 5 sont elles-mêmes des multiplications de polynômes, mais de polynômes de taille n au lieu de $2n$.

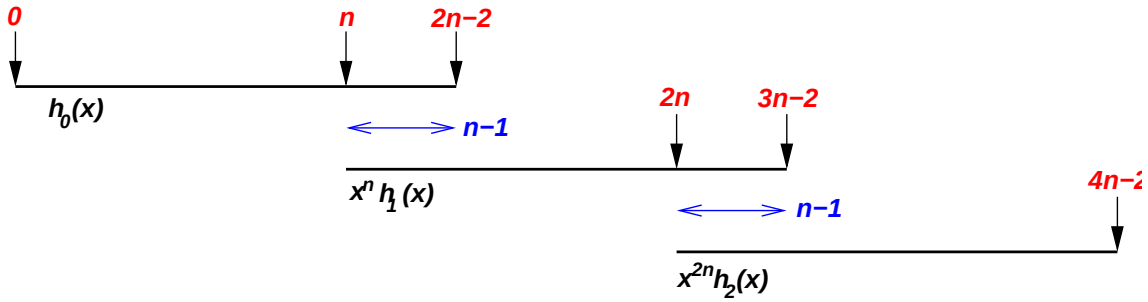
Nous montrons d'abord que le polynôme $h(x)$ calculé est en effet le produit $f(x) \cdot g(x)$. Pour le voir, on calcule la valeur de $h(x)$. Pour améliorer la lisibilité, nous écrivons h au

lieu de $h(x)$, etc. :

$$\begin{aligned} h &= x^{2n}h_2 + x^n h_1 + h_0 \\ &= x^{2n}f_1g_1 + x^n(uv - h_0 - h_2) + f_0g_0 \\ &= x^{2n}f_1g_1 + x^n \underbrace{[(f_0+f_1)(g_0+g_1) - f_0g_0 - f_1g_1]}_{(f_0g_1+f_1g_0)} + f_0g_0 \\ &= x^{2n}f_1g_1 + x^n(f_0g_1 + f_1g_0) + f_0g_0 \\ &= (x^n f_1 + f_0)(x^n g_1 + g_0) \\ &= f \cdot g. \end{aligned}$$

Ce que nous aimerions savoir maintenant est, bien sûr, s'il y a un avantage à utiliser l'Algorithme 6 au lieu de l'algorithme naïf introduit dans la section 2.2. Analysons donc son temps de parcours.

Pour multiplier deux polynômes de degré $< 2n$, nous avons besoin de 3 multiplications de polynômes de degré $< n$, 2 additions de polynômes de degré $< n$ et 2 additions de polynômes de degré $< 2n-1$ et de 2 additions de coût $2(n-1)$ (étape 7).



Commençons par supposer que nous utilisons l'algorithme naïf de multiplication de polynômes dans les étapes 1, 2 et 5 de l'algorithme de Karatsuba. Puisque, en utilisant l'algorithme naïf, nous avons besoin de $n^2+(n-1)^2$ opérations arithmétiques pour multiplier des polynômes de degré $< n$, le nombre total d'opérations pour multiplier deux polynômes de degré $< 2n$ utilisant la méthode de Karatsuba est :

$$3 \cdot (n^2 + (n-1)^2) + 2 \cdot n + 2 \cdot (2n-1) + 2 \cdot (n-1) = 6n^2 + 2n - 1,$$

tandis que l'algorithme naïf nécessite

$$8n^2 - 4n + 1$$

opérations, ce qui est plus grand par un facteur d'approximativement $4/3$ si n est grand. Ainsi, de cette manière nous obtenons un algorithme qui est plus rapide que l'algorithme naïf, mais ceci seulement par un facteur constant : il s'agit toujours d'un algorithme $O(n^2)$.

Heureusement, ce n'est pas la fin de l'histoire. Nous pouvons utiliser l'algorithme de Karatsuba lui-même dans les pas $1, 2$ et 5 pour obtenir un algorithme qui est encore plus

performant. De tels algorithmes, qui font appel à eux-mêmes pour accomplir leur tâche, s'appellent des algorithmes *récursifs*. Nous étudierons les algorithmes récursifs encore plus en détail.

Analysons maintenant le comportement du *vrai* algorithme de Karatsuba, i.e., de la variante récursive. Nous définissons les fonctions $M(\cdot), A(\cdot)$ et $T(\cdot)$ comme suit :

- $M(n)$: le nombre de multiplications pour multiplier deux polynômes de degré $< n$.
- $A(n)$: le nombre d'additions / soustractions pour multiplier deux polynômes de degré $< n$.
- $T(n)$: le nombre total d'opérations arithmétiques pour multiplier deux polynômes de degré $< n$, dans ce cas égal à $A(n) + M(n)$.

Nous avons clairement $M(1) = 1$, $A(1) = 0$, et nous pouvons calculer les valeurs de $M(\cdot), A(\cdot)$ et $T(\cdot)$ pour les puissances de deux en utilisant les récurrences suivantes :

$$\begin{aligned} M(2n) &= 3M(n) \\ A(2n) &= 3A(n) + 2n + 2(2n-1) + 2(n-1) \\ &= 3A(n) + 8n - 4, \\ T(2n) &= 3T(n) + 8n - 4 \end{aligned}$$

Par exemple, la formule pour $A(2n)$ ci-dessus se trouve comme suit : Le terme "$3A(n)$" vient des multiplications récursives dans les étapes 1, 2 et 5 de l'algorithme 6. Le terme "$2n$" vient des multiplications de polynômes de degré $< n$ dans les étapes 3 et 4. Le terme "$2(2n-1)$" vient des deux soustractions de polynômes de degré $< 2n-1$ dans l'étape 6. Finalement, le terme "$2(n-1)$" vient des additions dans l'étape 7. (Dans ce dernier cas, il suffit de compter les additions des parties qui se recouvrent ; c.f. l'illustration à la page précédente.)

Comment les fonctions $M(n)$, $A(n)$ et $A(n) + M(n)$ croissent-elles en fonction de n ? Nous apprendrons des techniques pour répondre à ce type de question plus tard dans ce chapitre.

Nous pouvons, par contre, déjà faire une comparaison numérique pour le cas où n est une puissance de deux. Nous obtenons

	Karatsuba			Naive		
n	Mult.	Add.	Total	Mult.	Add.	Total
2	3	4	7	4	1	5
4	9	24	33	16	9	25
8	27	100	127	64	49	113
16	**81**	**360**	**441**	**256**	**225**	**481**
32	243	1204	1447	1024	961	1985
64	729	3864	4593	4096	3969	8065
128	2187	12100	14287	16384	16129	32513

Tandis que pour n petit, la multiplication naïve semble être meilleure, nous voyons que pour $n \geq 16$, l'algorithme de Karatsuba est plus rapide. (En réalité, la multiplication est souvent plus coûteuse que l'addition, et l'algorithme de Karatsuba pourrait donc être meilleure même pour des n plus petits).

Nous montrerons plus tard que l'algorithme de Karatsuba est $O(n^{\log_2(3)})$, ce qui est beaucoup mieux que l'algorithme naïf $O(n^2)$, comme $\log_2(3) \sim 1.585$.

L'utilisation de structures de données adéquates rend aussi possible de réaliser ce gain sur des machines réelles et dans les applications pratiques. Par exemple, quelques implémentations de puces cryptographiques utilisent cet algorithme. Et même si de nos jours, des algorithmes meilleurs sont connus pour la multiplication de polynômes, l'algorithme de Karatsuba s'utilise souvent dans les logiciels informatiques (par exemple, PARI/GP), parce que c'est un algorithme qui donne une bonne performance en restant relativement facile à implémenter.

2.6 L'ALGORITHME DE STRASSEN

Dans la section précédente, nous avons appris que la multiplication de polynômes peut être effectuée plus rapidement en décomposant les polynômes en des parties plus petites et en effectuant ensuite les calculs sur ces polynômes plus petits. Cette stratégie de division de problèmes en sous-problèmes plus petits peut aussi être appliquée dans d'autres contextes ; ce type d'algorithme s'appelle un algorithme *diviser-pour-régner* (*divide-and-conquer*).

Dans cette section, nous présentons un algorithme diviser-pour-régner pour la multiplication de matrices.

Supposons que nous aimerions calculer le produit de deux matrices A et B,

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

où C_{ij}, A_{ij}, B_{ij} sont des matrices $n \times n$.

La décomposition en blocs ci-dessus suggère que si nous connaissions un algorithme performant pour la multiplication de matrices 2×2, ceci nous permettrait de construire un algorithme général (i.e., pour n arbitraire) performant de multiplication de matrices en utilisant la technique diviser-pour-régner.

Strassen a inventé un tel algorithme pour la multiplication de matrices 2×2 en 1969. Nous présentons ici la variante amélioré de Winograd de l'algorithme de Strassen qui nécessite encore moins d'additions. Les pas sont :

(1) Calculer

$$\begin{array}{lcllcl} S_1 &=& A_{21} + A_{22} & T_1 &=& B_{12} - B_{11} \\ S_2 &=& S_1 - A_{11} & T_2 &=& B_{22} - T_1 \\ S_3 &=& A_{11} - A_{21} & T_3 &=& B_{22} - B_{12} \\ S_4 &=& A_{12} - S_2 & T_4 &=& B_{21} - T_2 \end{array}$$

(2) Calculer

$$\begin{array}{lcllcl} P_1 &=& A_{11}B_{11} & P_5 &=& S_3T_3 \\ P_2 &=& A_{12}B_{21} & P_6 &=& S_4B_{22} \\ P_3 &=& S_1T_1 & P_7 &=& A_{22}T_4 \\ P_4 &=& S_2T_2 & & & \end{array}$$

et

$$\begin{array}{lcllcl} U_1 &=& P_1 + P_2 & U_5 &=& U_3 + P_3 \\ U_2 &=& P_1 + P_4 & U_6 &=& U_2 + P_3 \\ U_3 &=& U_2 + P_5 & U_7 &=& U_6 + P_6 \\ U_4 &=& U_3 + P_7. & & & \end{array}$$

(3) Ensuite,

$$\begin{array}{lcllcl} C_{11} &=& U_1, & C_{12} &=& U_7, \\ C_{21} &=& U_4, & C_{22} &=& U_5. \end{array}$$

Pour multiplier deux matrices $2n \times 2n$, l'algorithme naïf utilise 8 multiplications et 4 additions de matrices $n \times n$. L'algorithme de Strassen réduit ceci à 7 multiplications et 15 additions/soustractions de matrices $n \times n$. (Remarquons que le nombre d'additions de matrices n'est pas vraiment très important, puisqu'elle se fait élément par élément, donc en $O(n^2)$ tandis que la multiplication naïve est $O(n^3)$).

Pour analyser le temps de parcours de l'algorithme de Strassen, nous introduisons de nouveau les fonctions $M(\cdot), A(\cdot), T(\cdot)$:

- $M(n)$: Nombre de multiplications (de réels) pour multiplier des matrices $n \times n$.
- $A(n)$: Nombre d'additions (de réels) pour multiplier des matrices $n \times n$.

– $T(n)$: Le nombre total d'opérations arithmétiques pour la multiplication de matrices $n \times n$.

Pour les puissances de deux, ces fonctions sont données par les formules récursives

$$\begin{aligned} M(2n) &= 7M(n) \\ A(2n) &= 7A(n) + 15n^2 \\ T(2n) &= 7T(n) + 15n^2, \end{aligned}$$

et les conditions initiales $M(1) = 1$, $A(1) = 0$, $T(1) = 1$.

Dans la section 2.7 nous apprendrons comment déduire de ces formules que l'algorithme de Strassen est $O(n^{\log_2(7)})$. L'algorithme de Strassen est donc beaucoup plus performant que l'algorithme naïf $O(n^3)$, puisque $\log_2(7) \sim 2.81$. Le bon choix de structures de données aide à réaliser cet avantage en pratique.

2.7 RELATIONS DE RÉCURRENCE

Théorème 2.1 *Soit $T\colon \mathbb{N} \to \mathbb{R}$ une fonction telle qu'il existe $b, c, d \in \mathbb{R}_{>0}$, et $a \in \mathbb{N}$ avec*
(a) $T(n) \leq T(n+1)$ pour tout $n \geq 1$ (i.e. $T(\cdot)$ croît monotonement), et
(b) $T(an) \leq cT(n) + dn^b$.
On a alors
(1) Si $a^b < c$, alors $T(n) = O\left(n^{\log_a(c)}\right)$.
(2) Si $a^b = c$, alors $T(n) = O\left(n^{\log_a(c)} \cdot \log_a(n)\right)$.
(3) Si $a^b > c$, alors $T(n) = O\left(n^b\right)$.

Avant de prouver ce théorème, nous l'utilisons pour obtenir les temps de parcours estimés des algorithmes de Karatsuba et Strassen.

Corollaire 1 *L'algorithme de Karatsuba utilise $O(n^{\log_2(3)})$ opérations.*

Preuve. La fonction $T(n)$ dénote le nombre d'opérations pour multiplier des polynômes de degré $< n$. Nous pouvons clairement supposer $T(n) \leq T(n+1)$, car si n n'est pas une puissance de deux, nous pouvons résoudre le problème en remplissant les coefficients de f respectivement g plus grand que n avec des zéros, jusqu'à la prochaine puissance de deux. (C'est du zero-padding). Donc l'hypothèse (a) du théorème 2.1 est bien vérifiée.

Nous avons $T(2n) = 3T(n) + 8n - 4 \leq 3T(n) + 8n$. Donc, l'hypothèse (b) est vérifiée avec $a = 2$, $b = 1$, $c = 3$, et $d = 8$. En appliquant le théorème, nous obtenons le corollaire. ■

Corollaire 2 *L'algorithme de Strassen utilise $O(n^{\log_2(7)})$ opérations.*

Preuve. Nous avons $T(n) \leq T(n+1)$ pour la même raison qu'au corollaire précédent. Nous savons que $T(2n) \leq 7T(n) + 15n^2$. Donc nous pouvons appliquer le théorème avec $a = 2$, $b = 2$, $c = 7$, et $d = 15$. ■

Pour pouvoir prouver le théorème 2.1, nous avons besoin de la proposition suivante :

Proposition 2 *Soit $u \in \mathbb{R}$. Alors pour tout $k \geq 1$:*

$$\sum_{\ell=0}^{k-1} u^\ell = \begin{cases} k & \text{si } u = 1 \\ \frac{u^k - 1}{u - 1} & \text{si } u \neq 1. \end{cases}$$

De plus, si $|u| < 1$, alors

$$\sum_{\ell=0}^{k-1} u^\ell < \sum_{\ell=0}^{\infty} u^\ell = \frac{1}{1-u}.$$

Nous laissons la preuve comme exercice.

Preuve du théorème 2.1. La preuve consiste en 3 étapes :

Étape 1 : Nous montrons que pour tout $k \geq 1$,

$$T(a^k) \leq c^{k-1} \left(cT(1) + d \sum_{\ell=0}^{k-1} \left(\frac{a^b}{c} \right)^\ell \right).$$

Nous montrons ceci par induction sur k.

[**Base**] : $k = 1$. On a $T(a) \leq cT(1) + d$, en utilisant l'hypothèse (b) du théorème avec $n = 1$. ✓

[**Pas**] : L'application de la première étape suivi de l'hypothèse d'induction donne

$$\begin{aligned} T(a^{k+1}) &\leq cT(a^k) + da^{kb} \\ &\overset{\text{Hyp.}}{\leq} c^k \left(cT(1) + d \sum_{\ell=0}^{k-1} \left(\frac{a^b}{c} \right)^\ell \right) + da^{kb} \\ &= c^k \left(cT(1) + d \sum_{\ell=0}^{k-1} \left(\frac{a^b}{c} \right)^\ell + d \left(\frac{a^b}{c} \right)^k \right) \\ &= c^k \left(cT(1) + d \sum_{\ell=0}^{k} \left(\frac{a^b}{c} \right)^\ell \right). \end{aligned}$$

Étape 2 : nous montrons qu'il existe $\alpha, \beta, \gamma \in \mathbb{R}_{>0}$ tel que pour tout $k \geq 1$:

$$T(a^k) \leq \begin{cases} \alpha c^k & \text{si } a^b < c \\ \beta k c^k & \text{si } a^b = c \\ \gamma a^{kb} & \text{si } a^b > c \end{cases}$$

1. Si $a^b < c$, la Proposition 2 implique que

$$\sum_{\ell=0}^{k-1} \left(\frac{a^b}{c}\right)^\ell < \frac{1}{1-\frac{a^b}{c}}$$

pour tout k. En utilisant cette inégalité et l'étape 1, nous obtenons

$$\begin{aligned} T(a^k) &\leq c^{k-1}\Bigg(cT(1) + \underbrace{d \cdot \frac{1}{1-\frac{a^b}{c}}}_{=:\mu}\Bigg) \\ &= c^k T(1) + c^{k-1}\mu \\ &= \underbrace{\left(T(1) + \frac{\mu}{c}\right)}_{=:\alpha} c^k. \end{aligned}$$

2. Si $a^b = c$, alors $\sum_{\ell=0}^{k-1} \left(\frac{a^b}{c}\right)^\ell = k$, et donc avec l'étape 1 on obtient l'estimation suivante :

$$\begin{aligned} T(a^k) &\leq c^{k-1}\left(cT(1) + dk\right) \\ &= c^k\left(T(1) + \frac{kd}{c}\right) \\ &= \underbrace{\left(\frac{T(1)}{k} + \frac{d}{c}\right)}_{=:\beta} kc^k. \end{aligned}$$

3. Si $a^b > c$,

$$\sum_{\ell=0}^{k-1} \left(\frac{a^b}{c}\right)^\ell = \frac{\left(\frac{a^b}{c}\right)^k - 1}{\frac{a^b}{c} - 1} \leq \left(\frac{a^b}{c}\right)^k \underbrace{\frac{1}{\frac{a^b}{c} - 1}}_{=:\mu}.$$

Donc nous avons

$$\begin{aligned} T(a^k) &\leq c^k T(1) + c^{k-1} d \left(\frac{a^b}{c}\right)^k \mu \\ &= c^k T(1) + \frac{d\mu}{c} a^{kb} \\ &= a^{kb}\Big(\underbrace{\Big(\frac{c}{a^b}\Big)^k}_{<1} T(1) + \frac{d\mu}{c}\Big) \\ &\leq \underbrace{\left(T(1) + \frac{d\mu}{c}\right)}_{=:\gamma} a^{kb}. \end{aligned}$$

Étape 3 : Preuve du théorème.

Étant donné n, nous trouvons k tel que $a^{k-1} \leq n < a^k$. Nous aurons alors les trois inégalités suivantes :

$$a^k \leq an \tag{2.1}$$

$$k \leq 2\log_a(n) \tag{2.2}$$

$$T(n) \leq T(n+1) \leq \cdots \leq T(a^k) \text{ (Induction !)} \tag{2.3}$$

Si $a^b < c$, alors

$$\begin{aligned} T(n) &\leq T(a^k) \\ &\leq \alpha c^k \\ &= \alpha \cdot (a^k)^{\log_a(c)} \\ &\overset{(2.1)}{\leq} \alpha c \cdot n^{\log_a(c)}, \end{aligned}$$

donc, $T(n) = O(n^{\log_a(c)})$.

Si $a^b = c$, alors

$$\begin{aligned} T(n) &\leq T(a^k) \\ &\leq \beta k c^k \\ &= \beta k (a^k)^{\log_a(c)} \\ &\overset{(2.1)}{\leq} \beta c k \cdot n^{\log_a(c)} \\ &\overset{(2.2)}{\leq} 2\beta c \cdot n^{\log_a(c)} \log_a(n), \end{aligned}$$

d'où, $T(n) = O(n^{\log_a(c)} \log_a(n))$.

Si $a^b > c$, alors

$$\begin{aligned} T(n) &\leq T(a^k) \\ &\leq \gamma a^{kb} \\ &\overset{(2.1)}{\leq} \gamma a^b n^b, \end{aligned}$$

donc $T(n) = O(n^b)$.

Ceci termine la preuve du théorème. ■

2.8 REMARQUES FINALES

En analysant un algorithme il est important :
- D'identifier les opérations qui prennent du temps,
- D'analyser combien de telles opérations sont nécessaires,
- D'identifier une relation entre cette analyse et les vraies machines.

Nous avons développé la notation $O(\cdot)$ pour comparer les algorithmes asymptotiquement, i.e., quand le nombre de variables d'input est très grand. La notation $O(\cdot)$ ne prend pas en compte les constantes. Cependant, en pratique elles sont très importantes.

La notation $O(\cdot)$ prédit une certaine performance, mais le comportement d'un algorithme sur une vraie machine dépend de nombreux facteurs dont le choix des structures de données.

C'est ce qui nous concernera dans le prochain chapitre.

3
Structures de données élémentaires

Un programme C ou C++ contient d'habitude une partie algorithmique et une partie déclaratoire. La *partie algorithmique* (contenue typiquement dans les fichiers `.c`) contient proprement des algorithmes, i.e., une description de comment les données doivent être manipulées. La *partie déclaratoire* (en C dans les fichiers `.h`) contient les prototypes, les `typedef` et les déclarations de `struct`ures et `class`es. Autrement dit, la partie déclaratoire décrit comment les morceaux d'informations liés sont organisés, donc les *structures de données*.

> Algorithme + Structures de données = Programme

Le bon choix de structures de données est une partie crucial d'une implémentation performante d'un algorithme.

Dans ce chapitre nous étudierons quelques structures de données élémentaires et quelques types abstraits (*abstract data types*).

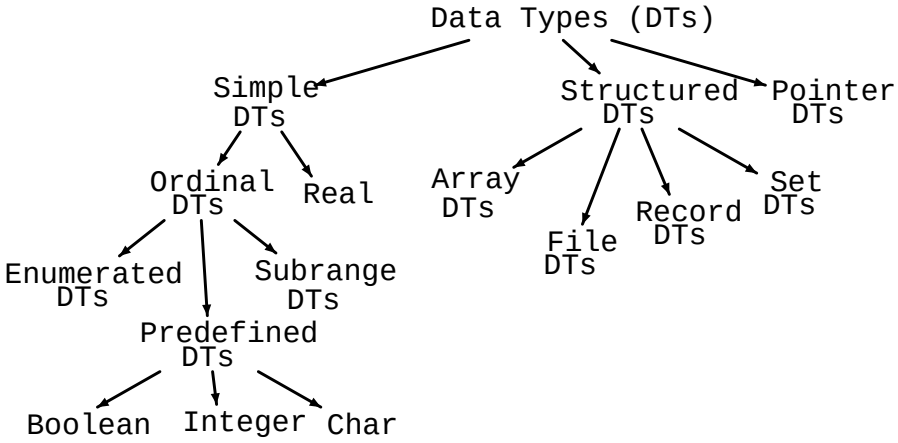
3.1 STRUCTURES DE DONNÉES STATIQUES

Une *structure de données statique* est une structure de données dont la taille ne change pas au cours de l'exécution du programme. Par exemple, tous les types encastrés de C (*built-in data types*) sont statiques.

Exemples :

- `int`eger : Un entier à 32 bit
- `bool`ean : Une variable booléenne à 1 bit
- `char`acter : Un caractère d'un octet
- `float` : Un nombre réel à précision simple.
- `double` : Un nombre réel à précision double.
- `int [5]` : Un tableau (array) de taille fixe contenant des structures de données statiques.
- etc.

On peut utiliser les types élémentaires ci-dessus pour obtenir des structures de données plus élaborées en utilisant des *constructeurs* (par exemple `struct`).



Nous supposerons par la suite que les structures de données statiques sont déjà connues, et nous étudierons dans ce cours surtout les structures de données dynamiques.

3.2 ENSEMBLES DYNAMIQUES

Très souvent, les algorithmes manipulent des ensembles de données. Ces ensembles ne sont pas statiques, mais peuvent grandir ou diminuer au cours du temps. Ce type d'ensemble est appelé *dynamique*.

Implémentation typique : Les ensembles dynamiques sont représentés comme des objets (structures) dont les membres individuels peuvent être inspectés et manipulés. L'accès aux membres est donné via un pointeur sur l'objet.

Pour la plupart des ensembles dynamiques, l'un des membres de l'objet est la *clé*, i.e., un identificateur unique de l'objet. Les autres membres de l'objet contiennent d'autres données d'intérêt.

Nous supposerons régulièrement que la clé appartient à un ensemble totalement ordonné (comme par exemple $\mathbb{N}$). Un ensemble S est totalement ordonné par rapport à une relation $<_0$ si pour tous $a, b \in S$, exactement une des trois conditions suivantes est vraie : $a <_0 b$, $b <_0 a$ ou $a = b$.

Les opérations sur les ensembles dynamiques appartiennent à deux types : les demandes et les opérations de modification.

- Les *demandes* donnent de l'information sur les éléments de l'ensemble.
- Les *opérations de modification* changent l'ensemble.

Écrivons S un ensemble dynamique, x un objet (élément) que S peut stocker, et k la clé d'un élément. Les opérations suivantes sur les ensembles dynamiques sont souvent utilisées en pratique :

Opérations de modification :
- `Insert`(S, x) : Opération de modification qui ajoute l'élément x à S
- `Delete`(S, x) : Opération de modification qui efface x de S si x appartient à S.

Demandes :
- `Search`(S, k) : Retourne un pointeur sur un élément $x \in S$ tel que `key`$[x] = k$ si un tel élément existe, et `NULL` sinon.
- `Minimum`(S) : Retourne l'élément avec la clé la plus petite.
- `Maximum`(S) : Retourne l'élément avec la clé la plus grande.
- `Successor`(S, x) : Retourne, pour un élément x donné, le prochain plus grand élément dans S, c'est-à-dire l'élément avec la plus petite clé qui soit plus grande que celle de x. Retourne `NULL` si x est l'élément maximal.
- `Predecessor`(S, x) : C'est l'analogue de la fonction `Successor`. Retourne pour x donné le prochain plus petit élément ou `NULL` si x est l'élément minimal.

Le temps nécessaire pour accomplir ces tâches est souvent donné en fonction de la taille de l'ensemble. Par exemple, nous serons souvent intéressés par des structures de données qui effectuent les opérations ci-dessus en temps $O(\log(n))$, où n est la taille de l'ensemble.

3.3 STRUCTURES DES DONNÉES DYNAMIQUES ÉLÉMENTAIRES

Dans ce chapitre nous étudierons les stacks, les files d'attente et les listes liées. Ce sont des types abstraits, c'est-à-dire que leur fonctionalité est définie mais ils peuvent être implémentés (réalisés) de différentes façons. Nous verrons pour chacun de ces trois types une implémentation (réalisation) qui utilise un tableau (array).

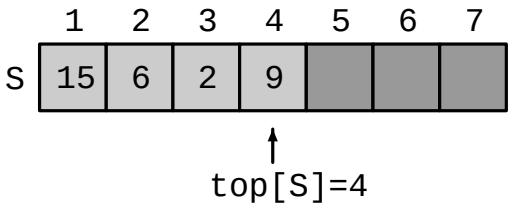
3.3.1 Stack (ou pile)

Un *stack* est une structure de données qui permet d'"empiler" des objets : L'intuition est qu'à chaque fois qu'on ajoute quelque chose sur une pile, on le met par dessus les objets déjà présents. On ne peut enlever que l'objet le plus haut. Ce principe s'appelle *dernier-entré, premier-sorti* (*LIFO, Last-in, First-out*).

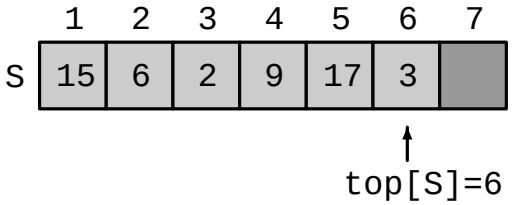
L'opération d'ajout d'un objet au stack s'appelle *push* et l'opération d'enlèvement est nommée *pop*. Une troisième opération indiquant si le stack est vide est parfois ajoutée.

On peut réaliser (implémenter) un stack avec un tableau (array) et un indice qui indique la première position vide (ou alternativement, la dernière position occupée).

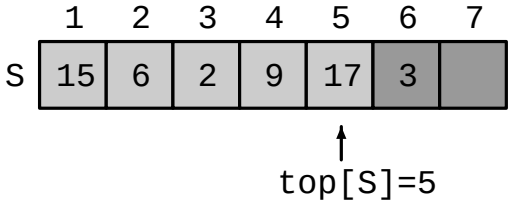
Illustration :



Voici un stack S contenant 4 éléments. Les cases sont des positions dans l'array de l'implémentation. Le stack possède un attribut $top[S]$ qui est l'indice indiquant la plus haute position occupée.



Le même stack S après $\texttt{Push}(S, 17)$ et $\texttt{Push}(S, 3)$.



L'opération $\texttt{Pop}(S)$ retourne 3 et efface ensuite cet élément de S (i.e. $top[S]$ est diminué de un).

Si on essaye d'effectuer un $\texttt{Pop}(S)$ sur un stack S vide, l'opération `Pop` retourne un code indiquant ce fait et on dit alors qu'on a *stack underflow*.

Les algorithmes 7, 8, 9 donnent une réalisation standard d'un stack basé sur un tableau (array).

Algorithme 7 $\texttt{StackIsEmpty}(S)$

```
1: if top[S] = 0 then
2:     return true
3: else
4:     return false
5: end if
```

Exemple: Les stacks sont très souvent utilisés dans la programmation. Quelques exemples :

Algorithme 8 `Push`(S, x)

1: top$[S] \leftarrow$ top$[S]$+1
2: S[top$[S]$] $\leftarrow x$

Algorithme 9 `Pop`(S)

1: **if** `StackIsEmpty`(S) **then**
2: **return** 'underflow'
3: **else**
4: top$[S] \leftarrow$ top$[S]$-1
5: **return** S[top$[S]$ + 1]
6: **end if**

– *Stack d'évaluation d'expressions* : Pour pouvoir évaluer des expressions comme

$$((a + b) * (c + d)) + (e * f) \tag{3.1}$$

on se sert d'un stack qui stocke les résultats intermédiaires. Les programmes qui évaluent ce type d'expression se servent typiquement d'un stack. L'expression (3.1) se traduit alors en la suite d'instructions suivante :

`Push`(S, a)
`Push`(S, b)
`Push`$(S,$ `Pop`$(S) +$ `Pop`$(S))$
`Push`(S, c)
`Push`(S, d)
`Push`$(S,$ `Pop`$(S) +$ `Pop`$(S))$
`Push`$(S,$ `Pop`$(S) *$ `Pop`$(S))$
`Push`(S, e)
`Push`(S, f)
`Push`$(S,$ `Pop`$(S) *$ `Pop`$(S))$
return `Pop`$(S) +$ `Pop`(S)

Pour des valeurs de a, b, c, d, e et f données, la suite ci-dessus retourne alors l'évaluation de l'expression.
(Certaines calculatrices de la marque HP laissent à l'utilisateur ce pas de traduction, et prennent comme input directement les opérations `Push` (la touche "enter"), et les opérateurs. Cette méthode s'appelle *reversed polish notation.*)

– *Stack d'adresse retour* : Pour stocker l'adresse retour d'une sous-routine. Un tel stack est utilisé par exemple dans les applications récursives. Les sous-routines sont ainsi exécutées dans l'ordre de leur ordre d'appel.
– *Stack de variable locale* : Pour gérer les variables locales dans des programmes récursifs.
– *Stack de paramètres* : Pour stocker les paramètres sur lesquels agit une sous-routine. (les valeurs d'input d'un programme par exemple).

Remarquons que quasi tous les programmes informatiques disposent au moins d'un stack : C'est le stack qui contient à la fois les variables locales, les adresses de retour et sur lequel les arguments pour les sous-routines sont passés. Ce stack sert aussi a stocker des résultats intermédiaires.

On peut par exemple utiliser un stack pour vérifier les parenthèses d'une expression :

Problème: Vérification de parenthèses
Input: Une expression contenant du texte et $(,), [,], \{,\}$
Output: "correct" ou "incorrect", selon si toutes les parenthèses ouvrantes sont ensuite fermées.

Nous utilisons un stack S, initialement vide.

Algorithme 10 VERIFICATIONPARENTHESES

```
1: while Input pas fini do
2:    c ← prochain caractère
3:    if c='(' ou c='[' ou c='{'  then
4:       Push(S, c) ;
5:    end if
6:    if c=')' ou c=']' ou c='}' then
7:       if StackIsEmpty(S) ou l'élément le plus haut de S ne correspond pas à c then
8:          return Incorrect
9:       else
10:         Pop(S) ;
11:      end if
12:   end if
13: end while
14: if StackIsEmpty(S) then
15:    return Correct
16: else
17:    return Incorrect
18: end if
```

La condition "l'élément le plus haut de S ne correspond pas à c" est définie par :
- '(' correspond à ')'
- '[' correspond à ']'
- '{' correspond à '}'

Exemple: $((1+9)*93-(98)/23-(43+2)$ est une expression incorrecte, la toute première parenthèse n'est pas fermée. L'expression $[(1+9)*93-(98)/23-(43+2)]$ est correcte.

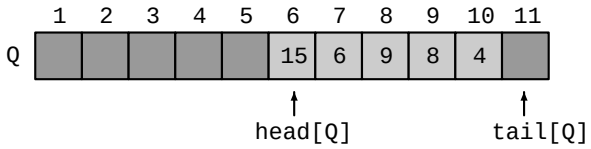
3.3.2 Files d'attente (Queues)

Les files d'attente sont des ensembles dynamiques qui permettent de nouveau d'enlever et d'ajouter des éléments. Cette fois, on ne peut ajouter qu'à la fin et enlever qu'au début, i.e., si on enlève, on enlève toujours le plus ancien élément encore dans la queue. C'est le principe *premier-entré, premier-sorti* (en anglais : *first-in-first-out, FIFO*). Nous appelons l'opération d'enlèvement d'un élément *dequeue*, et celle d'ajout *enqueue*.

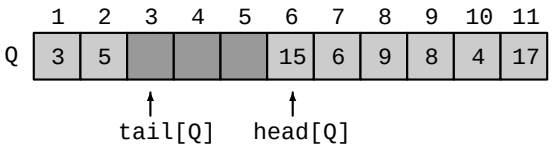
Il est de nouveau possible de se servir d'un tableau (array) pour réaliser une file d'attente, si nous savons que la queue ne dépassera pas une certaine taille. Nous avons besoin de deux attributs, à savoir d'un attribut qui indique la prochaine position libre à la fin de la queue, et un attribut qui indique la position du plus vieil élément de la queue (i.e. la position du prochain élément à enlever). Nous appellerons ces attributs respectivement `tail` et `head`.

L'idée est maintenant de remplir à chaque fois la position `tail` si `Enqueue` est appelé, et d'ensuite incrémenter `tail`. Si `tail` dépasse la fin du tableau, on recommence à la première position. La fonction `Dequeue` procède de manière similaire : Ayant lu la valeur de la position `head` à retourner, ce compteur est incrémenté et remis à un s'il dépasse la fin du tableau.

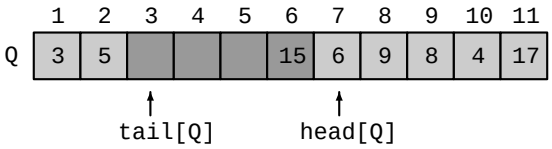
Exemple :



Le dessin ci-dessus représente une file d'attente Q avec 5 éléments aux positions $Q[6..10]$. Si nous faisons les opérations `Enqueue`$(Q, 17)$, `Enqueue`$(Q, 3)$ et `Enqueue`$(Q, 5)$, nous aurons alors la situation suivante :



Remarquons qu'après avoir ajouté 17, la valeur de `tail` est remise à 1. Si nous faisons ensuite l'opération `Dequeue`(Q), la valeur 15 est retournée, et la queue se trouve dans l'état suivant :



Les algorithmes 12 et 13 donnent des réalisations concrètes des opérations `Enqueue` respectivement `Dequeue` pour des queues basées sur un tableau. Remarquons que si head[Q] = tail[Q], la queue pourrait être soit pleine ou soit vide. Pour différencier ces deux cas nous posons head[Q]= `NULL` lorsque la queue est vide. Ainsi l'initialisation est la suivante :

Algorithme 11 `Initialize`(Q)

```
1: head[Q] ← NULL
2: tail[Q] ← 1
```

Algorithme 12 `Enqueue`(Q, x)

```
1: if head[Q]=tail[Q] then
2:    error 'overflow'
3: end if
4: Q[tail[Q]] ← x
5: if  head [Q] = NULL then
6:    head [Q] ← tail[Q]
7: end if
8: if tail[Q] = length[Q] then
9:    tail[Q]←1
10: else
11:    tail[Q] ← tail[Q]+1
12: end if
```

Les files d'attente sont utilisées quand des tâches doivent être traitées dans leur ordre d'arrivée.

Exemple: Les queues sont utilisées de manière visible à l'utilisateur dans les situations suivantes :

- *Routers* : Les tâches sont des paquets, il faut déterminer à quel router le paquet doit être envoyé. On utilise une file d'attente pour traiter les paquets dans l'ordre.
- *Appels téléphoniques* : Quand on appelle un numéro 0800 en Suisse, l'appel est mis dans une queue en attendant d'être traité. Dans la situation idéale, les appels sont traités dans leur ordre d'arrivée.
- *Imprimantes* : Quand on envoie une tâche à une imprimante, elle est insérée dans une file d'attente.

Les tampons (*buffers*) sont des types particuliers de files d'attente, contenant un flux de données à traiter. Les interfaces de transmission de données utilisent des tampons. Par exemple, les données à écrire sur un disque dur sont d'abord stockées sur un tampon.

Il existe des files d'attente qui prennent en compte les priorités, appelées des *files d'attente avec priorité* (*Priority Queues*). Cette variante ne nous concernera pas dans ce cours.

Algorithme 13 `Dequeue`(Q)

```
1: if head [Q] = NULL then
2:    error 'underflow'
3: end if
4: x ← Q[head[Q]]
5: if head[Q] = length[Q] then
6:    head[Q] ← 1
7: else
8:    head[Q] ← head[Q]+1
9: end if
10: if head[Q] = tail[Q] then
11:    head[Q] ← NULL
12: end if
13: return x
```

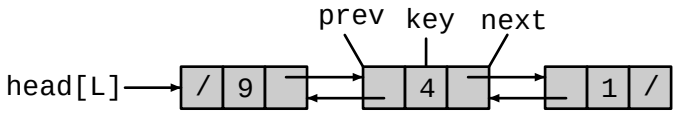
3.3.3 Listes liées (Linked lists)

Une liste liée est une façon simple et flexible de représenter un ensemble dynamique. Elle permet d'implémenter toutes les opérations mentionnées dans la section 3.2. Il s'agit d'une structure de données dans laquelle les objets sont ordonnés de façon linéaire. L'ordre est donné par des pointeurs associés à chaque objet.

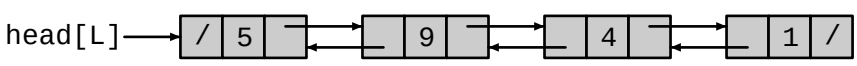
Il y a plusieurs types de listes liées. Nous regarderons ici les listes *doublement liées*. Les éléments de ce type de liste possèdent chacun deux pointeurs, `prev` et `next`. Le pointeur `prev` pointe sur le prédécesseur (dans la liste) de l'élément, et `next` sur le successeur. Si `prev`$[x]$ = `NULL` alors x est le premier élément (*head*) de la liste, et si `next`$[x]$ = `NULL`, alors x est le dernier élément (*tail*).

Nous avons encore besoin d'un attribut `head`$[L]$, qui est un pointeur indiquant le premier élément de la liste. Pour une liste L vide, on a `head`$[L]$ = `NULL`.

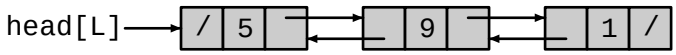
Exemple :



C'est une liste doublement liée L qui représente $(9, 4, 1)$.



La liste L après `Insert`(L, x), avec `key`$[x] = 5$.



La liste L après `Delete`(L, x), avec `key`$[x] = 4$.

Les algorithmes 14, 15 et 16 montrent comment chercher, insérer et effacer des objets dans une liste doublement liée.

Algorithme 14 `Search`(L, k)

```
1: x ← head[L]
2: while x ≠ NULL et key[x] ≠ k do
3:     x ← next[x]
4: end while
5: return x
```

Algorithme 15 `Insert`(L, x)

```
1: next[x] ← head[L]
2: prev[x] ← NULL
3: if head[L] ≠ NULL then
4:     prev[head[L]] ← x
5: end if
6: head[L] ← x
```

Algorithme 16 `Delete`(L, x)

```
1: if prev[x] ≠ NULL then
2:     next[prev[x]] ← next[x]
3: else
4:     head[L] ← next[x]
5: end if
6: if next[x] ≠ NULL then
7:     prev[next[x]] ← prev[x]
8: end if
```

3.4 GRAPHES ET ARBRES

Les graphes sont des structures de données générales qui peuvent être utilisées pour représenter des dépendances et des relations entre des objets. Ils sont très souvent utilisés en informatique, par exemple pour décrire la structure du hardware, des réseaux informatiques, des flux de données, des systèmes parallèles et des structures hiérarchiques (arbres). On rappelle la définition d'un graphe et des définitions s'y rapportant.

Définition. Un *graphe* (V, E) est un ensemble fini de sommets V et une relation $E \subseteq V \times V$.

Définition.

- Pour $a, b \in V$ nous disons qu'*il y a une arête de* a *à* b si $(a, b) \in E$.
- Un *chemin de longueur* n est une suite de n arêtes

$$(c_0, c_1), (c_1, c_2), \ldots, (c_{n-2}, c_{n-1}), (c_{n-1}, c_n).$$

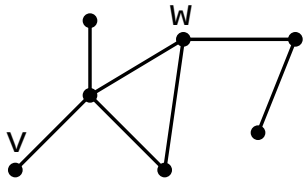
- Un chemin pour lequel $c_0 = c_n$ est appelé un *cycle*.
- Un graphe est dit *connexe* s'il existe un chemin entre a et b pour tout $(a, b) \in V^2$ tel que a est différent de b.
- Le *in-degré* d'un sommet a dans un graphe est le nombre d'arêtes qui vont vers a, i.e., $|\{b \in V \mid (b, a) \in E\}|$.
- Le *out-degré* d'un sommet a est le nombre d'arêtes qui sortent de a, i.e., $|\{b \in V \mid (a, b) \in E\}|$.
- Un *voisin* d'un sommet x d'un graphe est un autre sommet y tel que x et y sont liés (i.e., ont une arête en commun).
- Un graphe est dit *non-orienté* si E est symétrique.

Dans un graphe non-orienté, le degré d'un sommet a est le nombre de sommets adjacents à a, i.e.,

$$\deg(a) = |\{b \in E \mid (a, b) \in E\}|.$$

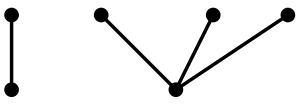
Exemple : G_1 et G_2 sont des graphes non-orientés.

G_1 :



connexe ;
a un cycle de longueur 3.

G_2 :



non-connexe ;
sans cycles

On voit que G_1 a 7 sommets et 7 arêtes, et que G_2 a 6 sommets et 4 arêtes. Dans G_1 le sommet V est de degré 1, et le sommet W est de degré 3.

3.4.1 Représenter des graphes

Comment peut-on représenter des graphes de manière efficace dans un ordinateur ? Nous allons étudier deux des plus importantes méthodes.

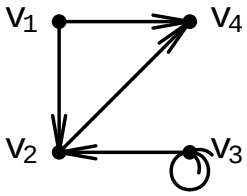
Représentation par une matrice d'adjacence

Considérons un graphe $G = (V, E)$, où $V = \{v_1, \ldots, v_n\}$. La *matrice d'adjacence* de G est une matrice $A = (a_{ij})_{1 \leq i,j \leq n}$, telle que pour tout i et j nous avons $a_{ij} \in \{0, 1\}$, définis comme suit :

$$a_{ij} := \begin{cases} 1, & \text{si } (v_i, v_j) \in E \\ 0, & \text{sinon.} \end{cases}$$

La quantité de mémoire nécessaire est n^2 bits pour des graphes orientés. Pour les graphes non-orientés, nous avons $a_{ij} = a_{ji}$, et donc la quantité de mémoire nécessaire est $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ bits.

Exemple : Considérons le graphe orienté G, et sa matrice d'adjacence A :

$G =$ 

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

On voit que le nombre de 1 dans la matrice d'adjacence est égal au nombre d'arêtes dans le graphe.

Remarquons les faits suivants :
- le in-degré de v_2 est 2, et il y a 2 fois la valeur 1 dans la $2^{\text{ème}}$ colonne de A.
- le out-degré de v_2 est 1, et il y a 1 fois la valeur 1 dans la $2^{\text{ème}}$ ligne de A.
- le in-degré de v_3 est 1.
- le out-degré de v_4 est 0.

En effet, par définition de A le nombre de 1 dans la colonne de A qui correspond au sommet V est égal au in-degré de V, et le nombre de 1 dans la ligne qui correspond à V est égal au out-degré de V.

Avantages :
- Représentation simple.
- On a un accès efficace, i.e., tester si deux sommets sont liés peut être rapidement effectué.

Désavantages :

– Nécessite beaucoup de mémoire ($O(n^2)$), indépendamment du nombre d'arêtes dans le graphe.
– Statique. Ne permet pas facilement d'agrandir le graphe.

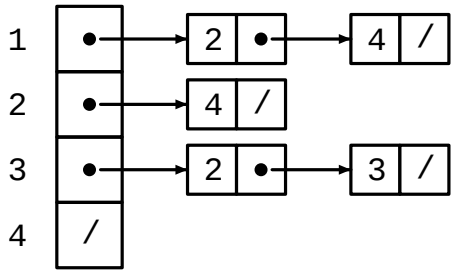
Représentation par des listes d'adjacence

Pour chaque sommet $v \in V$ nous stockons une liste L_v de sommets voisins de v. Par exemple, en C++ ceci peut être fait comme suit :

```
typedef int vertices ;
struct List {
    vertices value ;
    List* next ;
} ;
List graph[n] ;
```

Ainsi, le graphe est représenté comme un tableau (array) de listes d'adjacence. Il y a une liste par sommet v du graphe dans laquelle sont stockés tous ses voisins.

Exemple : Le graphe G de l'exemple précédent peut être représenté par les 4 listes suivantes :



En effet, il y a deux arêtes qui partent de v_1, l'une vers v_2 et l'autre vers v_4, et les éléments de la première liste (qui correspond à v_1) sont donc 2 et 4.

La deuxième liste correspond à v_2 qui a le seul voisin v_4. Donc 4 est le seul élément de la deuxième liste. Aucune arête ne part de v_4, la quatrième liste est donc vide.

Remarquons finalement qu'on a 4 listes car le graphe a 4 sommets, et ces 4 listes sont stockées dans un array.

Avantages :
- Nécessite peu de mémoire $O(|V| + |E|)$.
- On peut dynamiquement adapter le nombre d'arêtes.

Désavantages :
- Mémoire utilisée en plus par les pointeurs.
- Peut seulement opérer sur la liste d'adjacence de manière séquentielle (quoique ceci est souvent suffisant).

En utilisant une liste d'adjacence au lieu d'un array il est aussi possible d'augmenter ou de diminuer le nombre de sommets du graphe. Néanmoins, dans ce cas il est impossible d'avoir un accès aléatoire (*random access*) aux sommets.

3.4.2 Arbres

Les arbres sont l'une des plus importantes structures de données en informatique. Ils sont particulièrement utiles pour la représentation de données *hiérarchiques* qui apparaissent dans beaucoup d'applications. Très souvent, les arbres forment la base de solutions algorithmiques efficaces de problèmes.

Exemples :
- Arbres de décision
- Arbres de syntaxe
- Arbres de code
- Arbres de recherche
- Arbres couvrants (*spanning trees*)
- Arbres binaires
- Arbres AVL
- Arbres bicolores

Définition. Un *arbre* est un graphe qui est connexe et sans cycles.

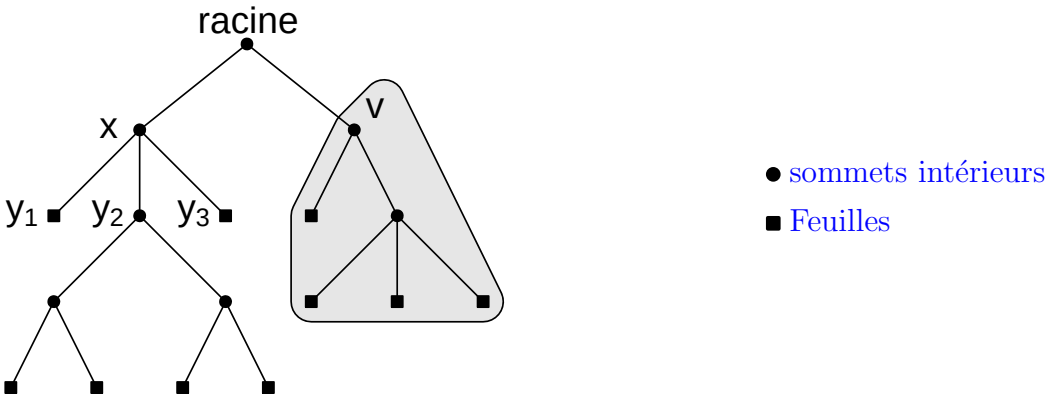
Nous considérons surtout les *arbres avec racine* (*rooted trees*), c'est-à-dire un arbre avec un sommet fixé appelé sa *racine* (*root*).

On a :
- Il existe pour tout $v \in V$ exactement un chemin sans répétitions de la racine v_0 à v. (Preuve ?)
- Si le chemin la racine à v est de longueur r, nous disons que v a *profondeur* r.
- Par définition, la racine est l'unique sommet de profondeur 0.

- La notion de profondeur induit une structure hiérarchique sur l'arbre. Si x est un sommet de profondeur r et $y_1, \ldots, y_d$ sont ses voisins de profondeur $r+1$, alors on les appelle les *fils* (*children*) de x, et x s'appelle le *père* (*parent*) des $y_1, \ldots, y_d$. L'entier d s'appelle le *degré* de x et est noté $\deg(x)$ (donc le degré de x est le nombre de fils de x).
- Un sommet de degré 0 est appelé une *feuille*
- Un sommet de degré > 0 est appelé un *sommet intérieur*.
- Le maximum $\max_{v \in V} \deg(v)$ s'appelle le *degré* de l'arbre.
- La *hauteur* d'un arbre est la plus grande profondeur dans l'arbre.
- Pour nous, un *nœud* est la même chose qu'un sommet.
- Si T est un arbre et x un sommet de T, alors le *sous-arbre* de racine x est l'arbre formé de tous les déscendants de x avec les mêmes arêtes.

Bien que nous avons défini un arbre comme un type particulier de graphe (connexe et sans cycles), les définitions ci-dessus ne sont pas les mêmes que celles pour un graphe (par exemple le définition de degré). Il est donc important quand on regarde des arbres avec racine de s'habituer à ces définitions.

Exemple : Considérons l'arbre avec racine T ci-dessous :



On a les faits suivants :

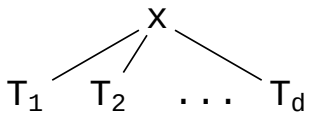
- T a hauteur 4
- x a profondeur 1
- y_1, y_2 et y_3 ont tous profondeur 2
- v a profondeur 1
- x a degré 3
- y_1 et y_3 ont degré 0, ce sont donc des feuilles
- y_2 et v ont degré 2
- x est le père de y_1, y_2 et y_3
- y_1, y_2 et y_3 sont les fils de x

- x et v sont les fils de la racine
- La racine est le père de x et v
- le degré de T est 3
- La partie grisée est un sous-arbre de T de racine v. Appelons ce sous-arbre B.
- B a hauteur 2
- B a degré 3

En considérant chaque fils de la racine comme racine d'un autre arbre avec racine, nous pouvons donner une définition *récursive* d'un arbre avec racine :

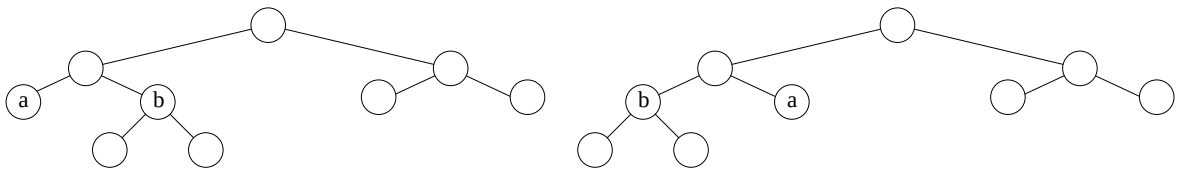
Définition. (Arbre avec racine)

1. L'arbre vide (sans sommets) est un arbre avec racine.
2. Si $d \geq 0$, $T_1, \ldots, T_d$ sont des arbres avec racine, et x est un nouveau sommet, alors T est un arbre avec racine x.

$T :=$ 

Remarque: Avec la définition ci-dessus, la notion d'arbre avec racine a été étendue et inclut maintenaut aussi les arbres vides et les sous-arbres vides.

D'habitude, l'ordre des sous-arbres d'un arbre avec racine n'est pas important. Si l'ordre est important, on dit que l'arbre est *ordonné*. Par exemple, les deux arbres suivants



ne sont pas distincts si considérés comme des arbres non-ordonnés, mais il le sont en tant qu'arbres ordonnés.

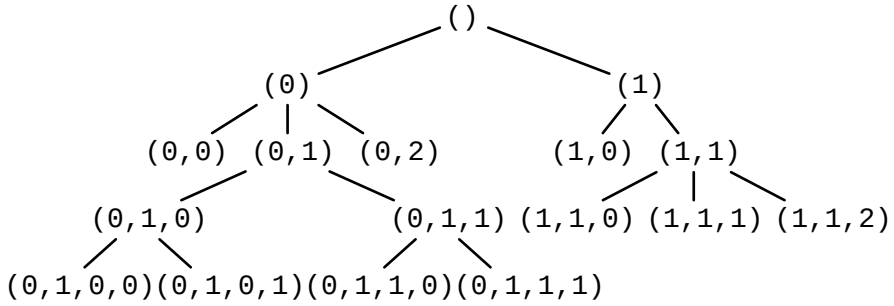
L'indice d'un sommet

Nous pouvons associer à chaque sommet dans un arbre ordonné avec racine une suite d'entiers non-négatifs, qui identifie ce sommet de manière unique. Cette suite s'appelle l'*indice*. Pour le faire, on procède récursivement :

Définition. (Indice d'un sommet)

1. L'indice de la racine est la suite vide ().
2. Supposons que le sommet x a l'indice $(a_0, \ldots, a_{k-1})$, et que $y_0, \ldots, y_{d-1}$ sont ses fils ordonnés. Alors à y_i on associe l'indice $(a_0, \ldots, a_{k-1}, i)$.

Exemple :



L'indice d'un sommet décrit la profondeur de ce sommet ainsi que l'unique chemin minimal de la racine à ce sommet. Par exemple le sommet en bas à gauche a l'indice

$$(0, 1, 0, 0)$$

C'est une suite de longueur 4, ce sommet a donc profondeur 4, et on voit que l'unique chemin de la racine à ce sommet est

$$() - (0) - (0, 1) - (0, 1, 0) - (0, 1, 0, 0)$$

Définition. Un *arbre binaire* (*binary tree*) est un arbre avec racine ordonné tel que chaque sommet a deux sous-arbres :
- un sous-arbre gauche (qui peut être l'arbre vide),
- un sous-arbre droit (qui peut aussi être l'arbre vide).

Une feuille est donc un sommet pour lequel le sous-arbre droit et le sous-arbre gauche sont tous les deux vides.

Définition. Un arbre binaire est *complet* si pour tout sommet, son sous-arbre gauche a la même hauteur que son sous-arbre droite.

3.4.3 Représentation d'arbres

Un arbre est un type particulier de graphe. On peut appliquer les mêmes techniques pour représenter des arbres que celles pour représenter des graphes.

Cependant la représentation par matrice d'adjacence est très inefficace : l'arbre sur n sommets a seulement $n-1$ arêtes, mais la matrice d'adjacence utilise $O(n^2)$ bits. De plus, une ligne entière de la matrice doit être parcourue pour trouver les fils d'un sommet.

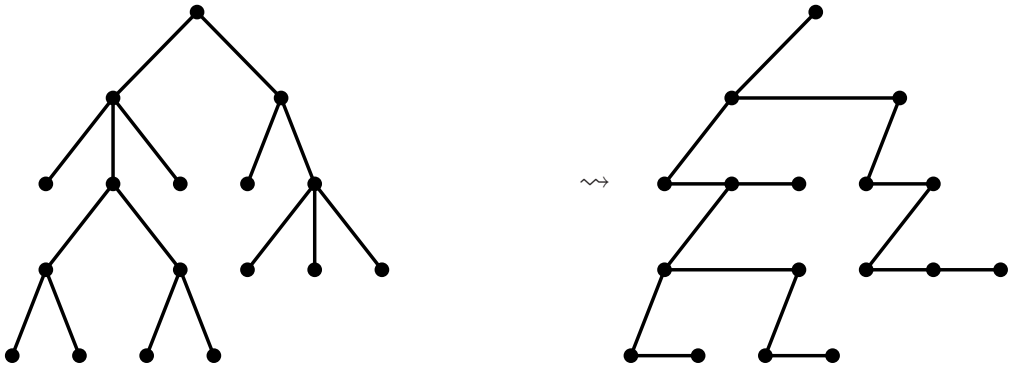
Les listes d'adjacence sont mieux adaptés pour représenter des arbres. Néanmoins, avec cette représentation, il est difficile de trouver le père d'un sommet donné, parce que ceci nous oblige a parcourir toutes les listes d'adjacence.

Dans cette partie, nous étudierons d'autres possibilités pour représenter les arbres.

Nous supposerons que l'arbre en question est de degré d (pour un d petit). C'est souvent vrai en pratique. Par exemple, pour les arbres binaires on a $d = 2$.

Il est même possible de construire un arbre binaire avec le même nombre de sommets à partir d'un arbre quelconque avec racine : si D est l'ensemble d'indices de l'arbre, alors l'arbre binaire correspondant (D, E) :

- $(0) \in D \quad \Rightarrow \quad \{(), (0)\} \in E$
- $(a, 0) \in D \quad \Rightarrow \quad \{(a), (a, 0)\} \in E$
- $(a, i), (a, i+1) \in D \quad \Rightarrow \quad \{(a, i), (a, i+1)\} \in E$



Désavantage : Les fils doivent être parcourus de manière séquentielle, ce qui ralentit le parcours de l'arbre.

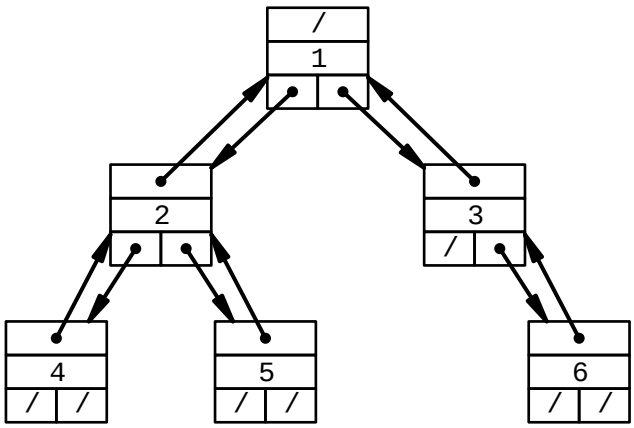
Structures de Pointeurs

Comme les listes, les arbres peuvent aussi être représentés en munissant chaque sommet avec des pointeurs sur les voisins : chaque sommet est une structure (record, struct) qui, en plus des données stockées au sommet, contient aussi des pointeurs vers le père et tous les fils. Si l'arbre est de degré $\leq d$, alors cette représentation peut être réalisée en C++ comme suit :

```
struct node {
    int val ;
    node* parent ;
    node* child[d] ;
} ;
typedef node* tree ;
```

Avec cette représentation, un arbre avec racine est donné par un pointeur vers sa racine.

Exemple : Un arbre binaire



(Il est important d'avoir une liste doublement liée pour être capable d'accéder aux pères directement.)

Avantages :
- Représentation simple
- Stockage dynamique
- Insertion, effacement, déplacement de sommets ou même de sous-arbres est très efficace.

Désavantages :
- Nécessite plus de mémoire pour les pointeurs.

Arbres comme tableaux (arrays)

Pour chaque sommet, on stocke les données et les indices des fils/pères dans un tableau (array). Cette méthode est raisonnable seulement si le nombre de sommets de l'arbre n'excède pas un maximum prédéterminé. Une telle représentation peut être implémenté en C++ de la manière suivante :

```
typedef int tree;
struct node {
    int val;
    tree left;
    tree right;
};
node table[MaxNodes+1];
```

Une valeur d'indice de -1 correspond à un arbre vide. Ainsi, l'arbre est donné par un tableau, et l'indice de sa racine se trouve dans le tableau.

Avantages :
- Accès rapide
- Pas de pointeurs
- Il est possible d'écrouler l'arbre

Désavantages :
- Nombre fixe de sommets (structure statique)
- Les opérations au milieu de l'arbre (telles que insertion et effacement) sont plus élaborées que pour le cas d'arbre basé sur les pointeurs.

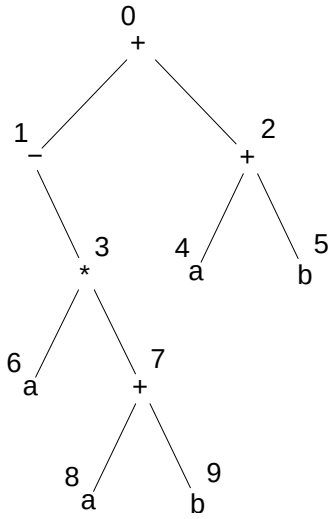
Exemple : Arbre d'expression

Considérons l'expression

$$(-(a \cdot (a + b))) + (a + b).$$

On peut la représenter avec un arbre binaire de la manière suivante :

B :



Dans cet arbre la racine correspond à l'indice 0. L'arbre peut être réalisé par ce tableau (array) :

i	val	gauche	droite
0	$+$	1	2
1	$-$	-1	3
2	$+$	4	5
3	$*$	6	7
4	a	-1	-1
5	b	-1	-1
6	a	-1	-1
7	$+$	8	9
8	a	-1	-1
9	b	-1	-1

Par exemple, la première ligne du tableau nous dit que la racine (le sommet d'indice 0) a la valeur $+$, et que ses deux fils ont les indices 1 (gauche) et 2 (droite). Donc si on veut voir le fils gauche du sommet on regarde la deuxième ligne du tableau (puisque ce sommet a l'indice 1) et on voit que sa valeur est $-$ et que ses fils sont l'arbre vide (qui correspond à l'indice -1) et le sommet d'indice 3.

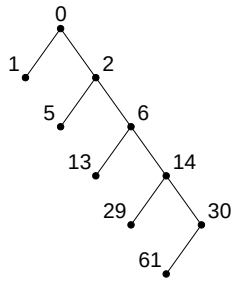
Représentation implicite d'arbres

Les sommets sont stockés dans un array, les arêtes s'obtiennent de manière implicite par la position des sommets dans l'array. Supposons que l'arbre est de degré $d \geq 2$: La racine se trouve en $A[0]$. Si x est stocké dans $A[i]$, alors les enfants $y_1, \ldots, y_d$ de x sont stockés dans $A[di+1], A[di+2], \ldots, A[di+d]$.

Avantage : Pas de pointeurs. Utilise peu de mémoire.

Désavantage : La mémoire est gaspillée si l'arbre est penché (i.e., il y a des sommets avec des sous-arbres de tailles différentes). De plus, les opérations au milieu de l'arbre sont compliquées.

Exemple :



Nous avons besoin d'un array avec 62 positions pour pouvoir représenter un arbre non-équilibré avec seulement 10 sommets.

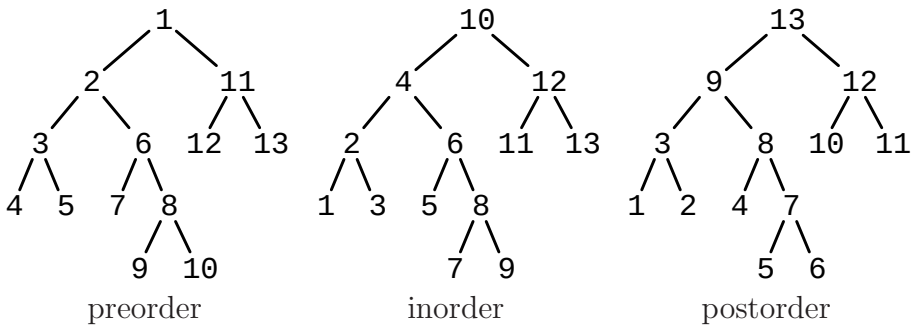
3.4.4 Parcourir des arbres

Une fois les données stockées dans un arbre, nous aimerions pouvoir l'inspecter pour trouver des sommets avec des propriétés données ou pour appliquer une opération à tous les sommets. Pour ce faire, il est nécessaire d'avoir des méthodes systématiques pour parcourir l'arbre.

Nous considérons ici le cas des arbres binaires ordonnés. Il y a trois méthodes de parcours :

- La méthode *preorder* consiste à d'abord parcourir la racine, ensuite le sous-arbre de gauche, et finalement le sous-arbre de droite. (Cette définition, comme les deux suivantes, est récursive : Nous supposons que les sous-arbres gauche et droite sont aussi parcourus en preorder.)
- Le parcours *inorder* parcourt d'abord le sous-arbre de gauche, ensuite la racine et enfin le sous-arbre de droite.
- Parcourir l'arbre en *postorder* veut dire de d'abord parcourir le sous-arbre de gauche, puis le sous-arbre de droite, et finalement la racine.

Exemple: Les nombres dans les arbres ci-dessous donnent les ordres de parcours



Exemple: Nous représentons ici un arbre par un pointeur sur un `struct` qui contient comme attributs un pointeur `left` et un pointeur `right` sur les sous-arbres de gauche et de droite respectivement. Un pointeur `NULL` représente un arbre vide.

L'algorithme récursif 17 effectue un parcours preorder. Dans cet exemple, inspect(t) effectue l'opération désirée sur le sommet t.

Algorithme 17 preorder(t)

Input: Un pointeur t sur la racine a parcourir.

```
if t ≠ NULL then
  inspect(t)
  preorder(left[t])
  preorder(right[t])
end if
```

3.4.5 Résumé de la section 3.4

- Nous avons vu qu'un graphe etait une paire (V, E) où V est un ensemble de sommets et $E \subseteq V \times V$ est une relation sur ces sommets. Nous avons vu les notions de chemin, cycle, connexité, in-degré, out-degré, graphe orienté, graphe non orienté.
- Nous avons vu deux façons de représenter un graphe sur un ordinateur :
  - Par une matrice d'adjacence
  - Par une liste d'adjacence
- Nous avons étudié le concept d'arbre avec racine, et les notions de profondeur, hauteur, degré (notion différente que celle d'un graphe en général), fils, père, feuille.
- Nous avons vu la définition de l'indice d'un sommet dans un arbre avec racine.
- Nous avons vu plusieurs façons de représenter un arbre sur un ordinateur :
  - Par une structure de pointeurs
  - Sous forme de tableau
  - De façon implicite
- Nous avons vu trois méthodes pour parcourir un arbre binaire : preorder, inorder et postorder.

3.5 STRUCTURES DE DONNÉES POUR DES ALGORITHMES DE RECHERCHE

Le problème que nous traiterons dans cette section consiste à trouver les éléments dans une base de données qui satisfont un certain critère (appelé le critère de recherche). Par exemple, la base de données pourrait être un dictionnaire, dans lequel nous voulons trouver la définitinon d'un mot. Il pourrait aussi s'agir d'un bottin, où nous voulons trouver le numéro de téléphone de quelqu'un (dont nous connaissons le nom). D'autres exemples :

- Une table de symboles (d'un compilateur). Étant donné le nom d'une variable, trouver les données correspondantes.
- Numéros de compte
- Données personnelles

Nous supposerons que l'information est déterminée de façon unique par une clé k. Pour l'exemple du bottin, la clé est le nom de la personne, et toutes les personnes auront, pour nous, des noms distincts.

Il y a deux différents types d'algorithmes de recherche. Les *algorithmes de recherche interne* travaillent sur une base de données qui est en mémoire (RAM) et accessible. Les *algorithmes de recherche externe* opèrent sur une base de données dont une partie des données se trouve sur un média secondaire.

Par exemple, la méthode de la recherche binaire, comme nous l'avons vu au chapitre 0 est un exemple typique d'un algorithme de recherche interne. Un autre exemple important est la recherche avec les arbres binaires que nous étudierons dans cette section plus en détail.

Un algorithme de recherche externe typique est la recherche en utilisant les arbres B (*B-trees*), mais ces arbres ne seront pas étudiés dans ce cours.

3.5.1 Arbres binaires de recherche

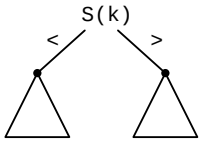
On peut stocker des données dans un arbre binaire en associant à chaque sommet une entrée de la base de données. Il est alors possible de retrouver une entrée arbitraire en faisant un parcours de l'arbre et en inspectant chaque sommet. Mais cette manière de procéder est $O(n)$ et donc très lente, et en effet l'intérêt des arbres est qu'on peut utiliser leur structure d'arbre si les entrées sont stockés de manière ordonnée.

Arbres de recherche

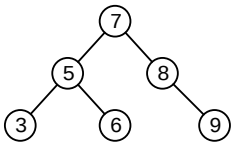
Définition. Un *arbre binaire de recherche* est un arbre ordonné binaire B muni d'une indexation des sommets $S\colon V \to M$, où M est un ensemble ordonné de clés (e.g., $M = \mathbb{N}_0$)

tel que pour tout $k \in V$, on a

$$S(k) > S(\ell) \text{ si } \ell \in \text{sous-arbre gauche de } k$$
$$S(k) < S(r) \text{ si } r \in \text{sous-arbre droit de } k.$$



Exemple: L'arbre suivant est un arbre binaire de recherche.



Les opérations que nous aimerions pouvoir effectuer rapidement dans un arbre de recherche sont :

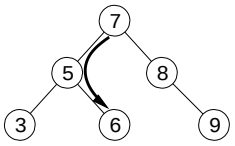
- `Search`(x) : Trouver la clé x ou montrer que x ne se trouve pas dans la base de données.
- `Insert`(x) : Insérer x dans l'arbre de recherche s'il n'existe pas encore.
- `Delete`(x) : Effacer x de l'arbre de recherche si x est un membre.

Sur un arbre binaire de recherche, ces algorithmes peuvent être réalisés comme suit :

Recherche (`Search`) : La recherche est une procédure récursive très simple.

- S'il s'agit de l'arbre vide, la recherche échoue (x ne se trouve pas dans l'arbre)
- Sinon on compare x avec la clé r de la racine :
  - Si $x = r$, on a fini ; on a trouvé le sommet r cherché.
  - Si $x < r$, on cherche pour x dans le sous-arbre de gauche.
  - Si $x > r$, on cherche dans le sous-arbre droite.

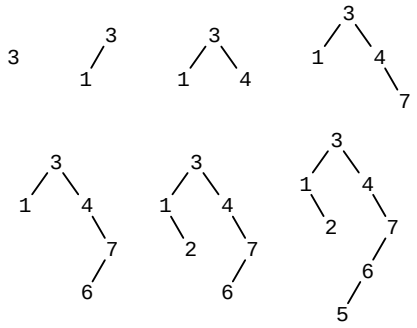
Par exemple, pour chercher la clé 6 dans l'arbre exemple ci-dessus, on commence à la racine et on visite, un après l'autre, les sommets comme indiqué par la flèche :



Insertion (`Insert`) : On commence en faisant une recherche de x. Si on trouve x alors qu'il est déjà dans l'arbre et on ne doit donc plus rien faire. Si on ne le trouve pas (i.e. la recherche echoue), l'algorithme se terminera sur une feuille. Appelons b la clé de cette feuille.

- Si $x < b$, on ajoute un sommet gauche de cette feuille, et on le munit de la clé x.
- Sinon, on ajoute un sommet droite à cette feuille, et on le munit de la clé x.

Exemple d'insertion : On insère $3, 1, 4, 7, 6, 2, 5$ dans un arbre initialement vide.

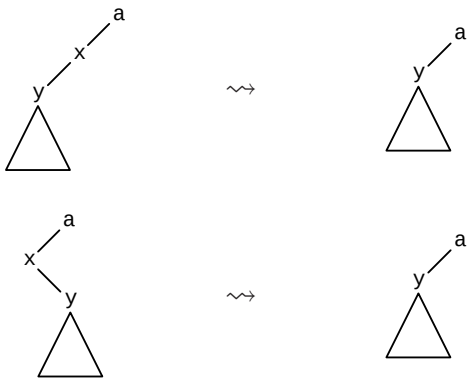


Effacement (`Delete`) : On commence comme pour `Search`(x). Si la recherche échoue, x ne se trouve pas dans l'arbre et nous n'avons donc plus rien à faire. Si x se trouve dans l'arbre, nous déterminons ainsi sa position.

Nous aimerions enlever x sans perdre la propriété de l'arbre de recherche.

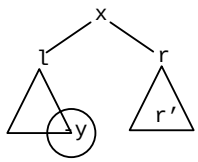
C'est facile si x est une feuille ou si x n'a qu'un seul fils, il suffit dans ce cas d'effacer le sommet correspondant, et de mettre l'unique fils à sa place (ou l'arbre vide s'il en a pas).

Illustration :



Dans le cas général, si x a deux fils, effacer est plus difficile. Écrivons ℓ la racine du sous-arbre gauche de x et r la racine du sous-arbre droite.

Soit y le sommet qu'on trouve en commençant en ℓ et en passant toujours au sous-arbre droite jusqu'à ce qu'il n'y en a plus.



Alors, comme y se trouve dans le sous-arbre gauche de x, nous avons $y < x$. D'autre part, y est le plus grand élément du sous-arbre gauche de x (pourquoi ?). Ayant trouvé y, nous pouvons effacer x comme suit :

`Delete`(x) : Remplacer x par y, et remplacer y par son sous arbre de gauche

En procédant de cette manière la propriété de l'arbre de recherche est préservée, comme $\ell < y < r'$ pour n'importe quel sommet r' du sous-arbre de droite.

Analyse du temps de parcours. Les temps de parcours des opérations considérées ci-dessus dépendent de la forme de l'arbre de recherche et de la position du sommet en question.

Dans le pire des cas, les temps de parcours sont proportionnels à la *hauteur* de l'arbre. La hauteur d'un arbre binaire avec N sommets est entre $\lfloor \log_2(N) \rfloor$ et $N-1$. Donc, le coût des opérations est $O(N)$ dans le pire des cas.

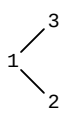
En ce sens, les arbres de recherche sont très similaires à des structures de données beaucoup plus simples telles que par exemple un tableau avec la recherche séquentielle : cette dernière structure permet aussi d'effectuer toutes ces opérations en $O(N)$.

N'aurions-nous donc rien gagné par rapport à des structures de données bien plus simples (telles qu'une liste liée avec recherche séquentielle) ? La réponse est qu'en pratique on ne s'intéresse pas seulement au comportement dans le pire des cas, mais aussi au comportement en moyenne. Nous verrons maintenant qu'en moyenne, les arbres de recherche se comportent en fait bien mieux que par exemple une liste liée pour ces opérations.

De quoi devrions-nous prendre la moyenne ?

Analyse de l'arbre aléatoire. Nous supposons que les $N!$ façons d'ordonner N clés ont toutes la même probabilité et nous considérons l'arbre de recherche obtenu en insérant les clés $1, 2, \ldots, N$ dans un ordre aléatoire dans un arbre vide. Nous prenons donc la moyenne sur tous les $N!$ arbres de recherche correspondant aux ordres possibles des clés.

Par exemple, si π est la permutation $\binom{123}{312}$, l'arbre de recherche correspondant est



$=: B(\pi)$.

Soit $B = (V, E)$ un arbre de recherche de sommets $v_1, \ldots, v_N$ et de clés correspondantes $1, 2, \ldots, N$. Nous définissons

$$\begin{aligned} A_B &= \sum_{k=1}^{N} \#\text{Recherches pour trouver } k \\ &= \sum_{k=1}^{N} (\text{profondeur}(v_k) + 1). \end{aligned}$$

Soit S_N l'ensemble de toutes les permutations sur N sommets, et pour $\pi \in S_N$ soit $B(\pi)$ l'arbre de recherche correspondant. Alors

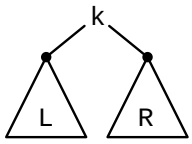
$$E_N := \frac{1}{N!} \sum_{\pi \in S_N} A_{B(\pi)}$$

est la moyenne du nombre total de recherches dans un arbre aléatoire de recherche. Est-il possible de trouver une formule plus explicite pour E_N ?

Pour E_0, E_1, nous avons

$$\begin{aligned} E_0 &= 0 \\ E_1 &= 1. \end{aligned}$$

Nous aimerions trouver une formule inductive pour E_N. L'illustration d'un arbre binaire de racine k suivante nous donne une idée :



Notons pour le moment $E_{N,k}$ la moyenne du nombre total de recherches dans un arbre de recherche de racine k. Nous avons

$$E_{N,k} = N + E_{k-1} + E_{N-k},$$

et donc, comme tout sommet $k \in \{1, \ldots, N\}$ a même probabilité d'être racine d'un arbre aléatoire de recherche à N sommets,

$$\begin{aligned} E_N &= \frac{1}{N} \left(\sum_{k=1}^{N} N + E_{k-1} + E_{N-k} \right) \\ &= N + \left(\sum_{k=0}^{N-1} E_k \right) + \left(\sum_{k=0}^{N-1} E_k \right) \\ &= N + \frac{2}{N} \sum_{k=1}^{N-1} E_k \end{aligned}$$

Pour $N+1$, la formule ci-dessus se réécrit

$$E_{N+1} = (N+1) + \frac{2}{N+1}\sum_{k=1}^{N} E_k.$$

Donc

$$\begin{aligned}
(N+1)E_{N+1} &= (N+1)^2 + 2\sum_{k=1}^{N} E_k \\
NE_N &= N^2 + 2\sum_{k=1}^{N-1} E_k \\
\Longrightarrow\ (N+1)E_{N+1} - NE_N &= 2N+1+2E_N
\end{aligned}$$

Cette dernière équation nous donne la formule récursive suivante :

$$\begin{aligned}
E_0 &= 0 \\
E_1 &= 1 \\
E_{N+1} &= \frac{N+2}{N+1}E_N + \frac{2N+1}{N+1}.
\end{aligned}$$

Écrivons

$$H_N := 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N}$$

le $N^{\text{ème}}$ *nombre harmonique*. Nous avons alors

$$E_N = 2 \cdot (N+1) \cdot H_N - 3N.$$

Or, pour H_N nous disposons de l'approximation suivante :

$$H_N = \ln N + \gamma + \frac{1}{2N} + O\left(\frac{1}{N^2}\right)$$

(où $\gamma = 0.5772\ldots$ est la *constante d'Euler*). Ceci nous donne pour E_N :

$$E_N = 2N\ln N - (3-2\gamma)N + 2\ln N + 1 + 2\gamma + O\left(\frac{1}{N^2}\right).$$

Enfin donc,

$$\frac{E_N}{N} \approx 1.386\ldots \cdot \log_2 N - (3-2\gamma) + \frac{2\ln N}{N} + \ldots$$

est le nombre moyen de recherches pour un sommet aléatoire dans un arbre binaire de recherche avec N sommets. Ceci prouve le théorème suivant :

Théorème 3.1 *Dans un arbre de recherche choisi aléatoirement, l'insertion et l'effacement peuvent être effectués en moyenne avec $O(\log_2(N))$ opérations.*

Nous insistons sur le fait que le théorème ci-dessus ne nous donne que le comportement d'arbres de recherche *en moyenne*. Dans le pire des cas, les temps de parcours peuvent être $O(N)$, ce qui est beaucoup plus grand.

Pour un arbre complet binaire (le meilleur cas), la moyenne correspondante est égale à $\frac{N+1}{N}\log_2(N+1) - 1$. (Sans preuve.) Si N est grand, nous avons $\frac{N+1}{N} \approx 1 < 1.39$, et donc l'opération de recherche prend en moyenne 40% de temps en plus dans un arbre aléatoire que dans un arbre complet.

3.5.2 Arbres bicolores

Bien qu'on puisse *en moyenne* faire des recherches dans un arbre binaire de façon rapide, ils se comportent très mal dans le *pire des cas*. En effet le temps de parcours d'une recherche dépend de la hauteur de l'arbre, qui sera $O(N)$ dans le pire des cas. Le problème est que si les éléments sont insérés dans un certain ordre (par exemple de façon croissante), la hauteur de l'abre grandit très vite.

Intuitivement, les arbres bicolores sont des arbres binaires de recherche dont la hauteur n'est pas trop grande, pour lesquels on peut ajouter et enlever des éléments en maintenant cette propriété.

Définition. Un *arbre bicolore* (*red-black tree*) est un arbre binaire de recherche dans lequel chaque sommet est attribué une couleur, soit rouge ou soit noir. De plus les conditions suivantes doivent être vérifiées :

1. La racine est noire.
2. Les fils d'un sommet rouge sont noirs.
3. Tout chemin d'un sommet à une de ses feuilles contient le même nombre de sommets noirs.

Remarquons que la deuxième condition est équivalente à dire que le père d'un sommet rouge est toujours noir.

Proposition 3 *Dans un arbre bicolore, le sommet le plus profond est au plus deux fois plus profond que le sommet le moins profond.*

Preuve : Puisque les fils d'un sommet rouge sont tous noirs (condition 2 dans la définition d'un arbre bicolore), un chemin de la racine à une feuille ne peut pas contenir deux sommets rouges de suite. Si un tel chemin contient m sommets noirs, il contiendra donc au plus $2m$ sommets au total (en alternant la couleur des sommets le long du chemin).

Nous avons de plus (condition 3) que tous les chemins de la racine à une feuille contiennent le même nombre de sommets noirs. Si nous appelons m ce nombre, nous voyons qu'un tel chemin contiendra au total au moins m et au plus $2m$ sommets. Puisque la profondeur d'un sommet est la longueur du chemin de la racine à ce sommet, le résultat suit immédiatement. ■

Cette propriété garantit que la hauteur d'un arbre bicolore n'est pas trop grande.

Insertion

Nous commencer par ajouter le sommet de la même façon que dans un arbre binaire normal, et le colorons en rouge. Soient x le sommet ajouté, soit y sont père, soit z sont grand-père, et soit u sont oncle.

- **Cas 1 :** x est la racine. Nous colorons x en noir.
- **Cas 2 :** y est noir. Nous ne faisons rien.
- **Cas 3 :** y est rouge, et u est rouge. Nous colorons y et u en noir.
- **Cas 4 :** y est rouge, u est noir, et x est le fils de droite de y. Nous faisons une *rotation à gauche*.
- **Cas 5 :** y est rouge, u est noir, et x est le fils de gauche de y. Nous faisons une *rotation à droite*.

Nous devons à présent vérifier que dans chaque cas, les 3 propriétés de la définition ci-dessus restent vraient après l'insertion. Il est clair que la propriété 1 restera vraie. Nous regardons à présent les deux propriétés restantes :

- **Cas 1 :** L'arbre n'a qu'un seul sommet noir, toutes les propriétés sont donc vérifiées.
- **Cas 2 :** Nous avons ajouté un sommet dont le père est noir, la propriété 2 reste donc vraie. De plus, le chemin d'un sommet quelconque t vers x aura le même nombre de sommets noirs que le chemin de t vers y, la propriété 3 est donc toujours vérifiée.
- **Cas 3 :**
- **Cas 4 :**
- **Cas 5 :**

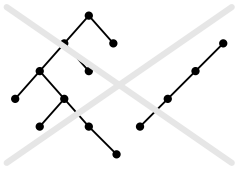
Effacement

- **Cas 1 :** x est la racine. Nous ne faisons rien
- **Cas 2 :** z est rouge. Nous colorons z en noir, y en rouge puis faison une roation à gauche en y.
- **Cas 3 :**
- **Cas 4 :**
- **Cas 5 :**

3.5.3 Arbres AVL

Les arbres de recherche binaires se comportent très mal dans le pire des cas, parce qu'ils peuvent être très penchés.

Un arbre binaire est dit *équilibré* (*balanced*), ou appelé *arbre AVL* (d'après Adelson, Velskij et Landis), si pour tout sommet k les hauteurs des sous-arbres gauches et droits diffèrent par au plus 1.



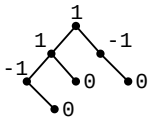
Définition. Soit T un arbre binaire avec racine. Soit v_k un sommet de T. Le *facteur d'équilibre* (*balance factor*) de v_k est défini comme suit :

$$bal(v_k) = (\text{hauteur du sous arbre de gauche}) - (\text{hauteur du sous arbre de droite}),$$

où nous supposons qu'un sous arbre vide a hauteur -1.

Définition. Un arbre AVL est un arbre binaire avec racine dans lequel le facteur d'équilibre de chaque sommets vaut -1, 0, ou 1.

Exemple: Le graphique suivant représente un arbre AVL dans lequel les sommets sont indexés par leurs facteurs d'équilibre respectifs.



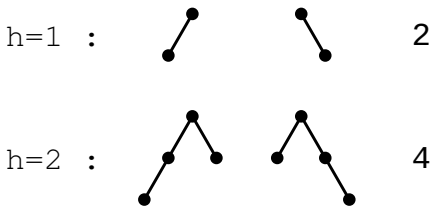
La propriété principale d'un arbre AVL est que sa hauteur ne peut croître que logarithmiquement avec le nombre de sommets. Un arbre AVL avec N sommets permet donc de chercher, d'insérer et d'effacer en $O(\log N)$ opérations dans le pire des cas.

Théorème 3.2 *La hauteur d'un arbre AVL avec n sommets vérifie l'inégalité suivante :*

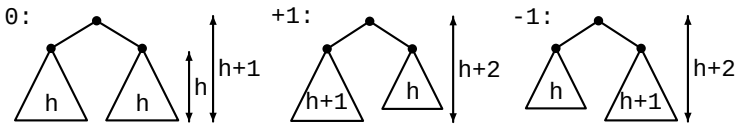
$$h < 1.4404 \log_2(n) + 0.672.$$

Preuve. Remarquons d'abord que les sous-arbres d'arbres AVL sont eux-mêmes des arbres AVL. Quel est le nombre minimal de sommets dans un arbre AVL de hauteur h ?

Pour $h = 1, 2$ nous avons :



En général, il y a trois cas : bal(racine) $= -1, 0, 1$.



Le nombre minimum A_h de sommets dans un arbre AVL de hauteur h vérifie

$$\begin{aligned} A_{h+1} &= 1 + \min\left(2A_h, A_h + A_{h-1}\right) \\ &= 1 + A_h + A_{h-1}. \end{aligned}$$

Nous prouvons par induction que

$$A_h > \frac{1+\sqrt{5}}{2\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^h. \tag{3.2}$$

[**Base**] Pour $h = 1$, on a $A_h = 2 > \frac{(1+\sqrt{5})^2}{4\sqrt{5}}$.

[**Pas**] Posons $c := \frac{1+\sqrt{5}}{2\sqrt{5}}$.

$$\begin{aligned} A_{h+1} &= 1 + A_h + A_{h-1} \\ &> 1 + \frac{1+\sqrt{5}}{2\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^h + \frac{1+\sqrt{5}}{2\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^{h-1} \\ &= 1 + c \left(\frac{1+\sqrt{5}}{2}\right)^h \left(1 + \frac{2}{1+\sqrt{5}}\right) \\ &= 1 + c \left(\frac{1+\sqrt{5}}{2}\right)^h \frac{3+\sqrt{5}}{1+\sqrt{5}} \\ &> c \left(\frac{1+\sqrt{5}}{2}\right)^{h+1}. \end{aligned}$$

Donc par induction (3.2) est vraie.

Si on pose $b = \frac{1+\sqrt{5}}{2}$, on sait à présent (par (3.2)) qu'un arbre AVL de hauteur h doit avoir au moins $c \cdot b^h$ sommets. Supposons maintenant que nous avons un arbre AVL avec n sommets, et soit h sa hauteur, par la phrase précédente on doit avoir

$$\begin{aligned} c \cdot b^h < n &\iff b^h < \tfrac{n}{c} \\ &\iff h \cdot \log(b) < \log(n) - \log(c) \\ &\iff h < \tfrac{\log(n)}{\log(b)} - \tfrac{\log(c)}{\log(b)} \\ &\implies h < 1.4404 \log_2(n) + 0.672 \end{aligned}$$

■

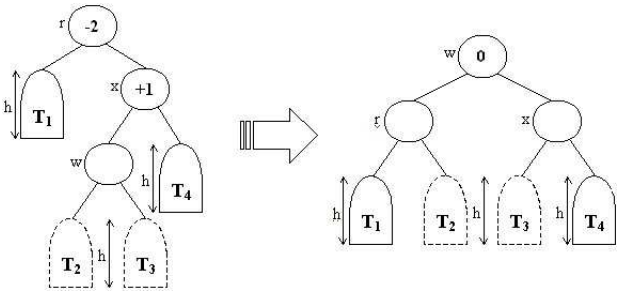
Ce théorème nous dit que la hauteur d'un arbre AVL avec N sommets est $O(\log(N))$. L'opération `Search` coûte donc $O(\log(N))$, même dans le pire des cas. Cependant, avec les méthodes d'insertion et d'effacement que nous avons vues pour les arbres binaires de recherche, si nous ajoutons ou effaçons un élément d'un arbre AVL il ne sera plus nécessairement AVL.

Si nous pouvions insérer et effacer des sommets d'un arbre AVL avec N sommets en temps $O(\log(N))$ *en maintenant la condition AVL*, alors nous aurions une structure de données qui permettrait d'effectuer les opérations d'insertion, d'effacement et de recherche en $O(\log(N))$, dans le pire des cas. De telles méthodes d'insertion et d'effacement existent, elles sont décrites ci-dessous :

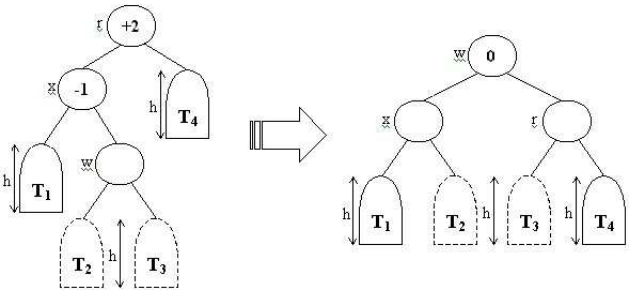
Insertion : On commence par insérer le sommet de la même manière que dans un arbre de recherche binaire. Ensuite, l'algorithme remonte du sommet inséré jusqu'à la racine en vérifiant l'équilibre à chaque sommet du chemin. Si un sommet non-équilibré est rencontré (i.e., tel que le facteur d'équilibre est égal à +2 ou -2), alors une *rotation* est effectuée. Le type de rotation à effectuer dépend de la position du sommet inséré par rapport au sommet non-équilibré.

Toutes les configurations possibles peuvent se classer en 4 cas, pour chacun desquels il existe une rotation qui rééquilibre le sommet. Les 4 rotations sont appelées RL, LR, RR et LL :

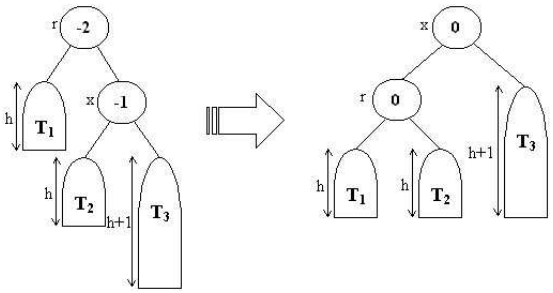
Rotation RL :



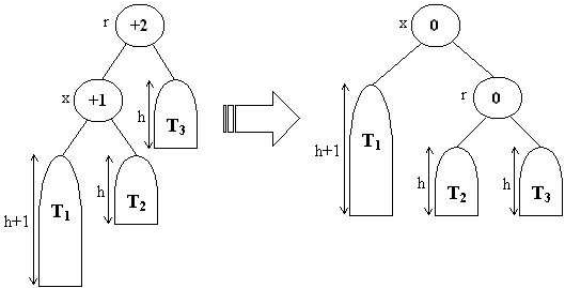
Rotation LR :



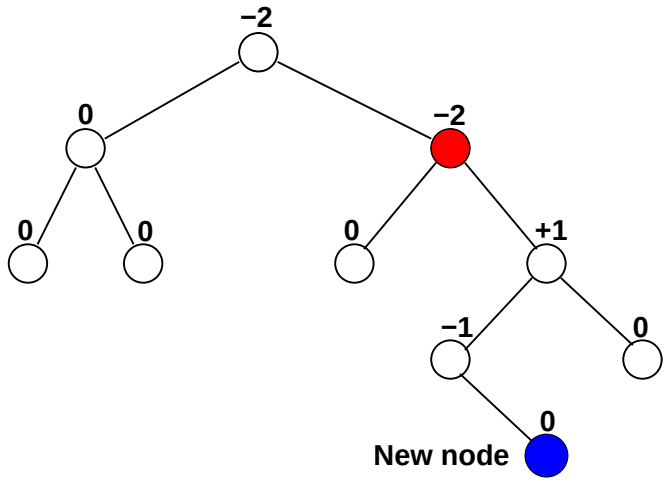
Rotation RR :



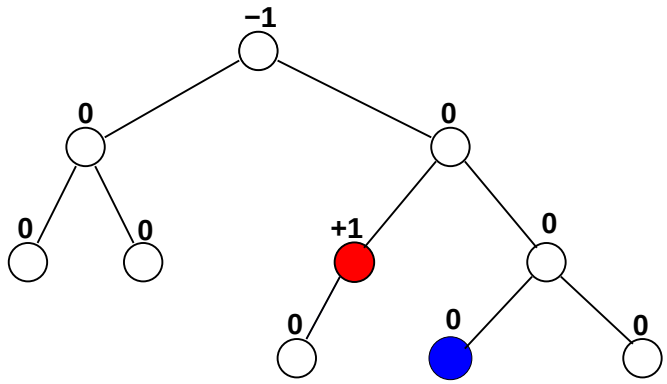
Rotation LL :



Exemple: Voici une illustration de la procédure de rééquilibrage. Le sommet bleu ("new node") vient d'être ajouté, et l'arbre n'est donc plus AVL. Il faut effectuer une rotation pour le rééquilibrer.

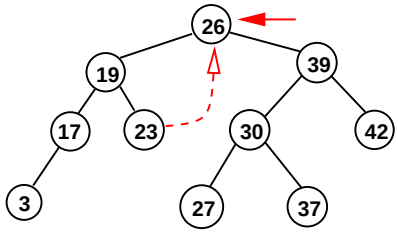


Nous suivons le chemin du sommet ajouté à la racine jusqu'au prochain sommet non-équilibré (dans l'exemple le sommet rouge, qui a comme facteur d'équilibre -2). Ce sommet subira une rotation RL :

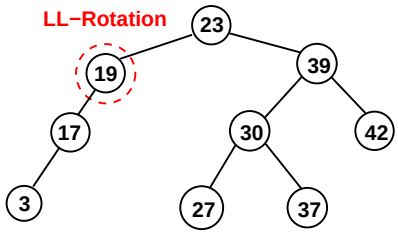


Effacement : L'algorithme d'effacement est un peu plus compliqué. On commence par effacer le sommet comme dans un arbre binaire de recherche arbitraire (c'est-à-dire non AVL). Comme pour l'insertion, l'arbre doit ensuite être rééquilibré en utilisant des rotations. Nous illustrons cette procédure à l'aide d'un exemple.

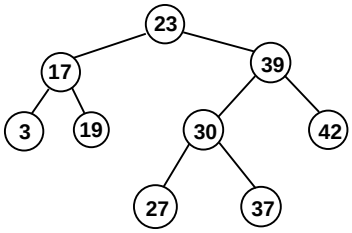
Exemple : Dans l'arbre ci dessous supposons que nous voulons effacer le sommet dont la clé vaut 26 (que nous appelons simplement le sommet 26). L'algorithme d'effacement que nous avons déjà vu (pour les arbres non AVL) enlève le sommet 26 et le remplace par le sommet 23 :



Nous voyons ensuite que l'arbre n'est plus équilibré (il n'est plus AVL), puisque le facteur d'équilibre du sommet 19 vaut +2.



On voit maintenant, en regardant les diagrames ci-dessus que cette configuration peut être rééquilibrée avec une rotation LL :



Quelques remarques :

- Contrairement au cas d'insertion, il est possible que plusieurs rotations soient nécessaires pour le rééquilibrage après un effacement.
- Le coût des opérations d'insertion et d'effacement est $O(\log(N))$, où N est le nombre total de sommets. Nous ne prouverons pas ce résultat.
- Remarquons pour terminer qu'une implémentation efficace d'un arbre AVL nécessite qu'on stocke quelque chose de plus dans chaque sommet, à savoir le facteur d'équilibre : On ne peut pas se permettre de le recalculer à chaque fois qu'on vérifie l'équilibre.

3.6 LE HACHAGE (*HASHING*)

Nous avons vu des méthodes pour stocker des données permettant une implémentation efficace des operations de recherche, d'insertion et d'effacement. Dans ces méthodes, chaque

élément est identifié par une clé unique. Cependant, seul un petit sous-ensemble K de l'ensemble $\mathbb{K}$ de toutes les clés possibles n'est utilisé à un moment donné.

Les méthodes vues jusqu'à présent procèdent en comparant les clés des éléments. Par contre, les *méthodes de hachage* (*hash methods*) tentent de trouver un élément en calculant sa position à partir de sa clé k. Les éléments sont stockés dans un tableau (array) linéaire avec des indices $0, 1, \ldots, m-1$, qu'on appelle la *table de hachage* (*hash table*).

Définition. La *fonction de hachage* (*hash function*) est une application

$$h\colon \mathbb{K} \to \{0, 1, \ldots, m-1\},$$

qui fait correspondre à chaque clé k un indice $h(k)$, appelé l'*adresse de hachage* (*hash address*).

Donc pour savoir où stocker un élément avec clé k dans la table de hachage, il suffit de calculer $h(k)$ qui nous donnera l'indice du tableau où mettre notre élément. De même pour rechercher un élément on calcule son adresse de hachage $h(k)$ puis on regarde la position correspondante dans la table de hachage.

Cependant, m est en général beaucoup plus petit que $\mathbb{K}$. La fonction de hachage n'est donc en général pas injective, c'est-à-dire que plusieurs clés peuvent avoir la même adresse de hachage.

Définition. Soit h une fonction de hachage, et soient $k, k' \in \mathbb{K}$. Si

$$k \neq k' \text{ et } h(k) = h(k').$$

alors on dit qu'on a une *collision d'adresse* (*address collision*), et k et k' sont appelés des *synonymes*.

Une bonne méthode de hachage doit avoir les propriétés suivantes :
- Elle doit donner aussi peu de collisions d'adresse que possible
- Les collisions doivent être résolues de manière efficace
- Il doit être possible de calculer l'indice de hachage $h(k)$ d'une clé de manière efficace.

Même les meilleures fonctions de hachage ne peuvent pas éviter les collisions. Les méthodes de hachage sont donc très inefficaces pour l'insertion et l'effacement dans le pire des cas. Cependant elles sont *en moyenne* beaucoup meilleures que les méthodes qui utilisent des comparaisons de clés. Par exemple, le temps de recherche en utilisant le hachage est indépendant du nombre d'éléments stockés.

Les méthodes de hachage doivent plutôt être utilisées quand les insertions sont toutes faites initialement, et que les opérations sont ensuite presque exclusivement des recherches, et sans que des éléments ne soient effacés.

Les méthodes de hachage introduites dans ce cours sont seulement *semi-dynamiques*, puisque les tables de hachage sont fixées au début.

Nous nous intéressons aux temps de parcours *moyens* des opérations de recherche, d'insertion et d'effacement, puisque les temps de parcours dans les pires des cas sont très mauvais.

Définition. Pour mesurer la performance d'une méthode de hachage on définit :

C_n = Espérance du nombre d'entrées visitées dans une table de hachage lors d'une recherche *réussie*

C'_n = Espérance du nombre d'entrées visitées dans une table de hachage lors d'une recherche qui *échoue*

Nous indiquerons toujours les distributions de probabilité qui nous intéressent.

Une bonne fonction de hachage doit avoir les propriétés suivantes :

- Elle doit être calculable de manière efficace,
- Elle doit distribuer les éléments de manière uniforme, même si les clés ne le sont pas (par exemple des noms de variables dans un programme)

Même dans le meilleur des cas il est impossible d'éviter les collisions.

Birthday Lemma. Si $q \gtrsim 1.78\sqrt{|M|}$, alors la probabilité qu'une fonction aléatoire uniforme

$$f\colon \{1, 2, \ldots, q\} \to M$$

soit injective est inférieure à $1/2$.

Corollaire. Si plus de 23 personnes sont dans une pièce, alors la probabilité qu'au moins deux d'entre elles aient le même anniversaire est supérieure à $1/2$.

Preuve du Birthday Lemma : On pose $m = |M|$. La probabilité que la fonction soit injective est

$$\begin{aligned} \tfrac{m\cdot(m-1)\cdots(m-q+1)}{m\cdot m\cdots m} &= \left(1-\tfrac{1}{m}\right)\cdots\left(1-\tfrac{q-1}{m}\right) \\ &< e^{-\frac{q(q-1)}{2m}} \end{aligned}$$

Si $q \geq \frac{1+\sqrt{1+8\ln(2)}}{2}\cdot\sqrt{m}$, on a

$$e^{-\frac{q(q-1)}{2m}} < \frac{1}{2}.$$

Remarquons que $\frac{1+\sqrt{1+8\ln(2)}}{2} \sim 1.78$.

Nous supposons à partir de maintenant que l'ensemble $\mathbb{K}$ de toutes les clés possibles est un sous-ensemble de $\mathbb{N}$. Si les clés nous sont données, par exemple sous la forme d'une liste de caractères (comme le nom d'une variable en C), nous pouvons interpréter les codes ASCII des caractères comme des entiers positifs et ainsi faire correspondre à chaque liste de caractères un entier.

La méthode de division avec reste

Dans cette méthode on choisit une valeur m qui sera la taille de la table de hachage. On définit la fonction de hachage $f\colon \mathbb{K} \to \{0, 1, \ldots, m-1\}$ comme suit :

$$h(k) := k \bmod m.$$

On a donc bien $h(k) \in \{0, 1, \ldots, m-1\}$.

Le choix de m est crucial. Par exemple, si m est pair, alors le bit le moins signifiant de k et $h(k)$ est le même (est en collision). On aura donc plus de collisions pour les éléments qui ont le même bit le moins signifiant. Avec le même raisonnement, on voit que m ne doit pas être une puissance de 2 puisque dans ce cas les éléments avec les mêmes bits les moins signifiants seraient en collision. Plus concrètement, si on choisissait $m = 2^\ell$, alors les ℓ bits les moins signifiants de k et de $h(k)$ seront les mêmes.

On choisit d'habitude pour m une puissance d'un nombre premier qui ne s'exprime pas sous la forme $2^\ell \pm j$ pour un certain ℓ et un certain j petit.

La méthode multiplicative

La clé est multipliée par un nombre irrationel θ, puis on regarde la partie fractionnaire (après la virgule) du résultat.

On obtient ainsi des résultats différents pour différentes valeurs entre 0 et 1 :

Si $k\theta - \lfloor k\theta \rfloor = \ell\theta - \lfloor \ell\theta \rfloor$, alors $(k-l)\theta \in \mathbb{Z}$. Et puisque θ est irrationel, on a $k - \ell = 0$.

Le Théorème suivant (de *Vera Turán Sós*) montre que pour les clés $1, 2, 3, \ldots, n$ ces valeurs sont assez uniformément distribuées dans l'intervalle $[0, 1)$:

Théorème 3.3 *Pour un θ irrationnel, les sous-intervalles formés par les nombres*

$$\theta - \lfloor \theta \rfloor, 2\theta - \lfloor 2\theta \rfloor, \ldots, n\theta - \lfloor n\theta \rfloor$$

ont au plus trois *longeurs différentes. De plus, le prochain point $(n+1)\theta - \lfloor (n+1)\theta \rfloor$ sera dans un de ces intervalles.*

Parmi tous les θ avec $0 \leq \theta \leq 1$ le *nombre d'or*

$$\phi^{-1} \;=\; \frac{\sqrt{5}-1}{2} \;\approx\; 0,6180339887$$

donne la distribution la plus uniforme. On obtient la fonction de hachage

$$h(k) \;:=\; \left\lfloor m\left(k\phi^{-1} - \left\lfloor k\phi^{-1} \right\rfloor\right)\right\rfloor$$

En particulier, la suite $h(1), h(2), \ldots, h(10)$ pour $m = 10$ est simplement une permutation des entiers $0, 1, \ldots, 9 : 6, 2, 8, 4, 0, 7, 3, 9, 5, 1$.

Lum, Suen, et Dodd ont étudié le comportement de nombreuses fonctions de hachage, et ont réalisé que la méthode de division avec reste donne les meilleurs résultats en moyenne.

Dans les discussion qui suivent sur la résolution de collisions, nous supposerons donc toujours que c'est cette méthode de hachage qui est utilisée.

Les Strategies pour la Résolution de Collisions

Supposons que nous avons une table de hachage qui contient la clé k. Si nous voulons insérer dans cette table un synonyme k' de k (i.e. une clé avec $h(k) = h(k')$), nous aurons alors une collision d'adresse qu'il faudra résoudre.

La position de $h(k) = h(k')$ est déjà occupée et la nouvelle clé doit donc être stockée autre part. De tels synonymes peuvent être stockés hors de la table de hachage, dans des listes dynamiques. Par exemple, les synonymes d'une clé peuvent être maintenus dans une liste liée linéaire. Cette liste est connectée à la position de la table de hachage donnée par la valeur de hachage des clé synonymes.

Exemple : Soit

$$m = 7$$

la taille de la table de hachage (donc le nombre d'addesses de hachage), soit

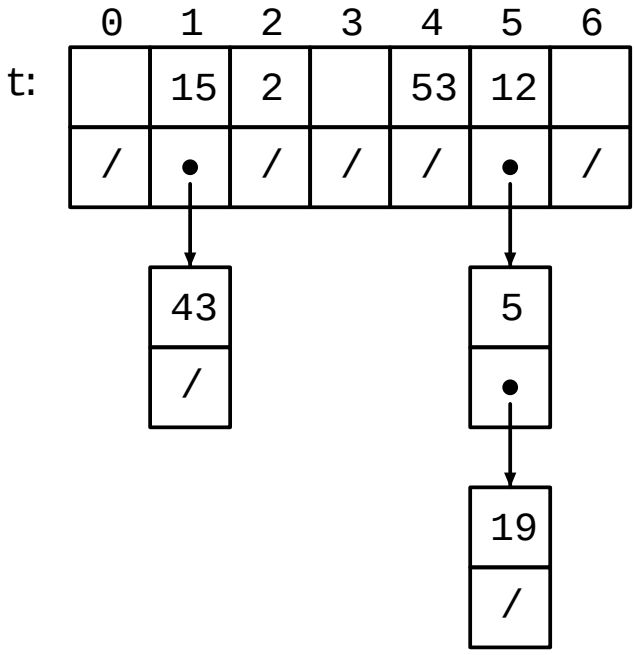
$$\mathbb{K} \;=\; \{0, 1, \ldots, 500\}$$

l'ensemble des clés possibles, et soit

$$h(k) \;=\; k \bmod m$$

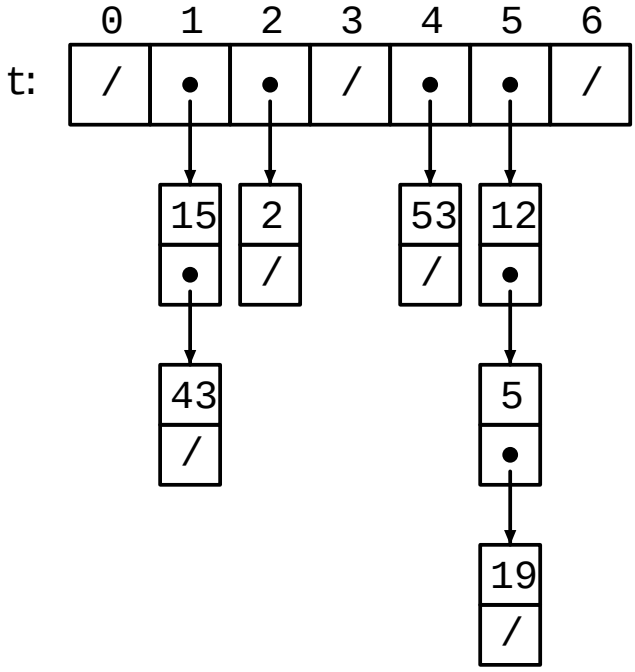
la fonction de hachage.

Après avoir entré les clés $12, 53, 5, 15, 2, 19, 43$, la table de hachage est dans l'état suivant :



Avec cette méthode chaque élément de la table de hachage est le début d'une liste liée.

La méthode d'adresse directe est plus facile à implémenter, mais nécessite plus de mémoire :



Nous analysons à présent la méthode d'adresse directe. Nous supposons :

– La fonction de hachage retourne les valeurs de hachage avec la même probabilité (*supposition de distribution uniforme*)

– Les valeurs de la fonction de hachage sont indépendantes pour chaque opération (*supposition d'indépendance*)

La deuxième supposition nous dit que la $j^{\text{ème}}$ opération choisit toujours l'adresse j' avec probabilité $1/m$, quel que soit j,

Définition. Si n est le nombre de clés possibles et m la taille de la table de hachage, alors le *facteur d'occupation* est défini comme suit :

$$\alpha := \frac{n}{m}.$$

Analyse d'une recherche qui échoue

Dans ce cas, l'adresse $h(k)$ est calculée, et tous les éléments dans la liste de synonymes sont traversés.

La longueur moyenne de chaque liste est $n/m = \alpha$, et donc

$$C'_n = \alpha$$

Analyse d'une recherche qui réussit

Supposons qu'il faille visiter visiter i éléments pour trouver la clé recherchée, i.e., le coût de trouver la clé est i.

Quand nous ajoutons la $j^{\text{ème}}$ clé, la longueur moyenne de la liste est $(j-1)/m$; Donc, en cherchant la $j^{\text{ème}}$ clé nous n'avons besoin de visiter que $1 + (j-1)/m$ éléments de la liste, si les insertions dans la liste ne sont faites qu'à la fin, et qu'il n'y a aucun effacement. Le nombre moyen de clés visitées dans le cas d'une recherche réussie est donc

$$\begin{aligned} C_n &= \frac{1}{n}\sum_{j=1}^{n}\left(1+\frac{j-1}{m}\right) \\ &= 1+\frac{n-1}{2m} \\ &\approx 1+\frac{\alpha}{2} \end{aligned}$$

si chaque clé est recherchée avec la même probabilité.

4

Construction d'algorithmes par induction

La méthode d'induction peut être utilisée pour construire des algorithmes, en réduisant la resolution d'une instance d'un problème à la resolution d'une ou plusieurs instances plus petites du même problème.

Deux points sont essentiels dans la conception d'algorithme par induction :

- Il est possible de résoudre une *petite* instance du problème : *le cas de base*.
- Une solution de chaque problème peut être construite à partir de solutions de *plus petits* problèmes : *le pas d'induction*.

Nous verrons des exemples de cette méthode, et introduirons plusieurs classes d'algorithmes basées sur cette méthode, telles que la programmation dynamique et les algorithmes diviser-pour-régner (*divide-and-conquer*).

4.1 LA MÉTHODE DE HORNER

La méthode de Horner est un algorithme efficace pour évaluer des polynômes *génériques* en un point donné. Le problème qui nous intéresse est donc le suivant :

Problème: Evaluation de Polynôme
Input: Polynôme $f(x) = \sum_{i=0}^{n} f_i x^i \in \mathbb{R}[x]$, et $\alpha \in \mathbb{R}$.
Output: $f(\alpha)$.

L'algorithme qui suit présente une méthode naïve pour évaluer un polynôme en un point donné.

Si nous utilisons la méthode naïve pour calculer α^i à la ligne 3, alors l'algorithme a besoin de $O(n^2)$ opérations sur des nombres réels. Si nous utilisons la méthode binaire à la ligne 3, alors l'algorithme a besoin de $O(n \log(n))$ opérations sur des nombres réels.

Algorithme 18 NAÏF$(f(x), \alpha)$

1: $s \leftarrow 0$ et $i \leftarrow 0$.
2: **while** $i \leq n$ **do**
3: Calculer $\beta \leftarrow \alpha^i$.
4: $s \leftarrow s + f_i * \beta$.
5: $i \leftarrow i + 1$.
6: **end while**
7: **return** s

Pouvons nous faire mieux ? *Oui*. La méthode de Horner simplifie le problème en utilisant l'observation suivante :

$$f(\alpha) = f_0 + \alpha g(\alpha),$$

où $g(x) = f_1 + f_2 x + \cdots + f_n x^{n-1}$.

Nous utilisons donc le fait que $f(\alpha)$ est égal à

$$((\cdots((f_n\alpha + f_{n-1})\alpha + f_{n-2})\alpha\cdots)\alpha + f_1)\alpha + f_0.$$

En pseudo-code, nous décrivons la méthode de Horner comme suit :

Algorithme 19 HORNER$(f(x), \alpha)$

1: $s \leftarrow 0$ and $i \leftarrow n$.
2: **while** $i \geq 0$ **do**
3: $s \leftarrow s * \alpha + f_i$.
4: $i \leftarrow i - 1$.
5: **end while**
6: **return** s

Sous cette forme nous voyons immédiatement qu'il faut à cet algorithme $n+1$ additions et $n+1$ multiplications pour calculer $f(\alpha)$ (nous ne comptons pas les opérations sur le compteur i), ainsi nous avons un algorithme $O(n)$.

La méthode de Horner est-elle optimale ? La réponse est oui, pour des polymômes génériques (preuve difficile). Pour des polynômes spécifiques, la méthode de Horner peut ne pas être optimale (on peut, par exemple, évaluer le polynôme $f(x) = x^n$ plus rapidement qu'en $O(n)$ opérations.)

4.2 ELIMINATION DE GAUSS

Supposons que nous voulons résoudre un système d'équations linéaires. Formellement nous avons :

Problème: Système d'équations linéaires
Input: Matrice $A \in \mathbb{R}^{n\times n}$, vecteur $b \in \mathbb{R}^n$
Output: Vecteur $x \in \mathbb{R}^n$ tel que $A \cdot x = b$.

L'algorithme pour résoudre un tel système est très utilisé en algèbre linéaire. On l'appelle la *méthode du pivot de Gauß*. Nous la présentons ici dans le cadre des algorithmes construits par induction en insistant sur le caractère inductif de l'algorithme.

Nous pouvons écrire la matrice A par blocs de la manière suivante :

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

où $a_{11} \in \mathbb{R}$, $A_{12} \in \mathbb{R}^{1\times(n-1)}$, $A_{21} \in \mathbb{R}^{(n-1)\times 1}$, et $A_{22} \in \mathbb{R}^{(n-1)\times(n-1)}$.

Nous appelons a_{11} le pivot, et supposons que $a_{11} \neq 0$. Soit I_m la matrice identité de dimension m. Alors

$$\overbrace{\left(\begin{array}{c|c} 1 & 0 \\ \hline -\frac{A_{21}}{a_{11}} & I_{n-1} \end{array}\right)}^{X} \cdot A \cdot \overbrace{\left(\begin{array}{c|c} 1 & -\frac{A_{12}}{a_{11}} \\ \hline 0 & I_{n-1} \end{array}\right)}^{Y} = \overbrace{\left(\begin{array}{c|c} a_{11} & 0 \\ \hline 0 & \hat{A} \end{array}\right)}^{Z}, \tag{4.1}$$

où

$$\hat{A} = A_{22} - A_{21} \cdot a_{11}^{-1} \cdot A_{12}. \tag{4.2}$$

Cette matrice $\hat{A}$ est appelée le *complément de Schur* de A par rapport à a_{11}. En utilisant X, Y et Z definis ci-dessus nous avons :

$$\begin{array}{lll} A \cdot x = b & \Longleftrightarrow & X \cdot A \cdot x = X \cdot b \\ & \Longleftrightarrow & (X \cdot A \cdot Y) \cdot Y^{-1} \cdot x = X \cdot b \\ & \Longleftrightarrow & Z \cdot Y^{-1} \cdot x = X \cdot b, \qquad \text{(avec (4.1) ci dessus)} \end{array}$$

et nous obtenons donc le système d'équations suivant :

$$\left(\begin{array}{c|c} a_{11} & 0 \\ \hline 0 & \hat{A} \end{array}\right) \cdot \left(\begin{array}{c|c} 1 & \frac{A_{12}}{a_{11}} \\ \hline 0 & I_{n-1} \end{array}\right) \cdot x = \left(\begin{array}{c|c} 1 & 0 \\ \hline -\frac{A_{21}}{a_{11}} & I_{n-1} \end{array}\right) \cdot b. \tag{4.3}$$

Definissons à présent

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad y = \begin{pmatrix} x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}, \quad c = \begin{pmatrix} b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}.$$

En decomposant (4.3) nous voyons que

$$x_1 = \frac{1}{a_{11}} (b_1 - A_{12} y) \tag{4.4}$$

$$\hat{A} \cdot y = c'. \tag{4.5}$$

où

$$c' = -\frac{b_1}{a_{11}} A_{21} + c. \tag{4.6}$$

Ces identités nous mènent à l'algorithme suivant pour calculer x :

Algorithme 20 RÉSOUDRE(A, b, n)

```
1: if n = 1 then
2:     x_1 ← b_1 / a_11
3:     return (x_1)
4: else
5:     Calculer Â et c'
6:     y ← RÉSOUDRE(Â, c', n − 1)
       (donc trouver y = (x_2, ..., x_n) tel que Â · y = c')
7:     s ← A_12 · y
8:     x_1 ← (1/a_11)(b_1 − s)
9:     return (x_1, x_2, ..., x_n)
10: end if
```

Analyse

Soit C_n le nombre d'opérations arithmétiques qu'il faut à cet algorithme pour résoudre un système de n équation à n inconnues (donc quand A est une matrice $n \times n$). Nous regardons chaque étape de l'algorithme :

• Calcul de $\hat{A}$ (ligne 5) : Nous savons que $\hat{A} = A_{22} - A_{21}a_{11}^{-1}A_{12}$ (voir (4.2)). A_{12}, A_{21} et A_{22} sont des sous-matrices de A elles sont donc toutes connues. Nous commençons par multiplier A_{12} par a_{11}^{-1}, il faudra $(n-1)$ multiplications (puisque A_{12} est un vecteur à $(n-1)$ entrées). Ensuite nous multiplions le résultat par A_{21}, il faudra $(n-1)^2$ multiplications (le résultat est une matrice $(n-1) \times (n-1)$). Finalement nous soustrayons le résultat à A_{22}, il faudra $(n-1)^2$ soustractions. Le nombre d'opération total est donc

$$(n-1) + (n-1)^2 + (n-1)^2 = 2(n-1)^2 + (n-1)$$

• Calcul de c' (ligne 5) : Nous savons que $c' = -\frac{b_1}{a_{11}}A_{21} + c$ (void (4.6)). Nous commençons donc par calculer $-\frac{b_1}{a_{11}}$ (1 division). Puis nous multiplions le résultat par le vecteur A_{21}, il faut $n-1$ multiplications. Finalement nous ajoutons le résultat à c ($n-1$ additions). Le nombre d'opération total est donc

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

• Calcul de y (ligne 6) : Il faut résoudre un système de $n-1$ équations à $n-1$ inconnues. Nous utilisons de nouveau le pivot de Gauß (c'est ici que l'*induction* est utilisée dans la construction), il faut donc C_{n-1} opérations.

• Calcul de s (ligne 7) : Nous multiplions le vecteur ligne A_{12} par le vecteur colonne y, c'est un produit scalaire de vecteurs à $n-1$ entrées, il faut donc $n-1$ multiplications et $n-2$ additions, un total de $2n-3$ opérations.

• Calcul de x_1 (ligne 8) : Il faut une soustraction et une division, donc 2 opérations.

Nous avons alors :

$$\begin{aligned} C_n &= C_{n-1} + 2n^2 + n - 1 \\ &= C_{n-1} + O(n^2). \end{aligned}$$

Nous pouvons en déduire que $C_n = O(n^3)$ (exercice).

Remarquons cependant qu'il est possible que l'algorithme ne puisse pas être appliqué de façon récursive puisque la condition $a_{11} \neq 0$ n'est pas forcément vérifiée. Il se peut qu'un changement de pivot doive être réalisé afin d'obtenir cette condition. On démontre en algèbre linéaire que ceci est toujours réalisable si la matrice A est non-singulière. Il s'agit de calculs auxiliaires qui ne changent pas l'ordre du temps de calcul.

4.3 LES ALGORITHMES DIVISER-POUR-RÉGNER

L'idée des algorithmes diviser-pour-régner (algorithmes DPR, *divide-and-conquer*) est de décomposer le problème en deux ou plusieurs sous-problèmes *indépendants* qui peuvent

ensuite être résolus séparément. Les solutions de ces sous-problèmes sont ensuite assemblées (c'est le *pas de combinaison*), de façon à obtenir une solution au problème original.

La technique diviser-pour-régner est très utile. Nous avons déjà rencontré quelques algorithmes de ce type dans ce cours : Les algorithmes de Karatsuba et de Strassen sont des algorithmes DPR. Nous étudierons le paradigme général des algorithmes DPR avant de voir quelques exemples.

4.3.1 Le paradigme DPR général

Le concept général est le suivant : étant donné une instance du problème à résoudre, la décomposer en plusieurs plus petites sous-instances (du même type), les résoudre indépendamment, et ensuite combiner les solutions des sous-instances pour obtenir une solution du problème original.

Cette description nous amène à une question importante : comment résoudre les sous-instances ? La réponse à cette question est primordiale pour les algorithmes DPR et explique en effet leur puissance. La forme précise d'un algorithme DPR est donnée par :

• La *taille limite* n_0 de l'input, qui donne la taille au dessous de laquelle un problème n'est plus divisé.

• La *taille* n/a de sous-instances résultant de la décomposition d'une instance.

• Le *nombre* c de telles sous-instances.

• L'algorithme utilisé pour *recombiner* les solutions des sous-instances.

La forme générale d'un algorithme DPR sur un input de taille n qui a une taille limite n_0 et qui divise le problème original en c sous-instances de taille n/a est la suivante :

Algorithme 21 DPR

Input: de taille n

1: **if** $n \leq n_0$ **then**
2: Résoudre le problème sans le sous-diviser
3: **else**
4: Décomposer en c sous-instances, chacune de taille n/a ;
5: Appliquer l'algorithme récursivement à chacune des sous-instances ;
6: Combiner les c sous-solutions résultantes pour obtenir la solution du problème original.
7: **end if**

Analyse

Pour calculer le temps de parcours de cet algorithme générique, nous supposons que le coût de la ligne 6 est dn^b (pour une certaine constante b et une certaine constante d). Soit

$T(n)$ le coût de l'algorithme sur un input de taille n. Alors pour $n \geq n_0$, nous avons

$$T(an) = cT(n) + dn^b.$$

Nous supposons de plus que pour tout $n \in \mathbb{N}$: $T(n) \leq T(n+1)$. Alors, par le Théorème 2.1 du chapitre 2, nous avons

$$T(n) = \begin{cases} O\left(n^{\log_a(c)}\right) & \text{si } a^b < c, \\ O\left(n^{\log_a(c)} \cdot \log_a(n)\right) & \text{si } a^b = c, \\ O\left(n^b\right) & \text{si } a^b > c. \end{cases} \tag{4.7}$$

Le temps de parcours d'un algorithme DPR dépend donc de la taille et du nombre de sous-instances, ainsi que du coût nécessaire pour les combiner.

4.3.2 L'algorithme de Karatsuba

Pour des polynômes $f(x)$ et $g(x)$ de degré $< n$, l'algorithme de Karatsuba retourne le produit $f(x) \cdot g(x)$.

Algorithme 22 KARATSUBA$(f(x), g(x))$

1: **if** $n = 1$ **then**
2: **return** $f_0 \cdot g_0$
3: **else**
4: $m \leftarrow \lceil n/2 \rceil$
5: $f(x) = f_0(x) + x^m f_1(x)$, $g(x) = g_0(x) + x^m g_1(x)$, $\deg(f_0), \deg(g_0) < m$
6: $h_0(x) \leftarrow$ Karatsuba$(f_0(x), g_0(x))$
7: $h_2(x) \leftarrow$ Karatsuba$(f_0(x) + f_1(x), g_0(x) + g_1(x))$
8: $h_1(x) \leftarrow$ Karatsuba$(f_1(x), g_1(x))$
9: $h(x) = h_0(x) + x^m(h_2(x) - h_0(x) - h_1(x)) + x^{2m} h_1(x)$
10: **return** $h(x)$
11: **end if**

Analyse

• Nous divisons notre problèmes en sous-problèmes plus petits jusqu'à ce que $n \leq 1$ (ligne 1), et donc $n_0 = 1$.
• Le problème est divisé en 3 sous-instances (lignes 6,7 et 8) donc $c = 3$.
• La taille de chaque sous instance est à peu près $n/2$, donc $a = 2$ (puisque les polynômes utilisés sont de degré $< m$ avec $m = \lceil n/2 \rceil$).
• L'algorithme pour combiner les sous-instances est décrit à la ligne 9. Son coût est $O(n)$, et donc $b = 1$.

Ainsi, comme $a^b = 2 < c = 3$, le coût de l'algorithme est

$$O(n^{\log_2(3)}).$$

4.3.3 Recherche binaire

L'algorithme de recherche binaire que nous avons vu dans le chapitre 0 est aussi un algorithme DPR. Nous rappelons d'abord la définition du problème de localisation :

Problème: Problème de localisation
Input: Une suite d'entiers $S_0 < S_1 < \cdots < S_{n-1}$, deux indices `start` et `end`, et $x \in \mathbb{Z}$
Output: Un indice i avec $\texttt{start} \leq i < \texttt{end}$, et tel que $S[i] = x$, ou FALSE si un tel indice n'existe pas

L'idée de la recherche binaire et de diviser la suite en deux parties à peu près de même taille en posant une unique question. L'algorithme BinSearch peut être décrit comme suit en pseudo-code :

Algorithme 23 BinSearch$(S, \texttt{start}, \texttt{end}, x)$

```
1: Length ← end − start ;
2: if Length = 1 then
3:    if S[start] = x then
4:       return start
5:    else
6:       return FALSE
7:    end if
8: else
9:    middle ← ⌊(end + start)/2⌋ ;
10:   if x < S[middle] then
11:      return Search(S, start, middle, x) ;
12:   else
13:      return Search(S, middle, end, x) ;
14:   end if
15: end if
```

Analyse

- Dans ce cas, le problème est divisé en une seule sous-instance, et donc $c = 1$.
- Le coût de chaque sous-instance est à peu près $n/2$, et donc $a = 2$.
- Le coût de combiner les sous-solutions est 0, donc $b = 0$.

Ainsi $a^b = 1 = c$, et le coût de l'algorithme est donc

$$O\left(n^{\log_a(c)} \log_a(n)\right) = O(\log(n)).$$

4.3.4 MergeSort

MergeSort est un algorithme de tri qui utilise le fait que deux suites triées peuvent être réunies (*merged*) en une seule suite à faible coût. Cet algorithme divise les objets à trier en deux groupes, puis trie chaque groupe et enfin les réunit en une suite finale et triée.

Nous commençons par étudier l'algorithme de réunion.

Algorithme 24 Merge(S_1, S_2)

Input: Suites triées S_1 et S_2 de tailles n et m
Output: Suite triée S de $S_1 \cup S_2$

```
1: i ← 0, i1 ← 0, i2 ← 0
2: while i < n + m do
3:    if i2 = m then
4:       S[i] = S1[i1]
5:       i1 ← i1 + 1
6:    else if i1 = n then
7:       S[i] = S2[i2]
8:       i2 ← i2 + 1
9:    else if S1[i1] < S2[i2] then
10:      S[i] = S1[i1]
11:      i1 ← i1 + 1
12:   else
13:      S[i] = S2[i2]
14:      i2 ← i2 + 1
15:   end if
16:   i ← i + 1
17: end while
18: return S
```

Le nombre de comparaisons de cet algorithme est $n + m$ qui est égal à la taille de l'input. On peut se servir de cet algorithme pour construire un algorithme de tri DPR.

Analyse

• Le problème est sous-divisé en 2 sous-instances (lignes 6 et 7), et donc $c = 2$.
• Chaque sous-instance est de taille à peu près égale $n/2$, donc $a = 2$.
• Le coût nécessaire pour combiner les sous-solutions est le coût de l'opération Merge et est donc égal à $O(n)$, d'où $b = 1$.

Ainsi, $a^b = c$, et le coût de l'algorithme est donc

$$O\left(n^{\log_a(c)} \log_a(n)\right) = O(n \log(n)).$$

Algorithme 25 MERGESORT(S)

Input: Suite S de n entiers
Output: Version triée S' de S

```
1: if n = 1 then
2:    return S
3: else
4:    m ← ⌈n/2⌉
5:    Soit S_1 la sous-suite de S composée des m premiers éléments de S, et S_2 la sous-suite
      formée des autres éléments.
6:    S'_1 ← MERGESORT(S_1)
7:    S'_2 ← MERGESORT(S_2)
8:    return MERGE(S'_1, S'_2)
9: end if
```

4.3.5 La paire la plus proche

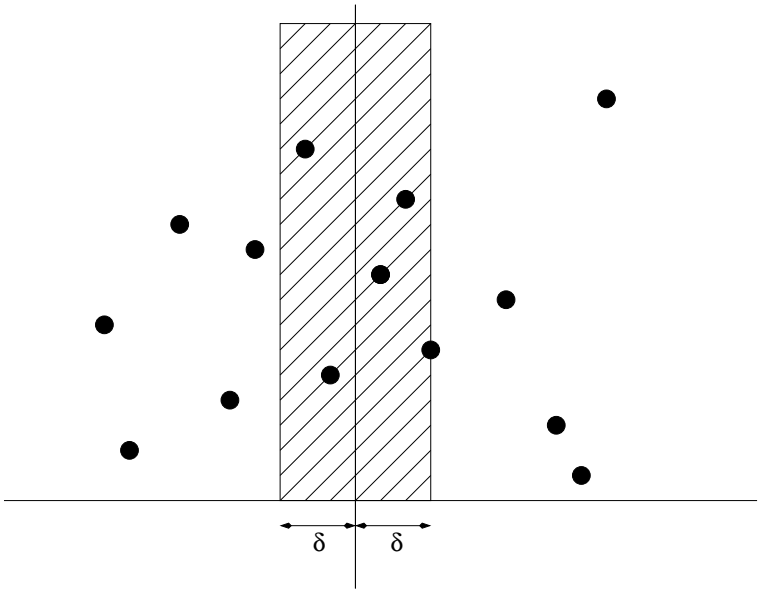
Supposons que nous voulons trouver la distance euclidienne minimale entre des points du plan $\mathbb{R}^2$. Nous supposerons de plus que tous nos points ont des coordonnées x et y différentes. Formellement :

Problème: La paire la plus proche
Input: Un ensemble de n points $\{P_1, \ldots, P_n\}$ dans le plan, avec $P_i = (x_i, y_i) \in \mathbb{R}^2$. Supposons de plus que $i \neq j \implies x_i \neq x_j$ et $y_i \neq y_j$.
Output: La distance euclidienne minimale entre deux points distincts de cet ensemble.

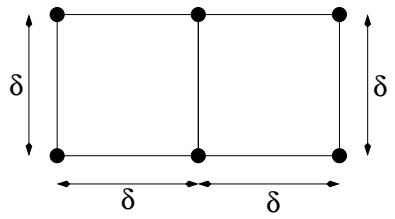
Dans un premier temps, les coordonnées x des points sont triées en ordre croissant. Ensuite, une valeur x_0 est déterminée telle que $\lceil n/2 \rceil$ des points ont une coordonnée x plus petite que x_0 tandis que les autres points ont une coordonnée x plus grande que x_0. Les deux sous-ensembles de points sont notés L_1 et L_2 ; l'algorithme est appliqué récursivement à ces sous-ensembles pour obtenir les distances minimales δ_1 et δ_2 de ces ensembles.

Il n'y a que trois possibilités en ce qui concerne la paire la plus proche dans l'ensemble : elle est soit dans L_1, soit dans L_2 ou le long de la frontière. Il ne nous reste donc qu'à vérifier les points le long de la frontière.

Nous déterminons l'ensemble S de points dont la coordonnée x est à une distance au plus $\delta := \min\{\delta_1, \delta_2\}$ de x_0.



Nous trions les coordonnées y de ces points et nous parcourons les points dans l'ordre croissant de la coordonnée y. Pour chaque point Q nous mesurons sa distance par rapport à tous les points dans S dont la coordonnée y est au plus δ plus grande que la coordonnée y de Q (tous les autres points ont une distance à Q qui est plus grande que δ). Il peut y avoir au plus 5 tels points dans S, en plus de Q, avec cette propriété. Voici une configuration de ces points :



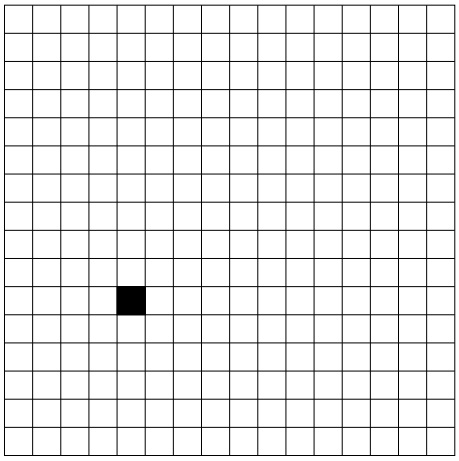
Ainsi, pour un point quelconque dans S nous devons calculer au plus 5 distances.

Après avoir traité tous les points de S de cette manière, nous saurons si la distance minimale le long de la frontière est plus petite que δ, et si c'est le cas, nous aurons déterminé la distance minimale. Les algorithmes de tri des valeurs le long les coordonnées x et y peuvent être effectués une fois au début des calculs avec coût $O(n \log(n))$ (voir le Chapitre 6 pour les algorithmes de tri).

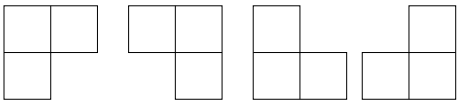
Le coût nécessaire pour combiner les sous-solutions (calculer la distance minimale le long de la frontière) est linéaire en n. Ainsi, le coût total de l'algorithme est $O(n \log(n))$.

4.3.6 Carrelage de formes L

Supposons que nous avons un carré de taille $2^k \times 2^k$ dans lequel nous enlevons un carré de taille 1×1 :

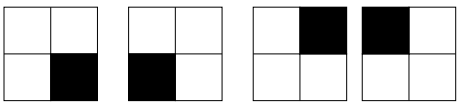


Le problème est de carreler cette structure avec des pièces de taille 3 en forme de L :



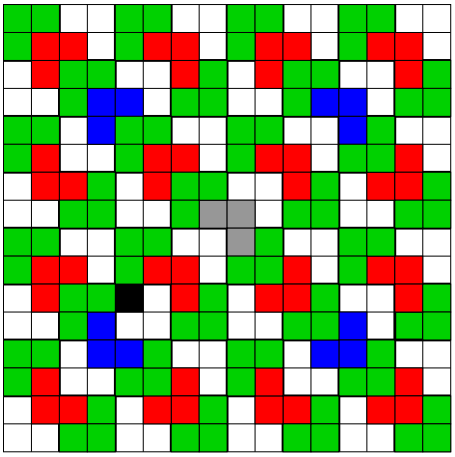
Nous allons utiliser un algorithme DPR pour cette tache.

Si $k = 1$ (donc la structure originale est un carré 2×2 avec un sous-carré manquant), alors la solution est triviale :



Étant donné un carré de taille $2^k \times 2^k$, nous le subdivisons en 4 carrés de taille $2^{k-1} \times 2^{k-1}$ chacun et nous enlevons dans chaque sous-carré qui ne contient pas la pièce 1×1 manquante la pièce 1×1 du coin de façon à ce que les carrés enlevés forment une pièce de forme L.

Remarquons que chacun des sous-carrés a un carré 1×1 manquant, et que donc l'algorithme s'applique récursivement. Nous appliquons donc récursivement l'algorithme aux sous-carrés. Le carrelage complet est montré dans l'exemple suivant :



4.4 PROGRAMMATION DYNAMIQUE

La méthode DPR s'applique si un problème peut être subdivisé en sous-problèmes *indépendants*, qui sont ensuite résolus récursivement. La programmation dynamique (*Dynamic Programming, DP*) peut être appliquée si les sous-problèmes ne sont pas indépendants et ont eux-mêmes des sous-problèmes communs. Chaque instance est résolue une seule fois et les résultats sont stockés dans un tableau pour éviter la détermination répétée des solutions des sous-problèmes.

Exemple: Supposons que nous voulons calculer le $n^{\text{ème}}$ nombre de Fibonacci, donné par la formule de récurrence

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2}, \quad n \geq 2. \end{aligned}$$

Voici la méthode DPR naïve :

Algorithme 26 FIB(n)

Input: Entier $n \geq 0$
Output: F_n

```
1: if n ≤ 1 then
2:     return n
3: else
4:     return FIB(n − 1) + FIB(n − 2);
5: end if
```

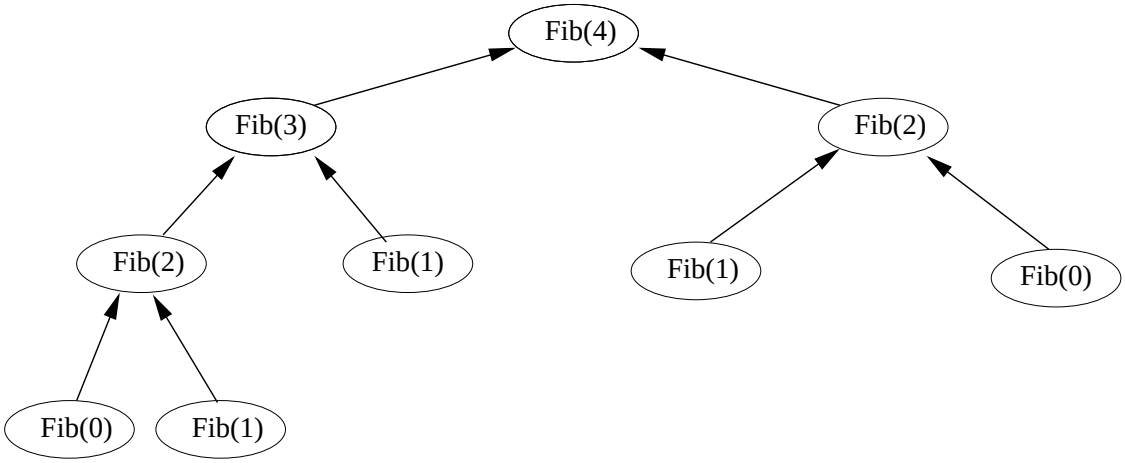
Cette méthode est simple, élégante et correcte. Cependant elle a un temps de parcours *exponentiel* en n. En effet, soit C_n le temps de parcours de FIB(n). Alors $C_n = C_{n-1} + C_{n-2}$ et $C_2 = 1, C_3 = 2$. On peut alors démontrer en partant de cette relation que $C_n = O(\varphi^n)$,

où $\varphi = (1+\sqrt{5})/2$. En fait, on peut démontrer par récurrence que

$$C_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n + \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n,$$

et comme $0 < (1-\sqrt{5})/2 < 1$, le résultat $C_n = O(\varphi^n)$ s'ensuit.

Le temps de parcours exponentiel de l'approche DPR est essentiellement dû au fait que le même sous-problème est calculé plusieurs fois. Par exemple, l'arbre suivant montre les appels récursifs pour le calcul de FIB(4) :



Nous voyons par exemple que FIB(2) est calculé deux fois. Il serait plus efficace de stocker sa valeur dans un tableau la première fois que nous en avons besoin, plutôt que de le recalculer à chaque fois.

La technique principale en DP est d'utiliser des tableaux qui stockent les solutions de sous-problèmes qui peuvent ensuite être réutilisées. La programmation dynamique est une technique qui calcule efficacement des étapes de récurrence en triant les résultats partiels. Ainsi, en mémorisant les résultats d'une récurrence, l'algorithme ne passe pas de temps à résoudre le même sous-problème plusieurs fois. Comme l'illustre l'exemple, cette technique aide parfois à réduire le temps de parcours d'exponentiel à polynomial.

Typiquement (mais pas exclusivement), on utilise DP pour résoudre des problèmes qui ont en général plusieurs solutions qui peuvent être comparées à l'aide d'une fonction de coût : Nous nous intéressons alors à une solution de coût minimal.

4.4.1 Multiplication de plusieurs matrices

Problème: **Disposition de parenthèses dans les multiplications de matrices**
Input: Une suite $(A_1, \ldots, A_n)$ de matrices de tailles $p_0 \times p_1$, $p_1 \times p_2, \ldots, p_{n-1} \times p_n$, respectivement.
Output: Les parenthèses de l'expression $A_1 \cdots A_n$ qui minimisent le nombre de multiplications scalaires si l'algorithme naïf de multiplication de matrices est utilisé.

Remarquons que de manière générale, la multiplication d'une matrice $p \times q$ avec une matrice $q \times r$ nécessite pqr multiplications.

Exemple: Pour $n = 4$, il y a 5 dispositions possibles des parenthèses :

$$(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)))$$
$$(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$$
$$((A_1 \cdot A_2) \cdot (A_3 \cdot A_4))$$
$$((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$$
$$(((A_1 \cdot A_2) \cdot A_3) \cdot A_4)$$

Pour n quelconque le nombre de dispositions possibles est un peu plus compliqué à trouver :

Théorème 4.1 *Pour le produit $A_1 \cdots A_n$, le nombre de façons dont on peut placer les parenthèses est exactement*

$$\frac{1}{n}\binom{2n-2}{n-1} = \Omega\left(\frac{4^n}{n^{3/2}}\right).$$

Preuve. Exercice. ■

Pour certaines tailles de matrices, les dispositions de parenthèses peuvent avoir des différences de coût importantes.

Exemple: Soient A_1, A_2 et A_3 des matrices de tailles suivantes :

$$\begin{aligned} A_1 &: 10 \times 100 \\ A_2 &: 100 \times 5 \\ A_3 &: 5 \times 50 \end{aligned}$$

Voici le nombre d'opérations nécessaires pour le calcul de $A_1A_2A_3$ selon la disposition des parenthèses :

Disposition	Nombre de multiplications scalaires
$(A_1 \cdot A_2) \cdot A_3$	$10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$
$A_1 \cdot (A_2 \cdot A_3)$	$10 \cdot 100 \cdot 50 + 100 \cdot 5 \cdot 50 = 75000$

Ainsi nous pouvons calculer le produit 10 fois plus rapidement en choisissant la bonne disposition de parenthèses.

Le théorème 4.1 montre que le nombre de dispositions de parenthèses est exponentiel en n. Ce n'est donc pas une bonne idée de toutes les essayer pour trouver la plus efficace. Nous résoudrons ce problème en utilisant DP. La solution sera établie en quatre étapes :

1. Structure de la disposition optimale des parenthèses
2. Calcul récursif de la valeur d'une solution optimale
3. Calcul du coût optimal
4. Construction d'une solution optimale

Structure d'une disposition optimale de parenthèses

Pour $1 \leq i \leq j \leq n$, posons

$$A_{i\ldots j} := A_i \cdot A_{i+1} \cdot \cdots \cdot A_j.$$

Pour une disposition optimale de $A_{1\ldots n}$, il existe un k tel que $A_{1\ldots k}$ et $A_{k+1\ldots n}$ sont d'abord calculés avec une disposition optimale de parenthèses, et les résultats sont ensuite multipliés. Ce type d'optimalité (*bottom-up optimality*) est typique pour des solutions DP.

Calcul récursif de la valeur d'une solution optimale

Soit m_{ij} le nombre minimal de multiplications scalaires pour le calcul de $A_{i\ldots j}$. Nous avons alors $m_{ii} = 0$ pour $1 \leq i \leq n$, et

$$m_{ij} = \min_{i \leq k < j} \{m_{ik} + m_{k+1,j} + p_{i-1} \cdot p_k \cdot p_j\}. \tag{4.8}$$

Pour stocker les solutions optimales, nous introduisons aussi s_{ij} qui est la valeur de k pour laquelle le minimum de la formule ci-dessus est atteint :

$$m_{ij} = m_{is_{ij}} + m_{s_{ij}+1,j} + p_{i-1} \cdot p_{s_{ij}} p_j. \tag{4.9}$$

Calcul du coût optimal

Nous allons construire deux matrices M et S de telle façon que $M[i,j] = m_{ij}$ et $S[i,j] = s_{ij}$. Remarquons que seules les valeurs au dessus de la diagonale nous intéressent (celles pour lesquelles $i \leq j$). Les matrices M et S seront mises à jour en commençant par la diagonale, puis les éléments au dessus de la diagonale et ainsi de suite. Pour $n = 4$ les entrées de M seraient donc calculées dans l'ordre suivant :

$$\begin{pmatrix} 1 & 5 & 8 & 10 \\ & 2 & 6 & 9 \\ & & 3 & 7 \\ & & & 4 \end{pmatrix}$$

Pour calculer $M[i,j]$ il faut comparer $j - i$ valeurs différentes (pour $k = i, \ldots, j-1$ dans (4.9)). Chacune de ces valeurs à comparer peut être trouvée avec un coût constant, puisque les éléments de M nécessaires dans (4.9) auront déjà été calculés. Ainsi pour tous les i, j qui nous intéressent (c'est à dire $i = 1, \ldots, n$ et $j = i, \ldots, n$), le coût pour calculer $M[i,j]$ est proportionel à $j - i$. Le coût total de l'algorithme est donc proportionel à

$$\sum_{i=1}^{n} \sum_{j=i}^{n} (j - i) = \frac{n \cdot (n+1) \cdot (2n-2)}{12} = \theta(n^3). \tag{4.10}$$

La preuve de (4.10) est laissée en exercice. Ce temps de parcours est donc bien meilleur que celui de l'algorithme naïf, qui est exponentiel. On obtient au final :

Exemple: Supposons que nous voulons multiplier les matrices $A_1, \ldots, A_6$, de tailles suivantes :

Matrice	Taille	Matrice	Taille
A_1	30×35	A_4	5×10
A_2	35×15	A_5	10×20
A_3	15×5	A_6	20×25

Algorithme 27 MATRIX-CHAIN-ORDER(p)

Input: $p = (p_0, p_1, \ldots, p_n)$ décrivant les tailles des matrices
Output: Matrices M et S comme décrites ci-dessus

```
1: n ← length(p) − 1
2: for i = 1...n do
3:    M[i, i] = 0
4: end for
5: for ℓ = 2...n do
6:    for i = 1...n − ℓ + 1 do
7:       j ← i + ℓ − 1
8:       M[i, j] = ∞
9:       for k = i...j − 1 do
10:         q ← M[i, k] + M[k + 1, j] + p_{i−1} p_k p_j
11:         if q < M[i, j] then
12:            M[i, j] ← q
13:            S[i, j] ← k
14:         end if
15:      end for
16:   end for
17: end for
18: return M et S
```

Nous avons donc

$$\begin{aligned} p_0 &= 30 \\ p_1 &= 35 \\ p_2 &= 15 \\ p_3 &= 5 \\ p_4 &= 10 \\ p_5 &= 20 \\ p_6 &= 25. \end{aligned}$$

Les matrices M et S sont alors :

$$M = \begin{pmatrix} 0 & 15750 & 7875 & 9375 & 11875 & 15125 \\ & 0 & 2625 & 4375 & 7125 & 10500 \\ & & 0 & 750 & 2500 & 5375 \\ & & & 0 & 1000 & 3500 \\ & & & & 0 & 5000 \\ & & & & & 0 \end{pmatrix} \qquad S = \begin{pmatrix} 0 & 1 & 1 & 3 & 3 & 3 \\ & 0 & 2 & 3 & 3 & 3 \\ & & 0 & 3 & 3 & 3 \\ & & & 0 & 4 & 5 \\ & & & & 0 & 5 \\ & & & & & 0 \end{pmatrix}$$

Par exemple, $M[1, 3]$ et $M[2, 5]$ on été calculés en utilisant (4.8) comme suit :

$$\begin{aligned} M[1,3] &= \min \begin{cases} M[1,2]+M[3,3]+p_0p_2p_3 &=& 15750+0+30\cdot 15\cdot 5 &=& 18000 \\ M[1,1]+M[2,3]+p_0p_1p_3 &=& 0+2625+30\cdot 35\cdot 5 &=& 7875 \end{cases} \\ &= 7875 \end{aligned}$$

$$\begin{aligned} M[2,5] &= \min \begin{cases} M[2,2]+M[3,5]+p_1p_2p_5 &=& 0+2500+35\cdot 15\cdot 20 &=& 13000 \\ M[2,3]+M[4,5]+p_1p_3p_5 &=& 2625+1000+35\cdot 5\cdot 20 &=& 7125 \\ M[2,4]+M[5,5]+p_1p_4p_5 &=& 4375+0+35\cdot 10\cdot 20 &=& 11375 \end{cases} \\ &= 7125 \end{aligned}$$

Construction d'une solution optimale

Nous utilisons la table $S[i,j]$. Remarquons que $S[i,j]=k$ signifie que $A_{i\ldots j}$ est construit de manière optimale à partir de $A_{i\ldots k}$ et $A_{k+1\ldots j}$.

Exemple: Pour les matrices $A_1,\ldots,A_6$ ci-dessus, nous obtenons

$$\begin{aligned} A_{1\ldots 6} &= ((A_{1\ldots 3})\cdot(A_{4\ldots 6})) \quad [S[1,6]=3] \\ &= ((A_1\cdot(A_2\cdot A_3))\cdot((A_4\cdot A_5)\cdot A_6)) \\ &\qquad [S[1,3]=1 \text{ et } S[4,6]=5]. \end{aligned}$$

Ainsi

$$((A_1\cdot(A_2\cdot A_3))\cdot((A_4\cdot A_5)\cdot A_6))$$

est une disposition de parenthèses de coût minimal pour $A_1\cdots A_6$.

4.4.2 Le problème LCS

Le problème de la plus longue sous-suite (LCS, *Longest Common Subsequence*) est un autre exemple pour lequel la DP est bien adaptée. Nous introduisons les notations suivantes : Si $X=\langle x_1,\ldots,x_m\rangle$ est une suite de longueur m, alors une suite $Z=\langle z_1,\ldots,z_k\rangle$ est appelée une *sous-suite* de X s'il existe des indices $1\le i_1<i_2<\cdots<i_k\le m$ tels que pour tout $1\le j\le k$ nous avons

$$z_j = x_{i_j}.$$

Exemple: La suite

$$X=\langle A,B,C,B,D,A,B\rangle$$

a

$$Z=\langle B,C,D,B\rangle$$

comme sous-suite. La suite des indices correspondante est $(2, 3, 5, 7)$.

Une suite Z est appelée une *sous-suite commune* de X et Y si Z est à la fois sous-suite de X et de Y. L'ensemble des sous-suites communes de X et Y est noté $CS(X, Y)$. La longueur de la plus longue sous-suite de X et Y est notée

$$lcs(X, Y),$$

et l'ensemble des sous-suites les plus longues de X et Y est noté par

$$LCS(X, Y).$$

Nous nous intéressons aux problèmes suivants, pour lesquels nous donnerons des solutions DP :
- Calculer $lcs(X, Y)$
- Construire un élément de $LCS(X, Y)$

Nous commençons par donner quelques propriétés de $LCS(X, Y)$. Soit $X = \langle x_1, \ldots, x_m \rangle$ une suite. Pour $1 \leq j \leq m$, nous notons par X_j le préfixe

$$X_j := \langle x_1, \ldots, x_j \rangle.$$

Théorème 4.2 *Soient $X = \langle x_1, \ldots, x_m \rangle$, $Y = \langle y_1, \ldots, y_n \rangle$ et soit $Z = \langle z_1 \ldots, z_k \rangle \in LCS(X, Y)$. Alors :*

(1) Si $x_m = y_n$, alors $z_k = x_m = y_n$ et $Z_{k-1} \in LCS(X_{m-1}, Y_{n-1})$.

(2) Si $x_m \neq y_n$, alors $z_k \neq x_m$ implique $Z \in LCS(X_{m-1}, Y)$.

(3) Si $x_m \neq y_n$, alors $z_k \neq y_n$ implique $Z \in LCS(X, Y_{n-1})$.

Preuve. (1) Si $z_k \neq x_m$, alors nous pouvons acoller à Z $x_m = y_n$, ce qui donnerait une sous-suite commune de X et Y de longueur $k + 1$, mais ceci contredit $Z \in LCS(X, Y)$. Donc, $z_k = x_m = y_n$ et le préfixe Z_{k-1} est une sous-suite commune de X_{m-1} et Y_{n-1}.

Nous montrons à présent que $Z_{k-1} \in LCS(X_{m-1}, Y_{n-1})$. Supposons que $W \in LCS(X_{m-1}, Y_{n-1})$ est de longueur $\geq k$. Alors la juxtaposition

$$Wx_m = Wy_n$$

est une sous-suite commune de X et Y de longueur $k + 1$, ce qui contredit $lcs(X, Y) = k$. Ainsi X_{m-1} et Y_{n-1} n'ont aucune sous-suite commune de longueur $> k - 1$ et donc $Z_{k-1} \in LCS(X_{m-1}, Y_{n-1})$.

(2) Si $z_k \neq x_m$, alors $Z \in CS(X_{m-1}, Y)$. S'il existait $W \in LCS(X_{m-1}, Y)$ de longueur $> k$, alors W appartiendrait aussi à $CS(X, Y)$, ce qui contredirait $lcs(X, Y) = k$.

(3) Symétrique à (2). ■

Remarquons que dans le cas $x_m \neq y_n$ au moins l'une des affirmations (2) ou (3) est vraie. Le théorème 4.2 permet un calcul récursif de $lcs(X, Y)$. Soit

$$c_{ij} = lcs(X_i, Y_j). \tag{4.11}$$

Alors le théorème nous dit que

$$c_{ij} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0, \\ c_{i-1,j-1} + 1 & \text{si } x_i = y_j, \\ \max(c_{i,j-1}, c_{i-1,j}) & \text{si } x_i \neq y_j. \end{cases} \tag{4.12}$$

Ainsi, comme nous l'avons fait dans les algorithmes DP précédants nous pouvons calculer et stocker les entrées de la matrice c_{ij} ligne par ligne, en utilisant à chaque fois les valeurs déjà stockées. Donc nous pouvons calculer chaque c_{ij} avec un coût constant puisque nous connaitrons toutes les entrées de la matrice c nécessaires dans (4.12). Puisqu'il y a $m \cdot n$ valeurs de c_{ij} à calculer, le coût total de l'algorithme sera $\theta(m \cdot n)$, et puisque la valeur que nous cherchons est l'entrée en bas à droite de la matrice :

$$lcs(X, Y) = c_{mn},$$

nous aurons bien trouvé la longueur de la plus grande LCS.

Nous pouvons aussi adapter cet algorithme pour qu'il retourne, en plus de la valeur de $lcs(X, Y)$, un élément de $LCS(X, Y)$. Pour ce, il nous faut une deuxième matrice b, qui va indiquer comment a été obtenu c_{ij} à partir de $c_{i-1,j-1}, c_{i,j-1}, c_{i-1,j}$ avec (4.12)

En effet, si $x_i = y_j$ alors cet élément commun sera inclu dans une LCS (nous mettrons alors $b_{ij} = "\nwarrow"$). Sinon si $c_{i-1,j} \geq c_{i,j-1}$ alors tout élément de $LCS(X_i, Y_j)$ sera aussi un élément de $LCS(X_{i-1}, Y_j)$, nous regardons donc l'entrée $b_{i-1,j}$ (dans ce cas nous mettons $b_{ij} = "\uparrow"$). Finalement si $c_{i-1,j} < c_{i,j-1}$ c'est le symétrique du cas précédant, et nous mettons donc $b_{ij} = "\leftarrow"$. A partir de cette matrice b nous pourrons construire une suite dans $LCS(X, Y)$ (voir l'exemple ci-dessous pour une matrice b, et comment l'utiliser pour trouver un élémet de $LCS(X, Y)$).

La procédure suivante, LCS-Length, calcule, pour deux suites $X = \langle x_1, x_2 \dots x_m \rangle$ et $Y = \langle y_1, \dots, y_n \rangle$ données, la valeur de $lcs(X, Y)$ et la matrice b décrite ci-dessus.

Exemple: Pour $X = \langle A, B, C, B, D, A, B \rangle$ et $Y = \langle B, D, C, A, B, A \rangle$ nous obtenons le tableau suivant (les matrices b et c sont superposées) :

Algorithme 28 LCS-LENGTH(X, Y)

Input: Suites X et Y

Output: $lcs(X, Y)$, et une matrice b décrivant un moyen de contruire un élément dans $LCS(X, Y)$.

```
1: m ← length[X]
2: n ← length[Y]
3: for i ← 1...m do
4:    c[i, 0] ← 0
5: end for
6: for j ← 0...n do
7:    c[0, j] ← 0
8: end for
9: for i ← 1...m do
10:   for j ← 1...n do
11:      if x_i = y_j then
12:         c[i, j] ← c[i − 1, j − 1] + 1
13:         b[i, j] ←"↖"
14:      else
15:         if c[i − 1, j] ≥ c[i, j − 1] then
16:            c[i, j] ← c[i − 1, j]
17:            b[i, j] ←"↑"
18:         else
19:            c[i, j] ← c[i, j − 1]
20:            b[i, j] ←"←"
21:         end if
22:      end if
23:   end for
24: end for
25: return c_mn and b
```

i	j	0 y_j	1 B	2 D	3 C	4 A	5 B	6 A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Comme $c_{mn} = 4$ nous avons $lcs(X, Y) = 4$. Les flèches permettent de lire un LCS en commençant à la position (m, n) et en suivant ensuite les flèches. Pour chaque case " $\nwarrow$" traversée il faut inclure l'élément correspondant dans notre LCS.

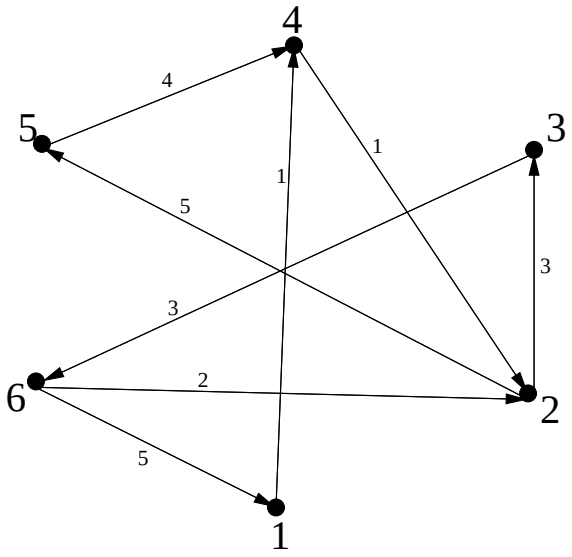
Ici nous commençons à la position $(7, 6)$, puis nous montons à $(6, 6)$. Comme cette case contient un " $\nwarrow$" nous ajoutons A à notre suite ($x_6 = y_6 = A$). Puis nous allons à la case $(5, 5)$, puis $(4, 5)$, et nous ajoutons B a notre suite etc... En notant bien que nous construisons notre LCS de la droite vers la gauche (donc en commençant par la fin) nous obtenons au final $\langle B, C, B, A\rangle$ pour notre élément de $LCS(X, Y)$.

Remarques :(1) En modifiant un peu cet algorithme nous pouvons obtenir une description de *tous* les éléments de $LCS(X, Y)$, avec le même temps de parcours.

(2) L'algorithme présenté ici nécessite $\theta(m \cdot n)$ en temps et espace. Nous verrons plus tard qu'il est possible de construire un algorithme de complexité $O(m \cdot p)$ en temps et en espace, où $p = lcs(X, Y)$.

4.4.3 Le plus-court-chemin-pour-toute-paire (Floyd-Warshall)

Soit G un graphe dont les arêtes sont étiquetées avec des entiers non-négatifs. La longueur d'un chemin dans le graphe est la somme des étiquettes des arêtes sur ce chemin. Le problème est de trouver pour toutes les paires de sommets du graphe la longueur du plus court chemin qui les relie.



Par exemple dans la situation ci-dessus, le chemin $6 \to 2 \to 3$ est de longueur $2+3=5$. Formellement, un graphe étiqueté est un graphe $G=(V,E)$ (V est l'ensemble des sommets et E celui des arêtes), muni d'une application $w\colon E \to \mathbb{R}_{\geq 0}$ qui associe à chaque arête un réel, appelé son *étiquette*.

La solution DP du problème du plus-court-chemin-pour-toute-paire a été découverte par Floyd et Warshall. Soit $V=\{1,...,n\}$. Alors l'algorithme se base sur l'idée suivante : Fixons $k \leq n$. Pour toute paire de points i et j, nous trouvons le plus court chemin entre i et j, mais qui n'utilise que les sommets $1,\ldots,k$. Nous commençons avec $k=1$ et continuons en incrémentant k jusqu'à n.

Soit c_{ij}^k la longueur du plus court chemin entre i et j dont les sommets intermédiaires sont un sous-ensemble de $\{1,\ldots,k\}$ (et $c_{ij}^k=\infty$ si un tel chemin n'existe pas). Nous posons aussi

$$c_{ij}^0 = \begin{cases} w(i,j) & \text{si } (i,j)\in E, \\ \infty & \text{sinon.} \end{cases}$$

Pour $k \geq 1$, nous avons (exercice : pourquoi ?) :

$$c_{ij}^k = \min\{c_{ij}^{k-1}, c_{i,k}^{k-1} + c_{k,j}^{k-1}\}. \tag{4.13}$$

Dans le même esprit que pour les deux problèmes précédants, nous calculons et stockons donc d'abord toutes les entrées c_{ij}^0, puis toutes les entrées c_{ij}^1, que nous calculons en utilisant (4.13) et les valeurs c_{ij}^0 déjà stockées. Nous continuons ainsi en calculant c_{ij}^k jusqu'à $k=n$, en utilisant à chaque fois (4.13) et les entrées c_{ij}^{k-1} déjà calculées.

Le temps de parcours de cet algorithme est clairement $O(n^3)$, et il utilise un espace $O(n^2)$.

Algorithme 29 APSP(G)

Input: $G = (V, E)$, un graphe étiqueté avec $V = \{1, \ldots, n\}$ et étiquette $w \colon E \to \mathbb{R}_{\geq 0}$
Output: Matrice $(c_{ij})_{1 \leq i \leq n}$, où c_{ij} est la longueur du chemin le plus court entre i et j.

```
1: for i = 1...n do
2:    for j = 1...n do
3:       if (i, j) ∈ E then
4:          c[i, j] ← w(i, j)
5:       else
6:          c[i, j] ← ∞
7:       end if
8:    end for
9: end for
10: for k = 1...n do
11:    for i = 1...n do
12:       for j = 1...n do
13:          d = c_ik + c_kj
14:          if d < c_ij then
15:             c_ij = d
16:          end if
17:       end for
18:    end for
19: end for
20: return c
```

Exemple: Utilisons le graphe précédent. Quand $k = 0$ nous pour c avons la matrice suivante :

$$\begin{pmatrix} \infty & \infty & \infty & 1 & \infty & \infty \\ \infty & \infty & 3 & \infty & 5 & \infty \\ \infty & \infty & \infty & \infty & \infty & 3 \\ \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 4 & \infty & \infty \\ 5 & 2 & \infty & \infty & \infty & \infty \end{pmatrix}.$$

Pour $k = 1$ nous avons

$$\begin{pmatrix} \infty & \infty & \infty & 1 & \infty & \infty \\ \infty & \infty & 3 & \infty & 5 & \infty \\ \infty & \infty & \infty & \infty & \infty & 3 \\ \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 4 & \infty & \infty \\ 5 & 2 & \infty & \mathbf{6} & \infty & \infty \end{pmatrix}.$$

$k = 2$:

$$\begin{pmatrix} \infty & \infty & \infty & 1 & \infty & \infty \\ \infty & \infty & 3 & \infty & 5 & \infty \\ \infty & \infty & \infty & \infty & \infty & 3 \\ \infty & 1 & \mathbf{4} & \infty & \mathbf{6} & \infty \\ \infty & \infty & \infty & 4 & \infty & \infty \\ 5 & 2 & \mathbf{5} & 6 & \mathbf{7} & \infty \end{pmatrix}.$$

$k = 3$:

$$\begin{pmatrix} \infty & \infty & \infty & 1 & \infty & \infty \\ \infty & \infty & 3 & \infty & 5 & \mathbf{6} \\ \infty & \infty & \infty & \infty & \infty & 3 \\ \infty & 1 & 4 & \infty & 6 & \mathbf{7} \\ \infty & \infty & \infty & 4 & \infty & \infty \\ 5 & 2 & 5 & 6 & 7 & \mathbf{8} \end{pmatrix}.$$

$k = 4$:

$$\begin{pmatrix} \infty & \mathbf{2} & \mathbf{5} & 1 & \mathbf{7} & \mathbf{8} \\ \infty & \infty & 3 & \infty & 5 & 6 \\ \infty & \infty & \infty & \infty & \infty & 3 \\ \infty & 1 & 4 & \infty & 6 & 7 \\ \infty & \mathbf{5} & \mathbf{8} & 4 & \mathbf{10} & \mathbf{11} \\ 5 & 2 & 5 & 6 & 7 & 8 \end{pmatrix}.$$

$k = 5$:

$$\begin{pmatrix} \infty & 2 & 5 & 1 & 7 & 8 \\ \infty & \mathbf{10} & 3 & \mathbf{9} & 5 & 6 \\ \infty & \infty & \infty & \infty & \infty & 3 \\ \infty & 1 & 4 & \mathbf{10} & 6 & 7 \\ \infty & 5 & 8 & 4 & 10 & 11 \\ 5 & 2 & 5 & 6 & 7 & 8 \end{pmatrix}.$$

$k = 6$:

$$\begin{pmatrix} \mathbf{13} & 2 & 5 & 1 & 7 & 8 \\ \mathbf{11} & \mathbf{8} & 3 & 9 & 5 & 6 \\ \mathbf{8} & \mathbf{5} & \mathbf{8} & \mathbf{9} & \mathbf{10} & 3 \\ \mathbf{12} & 1 & 4 & 10 & 6 & 7 \\ \mathbf{16} & 5 & 8 & 4 & 10 & 11 \\ 5 & 2 & 5 & 6 & 7 & 8 \end{pmatrix}.$$

Cette dernière matrice montre les chemins les plus courts entre toutes les paires de sommets. Par exemple la longueur du chemin le plus court entre le sommet 5 et le sommet 6 est 11.

4.4.4 Le Problème 0/1-Knapsack (Sac-à-dos 0/1)

Supposons que nous avons n objets avec :
- des poids $w_1, \ldots, w_n \in \mathbb{N}$,
- des valeurs $v_1, \ldots, v_n$,
- et un poids maximal $W \in \mathbb{N}$.

La tâche est de trouver un sous-ensemble $S \subseteq \{1, \ldots, n\}$ pour lequel la somme

$$\sum_{i \in S} v_i$$

est maximale, et tel que

$$\sum_{i \in S} w_i \leq W.$$

Comment pouvons nous trouver la valeur optimale ?

L'algorithme naïf regarderait tous les sous-ensembles de $\{1, \ldots, n\}$ et aurait donc un temps de parcours exponentiel (rappelons que le nombre de sous-ensembles possibles est 2^n). Nous pouvons utiliser ici un algorithme DP, car le problème a la propriété structurelle suivante :

Si nous retirons l'objet j de l'ensemble optimal, alors l'ensemble résultant doit être optimal pour le poids maximal $W - w_j$.

En effet, dans le cas contraire, la solution originale ne serait pas optimale.

Soit c_{iw} la valeur de la solution optimale en n'utilisant que les objets $1, \ldots, i$, et si le poids maximal est w. Nous avons alors

$$c_{iw} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } w = 0, \\ c_{i-1,w} & \text{si } w_i > w, \\ \max\{v_i + c_{i-1,w-w_i}, c_{i-1,w}\} & \text{si } i > 0 \text{ et } w \geq w_i. \end{cases} \tag{4.14}$$

Algorithme 30 KNAPSACK(w, v, W)

Input: $w = (w_1, \ldots, w_n)$, $v = (v_1, \ldots, v_n)$, $W \in \mathbb{N}$
Output: Valeur V telle que $V = \max \sum_{i \in S} v_i$ pour tout $S \subseteq \{1, \ldots, n\}$ avec $\sum_{i \in S} w_i \leq W$.

1: **for** $w = 0 \ldots W$ **do**
2: $\quad c_{0,w} \leftarrow 0$
3: **end for**
4: **for** $i = 1 \ldots n$ **do**
5: $\quad c_{i,0} \leftarrow 0$
6: $\quad$ **for** $w = 1 \ldots W$ **do**
7: $\quad\quad$ **if** $w_i \leq w$ **then**
8: $\quad\quad\quad$ **if** $v_i + c_{i-1,w-w_i} > c_{i-1,w}$ **then**
9: $\quad\quad\quad\quad c_{iw} = v_i + c_{i-1,w-w_i}$
10: $\quad\quad\quad$ **else**
11: $\quad\quad\quad\quad c_{iw} = c_{i-1,w}$
12: $\quad\quad\quad$ **end if**
13: $\quad\quad$ **else**
14: $\quad\quad\quad c_{iw} = c_{i-1,w}$
15: $\quad\quad$ **end if**
16: $\quad$ **end for**
17: **end for**
18: **return** c_{nW}.

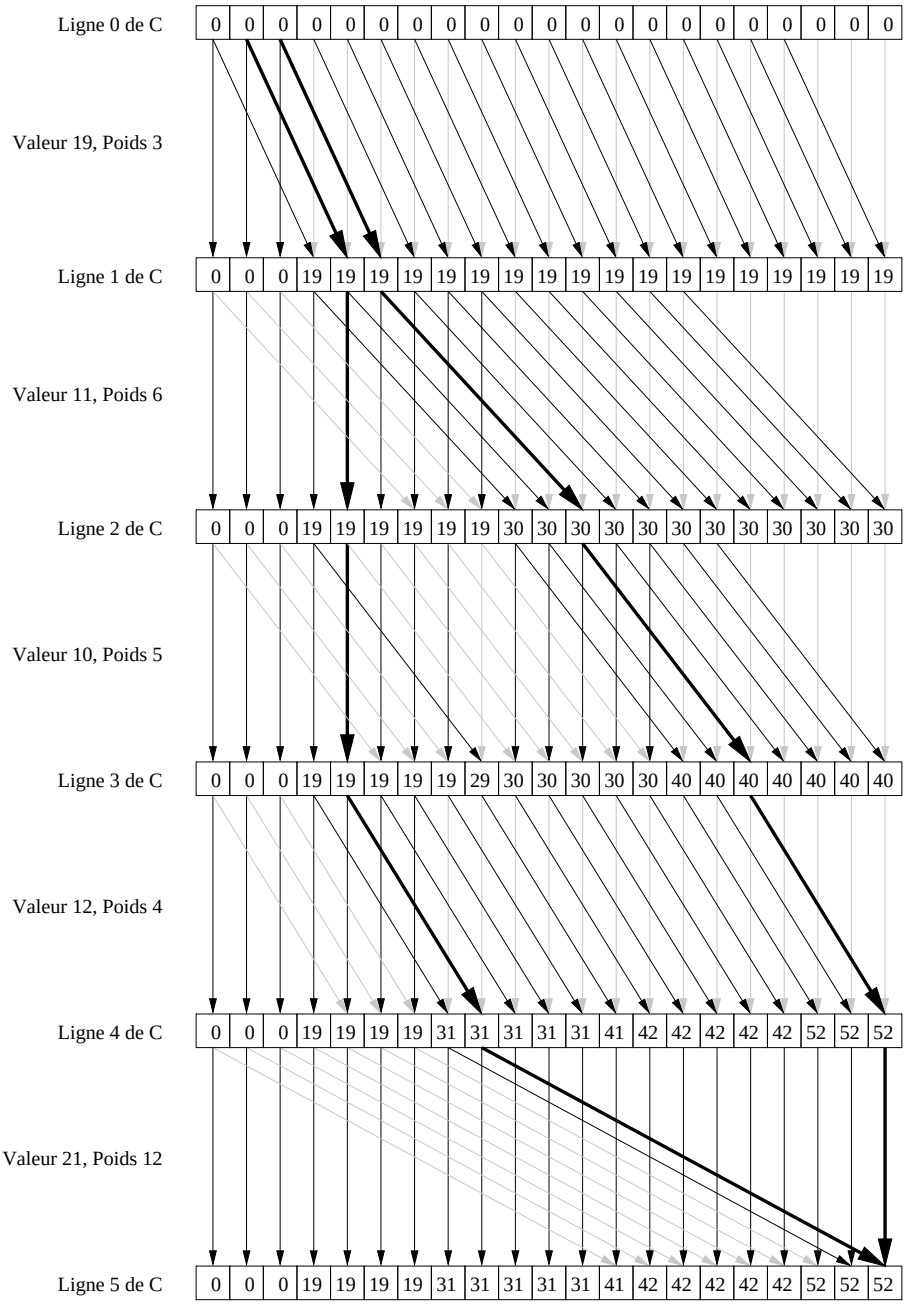
Comme pour les trois problèmes précédants, nous allons calculer et stocker les entrées c_{iw} ligne par ligne. A chaque fois la valeur de c_{iw} sera calculée en utilisant (4.14) et les entrées de c déjà stockées. Nous obtenons donc l'algorithme suivant :

Le temps de parcours de cet algorithme est clairement $O(nW)$ (boucles aux lignes 4 et 6). Il est donc bien meilleur que l'algorithme naïf qui est exponentiel en n.

Exemple: Supposons que notre poids maximal est $W = 20$, et que nous avons 5 objets avec les poids et valeurs suivants :

Objet	1	2	3	4	5
Valeur	19	11	10	12	21
Poids	3	6	5	4	12

Le parcours de l'algorithme peut être décrit par le graphique ci-dessous :



La valeur maximale est donc 52. Il y a deux ensembles d'objets qui atteignent ce maximum :

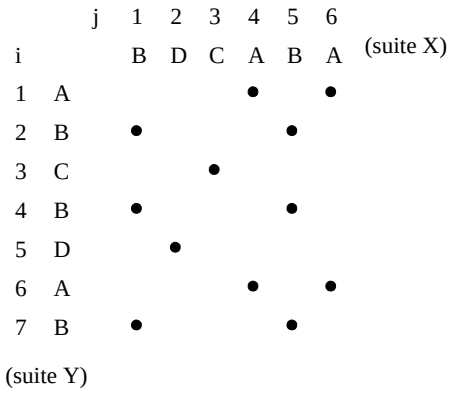
$$\{(19,3),(11,6),(10,5),(12,4)\} \qquad \text{et} \qquad \{(19,3),(12,4),(21,12)\}$$

4.5 UN ALGORITHME PLUS RAPIDE POUR LE PROBLÈME LCS

Soient X et Y deux suites de longueur n et m respectivement, avec $n \leq m$. Nous construirons dans cette section un algorithme qui trouve un élément de $LCS(X, Y)$ en temps et espace $O(mp)$, où m est la longueur de Y et $p = lcs(X, Y)$. Rappelons que l'algorithme pour le problème LCS vu dans la section 4.4.2 avait un temps de parcours $O(mn)$.

Nous commençons par étudier l'ensemble $M = \{(i, j) \mid X_i = Y_j\}$ de tous les éléments qui se correspondent dans X et Y.

Exemple: Si $X = \langle B, D, C, A, B, A\rangle$ et $Y = \langle A, B, C, B, D, A, B\rangle$ (Comme pour l'exemple de dans la section 4.4.2), l'ensemble M peut être représenté par les points dans le diagramme suivant :



Considérons la relation $\ll$ sur $(\mathbb{N} \times \mathbb{N})$ définie comme suit :

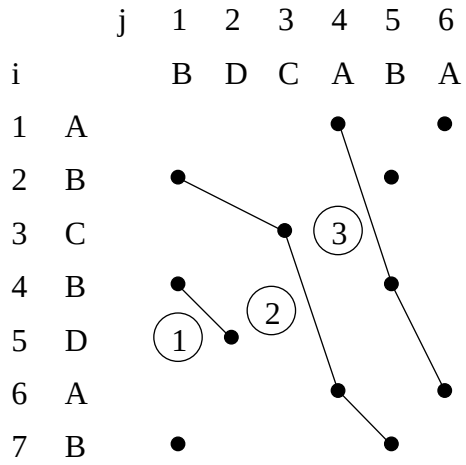
$$(k, l) \ll (k', l') \iff k < k' \wedge l < l'.$$

Cette relation définit un *ordre partiel* sur l'ensemble M. (Rappelons qu'un ordre partiel est une relation transitive et asymétrique.)

Remarque: Comme $\ll$ n'est pas un ordre total, il se peut que M ait des éléments non-comparables. Par exemple, pour $(4, 1), (3, 3) \in M$ nous n'avons ni $(4, 1) \ll (3, 3)$, ni $(3, 3) \ll (4, 1)$.

Un sous-ensemble $C \subseteq M$ ne contenant que des éléments comparables s'appelle une *chaîne*. Une *antichaîne* $A \subseteq M$ est un sous-ensemble dans lequel il n'y a pas deux éléments qui sont comparables.

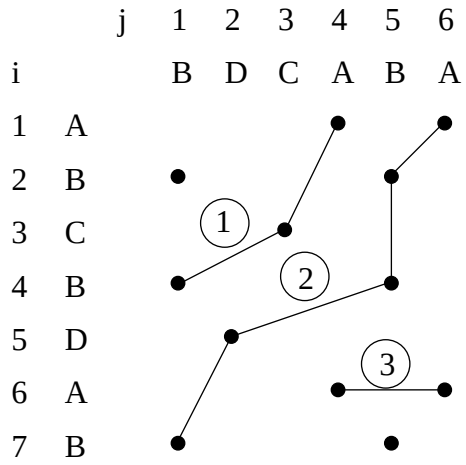
Exemple. Quelques chaînes :



Il est clair qu'une chaîne correspond à un élément dans $CS(X, Y)$. Ici nous avons

$$\begin{aligned} 1 &\mathrel{\widehat{=}} \langle B, D\rangle \\ 2 &\mathrel{\widehat{=}} \langle B, C, A, B\rangle \\ 3 &\mathrel{\widehat{=}} \langle A, B, A\rangle \end{aligned}$$

Exemple. Quelques antichaînes :



Notre but est donc de trouver une chaîne de longueur *maximale*. Nous regardons d'abord un lien intéressant entre chaînes et antichaînes. Pour expliquer ce lien, nous avons besoin de la notion de *décomposition* d'un ensemble fini et partiellement ordonné. C'est un ensemble d'antichaînes $\{A_1, A_2, \ldots, A_k\}$ tel que chaque élément de M est contenu dans au moins une des A_i :

$$M = A_1 \cup \cdots \cup A_k.$$

Une *décomposition minimale* est une décomposition contenant aussi peu d'antichaînes que possible.

Théorème 4.3 *Dans un ensemble fini et partiellement ordonné M, la longueur p d'une chaîne maximale est égale à la taille d'une décomposition minimale de M.*

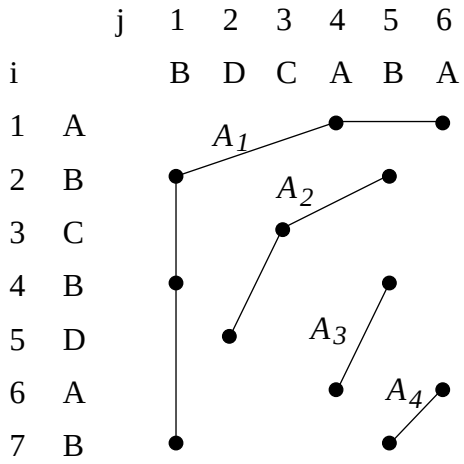
Preuve. Soit $\{A_1, \ldots, A_k\}$ une décomposition minimale de M, et soit C une chaîne maximale avec p éléments. Si $k < p$, alors il existe $c, c' \in C$, $c \neq c'$, tels que c et c' sont dans la même antichaîne (par le principe des tiroirs). C'est impossible, et donc $k \geq p$.

Pour montrer que $k \leq p$, nous construisons une décomposition en p antichaînes. Ainsi s'il existe une décomposition en p antichaînes, alors une décomposition minimale aura certainement au plus p antichaînes. Pour $c \in M$, soit $\ell(c)$ la longueur d'une chaîne maximale avec plus grand élément c. Si $\ell(c) = \ell(c')$, alors c et c' sont incomparables (Pourquoi ?). Donc

$$A_i := \{c \mid \ell(c) = i\}, \quad 1 \leq i \leq p$$

forment p antichaînes qui couvrent M. ■

La construction de la preuve dans notre cas est comme suit :



Le lemme suivant suggère qu'il est facile de calculer une chaîne maximale à partir de cette décomposition.

Lemme 1 *Soit $\{A_1, \ldots, A_p\}$ la décomposition minimale de M telle que*

$$A_i := \{c \mid \ell(c) = i\}, \quad 1 \leq i \leq p,$$

comme décrite ci-dessus. Soit $2 \leq i \leq p$. Alors nous avons

$$\forall c_i \in A_i \quad \exists c_{i-1} \in A_{i-1} \text{ tel que } c_{i-1} \ll c_i.$$

Preuve. Si $c_i \in A_i$, alors $\ell(c_i) = i$, et il existe une chaîne

$$m_1 \ll m_2 \ll \cdots \ll m_{i-1} \ll c_i.$$

Comme $m_1 \ll m_2 \ll \cdots \ll m_{i-1}$ nous avons $\ell(m_{i-1}) \geq i-1$. Mais $\ell(m_{i-1}) \geq i$ n'est pas possible, puisque ceci impliquerait $\ell(c_i) \geq i+1$. Donc, $\ell(m_{i-1}) = i-1$, i.e., $m_{i-1} \in A_{i-1}$. On choisit donc $c_{i-1} := m_{i-1}$. ■

Pour obtenir une chaîne de longueur maximale, nous commençons avec un élément $c_p \in A_p$ quelconque, puis nous choisissons $c_{p-1} \in A_{p-1}$ avec $c_{p-1} \ll c_p$; ensuite nous choisissons $c_{p-2} \in A_{p-2}$ avec $c_{p-2} \ll c_{p-1}$, etc. Ceci nous fournit une chaîne complète de longueur p :

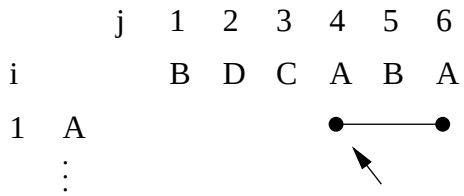
$$c_1 \ll c_2 \ll \cdots \ll c_{p-1} \ll c_p,$$

et par conséquent un élément de $LCS(X, Y)$.

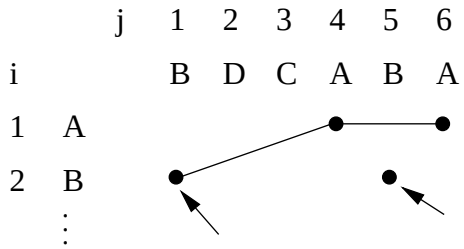
Il ne reste donc plus qu'à trouver une décomposition minimale $\{A_1, \ldots, A_p\}$ de notre ensemble M. Nous Pour ce, Nous construisons nos antichaînes de haut en bas. Nous parcourons M ligne par ligne, en ajoutant le point (i, j) à l'antichaîne A_k si et seulement s'il ne se trouve pas à droite du point le plus à gauche de A_k. Nous appelons s_k la colonne du point le plus à gauche de A_k.

Pour l'exemple ci-dessus nous obtenons :

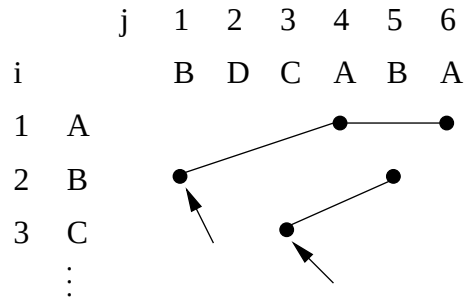
• Ligne 1 : Nous ajoutons $(1, 4)$ et $(1, 6)$ à A_1, et nous avons donc $s_1 = 4$. Pour l'instant nous n'avons qu'une seule antichaîne non vide :



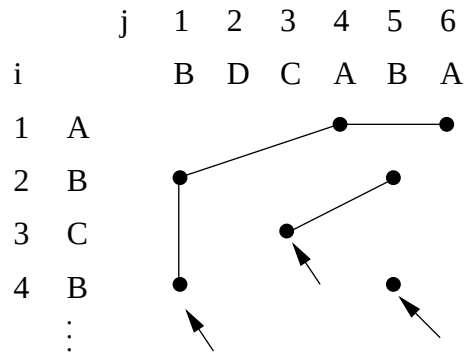
• Ligne 2 : Le premier point sur la ligne 2 est $(1, 2)$. Il est dans la 2^{eme} colonne et $2 \leq s_1$ nous l'ajoutons donc à A_1. Le prochain point est $(2, 5)$, cette fois comme $5 > s_1$ nous ne l'ajoutons pas à A_1. Par contre l'antichaîne A_2 est vide nous ajoutons donc $(2, 5)$ à A_2. Ainsi maintenant $s_1 = 1$ et $s_2 = 5$. Nous avons donc deux antichaînes non vides :



• Ligne 3 : Le seul point sur la ligne 3 est $(3,3)$. Comme $3 > s_1$ nous ne l'ajoutons pas à A_1. Par contre $3 \leq s_2$, nous ajoutons donc $(3,3)$ à A_3. Ainsi maintenant $s_1 = 1$ et $s_2 = 3$. Nous avons toujours deux antichaînes non vides :



• Ligne 4 : Nous ajoutons $(4,1)$ à A_1 et $(4,5)$ à A_3. Donc maintenant $s_1 = 1, s_2 = 3$ et $s_3 = 5$. Nous avons 3 antichaînes non vides : $A_1 = \{(1,6),(1,4),(2,1),(4,1)\}$, $A_2 = \{(2,5),(3,3)\}$ et $A_3 = \{(4,5)\}$:



etc.

Cette procédure est réalisée par l'algorithme suivant :

Dans cet algorithme nous avons
– i représente la ligne actuelle de M étudiée.
– $A[k]$ représente l'antichaîne A_k.
– $s[k]$ représente la colonne actuelle la plus à gauche de l'antichaîne A_k.

Remarquons que $s[j] = n+1$ si et seulement si A_j est vide. A la fin de l'algorithme, le nombre d'antichaines non vides est égal à

$$p = \max\{k \mid s[k] \leq n\},$$

et les antichaînes construites sont stockées dans $A[k]$, $1 \leq k \leq p$.

Théorème 4.4 *L'algorithme ci-dessus a un temps de parcours de $O(mp)$, où $p = lcs(X,Y)$.*

Algorithme 31 FastLCS(X, Y)

Input: Suites X et Y, de longueurs n et m.

Output: La décomposition minimale $(A_1, \ldots, A_p)$ de M, telle que $A_i = \{c \mid \ell(c) = i\}$, où $\ell(c)$ est la longueur d'une chaîne maximale avec plus grand élément c.

1: Construire l'ensemble $M = \{(i, j) \mid X_i = Y_j\}$
2: **for** $k = 1 \ldots n$ **do**
3: $\quad s[k] \leftarrow n + 1$
4: $\quad A[k] \leftarrow \emptyset$
5: **end for**
6: **for** $i = 1 \ldots m$ **do**
7: $\quad l \leftarrow \min\{k \mid (i, k) \in M\}$
8: $\quad k \leftarrow 1$
9: $\quad$ **while** $l \leq n$ **do**
10: $\quad\quad$ **if** $l \leq s[k]$ **then**
11: $\quad\quad\quad A[k] \leftarrow A[k] \cup \{(i, j) \in M \mid l \leq j \leq s[k]\}$
12: $\quad\quad\quad \text{temp} \leftarrow s[k]$
13: $\quad\quad\quad s[k] \leftarrow l$
14: $\quad\quad\quad l \leftarrow \min\{j \mid (i, j) \in M \wedge j > \text{temp}\}$
15: $\quad\quad$ **end if**
16: $\quad\quad k \leftarrow k + 1$
17: $\quad$ **end while**
18: **end for**
19: $p \leftarrow \max\{k \mid s[k] \leq n\}$.
20: **return** $(A[1], \ldots, A[p])$.

Preuve. Exercice. ■

Remarque : Une fois que nous avons trouvé cette décomposition minimale $(A_1, \ldots, A_p)$ de M nous utilisons le Lemme 1 pour en déduire une chaîne de longueur maximale. Nous la construisons de la droite vers la gauche.
Dans l'exemple ci-dessus nous commençons par prendre un élément de $A_p = A_4$, disons $(6, 6)$. Puis le Lemme nous dit qu'il existe un élément $(i, j) \in A_3$ avec $(i, j) \ll (6, 6)$, prenons par exemple $(4, 5)$. Ensuite il nous faut un élément de A_2, (prenons $(3, 3)$) et finalement un élément de A_2 (prenons $(2, 1)$).
Puisque nous l'avons construite en commençant à droite, notre chaine est

$$\Big((2, 1), (3, 3), (4, 5), (6, 6)\Big),$$

et donc la sous-suite commune correspondante est

$$\langle B, C, B, A\rangle.$$

5
Algorithmes gloutons

Pour résoudre certains problèmes d'optimisation, il suffit de faire un choix localement optimal à chaque pas de l'algorithme. En effet, dans certain cas, une famille de choix localement optimaux mène à une solution globalement optimale. Ce fait est au cœur des algorithmes gloutons : un algorithme glouton fait toujours le choix qui lui semble le meilleur sur le moment. Après l'étude d'un exemple typique, nous étudierons les aspects principaux de stratégies glouton. Ces stratégies seront illustrées à l'aide des exemples du problème de sac à dos et du code de Huffman, utilisé dans la compression des données.

5.1 EXEMPLE : HORAIRES DE SALLES DE COURS

Cet exemple concerne le problème de l'ordonnancement de plusieurs activités qui rivalisent pour l'utilisation exclusive d'une ressource commune, l'objectif étant de sélectionner un ensemble de taille maximale d'activités mutuellement compatibles.

Problème : n professeurs veulent utiliser la même salle de cours. Chaque professeur a une préférence pour un intervalle de temps : Le professeur i veut utiliser la salle dans l'intervalle $[s_i, f_i)$.

La tâche de l'organisateur est de trouver un plus grand sous-ensemble (c.à.d. un ensemble de taille maximale) A de $S = \{1, \ldots, n\}$ tel que pour $i, j \in A$, $i \neq j$, on a

$$[s_i, f_i) \cap [s_j, f_j) = \emptyset. \tag{5.1}$$

La stratégie gloutonne est la suivante :

Nous trions d'abord les intervalles selon leur temps d'arrêt f_i, afin d'avoir

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

On peut faire ce tri en temps $O(n \log(n))$. Puis l'algorithme opère de la manière suivante :

Théorème 5.1 *L'algorithme* GreedySelector *satisfait le nombre maximal de préférences.*

Algorithme 32 GREEDYSELECTOR(s, f)

Call : GREEDYSELECTOR(s, f)
Input: s et f sont des tableaux de longueur n, $f_1 \leq \cdots \leq f_n$.
Output: Sous-ensemble A de $\{1, \ldots, n\}$ vérifiant (5.1).

```
1: n ← length[s]
2: A ← {1}
3: j ← 1
4: for i = 2, ..., n do
5:    if s_i ≥ f_j then
6:       A ← A ∪ {i}
7:       j ← i
8:    end if
9: end for
10: return A
```

Nous donnons maintenant une preuve de ce théorème.

Preuve. Soit $S = \{1, 2, \ldots, n\}$ l'ensemble des indices des préférences. La préférence 1, réalisée par l'algorithme, a le temps d'arrêt le plus tôt f_1. Nous montrons qu'il y a une solution optimale avec le premier choix fait par l'algorithme, que nous appelons "choix glouton 1".

Soit $A \subseteq S$ une solution optimale. Supposons que $k \in A$ a le temps d'arret minimal dans cette solution, i.e., k vérifie $f_k \leq f_j \ \forall j \in A$. Si $k = 1$, nous avons fini. Si $k > 1$, nous pouvons remplacer A par $A' = (A \setminus \{k\}) \cup \{1\}$, comme $f_1 \leq f_k$ par hypothèse. Ainsi, il y a une solution optimale avec le choix glouton 1 que nous noterons de nouveau avec A.

Après ce choix, nous appliquons la même stratégie aux préférences restantes.

$$S' := \{i \in S \mid s_i \geq f_1\}.$$

Alors, $A \setminus \{1\}$ est une solution optimale pour S' : si la solution optimale B' pour S' est plus grande que $A \setminus \{1\}$, alors $B' \cup \{1\}$ serait une solution pour S avec plus d'éléments que A, ce qui contredit l'hypothèse que A est de taille maximale.

En appliquant une induction, on prouve le théorème. ■

5.2 ÉLÉMENTS D'UNE STRATÉGIE GLOUTONNE

Un algorithme glouton est un algorithme qui maximise son profit à chaque étape. Ce n'est pas toujours la bonne méthode pour résoudre un problème d'optimisation. Exemple : les flux de traffic ! Si chaque conducteur optimise son propre profit à chaque point, alors la solution convergente sera typiquement la pire pour tout le monde.

Comment peut-on décider si un problème d'optimisation peut être résolu de manière optimale en utilisant une stratégie gloutonne ? Il n'y a pas de recette général pour ce faire. Néanmoins, un algorithme glouton se propose si les propriétés suivantes sont vérifiées :

Stratégie Gloutonne : l'optimum *global* peut être atteint par une suite d'optima *locaux*. Ceci doit être prouvé dans chaque cas inviduel.

Sous-structure optimale : un problème d'optimisation possède une sous-structure optimale si une solution optimale est composée de solutions optimales de sous-problèmes.

Cette deuxième propriété est exploitée par la programmation dynamique ainsi que la méthode gloutonne. Dans l'exemple de la section précédente nous avons vu que pour une solution optimale A du problème de la salle de cours, si $1 \in A$, alors $A' = A \setminus \{1\}$ est une solution optimale pour les préférences dans $S' = \{i \in S \mid s_i \geq f_i\}$.

5.3 GLOUTON COMPARÉ À LA PROGRAMMATION DYNAMIQUE : LE PROBLÈME DU SAC À DOS

Le problème du sac à dos 0/1 a été présenté au chapitre précédent. Nous considérons une version affaiblie de ce problème : le *problème du sac à dos rationnel*. Supposons que nous ayons

- n objets de poids $w_1, w_2, \ldots, w_n$
- de valeurs $v_1, v_2, \ldots, v_n$,
- et un poids maximal $W \in \mathbb{N}$.

La tâche est de trouver $x_i \in \mathbb{Q}$, $0 \leq x_i \leq 1$, tel que la somme

$$\sum_{i=1}^{n} x_i v_i$$

soit maximale. La différence entre le problème du sac à dos 0/1 et celui du sac à dos rationnel est donc que ce dernier permet d'ajouter des morceaux d'objets dans le sac à dos, tandis dans le sac à dos 0/1, on choisit chaque objet soit en complet, soit pas du tout.

Le problème du sac à dos rationnel peut être résolu à l'aide d'un algorithme glouton :

- Pour chaque i nous calculons la valeur relative v_i/w_i.
- Trier (en un temps $O(n \log(n))$) les objets selon leurs valeures relatives. Après re-indexation nous aurons alors

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}.$$

- Trouver l'incide m avec

$$\sum_{i=1}^{m-1} w_i \leq W < \sum_{i=1}^{m} w_i.$$

On inclut alors les objets $1, \ldots, m-1$ complètement, et de l'objet m la fraction

$$\gamma_m := W - \sum_{i=1}^{m-1} w_i.$$

Les autres objets sont laissés en dehors du sac à dos. En d'autres termes, $x_1 = \cdots = x_{m-1} = 1$, $x_m = \gamma_m / w_m$, et $x_{m+1} = \cdots = x_n = 0$.

Exercice : Prouver que ce choix des x_i est optimal !

La même procédure ne donne pas nécessairement une solution optimale du problème de sac à dos 0/1. Par exemple, considérons l'instance suivante :

i	1	2	3
w_i	10	20	30
v_i	60	100	120
v_i/w_i	6	5	4

$W = 50.$

La solution optimale de cet exemple est donné par le choix des objet 2 et 3 pour une valeur totale de 220 et un poids de 50. Une solution incluant l'objet 1, dont la valeur relative est maximal, ne peut pas être optimale.

Pourquoi cela ? C'est parce que dans le problème du sac à dos 0/1, une solution optimale d'un sous-problème ne s'étend pas nécessairement à une solution optimale du problème plus grand. La programmation dynamique prend en compte toutes les solutions des sous-problèmes, et non pas seulement les meilleures. Donc, la programmation dynamique fonctionne dans ce cas, alors que la stratégie glouton échoue.

5.4 CODES DE HUFFMAN

Le codage de Huffman est une technique basique pour la *compression sans perte* (*lossless compression*). Par la suite on parlera de code de *longueur fixe* si tous les mots du code ont la même longueur. Si ce n'est pas le cas, on dira que le code est de *longueur variable*.

Exemple : Considérons un fichier w contenant 100000 lettres de l'alphabet $\{a, b, c, d, e, f\}$. (I.e., w est un mot, appartenant à l'ensemble $\{a, b, c, d, e, f\}^{100000}$.) Le mot w doit être stocké en représentation binaire.

Si nous utilisons un code de longueur fixe, alors nous devrions représenter chacune des six lettres avec trois bits. Par exemple :

$$\begin{array}{llllll} a & = & 000 & c & = & 010 \\ b & = & 001 & d & = & 011 \end{array} \qquad \begin{array}{llll} e & = & 100 \\ f & = & 101. \end{array}$$

Ceci résultera en 300000 bits pour une représentation binaire de ce fichier.

Un code de longueur variable est en général bien meilleur qu'un code de longueur fixe, parce qu'il est possible d'assigner des suites binaires plus courtes aux lettres qui apparaissent plus souvent. Pour l'utiliser, la fréquence des lettres doit être connue. (La *fréquence d'une lettre* est sa probabilité d'apparition. Par exemple, si 25% de toutes les lettres du mot sont des A, alors la fréquence de A est 0.25.)

Exemple :

Lettre	a	b	c	d	e	f
Fréquence	0.45	0.13	0.12	0.16	0.09	0.05
Représentation (code)	0	101	100	111	1101	1100

Ce code a seulement besoin de 224000 bits ($45000 \cdot 1 + 13000 \cdot 3 + 12000 \cdot 3 + 16000 \cdot 3 + 9000 \cdot 4 + 5000 \cdot 4$) et est 25% plus efficace que le code de longueur fixe comme décrit ci-dessus.

Par la suite, nous verrons une technique (la méthode de Huffman) qui permettra de trouver le code de longueur variable le plus efficace pour un alphabet avec fréquences des lettres donné. Mais d'abord, il faut préciser la terminologie. Soit C un alphabet fini.

Un *code binaire* pour C est une application injective

$$E\colon C \longrightarrow \{0,1\}^+.$$

Comme déjà annoncé, si tous les mots binaires dans l'image de E ont la même longueur, alors on dit que le code est un *code de longueur fixe*, sinon c'est un *code de longueur variable*.

Dans cette section, on écrira $c_1c_2\cdots c_n$ le mot de longueur n et de lettres $c_i \in C$. Aussi, si a et b sont deux mots quelconques, nous notons le mot obtenu en juxtaposant a et b par ab, à moins qu'il y ait une ambiguité.

Un *codage* via E est l'application (aussi notée par E) suivante :

$$\begin{array}{rccc} E: & C^+ & \longrightarrow & \{0,1\}^+ \\ & c_1\cdots c_n & \longmapsto & E(c_1)\cdots E(c_n). \end{array}$$

Autrement dit, le codage du mot $c_1\cdots c_n$ se fait en codant chaque lettre c_i individuellement et en juxtaposant les mots obtenus.

Si E est injective sur C^+, alors on dit que E est *uniquement décodable*. Dans ce cas, chaque mot de code $E(c_1)\cdots E(c_n)$ possède une unique préimage et l'application

$$D\colon E(c_1)\cdots E(c_n) \mapsto c_1\cdots c_n$$

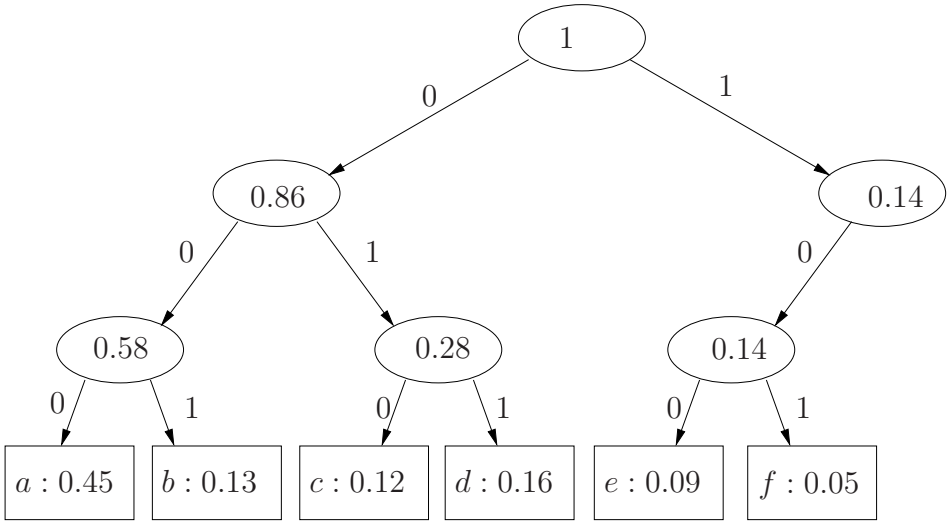
s'appelle un *décodage* pour E.

Un *code sans préfixes* (*prefix-code*) est un code pour lequel aucun $c \in \text{Image}(E)$ est le préfixe d'un autre élément dans $\text{Image}(E)$. Nous mentionnons le théorème suivant sans preuve.

Théorème 5.2 *La compression optimale de données sans perte est toujours possible en utilisant un code sans préfixes.*

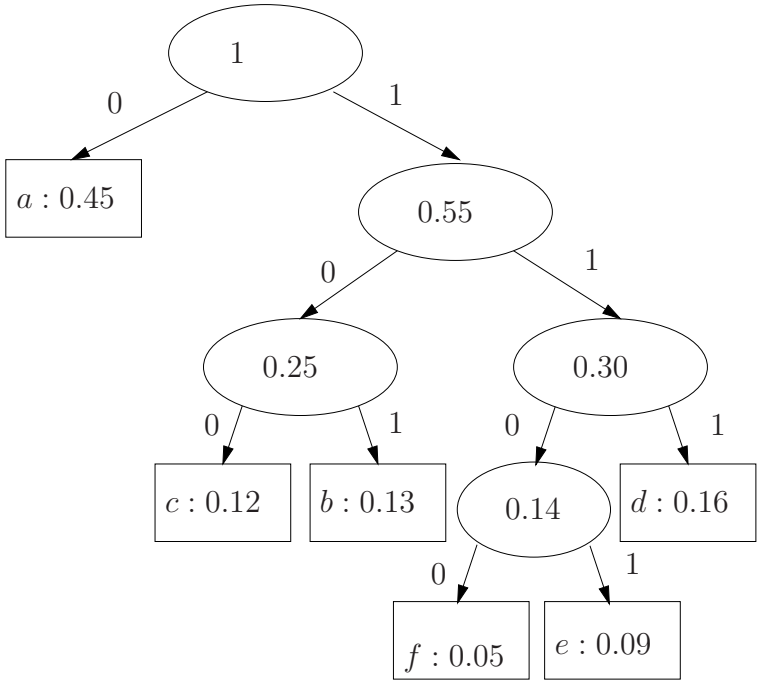
Il est possible de représenter des codes binaires par des arbres binaires. L'exemple suivant illustre comment ceci se fait, et le paragraphe suivant donnera ensuite une description précise du lien entre arbre et code.

Exemple: Code de longueur fixe :



L'arbre ci-dessus représente le code de longueur fixe pour l'alphabet $\{a, b, c, d, e, f\}$ vu avant. Le codage de la lettre a se lit sur les arêtes du chemin racine-a.

Nous avons vu que cet arbre n'est pas optimal : le fichier w était plus long que nécessaire. L'arbre du code de longueur variable de l'exemple précédent est représenté par l'arbre suivant :



Nous verrons que ce code est optimal, dans un sens à préciser.
- Mais que veut dire "optimal", et comment prouve-t-on qu'un arbre (ou un code) donné est optimal ?
- Comment trouve-t-on des arbres optimaux, pour alphabet et fréquences donnés ?

5.4.1 Codes representés par des arbres binaires

Un arbre binaire étiqueté est un arbre binaire raciné muni d'un étiquetage de ses arêtes par 0 et 1 tel que les étiquettes le long des arêtes des descendants sont différentes pour chaque sommet.

Soit C un alphabet fini donné avec les fréquences $f(c)$, $c \in C$. Les fréquences sont des nombres réels entre 0 et 1 dont la somme vaut 1. (Donc, $f(\cdot)$ définit une distribution de probabilités sur C.)

Un *arbre de compression* pour C est un arbre binaire étiqueté dont les feuilles sont en correspondance biunivoque avec les éléments de l'alphabet et dont les sommets ont des poids. Le poids d'une feuille est la fréquence de l'élément associé dans l'alphabet. Le poids d'un sommet interne est la somme des poids de tous ses descendants directs.

Le code associé à un arbre de compression est l'ensemble de toutes les suites binaires obtenu en juxtaposant les étiquettes des arêtes pour chaque chemin de la racine à une feuille.

Théorème 5.3 *Le code associé à un arbre de compression est un code sans préfixes.*

Soit $d_B(c)$ la profondeur de l'élément c de l'alphabet dans l'arbre de compression B. Alors $d_B(c)$ est la longueur du mot de code associé à c. La longueur moyenne d'un mot de code est alors

$$A(B) := \sum_{c \in C} f(c) d_B(c)$$

si l'on choisit les lettres à encoder selon la loi $f(c)$, i.e. une lettre c fixée est choisie avec probabilité $f(c)$. La longueur moyenne $A(B)$ est aussi appelée le *coût* de l'arbre de compression.

Un arbre de compression est *optimal* si son coût est minimal pour l'alphabet C et les fréquences $f(c)$, $c \in C$.

5.4.2 Le codage de Huffman

Le codage de Huffman est une méthode de construction d'un arbre de compression optimal pour un alphabet avec des fréquences données. Cette méthode a été inventée par David Huffman en 1952 alors qu'il était encore étudiant.

Cette méthode construit l'arbre optimal de bas en haut (*bottom-up*), et c'est un algorithme glouton. Nous introduirons d'abord cette construction. Ensuite nous prouverons l'optimalité de la construction. Puis nous discuterons des structures de données adaptées à une implémentation de cet algorithme.

La technique de base :

1. Commencer avec autant d'arbres qu'il y a d'éléments dans l'alphabet : Associer aux arbres (avec une seule feuille) l'élément correspondant de l'alphabet et sa fréquence.
2. Tant qu'il y a plus d'un arbre :
   i. Trouver deux arbres de fréquence minimale.
   ii. Combiner les arbres en un, en utilisant l'un des arbres comme sous-arbre gauche et l'autre comme sous-arbre droit. Etiquetter le sous-arbre gauche avec 0, et le sous-arbre droit avec 1. Etiquetter la racine du nouvel arbre avec la somme des coûts des deux sous-arbres.

5.4.3 Implémentation de l'algorithme de codage de Huffman

Nous utiliserons une *queue à priorité* Q pour cette tâche. Q contiendra les éléments de l'alphabet ainsi que leurs fréquences, $(c, f(c))$. Les éléments sont ordonnés par fréquence, de la plus petite à la plus grande.

La queue aura une fonction `deletemin` qui efface l'élément avec la plus petite fréquence. La meilleure façon d'implémenter cette structure de données est d'utiliser un binary heap. Nous verrons cette structure dans le chapitre suivant, dans lequel nous étudierons des *algorithmes de tri*.

Algorithme 33 HUFFMAN(C, f)

Input: Un ensemble de caractères C de taille $n = |C|$. (Q est une queue à priorité ordonnée par fréquence, initialement vide.)
Output: Arbre de compression Huffman T pour l'input.

1: **for** $c \in C$ **do**
2: Enqueue $((c, f(c)), Q)$.
3: **end for**
4: **for** $i = 1, \ldots, n-1$ **do**
5: Créer un nouveau sommet z
6: *left_child* = `deletemin`(Q)
7: *right_child* = `deletemin`(Q)
8: $f(z) = f(\textit{left\_child}) + f(\textit{right\_child})$
9: Les fils de z deviennent *left_child* et *right_child*
10: Enqueue $((z, f(z)), Q)$.
11: **end for**

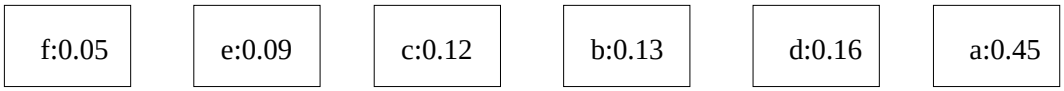
Le coût de cet algorithme est égal au coût des $2(n-1)$ `deletemin`'s et des n `enqueue`.

Si Q est implementée par un binary heap, ces opérations coûtent $O(\log(n))$, le coût total de l'algorithme sera alors égal à $O(n \log(n))$.

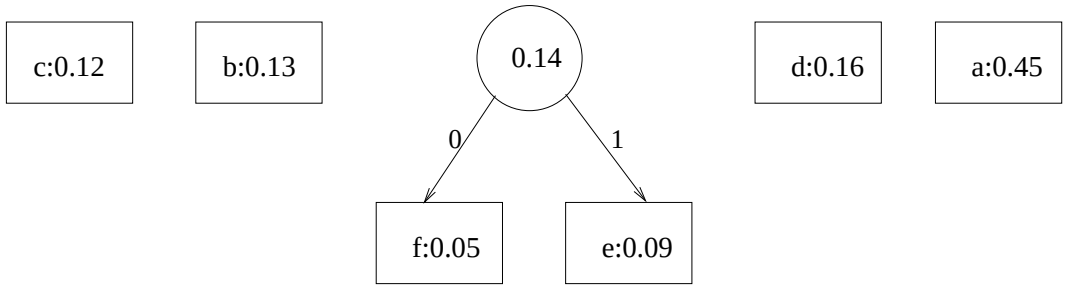
Nous verrons des binary heaps dans le chapitre suivant, dans lequel nous étudierons des algorithmes de tri.

Exemple : Voici comment l'algorithme opère sur l'exemple présenté en début de section :

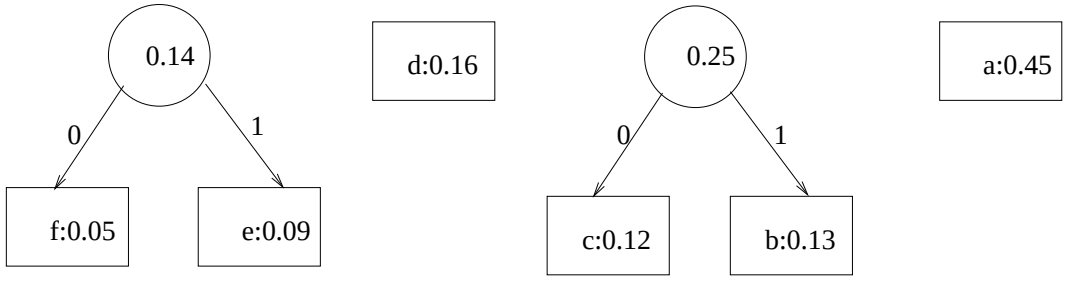
1. Mise en place de Q :



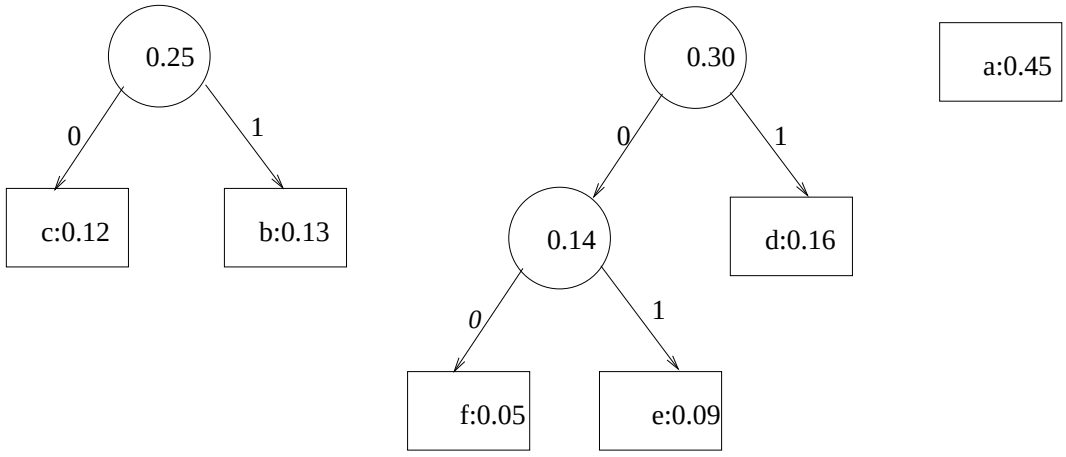
2. $i = 1$ avec $f(z) = 0.14$:



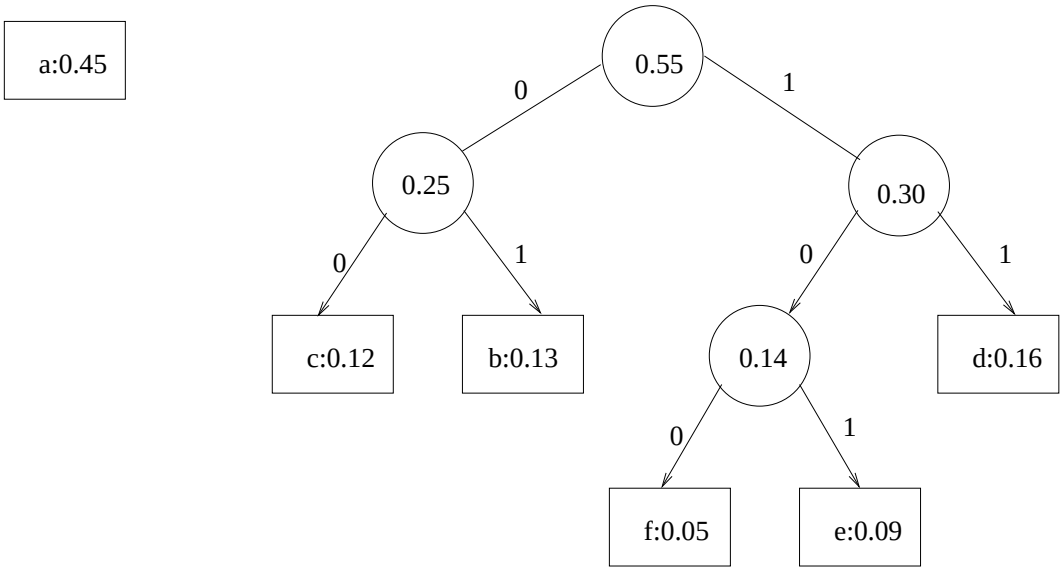
3. $i = 2$ avec $f(z) = 0.25$:



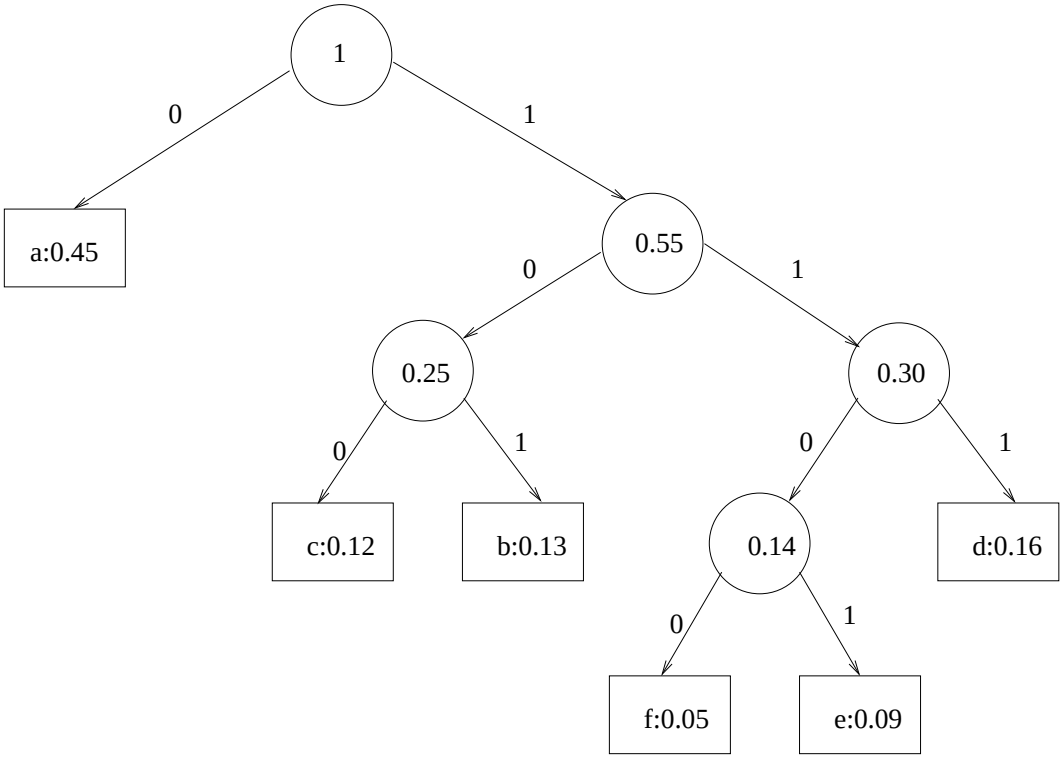
4. $i = 3$ avec $f(z) = 0.30$:



5. $i = 4$ avec $f(z) = 0.55$:



6. $i = 5$ avec $f(z) = 1$:



5.4.4 Optimalité de l'algorithme de codage de Huffman

Nous voulons montrer que l'arbre de compression Huffman obtenu avec une paire (C, f) où C est un alphabet fini et f est une distribution de probabilité a un coût optimal.

Soit x l'élément de C qui a la plus petite fréquence et y l'élément de C qui a la seconde plus petite fréquence. Nous montrons dans un premier temps qu'il existe un arbre de compression optimal dans lequel x et y ont le même parent.

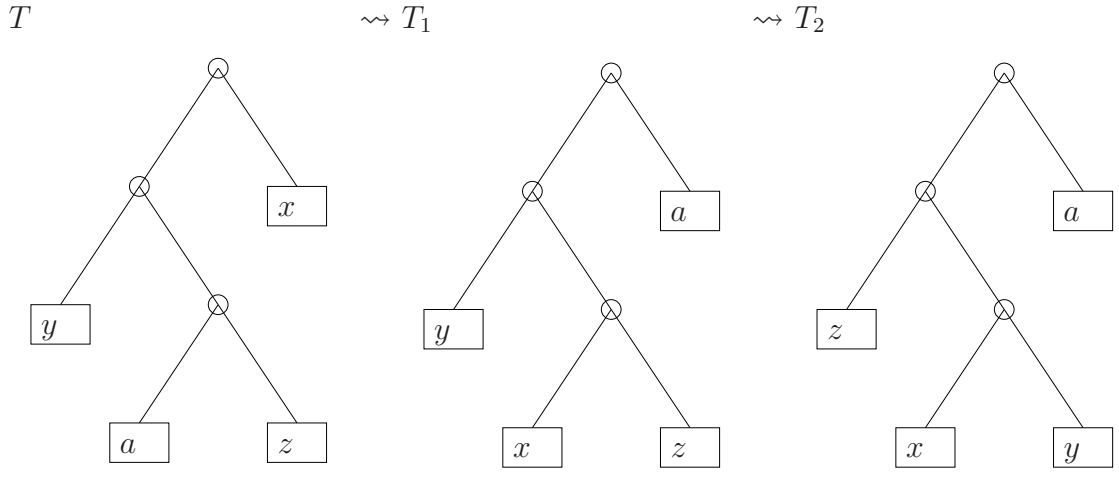
Soit T un arbre de compression optimal. Notons que T peut être transformé en un arbre optimal T_1 dans lequel x a une profondeur maximale : si x n'a pas déjà une profondeur maximale dans T, il existe un élément a dans T qui a une profondeur maximale et $d(a) > d(x)$. Par hypothèse $f(x) \leq f(a)$. En interchangeant x et a dans T, on obtient un autre arbre de compression T_1. On a

$$A(T) - A(T_1) = (f(a) - f(x))(\underbrace{d(a) - d(x)}_{>0}) \geq 0.$$

On constate que $f(a) = f(x)$ et que $A(T) = A(T_1)$. Ensuite, nous montrons que x a un frère z dans T_1. Dans le cas contraire, x peut être "monté" et a une profondeur plus petite, et donc réduit le coût de T_1, ce qui contredit l'optimalité de T_1. Notons que par hypothèse sur y, nous avons $f(z) \geq f(y)$. En interchangeant z et y, on obtient un autre arbre de compression T_2. Nous avons

$$A(T_1) - A(T_2) = (f(z) - f(y))(d(z) - d(y)) \geq 0.$$

Comme $d(z) \geq d(y)$ (Notons que $d(z)$ est la profondeur maximale dans T_1), nous voyons que $A(T_1) = A(T_2)$. Finalement, on a construit un arbre de compression optimal dans lequel x et y ont le même parent.



Pour montrer que l'arbre de Huffman est un arbre de compression optimal pour (C, f), nous procédons par induction sur n, le nombre de caractères dans C. Si $|C| = 2$, il n'y a rien à montrer.

Supposons maintenant que $|C| > 2$. Soit le nouvel alphabet $\overline{C} = C - \{x, y\} \cup \{z\}$, qui contient $|C| - 1$ éléments et où $f(z) = f(x) + f(y)$. Par hypothèse d'induction, l'algorithme de Huffman construit un arbre de compression optimal $\overline{T}$ pour $\overline{C}$. Supposons que le coût de l'arbre de Huffman sur C n'est pas optimal, et soit T_1 un arbre de compression optimal pour C dans lequel x et y ont le même parent (un tel arbre existe par la preuve ci-dessus). Comme x et y ont le même père dans T, on a

$$\begin{aligned} A(T) &= A(\overline{T}) - d(z)(f(x) + f(y)) + (d(z) + 1)(f(x) + f(y)) \\ &= A(\overline{T}) + f(x) + f(y). \end{aligned}$$

De la même manière, $A(T_1) = A(\overline{T}_1) + f(x) + f(y)$, où $\overline{T}_1$ est obtenu de T_1 en enlevant x et y et en donnant à l'ancêtre commun de x et y la valeur $f(x) + f(y)$. Comme, par induction, $A(\overline{T})$ est optimal, on a

$$\begin{aligned} A(T) &= A(\overline{T}) + f(x) + f(y) \\ &\leq A(\overline{T}_1) + f(x) + f(y) \\ &= A(T_1). \end{aligned}$$

Par optimalité de T_1, on voit que $A(T) = A(T_1)$, et donc T est un arbre de compression optimal.

6
Algorithmes de tri

On estime que plus de 25% du temps de calcul utilisé commercialement est passé pour faire des tris (*sorting*). Le développement de méthodes de tri puissantes est donc très utile et important.

Qu'est-ce que "trier", et que signifie "être efficace" dans ce contexte ? Le problème du tri peut être énoncé de manière précise comme suit :

Problème: Tri
Input: Des objets $s_0, \ldots, s_{N-1}$, dont chacun est muni d'une clé k_i appartenant à un ensemble totalement ordonné (par exemple $(\mathbb{Z}, \leq)$).
Output: Permutation π de $0, \ldots, N-1$ telle que $k_{\pi(0)} \leq k_{\pi(1)} \leq \cdots \leq k_{\pi(N-1)}$.

Nous développerons des algorithmes de tri qui n'utilisent que des comparaisons entre les clés. Nos algorithmes *modifieront* l'input afin d'obtenir un output pour lequel les clés seront triées dans l'ordre croissant.

De manière générale, on distingue les algorithmes de tri internes et externes :

- Les algorithmes de tri *internes* supposent que l'accès aléatoire à chaque élément est possible avec le même coût. C'est le cas si les données sont toutes stockées dans la mémoire vive.
- Les algorithmes de tri *externes* supposent que objets sont stockés sur des média extérieurs et qu'ils ne peuvent pas être accédés avec le même temps.

Dans ce chapitre, nous regarderons uniquement des algorithmes de tri internes.

Nous supposons que les clés sont des nombres entiers, et que les objets sont stockés dans un tableau (array). Nous pouvons représenter les objets eux-mêmes (que nous appelons aussi des éléments) avec la structure

```
typedef struct {
```

```
   int key;
   Payload P;
} element;
```

et nos objets sont stockés dans un tableau

```
element a[N];
```

6.1 ALGORITHMES ÉLÉMENTAIRES

Dans cette section nous étudierons quatre algorithmes de tri :

- Selection Sort
- Insertion Sort
- Shell Sort
- Bubble Sort

Quand nous mesurons les temps de parcours de ces algorithmes nous regardons deux propriétés :

- **Le nombre de comparaisons de clés** C
- **Le nombre de mouvements** M **(déplacements)**

Dans les deux cas nous nous intéressons au meilleur des cas, au pire des cas et au comportement moyen. Typiquement, il faut aux méthodes décrites dans cette section $O(N^2)$ comparaisons de clés pour trier N objets dans le pire des cas.

Remarque : Techniquement, un *mouvement* consiste à copier un élément d'un endroit vers un autre (par exemple d'une position dans le tableau a vers une position en mémoire). Cependant, la plupart des algorithmes dans ce chapitre font exclusivement des *échanges* d'éléments (échanger l'élément à la position $a[i]$ avec celui à la position $a[j]$).
Formellement, pour faire un échange il faut 3 mouvements (on copie le contenu de $a[i]$ vers une position temporaire t en mémoire, on copie le contenu de $a[j]$ vers $a[i]$, et on copie finalement t vers $a[j]$).
Cependant, *nous compterons un échange comme un seul mouvement*. En effet nous nous intéressons surtout au comportement asymptotique des algorithmes (donc avec la notation O), et celui-ci reste le même si le nombre de mouvements est multiplié par une constante.

Soient $C_{\min}(N)$, $C_{\max}(N)$ et $C_{\text{avg}}(N)$ le nombre de comparaisons qu'utilise un algorithme dans le *meilleur cas*, le *pire des cas*, et le *cas moyen*, respectivement. Par "cas moyen", on entend le comportement de l'algorithme en moyenne, en supposant que chaque permutation possible de l'input a la même probabilité. $C_{\text{avg}}(N)$ est donc l'*espérance* du nombre de comparaisons.

De même, nous notons par $M_{\min}(N)$, $M_{\max}(N)$, et $M_{\text{avg}}(N)$ le nombre de mouvements effectués par un algorithme dans le *meilleur cas*, le *pire des cas*, et le *cas moyen*. (Nous rappelons qu'un échange de deux éléments est considéré comme un seul mouvement.)

Remarques :
- **(1)** Quand nous parlons du *temps de parcours* d'un algorithme de tri nous voulons en général dire la somme du nombre de mouvements et du nombre de comparaisons dont il a besoin.
- **(2)** Soit $a[0], \ldots, a[N-1]$ le tableau contenant nos objets. Nous abuserons parfois de la notation en disant par exemple que l'élément $a[i]$ est plus petit que l'élément $a[j]$ (alors que formellement il faudrait dire que *la clé de l'élément* $a[i]$ *est plus petite que la clé l'élément* $a[j]$).
- **(3)** Soit $a[0], \ldots, a[N-1]$ le tableau contenant nos objets. Quand nous parlons du *premier élément* du tableau (ou de la suite) nous voulons dire $a[0]$, quand nous parlons du *deuxième élément* nous voulons dire $a[1]$. Donc de la même manière quand nous parlons du $i^{ème}$ *élément* de la suite nous voulons dire $a[i-1]$.

Dans tout le chapitre nous posons les hypothèses suivantes :
- Les clés sont les nombres $1, \ldots, N$.
- Quand nous condidérons le comportement *moyen* de nos algorithmes, nous supposons que les $N!$ permutations possibles des clés ont toutes la même probabilité.

6.1.1 Selection Sort

L'idée de cet algorithme est de trier une suite en déterminant son plus petit élément, puis son deuxième plus petit élément, puis son troisième plus petit, etc.

Plus précisément : Trouver la position j_0 du plus petit élément dans la suite puis échanger $a[0]$ et $a[j_0]$. Ensuite, déterminer la position j_1 du plus petit l'élément parmi $a[1], \ldots, a[N-1]$, et échanger $a[1]$ et $a[j_1]$. Nous continuons de cette manière jusqu'à ce que tous les éléments soient à la bonne position.

Analyse

Nombre de comparaisons : Pour trouver le plus petit élément parmi ℓ possibilités, il faut toujours faire $\ell - 1$ comparaisons. Donc :

$$C_{\max}(N) = C_{\min}(N) = C_{\text{avg}}(N) = \sum_{i=0}^{N-2}(N-1-i) = \frac{N(N-1)}{2} = O(N^2).$$

Nombre de mouvements : Nous rappelons d'abord que les $N!$ permutations possibles des clés ont toutes la même probabilité. Donc chacun des nombres k, $1 \leq k \leq N$ apparaît

Algorithme 34 SELECTIONSORT(a)

Input: Suite a d'éléments avec des clés entières.
Output: Transformation de a telle que $a[i].key \leq a[i+1].key$ pour $0 \leq i < N-1$

```
1: for i = 0, ..., N − 2 do
2:    min ← i
3:    for j = i + 1, ..., N − 1 do
4:       if a[j].key < a[min].key then
5:          min ← j
6:       end if
7:    end for
8:    if min ≠ i then
9:       Echanger a[min] et a[i]
10:   end if
11: end for
```

à une position donnée i avec probabilité $1/N$.
On obtient :

$$M_{\max}(N) = N-1.$$

$$M_{\min}(N) = 0 \quad \text{(la ligne 9 de l'algorithme n'effectue l'échange que s'il est nécessaire).}$$

$$M_{\text{avg}}(N) = \tfrac{N-1}{N} + \tfrac{N-2}{N-1} + \cdots + \tfrac{1}{2} = N - \ln(N) - \Theta(1) \quad \text{(exercice).}$$

Exemple :

Position	0	1	2	3	4	5	6
	15	2	43	17	4	7	47
	2	15	43	17	4	7	47
	2	4	43	17	15	7	47
	2	4	7	17	15	43	47
	2	4	7	15	17	43	47
	2	4	7	15	17	43	47
	2	4	7	15	17	43	47

Est-il possible de rendre l'algorithme plus efficace en choisissant une meilleure méthode pour le calcul du minimum à chaque étape ?

Théorème 6.1 *N'importe quel algorithme pour le calcul du minimum de i clés utilise au moins $i-1$ comparaisons (si seulement des comparaisons de clés sont admises).*

Preuve (esquisse). On dessine le graphe qui a pour chacune des i entrées un sommet, et qui a une arête entre chaque paire de sommets qui sont comparés. Par la formule d'Euler pour les arbres (c.f. section 1.3.1), si on effectue moins que $i-1$ comparaisons, ce graphe n'est pas connexe. Mais alors une composante peut être strictement plus grande que l'autre ou aussi strictement plus petite, sans influence sur les résultats des comparaisons.
Mais le minimum se trouvera alors forcément dans des composantes différentes dans ces deux cas. Il s'en suit que dans un de ces cas au moins, l'output de l'algorithme sera faux. ■

6.1.2 Insertion Sort

Algorithme de base

L'idée est d'insérer les éléments un par un dans une suite triée. En insérant $a[i]$, il faut s'assurer que la suite $a[0], \ldots, a[i-1]$ est déjà triée et insérer $a[i]$ à la bonne position.

Cet algorithme peut être vu comme la méthode utilisée par un joueur de cartes qui place avec la main droite les cartes distribuées, une à une, dans sa main gauche, en plaçant au bon endroit chaque carte dans les cartes déjà triées qu'il tient dans la main gauche.

Algorithme 35 InsertionSort(a)

Input: Suite a d'objets avec des clés entières.
Output: Transformation de a telle que $a[i].key \leq a[i+1].key$ pour $0 \leq i < N-1$

```
1: for i = 1, ..., N-1 do
2:     j ← i − 1
3:     t ← a[i]
4:     while a[j].key > t.key and j ≥ 0 do
5:         a[j+1] ← a[j]
6:         j ← j − 1
7:     end while
8:     a[j+1] ← t
9: end for
```

Exemple

Position	0	1	2	3	4	5	6
	15	2	43	17	4	7	47
	2	15	43	17	4	7	47
	2	15	43	17	4	7	47
	2	15	17	43	4	7	47
	2	4	15	17	43	7	47
	2	4	7	15	17	43	47
	2	4	7	15	17	43	47

Analyse

Nombre de comparaisons : Pour trouver à quelle position ajouter le $i^{\text{ème}}$ élément $a[i-1]$ dans $a[0], \ldots, a[i-2]$, nous avons au moins 1 et au plus $i-1$ comparaisons à faire. On a donc

$$\begin{aligned} C_{\max}(N) &= \sum_{i=2}^{N}(i-1) = O(N^2) \\ C_{\min}(N) &= N-1. \end{aligned}$$

Il est aussi possible de montrer que

$$C_{\text{avg}}(N) = O(N^2).$$

Nombre de mouvements : Pour insérer le $i^{\text{ème}}$ élément $a[i-1]$ dans $a[0], \ldots, a[i-2]$ nous avons au moins 2 et au plus $2+(i-1)$ mouvements à effectuer (pourquoi ?). Donc

$$\begin{aligned} M_{\max}(N) &= 2(N-1) + \sum_{i=2}^{N}(i-1) = O(N^2) \\ M_{\min}(N) &= 2(N-1), \end{aligned}$$

et il est aussi possible de montrer que

$$M_{\text{avg}}(N) = O(N^2).$$

Binary Insertion Sort

Dans la version de base de InsertionSort, l'algorithme fait une recherche linéaire pour trouver dans $a[0], \ldots, a[i-1]$ la position à laquelle $a[i]$ doit être inséré. Cependant puisque

$a[0], \ldots, a[i-1]$ est triée, nous pouvons aussi utiliser l'algorithme de recherche binaire (ce qui réduit le nombre de comparaisons nécessaires, mais pas le nombre de mouvements).

Dans ce qui suit, nous supposons que la procédure BinSearch(b, x) retourne, étant donné une suite triée b de longueur N, la position j à laquelle il faudrait insérer x pour que b reste triée. Plus formellement, l'output j doit vérifier l'une des 3 propriétés suivantes :

- $j \neq 0, N$ et $b[j-1].key < x.key \leq b[j].key$.
- $j = 0$ et $x.key \leq b[0].key$.
- $j = N$ et $b[N-1] < x.key$.

Par "insérer x à la position j", nous entendons copier $a[i]$ vers $a[i+1]$ pour tout $i = N-1, \ldots, j$, puis copier x vers $a[j+1]$.

Algorithme 36 InsertionSortBin(a)

Input: Suite a d'éléments avec des clés entières.
Output: Transformation de a telle que $a[i].key \leq a[i+1].key$ pour $0 \leq i < N-1$

1: Poser $a^{(0)} := (a[0])$
2: **for** $i = 1, \ldots, N-1$ **do**
3: Poser $j \leftarrow$ BinSearch$(a^{(i-1)}, a[i])$
4: Insérer $a[i]$ dans $a^{(i-1)}$ à la position j, et appeler la suite résultante $a^{(i)}$
5: **end for**

Analyse

Nombre de comparaisons : Nous supposons que toutes les clés dans la suite sont distinctes. Pour insérer $a[i]$ dans la suite $a^{(i-1)}$, il nous faut faire une recherche binaire. Le nombre minimum de comparaisons pour insérer $a[i]$ est donc $\lfloor \log_2(i) \rfloor$. Le nombre maximum de comparaisons est $\lceil \log_2(i) \rceil$. Donc

$$\begin{aligned} C_{\min}(N) &= \sum_{i=1}^{N-1} \lfloor \log_2(i) \rfloor \\ &\leq \log_2((N-1)!) \\ &= O(N \log(N)). \end{aligned}$$

Pour des N arbitrairement grands, il existe des suites qui atteignent cette borne. De la même façon, poue le nombre maximal de comparaison on a

$$C_{\max}(N) = O(N \log(N)).$$

Le nombre moyen de comparaisons de l'algorithme de recherche binaire est $O(\log(N))$. Ainsi :

$$C_{\text{avg}}(N) \quad = \quad O(N \log(N)).$$

Nombre de mouvements : Le nombre de mouvements ne change pas avec BinSearch. Les calculs restent donc les mêmes que pour la version de base de InsertionSort.

6.1.3 Shell Sort

Cet algorithme de tri a été développé en 1959 par D.L. Shell. La définition suivante est la base de ShellSort.

Définition. Soit $0 \leq \ell < N$. On dit qu'une suite de clés $k_0, \ldots, k_{N-1}$ est *ℓ-triée* si pour tout $0 \leq i < N - \ell$ on a $k_i \leq k_{i+\ell}$.

En particulier, une suite triée est donc la même chose qu'une suite 1-triée.

L'idée est de choisir une suite d'*incréments* $h_t > h_{t-1} > \cdots > h_1 = 1$, et d'utiliser InsertionSort pour transformer la suite originale en une suite h_t-triée S_t, puis d'utiliser de nouveau InsertionSort pour tranformer S_t en une suite h_{t-1}-triée S_{t-1}, etc, pour finalement obtenir une suite h_1-triée (donc triée) S_1. Un bon choix d'incréments diminue le nombre de mouvements nécessaires par rapport à InsertionSort.

Exemple

Position	0	1	2	3	4	5	6
h_3=5	15	2	43	17	4	7	47
h_2=3	7	2	43	17	4	15	47
h_1=1	7	2	15	17	4	43	47
	2	4	7	15	17	43	47

Quand $h_3 = 5$, on considère les sous-suites suivantes obtenues par incrément de 5 :

$$\begin{array}{cccccc} 15 & 7 & \xrightarrow{\text{InsertionSort}} & \cdots & \longrightarrow & 7 \quad 15 \\ 2 & 47 & \xrightarrow{\text{InsertionSort}} & \cdots & \longrightarrow & 2 \quad 47 \\ 43 & & & & & \\ 17 & & & & & \\ 4 & & & & & \end{array}$$

Puis on remplace dans leur position originale les sous-suites triées :

$$7 \quad 2 \quad 43 \quad 17 \quad 4 \quad 15 \quad 47$$

Puis $h_3 = 3$ et on considère les sous-suites suivantes obtenues par incrément de 3 :

$$\begin{array}{cccccccc} 7 & 17 & 47 & \xrightarrow{\text{InsertionSort}} & 7 & 17 & 47 \\ 2 & 4 & & \xrightarrow{\text{InsertionSort}} & 2 & 4 & \\ 43 & 15 & & \xrightarrow{\text{InsertionSort}} & 15 & 43 & \end{array}$$

A nouveau, on remplace dans leur position originale les sous-suites triées :

$$7 \quad 2 \quad 15 \quad 17 \quad 4 \quad 43 \quad 47$$

Le cas $h_1 = 1$ revient à trier, avec InsertionSort, la suite entière.

Dans cet exemple, le nombre de mouvements est 6, par opposition aux 8 mouvements nécessaires si l'on utilise le tri par insertion.

Algorithme 37 ShellSort(a, h)

Input: Suite a d'éléments avec des clés entières, suite d'incréments $h_t > h_{t-1} > \cdots > h_1 = 1$.

Output: Transformation de a telle que $a[i].key \leq a[i+1].key$ pour $0 \leq i < N - 1$

```
1: for k = t, ..., 1 do
2:    h ← h_k
3:    for i = h, ..., N − 1 do
4:       v ← a[i]
5:       j ← i
6:       while j ≥ h and a[j − h].key > v.key do
7:          a[j] ← a[j − h]
8:          j ← j − h
9:       end while
10:      a[j] ← v
11:   end for
12: end for
```

Analyse

La partie la plus importante est de trouver le meilleur choix des incréments pour que le nombre de mouvements aussi petit que possible. Une réponse complète à cette question n'est toujours pas connue. On peut cependant démontrer les faits suivants :

– Si une suite est k-triée et l'algorithme ci-dessus produit une suite h-triée pour $h < k$, alors la nouvelle suite reste k-triée.

- Si une suite de longueur N est 2-triée et 3-triée, alors on peut la trier avec un passage de InsertionSort en utilisant N comparaisons.
- Pour tout $n \geq 1$, si une suite de longueur N est $2n$-triée et $3n$-triée, alors la suite peut être n-triée en un passage de InsertionSort en utilisant N comparaisons.
- Soit $1 = h_1 < h_2 < h_3 < \cdots$ la suite contenant tous les entiers de la forme $2^a 3^b$ pour $a, b \geq 0$. Le temps de parcours de l'algorithme ShellSort avec cette suite d'incréments est $O(N(\log N)^2)$ (nombre de comparaisons et de mouvements dans le pire des cas)

6.1.4 Bubble Sort

L'idée de BubbleSort est très simple : L'algorithme procède par étapes, où à chaque étape nous parcourons la suite et comparons les éléments adjacents. Nous les échangeons s'ils ne sont pas dans le bon ordre. Nous continuons ces étapes jusqu'à ce qu'il n'y ait plus d'échanges nécessaires.

Après la première étape (aussi appelé premier passage), $a[N-1]$ est l'élément maximal de la suite. Après le deuxième passage, $a[N-2]$ est l'élément maximal parmi $a[0], \ldots, a[N-2]$ (donc le $2^{\text{ème}}$ plus grand 'élément de la suite). De même, après la $i^{\text{ème}}$ étape, $a[N-i]$ est l'élément maximal parmi $a[0], \ldots, a[N-i]$ (donc le $i^{\text{ème}}$ plus grand 'élément de la suite).

Donc après l'étape i, l'élément maximal parmi $a[0], \ldots, a[N-i]$ a été déplacé jusqu'à la fin de cette suite (à la position $a[N-i]$) ; Comme une bulle qui monte à la surface de l'eau (et donc le nom *bubble sort*).

Exemple

Position	0	1	2	3	4	5	6
1^{er} pass.	**15**	**2**	43	17	4	7	47
	2	15	**43**	**17**	4	7	47
	2	15	17	**43**	**4**	7	47
	2	15	17	4	**43**	**7**	47
	2	15	17	4	7	43	47
$2^{\text{ème}}$ pass.	2	15	**17**	**4**	7	43	47
	2	15	4	**17**	**7**	43	47
	2	15	4	7	17	43	47
$3^{\text{ème}}$ pass.	2	**15**	**4**	7	17	43	47
	2	4	**15**	**7**	17	43	47
	2	4	7	15	17	43	47
Résultat	2	4	7	15	17	43	47

Algorithme 38 BUBBLESORT(a)

Input: Suite a d'éléments avec des clés entières.
Output: Transformation de a telle que $a[i].key \leq a[i+1].key$ pour $0 \leq i < N-1$

```
1:  Continue ← TRUE
2:  r ← N − 1 ;
3:  while Continue est TRUE do
4:     Continue ← FALSE
5:     for i = 0, . . . , r − 1 do
6:        if a[i].key > a[i + 1].key then
7:           Swap a[i] and a[i + 1]
8:           Continue ← TRUE ;
9:        end if
10:    end for
11:    r ← r − 1
12: end while
```

Analyse

Meilleur cas : $N-1$ comparaisons et pas de mouvements (si la suite est déjà trée).

$$\begin{aligned} C_{\min}(N) &= N-1 \\ M_{\min}(N) &= 0. \end{aligned}$$

Pire des cas : Quand la suite est triée de façon décroissante (pourquoi ?).

$$\begin{aligned} C_{\max}(N) &= \frac{N(N-1)}{2} = O(N^2), \\ M_{\max}(N) &= \sum_{i=1}^{N}(i-1) = O(N^2). \end{aligned}$$

Cas moyen : On peut montrer (plus difficilement) que

$$C_{\text{avg}}(N) = M_{\text{avg}}(N) = O(N^2).$$

Nous terminons ici notre discussion des algorithmes de tri élémentaires. Nous étudions maintenant des algorithmes de tri plus sophistiqués avec des temps de parcours bien meilleurs.

6.2 QUICKSORT

L'algorithme QUICKSORT a été inventé en 1962 par C.A.R. Hoare. Le nom "quicksort" est dû au fait que c'est l'un des algorithmes de tri les plus rapides connus.

Dans des applications typiques de tri, les éléments associés aux clés peuvent être très grands. Par exemple, il pourrait s'agir d'un site web formé de beaucoup d'images, de fichiers de son et de vidéo, etc. Donc, la quantité de mémoire associée à un élément est une ressource précieuse qu'il ne faut pas gaspiller. Nous appelons ce type de mémoire du "mémoire élément" pour pouvoir distinguer ce type de mémoire de celui qu'utilise un programme pour fonctionner. Nous disons qu'un algorithme est "in-place" s'il utilise seulement une quantité de mémoire élément supplémentaire $O(1)$. QUICKSORT est un algorithme in-place et diviser-pour-régner. Dans le pire des cas son temps de parcours est $O(N^2)$ pour trier une suite de longueur N (donc pas meilleur que les algorithmes de la section précédante), mais son temps de parcours *en moyenne* est

$$O(N \log(N)).$$

L'idée est de choisir un élément arbitraire b appelé le *pivot* dans la suite $a[0], \ldots, a[N-1]$ et de diviser la suite en deux sous-suites S_1 et S_2 avec les propriétés suivantes :

- S_1 contient les éléments $a[i]$ avec
$$a[i].key \leq b.key,$$
- S_2 contient les $a[i]$ avec
$$a[i].key > b.key.$$

L'algorithme QUICKSORT est ensuite appliqué récursivement à ces deux sous suites.

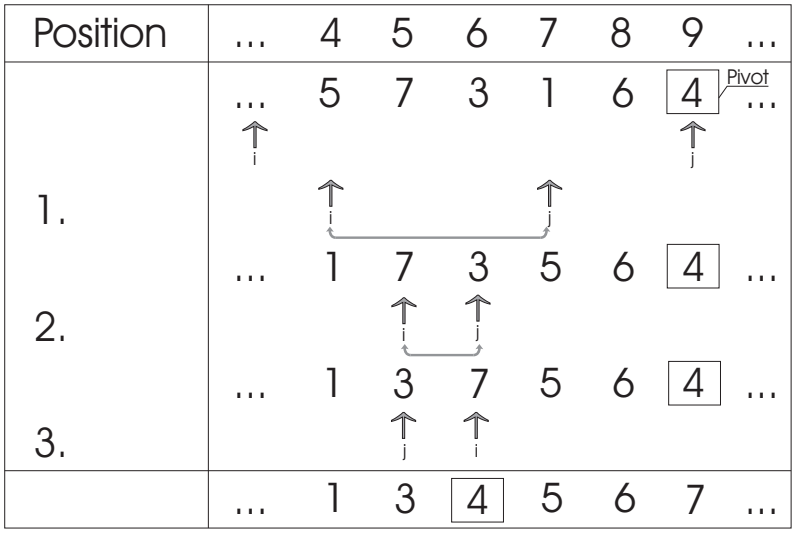
Comment pouvons-nous construire une version in-place de cet algorithme ?

Une sous-suite de la suite originale est spécifiée de façon unique par deux indices ℓ ($a[\ell]$ est l'élément le plus à gauche) et r ($a[r-1]$ est l'élément le plus à droite). A chaque appel récursif de QUICKSORT nous appelons la routine en spécifiant ℓ et r. L'étape la plus importante de l'implémentation de cet algorithme est la subdivision en sous-suites.

Ici : L'élément pivot est le dernier élément de la suite, donc $a[r-1]$.

Nous prenons deux pointeurs p et q ; Au début p pointe vers l'élément $a[r-1]$ et q vers l'élément $a[\ell]$. p se déplace vers la gauche jusqu'à ce qu'il trouve un élément plus petit que $a[r-1]$ (le pivot) ; de la même façon, q se déplace vers la droite jusqu'à ce qu'il trouve un élément plus grand que $a[r-1]$ (le pivot). Les contenus de $a[p]$ et $a[q]$ sont échangés, et le processus est répété jusqu'à ce que $p < q$. Puis $a[r-1]$ et $a[q]$ sont échangés.

Exemple (de l'étape de subdivision)



Donc ici $\ell = 4$ et $r = 10$ (attention, la suite qui nous intéresse est $a[\ell], a[\ell+1], \ldots, a[r-1]$, soit $r-\ell$ éléments). Le pivot est $a[r-1]$, donc dans notre exemple on a $pivot = a[9] = 4$. Au départ p pointe vers $a[r-1]$ (donc ici $a[9]$), et q vers $a[\ell]$ (donc ici $a[4]$).

• Nous comparons $a[p]$ avec le pivot et voyons que $pivot \leq a[p]$ $(4 \leq 4)$, donc nous continuons avec p.
• Nous déplaçons p vers la gauche (donc maintenant $p = 8$) et comparons $a[p]$ avec le pivot. Puisque $pivot \leq a[p]$ $(4 \leq 6)$ nous continuons avec p.
• Nous déplaçons p vers la gauche (donc maintenant $p = 7$) et comparons $a[p]$ avec le pivot. Cette fois nous avons $a[p] < pivot$ $(1 < 4)$, nous laissons donc p en place et regardons q.
• Nous comparons $a[q]$ avec le pivot et voyons que $pivot < a[q]$ $(4 < 5)$ donc nous laissons q en place.
• Nous échangeons a présent $a[q]$ et $a[p]$. Notre suite est donc maintentant $1, 7, 3, 5, 6, 4$.
• Nous recommençons.

• Nous comparons $a[p]$ ($p = 7$ toujours) avec le pivot et voyons que $pivot \leq a[p]$ $(4 \leq 5)$, nous continuons donc avec p.
• Nous déplaçons p vers la gauche (donc maintenant $p = 6$). $a[p] < pivot$ $(3 < 4)$, nous laissons donc p en place et regardons q.
• Nous comparons $a[q]$ ($q = 4$ toujours) avec le pivot et voyons que $a[q] \leq pivot$ $(1 \leq 4)$ donc nous continuons avec q.
• Nous déplaçons q vers la droite (donc maintenant $q = 5$) et comparons $a[q]$ avec le pivot. Cette fois nous avons $pivot < a[q]$ $(4 < 7)$ nous laissons donc q en place.
• Nous échangeons a présent $a[q]$ et $a[p]$. Notre suite est donc maintentant $1, 3, 7, 5, 6, 4$.
• Nous recommençons.

• Nous comparons $a[p]$ ($p = 6$ toujours) avec le pivot et voyons que $pivot \leq a[p]$ $(4 \leq 7)$, nous continuons donc avec p.

• Nous déplaçons p vers la gauche (donc maintenant $p = 5$) $a[p] < pivot$ $(3 < 4)$, nous laissons donc p en place et regardons q.
• Nous comparons $a[q]$ ($q = 5$ toujours) avec le pivot et voyons que $a[q] \leq pivot$ $(3 \leq 4)$ donc nous continuons avec q.
• Nous déplaçons q vers la droite (donc maintenant $q = 6$) et comparons $a[q]$ avec le pivot. Cette fois nous avons $pivot < a[q]$ $(4 < 7)$ nous laissons donc q en place.
• Comme $p < q$ $(5 < 6)$ ne n'échangeons pas $a[p]$ et $a[q]$, mais sortons de la boucle principale.

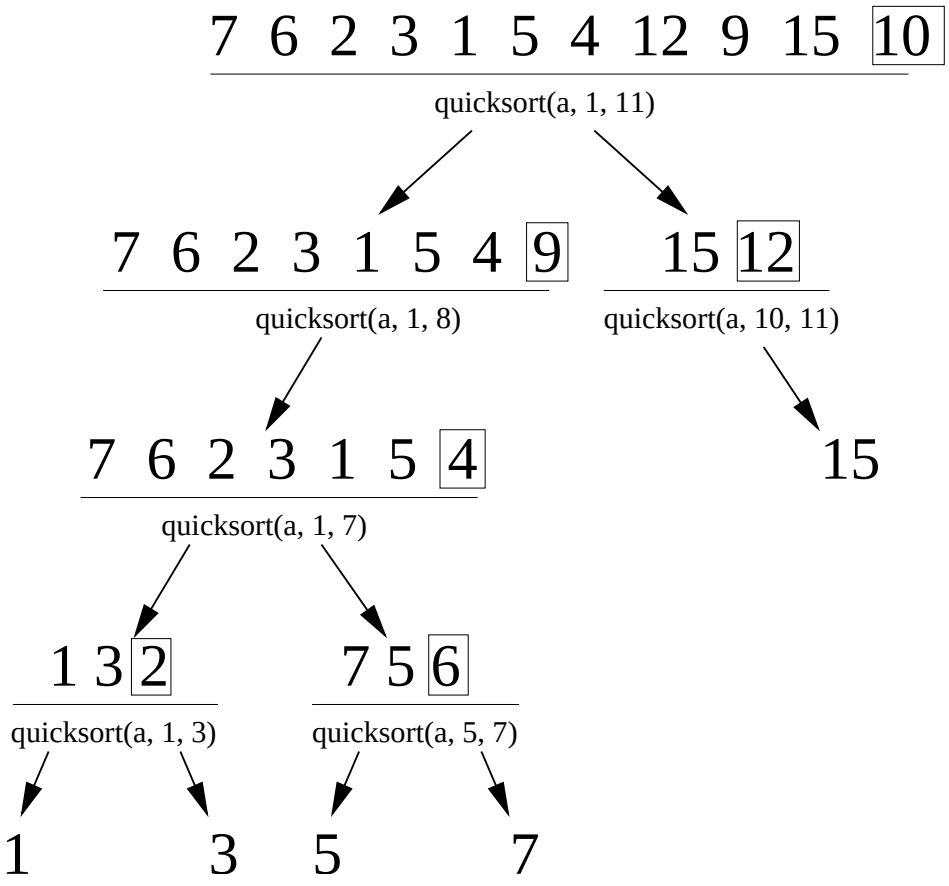
• Nous echangeons $a[q]$ et $a[r-1]$ (donc $a[6]$ et $a[9]$) pour obtenir comme suite $1, 3, 4, 5, 6, 7$. Les deux sous-suites sont donc $1, 3$ $(a[4], a[5])$ et $5, 6, 7$ $(a[7], a[8], a[9])$ avec le pivot 4 au milieu.
• Nous appelons donc QuickSort$(a, 3, 6)$ et QuickSort$(a, 7, 10)$

Algorithme 39 QuickSort(a, ℓ, r)

Input: Suite a d'éléments avec des clés entières, indices ℓ et r
Output: Transformation de $a[\ell], \ldots, a[r-1]$ telle que $a[i].key \leq a[i+1].key$ pour $\ell \leq i < r-1$

```
1: if r ≤ ℓ + 1 then
2:    return
3: end if
4: q ← ℓ
5: p ← r − 1
6: pivot ← a[r − 1].key
7: while p > q do
8:    while a[p].key ≥ pivot and p > l do
9:       p ← p − 1
10:   end while
11:   while a[q].key ≤ pivot and q < r − 1 do
12:      q ← q + 1
13:   end while
14:   if p > q then
15:      Échanger a[p] et a[q]
16:   end if
17: end while
18: if q ≠ r − 1 then
19:    Echanger a[q] et a[r − 1]
20: end if
21: QuickSort(a, ℓ, q)
22: QuickSort(a, q + 1, r)
```

Exemple :



Analyse

On voit facilement que le temps de parcours est au plus $O(N^2)$. Pour une suite ordonnée de façon décroissante, le nombre de comparaisons et de déplacements est $\Omega(N^2)$, si le pivot est choisi comme précédemment. Ainsi,

$$C_{\max}(N) = \Omega(N^2),$$

$$M_{\max}(N) = \Omega(N^2).$$

On voit facilement que le temps de parcours est $\Omega(N)$, puisque chaque élément de la suite doit être inspecté. Pour une suite ordonnée de façon croissante, le nombre de mouvements est 0. Le nombre de comparaisons doit satisfaire $C_{\min}(N) = 2C_{\min}(N/2) + O(N)$. Donc en utilisant le théorème 2.1, on peut conclure que

$$C_{\min}(N) = O(N \log(N))$$

$$M_{\min}(N) = 0.$$

Analysons maintenant le comportement moyen de QUICKSORT. Soit $T_k(N)$ le coût total (comparaisons + mouvements) moyen de QUICKSORT si le pivot est le $k^{\text{ème}}$ élément le plus petit, et $T_{\text{avg}}(N)$ le coût moyen de QUICKSORT pour trier une suite de longueur N. Alors

$$T_{\text{avg}}(N) = \frac{1}{N}\sum_{k=1}^{N} T_k(N),$$

et

$$T_k(N) = \underbrace{bN}_{\text{sous-division}} + T_{\text{avg}}(k-1) + T_{\text{avg}}(N-k),$$

pour une constante b.

$$\begin{aligned} T_{\text{avg}}(N) &= bN + \frac{1}{N}\sum_{k=1}^{N}\left(T_{\text{avg}}(k-1) + T_{\text{avg}}(N-k)\right) \\ &= bN + \frac{2}{N}\sum_{k=1}^{N} T_{\text{avg}}(k). \end{aligned}$$

De là, on peut montrer (par induction) que

$$T_{\text{avg}}(N) \leq cN\log(N),$$

pour une constante c indépendante de N.

6.2.1 Variantes

Le choix du pivot est crucial pour la performance de l'algorithme QUICKSORT. Il y a plusieures stratégies pour choisir le pivot, les plus connues sont 3-median strategy (*stratégie 3-médianes*) et randomized strategy (*stratégie aléatoire*) :

- *3-Median Strategy* : On choisit comme pivot la médiane de 3 éléments de la suite (par exemple la médiane du premier, dernier et élément du milieu).
- *Randomized Strategy* : On choisit comme pivot un élément aléatoire dans la suite. Avec cette stratégie, la probabilité que QUICKSORT ait une mauvaise performance sur une suite donnée est très petite.

6.3 HEAP SORT

Heapsort est une technique de tri qui avec un temps de parcours *moyen* comparable à celui de QUICKSORT ($O(N\log(N))$), mais qui a aussi un temps de parcours $O(N\log(N))$ *dans le pire des cas.*

Heapsort est aussi une methode in-place. Un array a de longueur N peut être considéré comme un arbre binaire avec racine de la façon suivante (représentation implicite) : Les

sommets de l'arbre correspondent aux positions $0, \ldots, N-1$. La racine est à la position 0, et les fils des positions i sont aux positions $2i+1$ et $2i+2$, à condition que ces valeurs soient plus petites que N. Nous appelons un tel arbre un "array-tree" binaire.

Il est évident que la hauteur du array-tree binaire d'un array de longueur N est $O(\log(N))$. (Pourquoi ?)

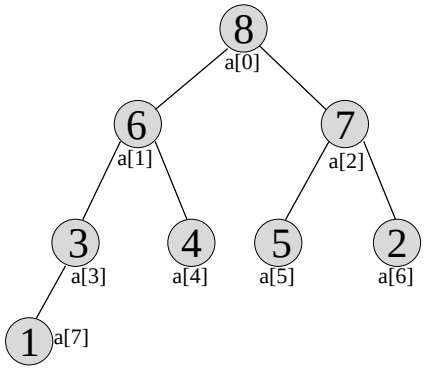
Un tableau a satisfait la *heap property* (*propriété de monceau*) si

$$\begin{array}{llll} a[i].key \geq a[2i+1].key & \text{pour tout } i \text{ vérifiant} & 2i+1 < N \\ a[i].key \geq a[2i+2].key & \text{pour tout } i \text{ vérifiant} & 2i+2 < N \end{array}$$

On appelle un arbre binaire qui correspond à un tableau a un *heap* (*monceau*) si le tableau vérifie la propriété heap.

Exemple : Le tableau `[8,6,7,3,4,5,2,1]` vérifie la propriété de heap, parce que

$$\begin{array}{c} 8 \geq 6, 8 \geq 7 \\ 6 \geq 3, 6 \geq 4; \qquad 7 \geq 5, 7 \geq 2 \\ 3 \geq 1. \end{array}$$



Comme déjà mentionné, un *heap contenant N sommets a une hauteur de $O(\log(N))$.*

6.3.1 Sifting

La procédure de sift (le *criblage* en français) est une procédure efficace qui peut être utilisée pour insérer ou enlever des éléments dans un heap.

Nous apprendrons deux techniques de sifting : le "sift up" et le "sift down". En utilisant la terminologie des graphes, on peut décrire la procédure de SiftDown de l'élément d'indice i dans le tableau a de taille N comme suit :

Si le sommet i est une feuille ou s'il est plus grand que ses fils on ne fait rien. Sinon on échange le sommet i avec le plus grand de ses fils et on recommence ce procédé de façon récursive sur le fils qu'on vient d'échanger.

La performance de SiftDown est facile à analyser :

Algorithme 40 SiftDown(a, i)

Input: Suite a de N 'éléments à clés entières, et un entier i, $0 \leq i < N$ pour lequel $a[i+1], \ldots, a[N-1]$ vérifie la propriété de heap.

Output: Transformation de a telle que $a[i], a[i+1], \ldots, a[N-1]$ vérifie la propriété de heap.

```
1: Swapped ← TRUE
2: while Swapped = TRUE and 2i + 1 < N do
3:    Swapped ← FALSE
4:    Trouver le plus grand fils a[j] de a[i]
5:    if a[j].key > a[i].key then
6:       Echanger a[i] et a[j]
7:       i ← j
8:       Swapped ← TRUE
9:    end if
10: end while
```

Théorème 6.2 SiftDown *utilise au plus* $O(\log(N/i))$ *comparaisons et échanges.*

Preuve. Remarquons d'abord que le nombre de comparaisons est au plus deux fois la distance entre le sommet i et le bas de l'arbre (deux comparaisons par niveau), et le nombre d'échanges est au plus égal à ce nombre. La différence entre la hauteur totale du heap et la hauteur du sommet i est $O(\log(N) - \log(i))$, et l'affirmation s'ensuit. ■

La procédure SiftDown a un analogue appelé SiftUp. La procédure SiftUp peut être appliquée au i-ème sommet, si $a[0], \ldots, a[i-1]$ vérifient déjà la propriété de heap ; après SiftUp on aura alors que $a[0], \ldots, a[i-1], a[i]$ vérifient la propriété de heap. L'idée est d'échanger $a[i]$ avec son parent si nécessaire, et de continuer ensuite avec le parent.

Il est facile de montrer que le coût de cette procédure est $O(\log(i))$ (Exercice.)

6.3.2 Création d'un heap

Comment transformer un tableau donné en un heap ? Nous verrons deux méthodes pour créer un heap : La méthode *top-down* et la méthode *bottom-up*.

La méthode top-down consiste à traverser la suite $a[0], \ldots, a[N-1]$ de la gauche vers la droite. Au pas $i \geq 1$, nous effectuons un SiftUp de l'élément $a[i]$ dans le heap $a[0], \ldots, a[i-1]$ déjà créé. Un argument d'induction montre que le résultat final est aussi un heap.

Le temps de parcours de cet algorithme peut être facilement analysé. A l'étape i, le

Algorithme 41 SiftUp(a, i)

Input: Suite a de N éléments à clés entières, et un entier i, $0 \leq i < N$ avec $a[0], \ldots, a[i-1]$ vérifiant la propriété de heap.
Output: Transformation de a telle que $a[0], \ldots, a[i]$ verifie la propriété de heap.

```
1: Swapped ← TRUE
2: while Swapped = TRUE et i > 0 do
3:    Swapped ← FALSE
4:    j ← ⌊(i − 1)/2⌋      (a[j] est alors le parent de a[i].)
5:    if a[j].key < a[i].key then
6:       Echanger a[j] et a[i]
7:       i ← j
8:       Swapped ← TRUE
9:    end if
10: end while
```

Algorithme 42 TopDownHeapCreate(a)

Input: Suite a d'éléments avec N clés entières.
Output: Transformation de $a[0], \ldots, a[N-1]$ en un heap.

```
1: for i = 1, ..., N − 1 do
2:    SiftUp(a, i)
3: end for
```

coût de SiftDown est $O(\log(i))$, le coût total de l'algorithme est donc

$$\sum_{i=2}^{N} O(\log(i)) = O(\log(N!)) = O(N \log(N)).$$

La deuxième méthode pour créer un heap s'appelle *bottom-up* et consiste à créer plusieurs petits heaps. Cette fois nous traversons $a[0], \ldots, a[N-1]$ de la droite vers la gauche. A chaque étape i la condition heap est vérifiée pour les éléments $a[i], a[i+1], \ldots, a[N-1]$. L'élément $a[i-1]$ est inséré dans le heap en le siftant dans l'arbre binaire original.

Exemple: Creation d'un heap utilisant l'approche bottom-up. Dans cet exemple, les lettres en gras indiquent la partie qui est déjà transformée en heap. Valeurs sous-lignées : Sift-down.

Tableau :	2	1	5	3	**4**	**8**	**7**	**6**
$a[3].key = 3$:	2	1	5	**6**	**4**	**8**	**7**	**3**
$a[2].key = 5$:	2	1	**8**	**6**	**4**	**5**	**7**	**3**
$a[1].key = 1$:	2	**6**	**8**	**3**	**4**	**5**	**7**	**1**
$a[0].key = 2$:	**8**	**6**	**7**	**3**	**4**	**5**	**2**	**1**.

Algorithme 43 BottomUpHeapCreate(a)

Input: Suite a à N éléments, avec clés entiers.
Output: Transformation de $a[0], \ldots, a[N-1]$ en un heap.

```
1: for i = ⌊(N − 2)/2⌋, . . . , 0 do
2:     SiftDown(a, i) ;
3: end for
```

Le temps de parcours de cet algorithme est bien meilleur que celui de l'approche top-down : pour chaque sommet dans l'arbre original, le coût de SiftDown est (au plus) proportionnel à la distance du sommet au dernier niveau de l'arbre.

Pour un arbre complet binaire à $2^h - 1$ sommet (et de hauteur $h-1$), on peut montrer par induction que la somme des distances des sommet au dernier niveau est

$$2^h - (h+1).$$

Il s'ensuit que le temps de parcours de l'approche bottom-up est $O(N)$.

6.3.3 L'algorithme HeapSort

Nous sommes maintenant prêts à définir l'algorithme de tri HeapSort. La démarche est la suivante :

- Nous commençons avec la suite $a[0], \ldots, a[N-1]$.
- Nous la transformons en un heap en utilisant l'approche bottom-up ou top-down.
- Pour chaque $s = 1, \ldots, N-1$, nous échangeons la racine de l'arbre ($a[0]$) avec $a[N-s]$, puis faisons un SiftDown de $a[0]$ dans l'arbre donné par les éléments $a[0], \ldots, a[N-s-1]$.

Analyse

Le temps de parcours de HeapSort est la somme des temps de parcours de la création du heap (en $O(N)$), et des $N-1$ opérations de sift (chacune en $O(\log i)$). Donc, le temps

Algorithme 44 HeapSort(a)

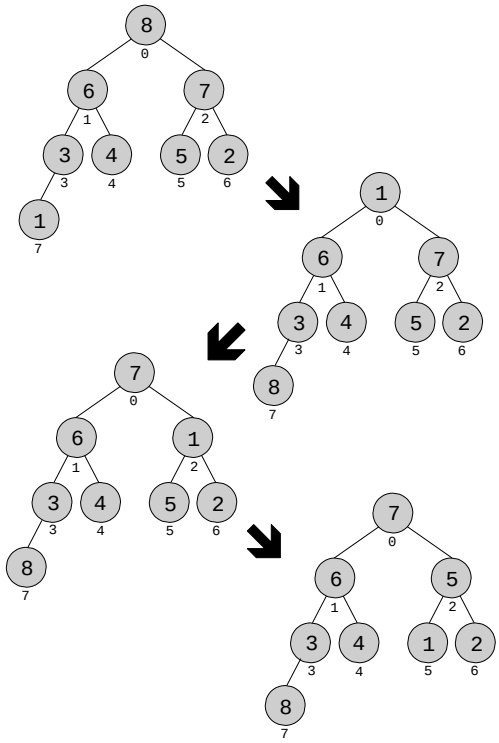
Input: Suite a de N éléments à clés entiers
Output: Transformation de $a[\ell], \ldots, a[r-1]$ telle que $a[i].key \leq a[i+1].key$ pour $\ell \leq i < r-1$

```
1: BottomUpHeapCreate(a) ;
2: for i = N − 1, ..., 1 do
3:    Echanger a[0] et a[i]
4:    Swapped ← TRUE
5:    j ← 0
6:    while Swapped = TRUE and j < i do
7:       Swapped ← FALSE
8:       if 2j + 1 ≥ i (a[j] n'a aucun fils avec indice < i) then
9:          NEXT i
10:      else
11:         Soit a[ℓ] le plus grand fils de a[j] tel que ℓ < i
12:      end if
13:      if a[ℓ].key > a[j].key then
14:         Echanger a[j] et a[ℓ]
15:         j ← ℓ
16:         Swapped ← TRUE
17:      end if
18:   end while
19: end for
```

de parcours total dans le pire des cas est

$$\begin{aligned} O(N) + \sum_{i=1}^{N-1} O(\log(i)) &= O(N) + O(\log(N!)) \\ &= O(N \log(N)). \end{aligned}$$

Exemple:



6.3.4 Priority Queues

Les heaps implémentent des queues à priorité (*Priority Queues*) de manière très efficace. Une queue à priorité est une structure de données abstraite pour stocker des objets avec des clés. Deux opérations sont possibles : Enqueue ajoute un élément et DeleteMin enlève l'élément avec la clé la plus petite.

De manière analogue, il y a une queue "max-priority queue" qui implémente les opérations Enqueue et DeleteMax. Il est clair qu'une de ces structures de données peut être utilisée pour implémenter l'autre avec une modification triviale (Exercice). Dans l'implémentation, une queue à priorité pourrait être une paire consistant en un tableau (*array*) de longueur variable ainsi que la longueur du tableau : de cette façon il est possible d'ajouter des

éléments au tableau, ou d'en enlever. Dans certaines applications un tel tableau peut être de longueur fixe (par exemple pour les codes de Huffman).

Pour une queue à priorité P, on dénote par Array(P) le tableau correspondant à P, par $P[i]$ le i^{eme} élément du tableau, et par Length(P) la longueur du tableau.

Algorithme 45 EnQueue(P, a)

Input: Queue à priorité P, élément a ;
Output: Addition de i à P de telle façon que le nouveau tableau reste une queue à priorité

1: Length(P) $\leftarrow$ Length(P) + 1
2: P[Length(P) − 1] $\leftarrow$ a
3: SiftUp (Array(P), Length(P) − 1)

Algorithme 46 DeleteMax(P)

Input: Queue à priorité P
Output: Effacement de l'élément maximal de P de telle façon que le nouveau tableau reste une queue à priorité ; L'élément maximal est retourné

1: $a \leftarrow P[0]$
2: $P[0] \leftarrow P$[Length(P) − 1]
3: Length(P) $\leftarrow$ Length(P) − 1
4: SiftDown (Array(P), 0)
5: **return** a

7
Algorithmes de graphes

Nous avons étudié des structures de données pour représenter des graphes dans le chapitre 3. Nous verrons dans ce chapitre plusieurs problèmes de graphes fondamentaux, et nous introduirons des solutions algorithmiques pour ces problèmes.

Dans ce chapitre, l'ensemble des sommets sera en général noté V, celui des arêtes E et le graphe G, i.e., $G = (V, E)$.

7.1 PARCOURIR DES GRAPHES

Nous étudions d'abord le problème de *parcours* de graphe. "Parcourir un graphe" veut dire énumérer tous les sommets atteignables depuis un sommet s donné, en considérant d'abord les voisins de s, et itérativement les voisins des voisins, les voisins des voisins des voisins, etc.

Le but est de *visiter* tous les sommets atteignables, i.e. d'effectuer une opération sur ces sommets. Cette opération dépend de l'application concrète en question. Pour donner un exemple simple, si nous voulons compter le nombre de sommets atteignables depuis un sommet s, nous pouvons parcourir le graphe et la visite d'un sommet consisterait alors à incrémenter un compteur.

En général, on veut visiter chaque sommet atteignable exactement une fois, et il faut alors résoudre deux problèmes. Il faut d'une part n'oublier aucun sommet, en s'assurant d'autre part qu'aucun sommet ne soit visité plus d'une fois.

Nous verrons que le premier problème peut être résolu en maintenant un ensemble T énumérant les sommets déjà vus mais pas encore traités. Pour résoudre le deuxième problème, nous marquons les sommets traités de façon à savoir quels sommets sont encore à visiter.

Nous introduirons d'abord une solution abstraite pour le problème du parcours de graphe, avant de spécialiser cette solution afin d'obtenir deux des algorithmes les plus utilisés : *Depth First Search* (*DFS*, profondeur d'abord) et *Breadth First Search* (*BFS*, largeur d'abord). La différence entre les solutions spécialisées concerne l'ordre dans lequel le parcours a lieu.

Dans la méthode abstraite nous maintenons un ensemble T qui contient des sommets qui sont connus atteignables depuis s, mais qui n'ont pas encore été visités. Au début le sommet s est le seul élément de l'ensemble T. L'algorithme prend un sommet v dans T, le marque puis l'enlève de T et ajoute à T tous ses voisins non marqués. Dans cette section, $N[v]$ dénote l'ensemble des voisins d'un sommet v (N comme neighbor, *voisin*).

L'algorithme est décrit en pseudo-code ci-dessous.

Algorithme 47 AbstractTraversal(G, s)

Input: Graphe $G = (V, E)$, sommet $s \in V$.
Output: Visiter et marquer les sommets de V qui peuvent être atteints depuis s.

```
1: T ← {s}.
2: Marquer s.
3: while T ≠ ∅ do
4:    Choisir v ∈ T.
5:    Visiter v.
6:    T ← T \ {v}
7:    while N[v] ≠ ∅ do
8:       Choisir v' ∈ N[v]
9:       N[v] ← N[v] \ {v'}
10:      if v' n'est pas marqué then
11:         T ← T ∪ {v'}
12:         Marquer v'
13:      end if
14:   end while
15: end while
```

Théorème 7.1 *Quand cet algorithme se termine, l'ensemble des sommets visités est exactement l'ensemble des sommets atteignables depuis s.*

Preuve. Nous montrons que l'ensemble de sommets *marqués* est égal à l'ensemble des sommets atteignables depuis s. Le résultat suivra alors parce que chaque sommet marqué est mis dans l'ensemble T juste après qu'il est marqué, et chaque sommet qui est mis dans T est aussi visité, étant donné la condition d'arrêt à la ligne 3.

Supposons par l'absurde que $v \in V$ soit atteignable depuis s, mais pas marqué. Il existe un chemin qui va de s a v, i.e, $s = v_0 \to v_1 \to \cdots \to v_{k-1} \to v_k = v$. Soit ℓ le plus petit indice pour lequel v_ℓ n'est pas marqué. On a alors $0 < \ell \leq k$. Le sommet $v_{\ell-1}$ est marqué, il a donc été choisi à la ligne 4 à un certain moment dans l'algorithme. Mais alors, v_ℓ a dû être choisi à la ligne 8 à un certain moment, puis marqué à la ligne 12, une contradiction.

Inversement, supposons par l'absurde qu'il existe un sommet $v' \in V$ qui est marqué mais qui n'est pas atteignable depuis s. Soit v' le premier sommet à être marqué avec cette

propriété. Donc, $v' \in N[v]$ pour un certain sommet marqué v. Puisque v' était le premier sommet non atteignable à être marqué, et v a été marqué avant v', v est atteignable depuis s. Mais alors v' est atteignable depuis s, une contradiction. ■

Théorème 7.2 *L'algorithme ci-dessus peut être implémenté de façon à ce que son temps de parcours soit $O(|E|)$.*

Preuve. Chaque sommet est inséré dans T au plus une fois, puisque seuls des sommets non marqués sont insérés dans T (ligne 10). Pour chaque sommet inséré dans T, au plus tous ses voisins seront considéreś aux lignes 7–14.

Définissons $d(v)$ comme étant le (out-)degree du sommet v. Le nombre de fois que la boucle de la ligne 7 est executée est au plus

$$\sum_{v \in V} d(v) = \begin{cases} |E| & \text{si } G \text{ est orienté,} \\ 2|E| & \text{si } G \text{ est non orienté.} \end{cases}$$

Si les ensembles $N[v]$ sont réalisés avec une structure de données qui permet d'accéder à v' en temps constant (ligne 8), et d'enlever ce même élément en temps constant (ligne 9), et de même pour l'ensemble T, alors le temps de parcours de l'algorithme est $O(|E|)$. ■

Exercice : Quelles structures de données doivent être utilisées pour stocker $N[v]$ en permettant d'insérer et d'enlever des éléments en temps constant ?

Nous n'avons pas spécifié comment l'ensemble T devait être géré, et c'est exactement dans ce choix que se trouve la différence entre les algorithmes concrets de parcours de graphes.

• Si nous utilisons un *stack*, nous obtenons alors l'algorithme *Depth First Search* (*DFS*).

• Si nous utilisons une *file d'attente (queue)*, nous obtenons alors l'algorithme *Breadth First Search* (*BFS*).

7.2 L'ALGORITHME DEPTH FIRST SEARCH (DFS)

L'algorithme DFS est obtenu en réalisant AbstractTraversal avec T étant un stack. Rappelons que l'on peut effectuer sur un stack T les opérations suivantes :

• `Top`(T) : Retourne l'élément sur le haut du stack.
• `Pop`(T) : Enlève l'élément sur le haut du stack.
• `Push`(T, v) : insère v sur le haut du stack (*"push v onto T"*).
• `StackEmpty`(T) : retourne vrai si T est vide, faux sinon.

Algorithme 48 DFS(G, s)

Input: Graphe $G = (V, E)$, sommet $s \in V$.
Output: Visiter les sommets de V qui peuvent être atteints depuis s.

```
1: Push(T, s)
2: Marquer s
3: while Not StackEmpty(T) do
4:    v ← Top(T)
5:    Visiter v
6:    Pop(T)
7:    while N[v] ≠ ∅ do
8:       Choisir v' ∈ N[v]
9:       N[v] ← N[v] \ {v'}
10:      if v' n'est pas marqué then
11:         Push(T, v')
12:         Marquer v'
13:      end if
14:   end while
15: end while
```

7.3 L'ALGORITHME BREADTH FIRST SEARCH (BFS)

L'algorithme BFS est la variante de AbstractTraversal qui utilise pour T la structure de données queue. Rappelons que l'on peut effectuer sur T les opérations suivantes :

- `Head`(T) retourne le premier élément de la queue.
- `Dequeue`(T) enlève le premier élément de la queue.
- `Enqueue`(T, v) insère v à la fin de la queue.
- `QueueEmpty`(T) retourne vrai si et seulement si T est vide.

Exercices :

- Modifier l'algorithme BFS pour qu'il retourne pour chaque sommet sa distance par rapport à s.
- Modifier les algorithmes BFS et DFS pour qu'ils retournent pour chaque sommet la taille de sa composante connexe.
- Construire une structure de données qui peut maintenir la liste précédente dans le cas d'un graphe dynamique auquel on peut ajouter des sommets (mais duquel on ne peut pas enlever de sommets), en gardant pour chaque sommet un pointeur sur un autre sommet dans la composante connexe. Quel est le temps de parcours de votre algorithme ?

Algorithme 49 BFS(G, s)

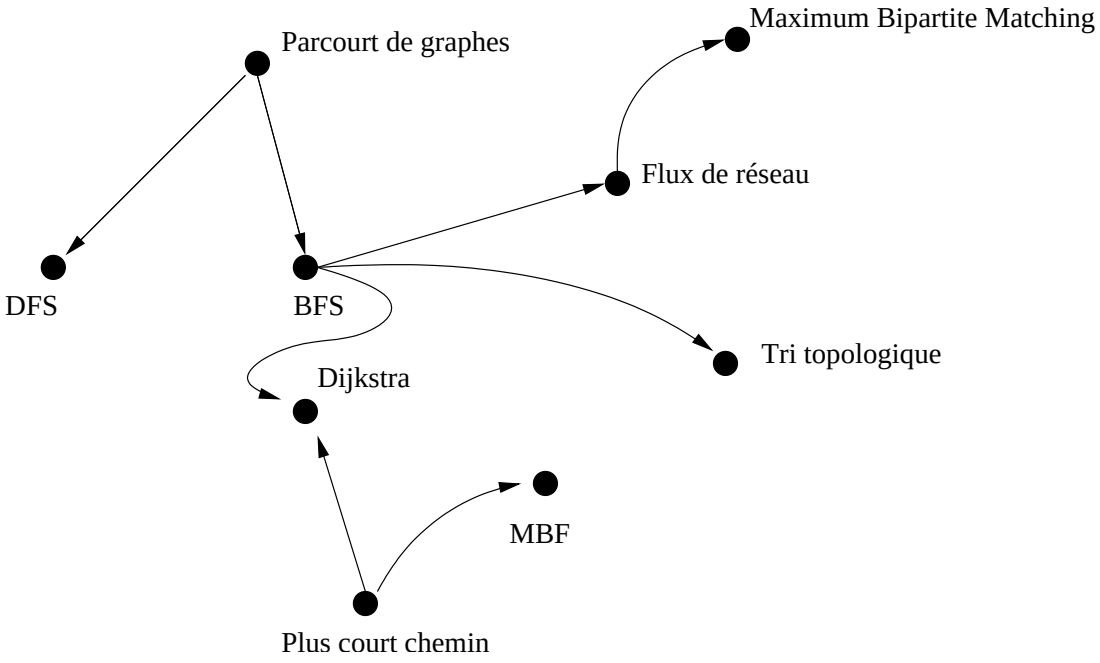
Input: Graphe $G = (V, E)$, sommet $s \in V$.
Output: Visiter les sommets de V qui peuvent être atteints depuis s.

```
1: Enqueue(T, s)
2: Marquer s
3: while Not QueueEmpty(T) do
4:    v ← Head(T)
5:    Visiter v
6:    Dequeue(T)
7:    while N[v] ≠ ∅ do
8:       Choisir v' ∈ N[v]
9:       N[v] ← N[v] \ {v'}
10:      if v' n'est pas marqué then
11:         Enqueue(T, v')
12:         Marquer v'
13:      end if
14:   end while
15: end while
```

7.4 TRI TOPOLOGIQUE (TOPOLOGICAL SORTING)

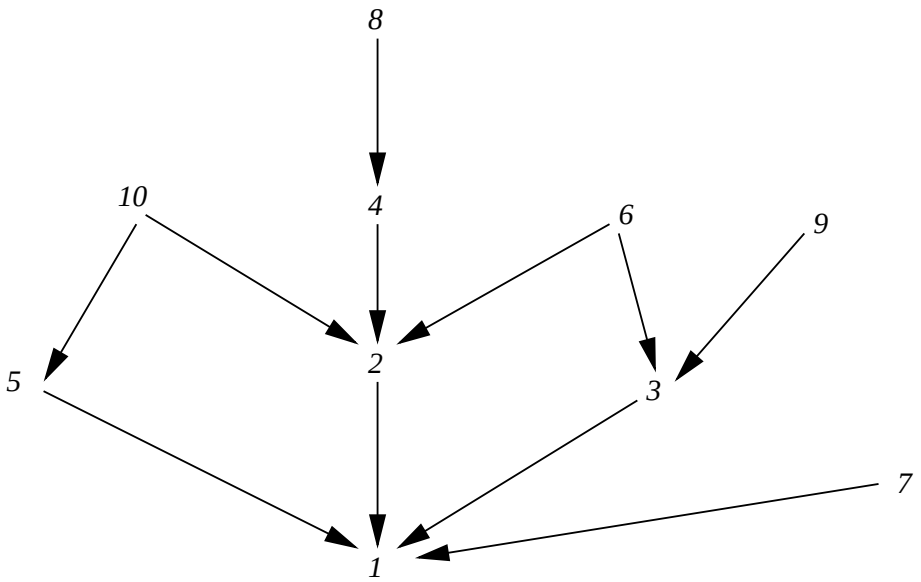
Supposons que nous voulons écrire un chapitre sur les algorithmes de graphes dans un support de cours d'algorithmique. Nous avons tout d'abord une liste de sujets que nous aimerions traiter, comme "l'algorithme BFS", "le tri topologique", etc. Evidemment, nous ne pouvons pas traiter les sujets dans n'importe quel ordre ; par exemple avant d'étudier l'algorithme BFS, il est obligatoire de comprendre ce que c'est qu'un parcourt de graphe ; de manière générale, les sujets ont comme préalable d'autres sujets. Nous pouvons décrire ces dépendances à l'aide d'un graphe : chaque sommet correspond à un sujet, et on met une arête $a \to b$ s'il est nécéssaire (ou souhaitable) de traiter le sujet a avant le sujet b.

Pour l'exemple du livre d'algorithmique, on aura peut-être un graphe comme le suivant :

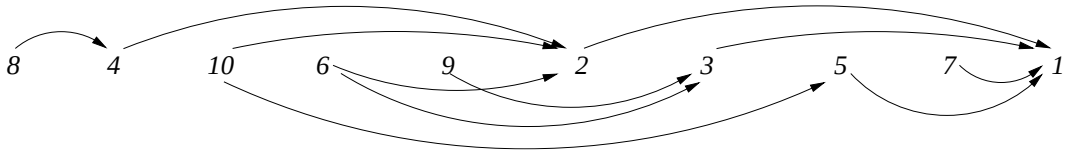


Le problème que l'auteur doit résoudre, est le suivant : dans quel ordre dois-je traiter les sujets afin de ne jamais traiter un sujet avant d'avoir traité les préalables nécessaires. En termes du graphe, la tâche est : trouver un ordre des sommets tel que pour toute paire de sommets $a, b \in V$, si a vient avant b, alors il n'y a pas d'arête $b \to a$. C'est un problème de tri topologique.

Plus formellement, le *tri topologique* d'un graphe orienté acyclique $G = (V, E)$ consiste à donner un ordre aux sommets $v_1, \ldots, v_n$ de G tel que $(v_i, v_j) \in E$ implique que $i < j$. Voici un autre exemple :



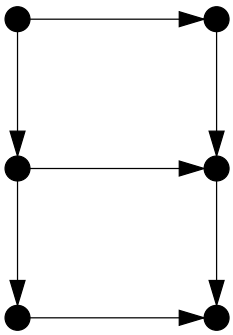
Un tri topologique possible d'un tel graphe est le suivant :



Donc nous prenons les sommets de droite à gauche dans la figure ci-dessus pour obtenir l'ordre voulu. Parmi les différentes applications de cette méthode de tri, signalons les exemples suivants :

- Job scheduling : une famille de tâches (*jobs*) doit être accomplie avec certaines contraintes quant à l'ordre dans lequel ceci doit être fait. On utilise alors un graphe orienté acyclique où les sommets sont les tâches et une arête (u, v) représente la contrainte "la tâche u doit être effectuée avant la tâche v".

- Les parenthèsements valables avec 3 paires de parenthèses sont en bijection avec tous les tris topologiques possibles du graphe suivant :



La bijection s'obtient en associant aux sommets de gauche des parenthèses ouvertes à droite et aux sommets de droite des parenthèses ouvertes à gauche. Ce cas est facile a généraliser.

Comment trouver un ordre topologique d'un graphe ? Pour reprendre l'exemple du cours d'algorithmique ci-dessus, une approche simple est de d'abord traiter tous les sujets sans préalables, ensuite les sujets qui ont juste comme préalables les sujets déjà traités et ainsi de suite.

En effet, cette approche est l'idée de base de l'algorithme TopSort que nous discuterons maintenant plus en détails.

Dans l'algorithme suivant nous donnons des indices aux sommets v. L'indice du sommet v est dénoté par v.label. Nous utiliserons une queue Q qui contiendra l'ensemble de tous les sommets visités jusqu'à présent qui pourraient être utilisés en tant que prochain élément dans le tri topologique. L'algorithme traverse les sommets déjà dans présents dans Q, et les enlève du graphe. Si pendant cette procédure le in-degré d'un sommet est réduit à zero, alors ce sommet est ajouté à Q.

Algorithme 50 TopSort(G)

Input: Graphe acyclique orienté $G = (V, E)$.
Output: Tri topologique de G, donné par les indices des sommets $v.label$.

```
1: Initialiser v.Indegree pour tout v ∈ V (e.g., via DFS).
2: t ← 0
3: for i = 1, ..., n do
4:    if v_i.Indegree = 0 then
5:       Enqueue(Q, v_i)
6:    end if
7: end for
8: while Not QueueEmpty(Q) do
9:    v = Head(Q)
10:   Dequeue(Q)
11:   t ← t + 1
12:   v.label = t
13:   for w ∈ N[v] do
14:      w.Indegree ← w.Indegree − 1
15:      if w.Indegree = 0 then
16:         Enqueue(Q, w)
17:      end if
18:   end for
19: end while
```

Rappelons que $N[v]$ dénote l'ensemble des voisins du sommet v.

Théorème 7.3 *Le temps de parcours de* TopSort *est* $O(|V| + |E|)$*, et il a besoin de* $O(|V|)$ *mémoire en plus.*

Preuve. Exercice.

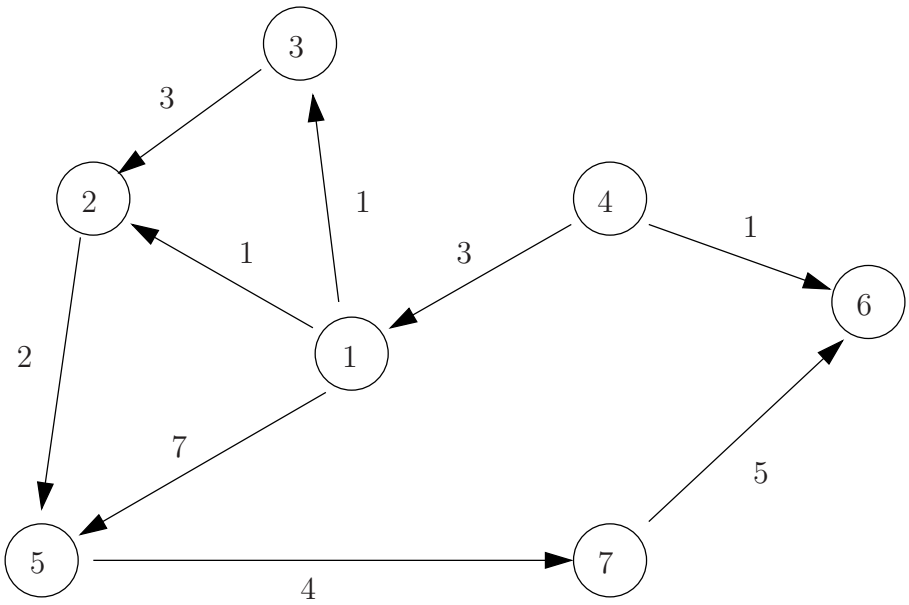
Exercice. En quoi consiste la visite d'un sommet si l'on veut utiliser DFS pour initialiser v.Indegree dans le pas 1 de TopSort ?

7.5 CHEMINS LES PLUS COURTS

Supposons que nous disposons d'une carte du réseau routier du canton de Vaud. Nous nous trouvons à Forel et nous aimerions trouver les plus courts chemins de Forel à tous les autres villages du canton. Nous pouvons modéliser ce problème comme un problème de graphe : chaque village est un sommet et les routes sont des arêtes entre les villages. Chaque arête (route entre villages) a une longueur. Le problème est alors de trouver pour chaque sommet x, le chemin du sommet de départ (Forel dans l'exemple) arrivant à x tel que la somme des distances sur le chemin soit minimale.

Nous décrivons d'abord le problème plus formellement. Soit $G = (V, E)$ un graphe orienté avec une fonction poids $c\colon E \to \mathbb{R}_{>0}$. Donc chaque arête e a un certain poids $c(e)$. Nous voulons trouver les chemins les plus courts d'un sommet $s \in V$ donné à tous les sommets $v \in V$ atteignables depuis s. Il faut préciser ici quelle notion nous attribuons à la définition de "plus court". Le coût d'un chemin est la somme des poids des arêtes sur le chemin et par *chemin le plus court*, on entend *le chemin pour lequel le coût est minimal*.

Exemple : $V = \{v_1, \ldots, v_7\}$; $s = v_1$



$\ell(v)$: Longueur du chemin le plus court du sommet 1 au sommet v, $\ell(v) = \infty$ si v n'est pas connecté à s.

$$\begin{array}{llllll} \ell(v_1) = 0; & \ell(v_2) = 1; & \ell(v_3) = 1; \\ \ell(v_4) = \infty; & \ell(v_5) = 3; & \ell(v_6) = 12; \\ \ell(v_7) = 7; \end{array}$$

L'*algorithme de programmation dynamique* suivant, dû à Dijkstra (1959) calcule les chemins les plus courts de s à tous les $v \in V$ auxquels s est connecté. Il maintient un

ensemble T qui, à la fin du parcours de l'algorithme, contient tous les sommets dans V atteignables depuis s. A chaque étape pour tout $w \in T$ la valeur $\ell(w)$ sera la longueur du chemin le plus court de s à w.

Algorithme 51 DIJKSTRA(G, s)

Input: Graphe orienté $G = (V, E)$ avec fonction de poids $c\colon E \to \mathbb{R}_{>0}$, sommet $s \in V$.
Output: Pour tout $v \in V$ la distance la plus courte $\ell(v)$ entre s et v.

```
1: for v ∈ V \ {s} do
2:    ℓ(v) ← ∞
3: end for
4: ℓ(s) ← 0, T ← ∅, v ← s
5: while v ≠ NULL and ℓ(v) ≠ ∞ do
6:    T ← T ∪ {v}
7:    for w ∈ V \ T do
8:       if (v, w) ∈ E then
9:          d ← ℓ(v) + c(v, w)
10:         if d < ℓ(w) then
11:            ℓ(w) ← d
12:         end if
13:      end if
14:   end for
15:   Choisir v dans V \ T de façon à ce que ℓ(v) soit minimal,
      poser v := NULL si V \ T = ∅.
16: end while
```

Théorème 7.4 *L'algorithme de Dijkstra est juste, i.e., la valeur $\ell(w)$ déterminée est en effet la longueur du plus court chemin de s à w pour tout $w \in V$.*

Preuve. Pour la preuve, notons $\ell_{\text{vrai}}(v)$ la vraie longueur du plus court chemin de s à v. Il s'agit donc de montrer qu'à la fin de l'algorithme, nous avons $\ell(v) = \ell_{\text{vrai}}(v)$ pour tout $v \in V$. Pour simplifier les notations, nous posons $c(v, w) = \infty$ si $(v, w) \notin E$.

Nous montrons d'abord que l'algorithme ne se termine pas avant que T couvre toute la composante connexe de s. Nous montrons ceci par l'absurde. Notons S la composante connexe de s. Si T est un sous-ensemble propre non-vide de S, alors il existe un voisin w de T. Soit $v \in T$ tel que $(v, w) \in E$. On a $\ell(v) < \infty$ (puisqu' aucun élément v avec $\ell(v) = \infty$ n'est jamais ajouté à T) et $c(v, w) < \infty$ et donc $\ell(w) \leq \ell(v) + c(v, w) < \infty$. Donc w peut être choisi à l'étape 15, et il ne s'agit alors pas de l'itération finale. Donc à la fin de l'algorithme, on a $T = S$.

Montrons maintenant que pour tout $v \in T$, $\ell(v)$ est la longueur d'un chemin de s à v ; il s'en suivra que $\ell(v) \geq \ell_{\text{vrai}}(v)$.

Pour montrer cela, on procède par induction sur $|T|$. Il n'y a rien à montrer pour $|T| = 0$. Supposons maintenant qu'un élément w est ajouté à T. La propriété est vraie sur $T \setminus \{w\}$ par hypothèse d'induction et il suffit donc de la montrer pour w. La valeur de $\ell(w)$ vaut $\ell(w) = \ell(v) + c(v, w)$ pour un certain $v \in T$; par hypothèse d'induction, un chemin $s \to \cdots \to v$ de longueur $\ell(v)$ existe. Le chemin $s \to \cdots \to v \to w$ est alors de longueur $\ell(v) + c(v, w) = \ell(w)$, comme voulu.

Pour la suite, nous aurons besoin de l'égalité

$$\ell(w) = \min_{v \in T} \big(\ell(v) + c(v, w)\big),$$

qui est vraie parce que la boucle des lignes 7 - 14 met à jour $\ell(w)$ pour chaque v ajouté à T.

Montrons que pour tout $v \in T$, $\ell(v) \leq \ell_{\text{vrai}}(v)$ (et que alors $\ell(v) = \ell_{\text{vrai}}(v)$). Nous procédons de nouveau par récurrence sur $|T|$. Nous considérons l'étape où $w \in V$ est ajouté à T. Soit $s = w_0 \to w_1 \to \cdots \to w_m = w$ un plus court chemin de s à w.

Il nous faut d'abord voir que alors $w_{m-1} \in T$. (En fait nous montrerons même que $w_0, w_1, \ldots, w_{m-1} \in T$). Soit, par l'absurde, $0 < k < m$ le plus petit indice avec $w_k \notin T$. Alors

$$\begin{aligned}
\ell(w) &\geq \ell_{\text{vrai}}(w) \\
&> \ell_{\text{vrai}}(w_k) \\
&= \ell_{\text{vrai}}(w_{k-1}) + c(w_{k-1}, w_k) \\
&= \ell(w_{k-1}) + c(w_{k-1}, w_k) \qquad \text{(par hypothèse d'induction)} \\
&\geq \min_{v \in T} \big(\ell(v) + c(v, w_k)\big) \\
&= \ell(w_k),
\end{aligned}$$

contradisant le choix de w à l'étape 15 : w_k aurait été un meilleur choix.

Donc $w_{m-1} \in T$, et alors

$$\begin{aligned}
\ell_{\text{vrai}}(w) &= \ell_{\text{vrai}}(w_{m-1}) + c(w_{m-1}, w) \\
&= \ell(w_{m-1}) + c(w_{m-1}, w) \qquad \text{(par hypothèse d'induction)} \\
&\geq \min_{v \in T} \big(\ell(v) + c(v, w)\big) \\
&= \ell(w),
\end{aligned}$$

comme voulu. ■

Voici le résultat concernant la vitesse de l'algorithme :

Théorème 7.5 *L'algorithme de Dijkstra a un temps de parcours $O(n^2)$ et a besoin de $O(n)$ de mémoire, où $n = |V|$.*

Preuve. La boucle while à la ligne 5 est executée au plus n fois. A chaque étape tous les voisins de v en dehors de T sont considérés aux lignes 8–13. Le nombre de tels sommets est au plus $d(v)$, le out-degree de v. Ainsi, les lignes 8–13 sont executées $|E|$ fois au total. Ligne 15 : Sans une queue à priorité, tous les éléments de $V \setminus T$ doivent être inspectés, le temps de parcours de la ligne 15 sur tout l'algorithme est donc $O(n^2)$. Puisque $|E| = O(n^2)$, le premier résultat s'ensuit.

Finalement, nous devons stocker $\ell(v)$ pour tout $v \in V$, il nous faut dont $O(n)$ mémoire en plus (puisque $|V| = n$). Remarquons que nous ne prenons pas en compte le fait que les nombres $\ell(v)$ deviendront plus grands quand n augmente, et nécessiteront donc plus de mémoire. ■

Avec la bonne structure de données (Fibonacci-Heaps), le temps de parcours peut être réduit à $O(n \log(n) + |E|)$. (sans preuve)

Exercice : Modifier l'algorithme de Dijkstra pour qu'il retourne pour chaque sommet son prédécesseur sur un chemin le plus court depuis s. (Au lieu de simplement retourner la longueur du chemin le plus court). *Indice :* Ajouter un membre `pred` (pour mémoriser le prédécesseur sur le plus court chemin) aux sommets.

7.6 L'ALGORITHME DE MOORE-BELLMAN-FORD

Le grand désavantage de l'algorithme de Dijkstra est que les coûts des arêtes doivent être non-négatifs.

Il existe un algorithme qui n'a pas cette restriction. Il s'agit de l'algorithme de Moore-Bellman-Ford dans lequel les coûts peuvent être négatifs. Cependant, la seule exigence est la non-existence de cycles de longueur négative.

Remarquons que cette dernière condition est en fait une restriction essentielle, plutôt qu'une restriction de l'algorithme : s'il y a un cycle de poids négatif, alors le problème n'est pas bien posé ; dans ce cas, un chemin de poids arbitrairement petit peut être construit simplement en tournant au rond dans ce cycle.

Théorème 7.6 *L'algorithme de Moore-Bellman-Ford est correct et calcule son output en utilisant $O(mn)$ opérations, où $m = |E|$ et $n = |V|$.*

Preuve. L'assertion sur le temps de parcours est évidente. Montrons à présent que l'algorithme est correct :

Remarquons que pendant le parcours de l'algorithme, si $\ell(v) \neq \infty$, on a toujours qu'il existe un chemin $s \to \cdots \to v$ de longueur $\ell(v)$ pour n'importe quel $v \in V$. On voit cela par induction sur le nombre de fois que l'étape 9 est executée : il faut alors montrer que la

Algorithme 52 MBF(G, s)

Input: Graphe orienté $G = (V, E)$ avec fonction poids $c\colon E \to \mathbb{R}$ qui ne contient pas de cycles de longueur négative, $s \in V$.
Output: Pour tout $v \in V$ la distance la plus courte $\ell(v)$ entre s et v.

```
1: for v ∈ V \ {s} do
2:     ℓ(v) ← ∞
3: end for
4: ℓ(s) ← 0
5: for i = 1, ..., n − 1 do
6:     for (v, w) ∈ E do
7:         d ← ℓ(v) + c(v, w)
8:         if d < ℓ(w) then
9:             ℓ(w) ← d
10:        end if
11:    end for
12: end for
```

mise à jour

$$\ell(w) \leftarrow d = \ell(v) + c(v, w)$$

n'invalide pas l'affirmation. C'est vrai parce que par hypothèse d'induction, on a un chemin $s \to \cdots \to v$ de longueur $\ell(v)$. Mais alors le chemin $s \to \cdots \to v \to w$ est de longueur $\ell(v) + c(v, w)$, comme désiré.

Montrons maintenant que les longueurs trouvées sont les plus courtes possibles.

Remarquons d'abord qu'il suffit de prouver que l'algorithme trouve la longueur du plus court chemin passant par $n - 1$ arêtes, car il existe toujours un chemin le plus court de s à w qui utilise au plus $n - 1$ arêtes. En effet si un chemin en avait plus que $n - 1$, alors au moins un sommet apparaîtrait au moins deux fois sur le chemin, il devrait donc contenir un cycle. Ce cycle pourrait être enlevé, sans augmenter le poids du chemin (puisqu'il n'y a pas de cycles négatifs). Ce procédé pourrait être continué jusqu'à ce que chaque sommet sur le chemin apparaisse au plus une fois, résultant en un plus court chemin passant par au plus $n - 1$ arêtes.

Étant donné l'affirmation précédente, il suffit de montrer que après la i-ème itération de la boucle des lignes 5-12, $\ell(v)$ est inférieur où égal à la longueur du plus court chemin $s \to \cdots \to v$ passant par au plus i arêtes.

On prouvera cela par induction sur i. Soit $\ell_{\text{vrai},i}(v)$ la longueur du plus court chemin $s \to \cdots \to v$ passant par au plus i arêtes. Soit $\ell_i(v)$ la valeur de $\ell(v)$ déterminée par l'algorithme *à la fin* de la i-ème itération.

Soit $\ell_{i,w}(v)$ la valeur de $\ell(v)$ à la ligne 7 lorsque la variable $w = w$ et $i = i$. Notons que pendant la i-ème itération, on a alors toujours $\ell_{i-1}(v) \geq \ell_{i,w}(v)$, vu que $\ell(v)$ n'augmente jamais.

À la i-ème itération, on a

$$\begin{aligned}
\ell_{\text{vrai},i}(w) &= \min\{\ell_{\text{vrai},i-1}(w), \min_{(v,w)\in E}\{\ell_{\text{vrai},i-1}(v) + c(v,w)\}\} \\
&\geq \min\{\ell_{i-1}(w), \min_{(v,w)\in E}\{\ell_{i-1}(v) + c(v,w)\}\} \qquad \text{(hyp. d'induction terme par terme)} \\
&\geq \min\{\ell_{i-1}(w), \min_{(v,w)\in E}\{\ell_{i,w}(v) + c(v,w)\}\} \qquad \text{(comme } \ell_{i-1}(v) \geq \ell_{i,w}(v)\text{)} \\
&= \ell_i(w),
\end{aligned}$$

prouvant l'affirmation et terminant la preuve. ■

7.7 FLUX DE RÉSEAU

Définition. Un *réseau* (*network*) (G, c, s, t) est un graphe orienté $G = (V, E)$ avec une *fonction de capacité* $c\colon E \to \mathbb{R}_{>0}$ et deux sommets distingués $s \in V$ et $t \in V$ appelés respectivement *source* et *sink*. Nous supposons que tout autre sommet dans V est sur un chemin de s vers t.

Définition. Un *flux* (*flow*) est une application $f\colon E \to \mathbb{R}_{\geq 0}$ telle que :

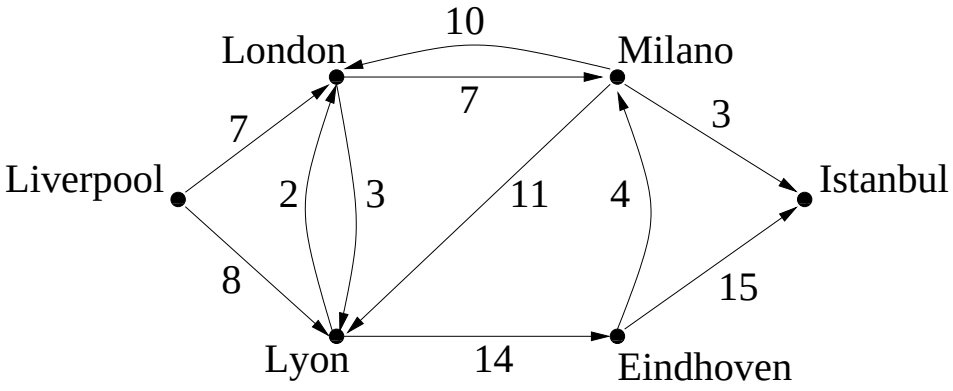
1. Pour tout $e \in E$ on a $f(e) \leq c(e)$ (contrainte de capacité)
2. Pour tout $v \in V \setminus \{s, t\}$:

$$\sum_{e\in E(v,V\setminus\{v\})} f(e) = \sum_{e\in E(V\setminus\{v\},v)} f(e),$$

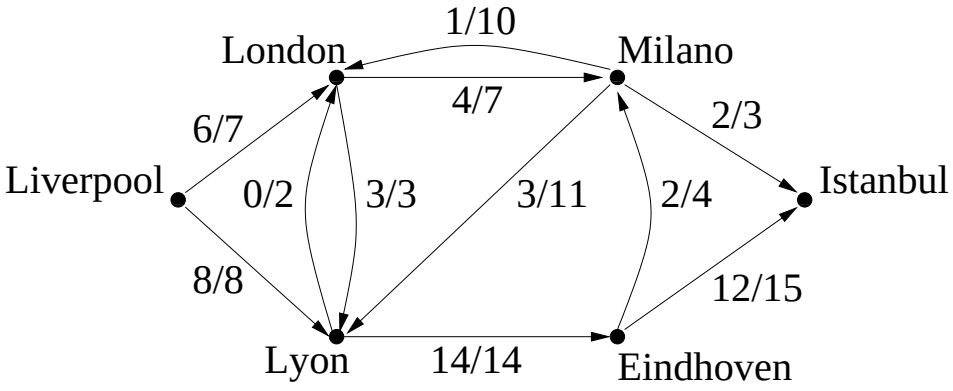
i.e., tout ce qui rentre dans le sommet v doit aussi en sortir (sauf pour s et t).

On peut par exemple avoir un réseau de tuyaux (les arêtes). On veut faire couler de l'eau de s vers t. La contrainte de capacité (condition 1 dans la définition de flow) exprime alors le fait que chaque tuyau a un débit (capacité) maximal (en litres par secondes), et la condition 2 dit que à chaque sommet (p.e. bifurcation de tuyaux), il entre autant d'eau par unité de temps qui sort.

Dans le réseau suivant, les sommets sont des villes et les capacités des capacités de transport entre les villes. Ici, on a $s =$ Liverpool et $t =$ Istanbul.



Voici un flux possible sur ce réseau :



Définition. La *valeur* d'un flux f est

$$|f| = \sum_{e \in E(s, V \setminus \{s\})} f(e) - \sum_{e \in E(V \setminus \{s\}, s)} f(e).$$

Donc $|f|$ donne la *quantité nette* transportée de s à t.

Dans l'exemple ci-dessus on a $|f| = 14$.

Définition. Etant donné un réseau, un *flux maximal* est un flux qui a la plus grande valeur parmi tous les flux possibles.

Remarquons qu'un flux maximal n'est pas forcément unique, c'est-à-dire qu'il peut y avoir plusieurs flux qui ont la valeur maximale. Nous nous intéressons au problème suivant :

Problème. Etant donné un réseau, trouver un flux maximal.

La notion de cut nous fournira un critère de maximalité d'un flux.

Définition. Un *cut* (S,T) avec $S,T \in V$ est une partition de V : $S \cap T = \emptyset$, $S \cup T = V$, avec $s \in S, t \in T$. L'ensemble des arêtes allant de S à T est

$$E(S,T) := (S \times T) \cap E.$$

Pour $x \in S$ on pose $E(x,T) := E(\{x\},T)$ (de même pour S et $y \in T$).

Dans l'exemple ci-dessus, prenons pour S les villes en Angleterre et pour T les autres villes, i.e., $S = \{\text{London}, \text{Liverpool}\}$ et $T = V \setminus S$. La masse partant de Liverpool et arrivant à Istanbul doit bien passer la frontière anglaise à un moment donné. Le lemme suivant exprime cette observation.

Lemme 2 *Soit (G,c,s,t) un réseau, (S,T) un cut et f un flux. Alors*

$$|f| = \sum_{e \in E(S,T)} f(e) - \sum_{e \in E(T,S)} f(e).$$

Preuve.

$$\begin{aligned}
|f| &= \sum_{e \in E(s,V\setminus\{s\})} f(e) - \sum_{e \in E(V\setminus\{s\},s)} f(e) \\
&= \sum_{v \in S} \left(\sum_{e \in E(v,V\setminus\{v\})} f(e) - \sum_{e \in E(V\setminus\{v\},v)} f(e) \right) \\
&= \sum_{v \in S} \left(\sum_{\substack{x \in S \\ (v,x) \in E}} f((v,x)) + \sum_{\substack{x \in T \\ (v,x) \in E}} f((v,x)) \right. \\
&\qquad \left. - \sum_{\substack{x \in S \\ (x,v) \in E}} f((x,v)) - \sum_{\substack{x \in T \\ (x,v) \in E}} f((x,v)) \right) \\
&= \sum_{\substack{v,x \in S \\ (v,x) \in E}} f((v,x)) + \sum_{\substack{v \in S, x \in T \\ (v,x) \in E}} f((v,x)) \\
&\qquad - \sum_{\substack{v,x \in S \\ (x,v) \in E}} f((x,v)) - \sum_{\substack{v \in S, x \in T \\ (x,v) \in E}} f((x,v)) \\
&= \sum_{\substack{v \in S, x \in T \\ (v,x) \in E}} f((v,x)) - \sum_{\substack{v \in S, x \in T \\ (x,v) \in E}} f((x,v)) \\
&= \sum_{e \in E(S,T)} f(e) - \sum_{e \in E(T,S)} f(e)
\end{aligned}$$

■

Aussi, la capacité transporté de Liverpool à Istanbul ne peut pas excéder la capacité de la frontière anglaise. Plus formellement,

Lemme 3 *Soit (G, c, s, t) un réseau, (S, T) un cut et f un flux. Alors*

$$|f| \leq \sum_{e \in E(S,T)} c(e).$$

Preuve. Suit du lemme précédent, puisque

$$\forall e \in E\colon \quad 0 \leq f(e) \leq c(e).$$

■

Remarquons qu'on peut choisir un cut et un flux quelconque dans le lemme précédent. En particulier, le lemme implique que le flux maximal ne peut pas être plus grand que le cut minimal.

Nous allons voir qu'en fait, le flux maximal est *égal* au cut minimal. Pour prouver cette assertion nous avons besoin de nouvelles notations :

Pour une arête $e = (u, v) \in E$ donnée, l'*arête inverse* de e est l'arête $\overleftarrow{e} := (v, u)$.

A un flux f on peut associer le *graphe résiduel* (G_f, c', s, t) de f. Les arêtes du graphe résiduel sont

$$E' := \{e \mid e \in E \wedge f(e) < c(e)\} \cup \{\overleftarrow{e} \mid e \in E \wedge f(e) > 0\}.$$

En français, on met une arête entre deux sommets pour chaque arête pour laquelle la capacité n'est pas épuisée par le flux, et on met une arête en sens inverse pour chaque arête de flux non-nul. (Techniquement, le graphe résiduel est un multigraphe, i.e. il peut y avoir plusieurs arêtes entre deux sommets. Nous ignorons ce problème ici.)

$$c_{f(e')} := \begin{cases} c(e') - f(e') & \text{si } e' \in E, \\ f(e) & \text{si } e' = \overleftarrow{e} \text{ pour un } e \in E. \end{cases}$$

On a $c_f(e) > 0$ pour tout $e \in E'$.

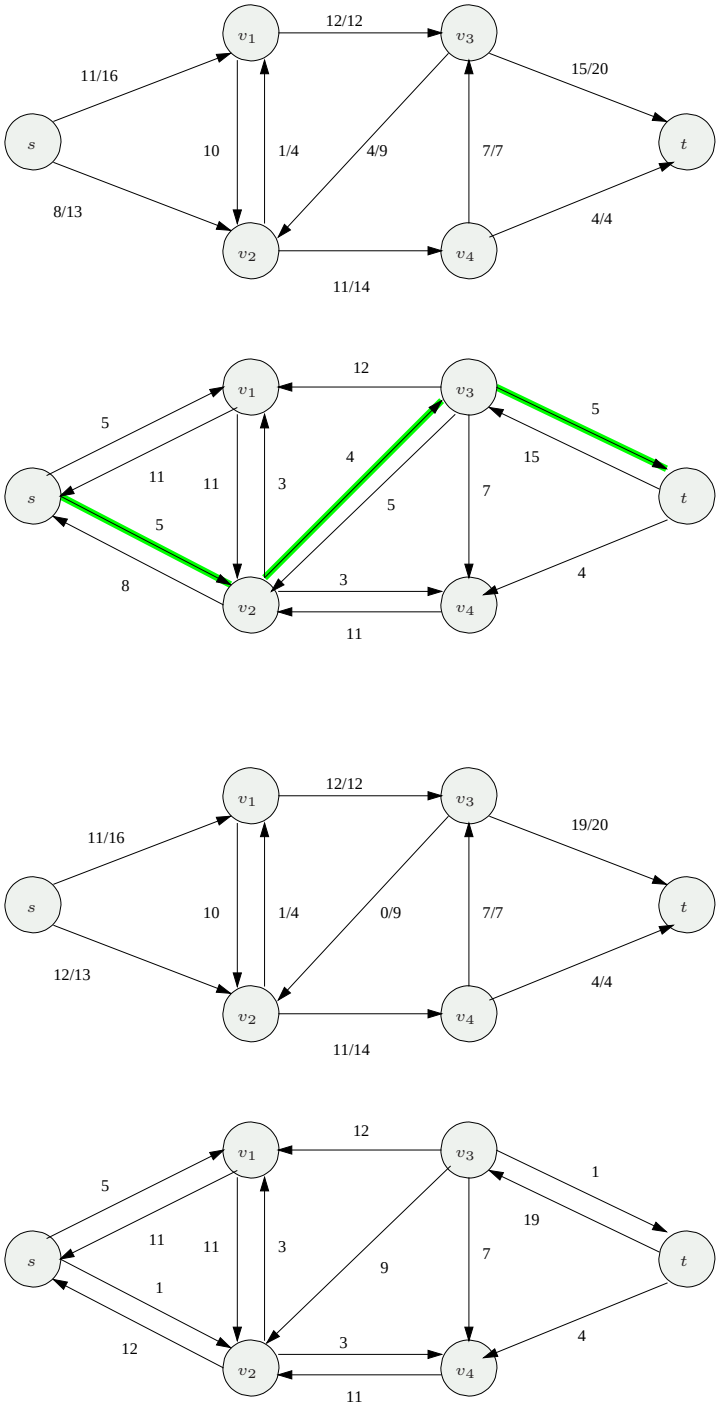
Le graphe résiduel G_f représente donc les différentes façons dont on peut modifier le flux $f(e)$ sur chaque arête. Si $f(e) < c(e)$ alors on peut augmenter $f(e)$ puisqu'il n'a pas atteint la capacité de l'arête. Si $f(e) > 0$ alors on peut diminuer le flux sur l'arête e.

Un *chemin f-augmentant* P du réseau (G, c, s, t) par rapport au flux f est un s-t-chemin dans le graphe résiduel G_f. f est augmenté par $\gamma \in \mathbb{R}_{>0}$ via P, si pour toute arête e' de P

on a la propriété suivante : si $e' \in E$, alors $f(e)$ est augmenté de γ, si $e' = \overleftarrow{e}$, alors $f(e)$ est diminué de γ.

Le graphe résiduel pour un certain flux représente les modifications possibles pour ce flux.

Exemples :



La valeur du flux a été augmentée de 4. Augmenter de $\gamma \in \mathbb{R}_{>0}$ augmente la valeur du flux de γ (en général). Restriction : γ ne doit pas être plus grand qu'une capacité résiduelle.

Théorème 7.7 *Soit f un flux dans un réseau (G, c, s, t). Les assertions suivantes sont équivalentes :*

(a) f est un flux maximal,

(b) G_f ne contient pas de chemins augmentants,

(c) Il existe un cut (S,T) avec

$$|f| = \sum_{e \in E(S,T)} c(e).$$

Preuve. $(a) \Rightarrow (b)$: Supposons que G_f contient un chemin augmentant P. Soit $\gamma := \min_{e' \in E'} c_f(e')$. Par définition de c_f on a $\gamma > 0$. Donc, la valeur de f serait diminuée en augmentant f via P de γ, une contradiction.

$(b) \Rightarrow (c)$: G_f ne contient pas de chemins augmentants, donc t n'est pas atteignable depuis s dans G_f. Soit S l'ensemble des sommets atteignables dans G_f depuis s, donc $s \in S$, $t \notin S$, donc (S,T), $T = V \setminus S$, est un cut. Par définition de c_f :

$$\begin{array}{lll} \forall e \in E(S,T) & f(e) & = \; c(e) \\ \forall e \in E(T,S) & f(e) & = \; 0 \end{array}$$

En effet, si on n'avait pas $f(e) = c(e)$ pour un certain $e \in (S,T)$, alors il y aurait une arête dans G_f entre S et T, donc un sommet de T serait atteignable depuis s, une contradiction. De même pour l'autre égalité. Par le Lemme 7.1 : $|f| = \sum_{e \in E(S,T)} c(e)$.

$(c) \Rightarrow (a)$: Suit du Lemme 7.2. ■

Ce résultat implique le célèbre Théorème de Ford-Fulkerson qui suit :

Théorème 7.8 (Ford-Fulkerson) *La valeur du flux maximal dans un réseau est égale à la valeur du cut minimal du réseau.*

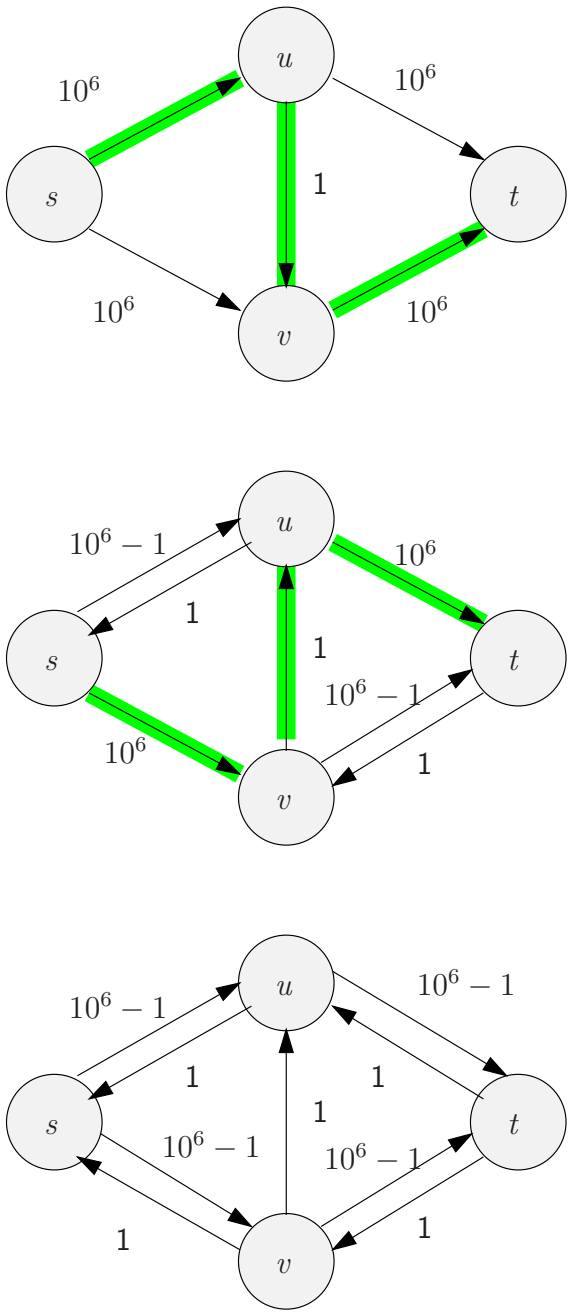
L'algorithme suivant est dérivé directemement de la preuve du théorème de Ford-Fulkerson :

Algorithme 53 MaxFlowMinCut(G, c, s, t)

Input: Réseau (G, c, s, t)
Output: Flow maximal dans le réseau de s à t.

1: **for** $e \in E$ **do**
2: $\quad f(e) = 0$
3: **end for**
4: Construire un chemin f-augmentant P ; STOP si un tel chemin n'existe pas.
5: Poser $\gamma := \min_e c_f(e)$, où e parcourt toutes les arêtes de P.
6: Augmenter f via P de γ.
7: GOTO Etape 4.

Le temps de parcours de cet algorithme est problématique. Si les capacités sont des valeurs entières, alors il est possible d'obtenir un algorithme avec un temps de parcours exponentiel. Voir l'exemple ci-dessous.

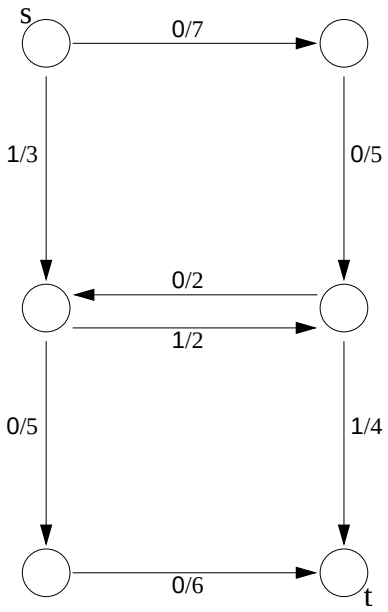


Quand des capacités irrationnelles sont utilisées, il n'est même pas certain que l'algorithme termine. Le problème est que nous n'avons rien dit sur comment ces chemins peuvent être construits. Si nous utilisons un chemin augmentant le plus court à chaque

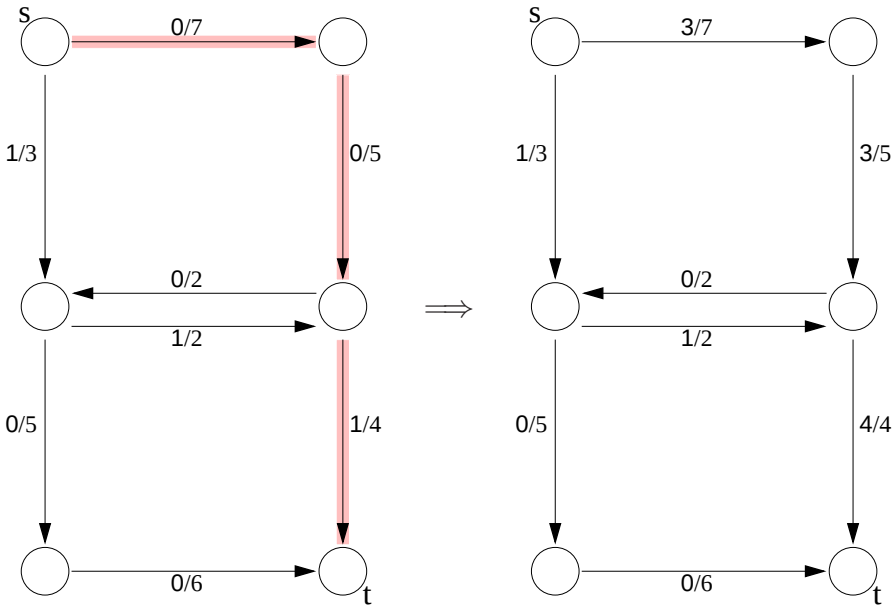
étape (par exemple, en utilisant l'algorithme BFS), alors on peut montrer que l'algorithme termine après au plus $mn/2$ augmentations. Puisque chaque recherche BFS nécessite un temps de parcours de l'ordre de $O(m)$, on obtient un temps de parcours $O(m^2 \cdot n)$ pour l'algorithme.

Nous ne prouvons pas ce résultat ici.

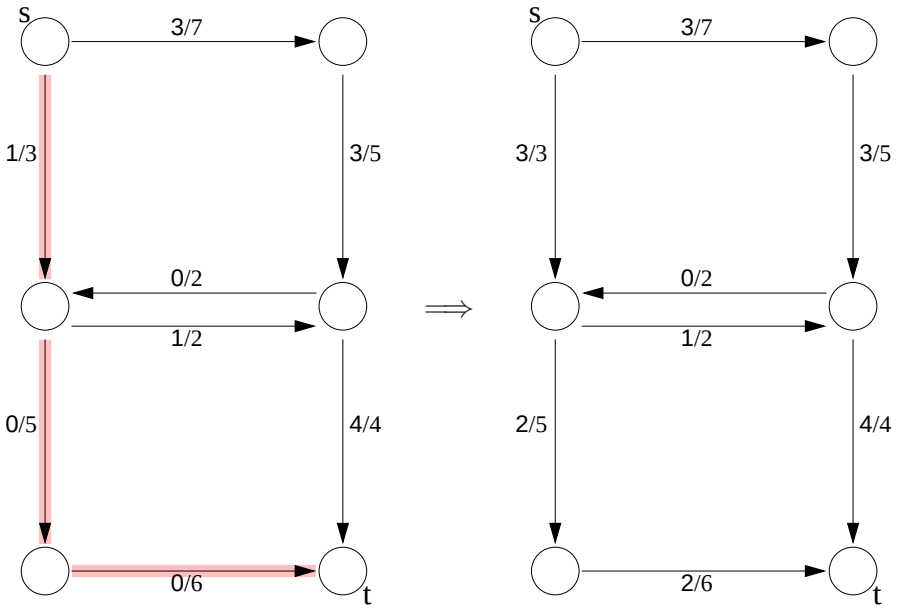
Exemple : Maximiser le flux de s à t ci-dessous :



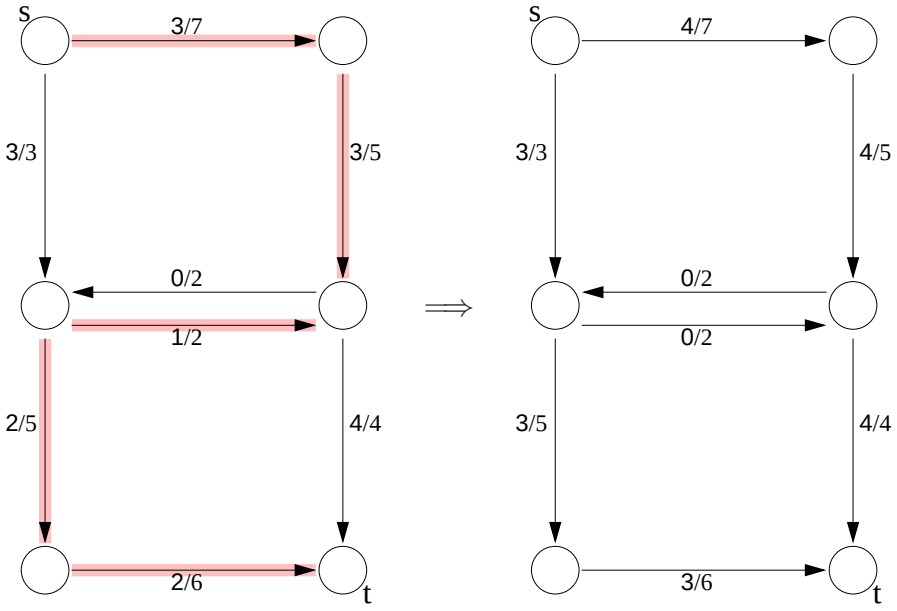
On trouve un chemin augmentant et l'amélioration correspondante :



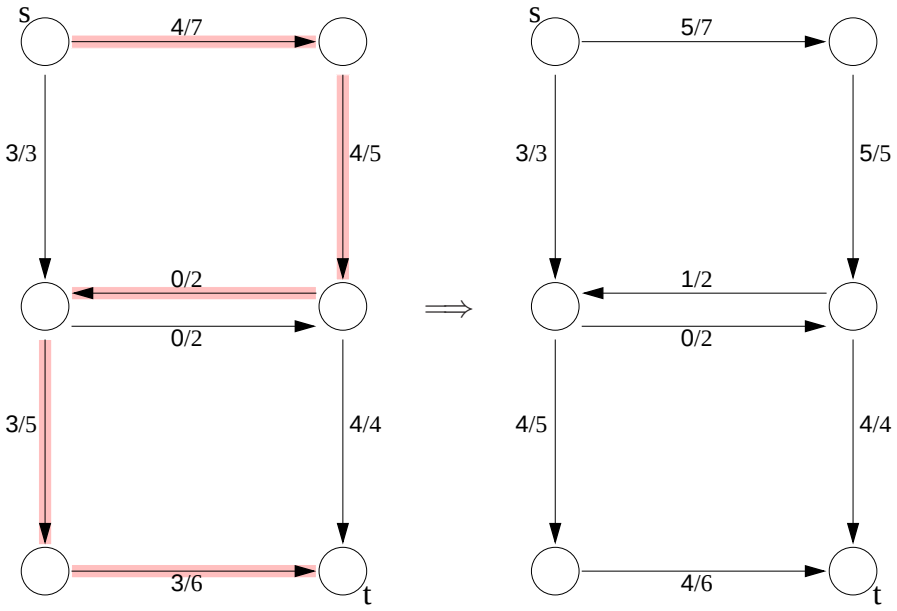
On répète la procédure :



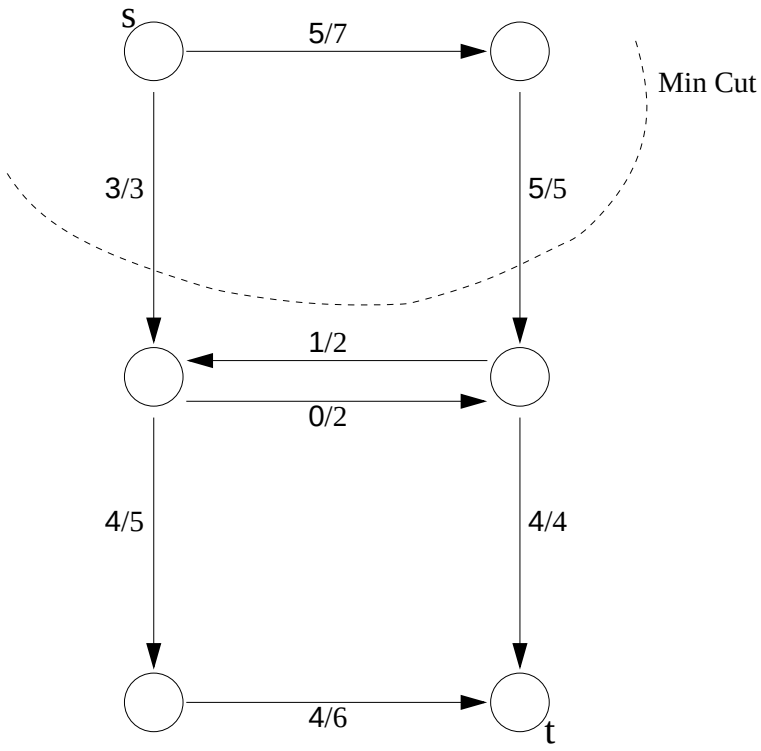
Encore une fois :



Encore :



Finalement, on obtient le flux optimal, comme le montre le cut :



7.8 APPLICATION : MAXIMUM BIPARTITE MATCHING

Considérons le problème suivant : étant donné un ensemble de m élèves, et un ensemble de n chambres, et pour chaque élève la liste des chambres qui l'intéressent, donner à chaque élève une chambre de telle façon qu'autant d'élèves que possible soient satisfaits. (Une chambre peut contenir exactement un élève).

C'est le problème de "maximum bipartite matching".

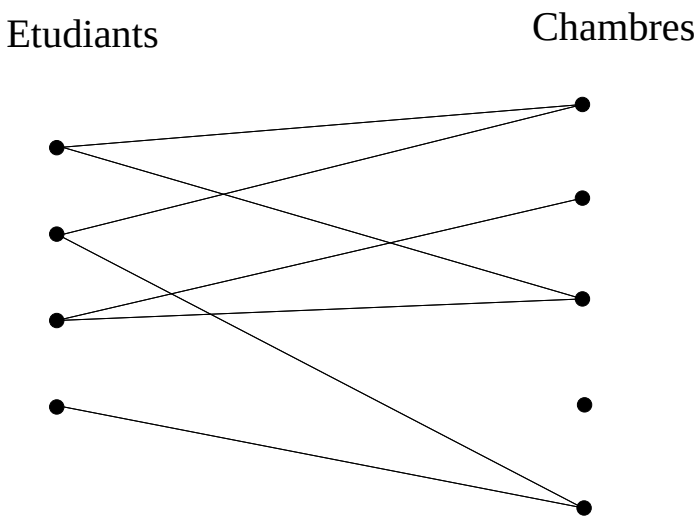
En général : étant donné un graphe biparti avec m sommets sur la gauche et n sommets sur la droite, un matching de taille ℓ dans le graphe consiste en

- ℓ sommets de gauche,
- ℓ sommets de droite,
- un sous-ensemble de ℓ arêtes qui connecte chacun des ℓ sommets de gauche à un des sommets de droite et vice versa.

Un *maximum bipartite matching* est un matching pour lequel ℓ est aussi grand que possible. Nous exposons une solution a ce problème basé sur l'algorithme de flux de réseau :

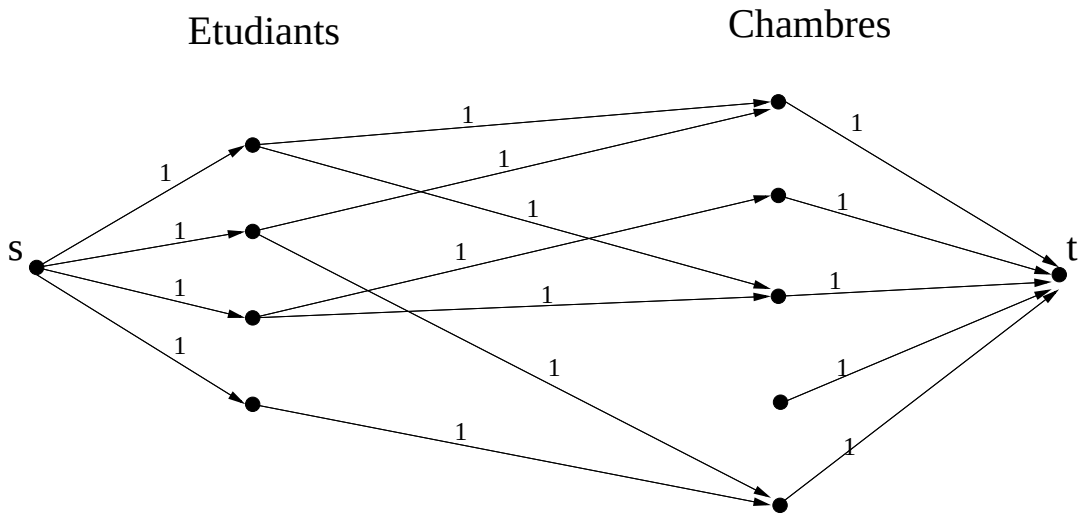
Soient L et R les ensembles de sommets sur la gauche et sur la droite. Ajouter des sommets s et t, pour lesquels s est connecté a tous les sommets dans L, et tous les sommets dans R sont connectés à t. On oriente tous les sommets de la gauche vers la droite, et on leur donne une capacité de 1. On peut alors appliquer l'algorithme MaxFlowMinCut du paragraphe précédent pour trouver un flux maximal auquel correspondra alors un matching maximal.

Exemple : Supposons que nous avons 4 étudiants et 5 chambres. Nous commençons avec un graphe biparti qui indique quelles sont les chambres par lesquelles est intéressé chaque élève :

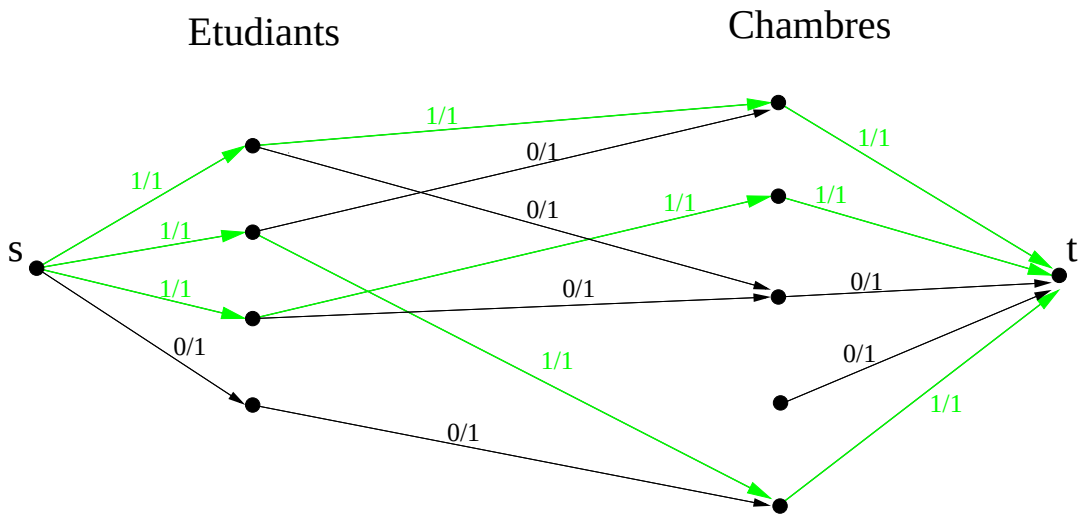


Nous ajoutons les sommets s et t, orientons le graphe et ajoutons les capacités, comme

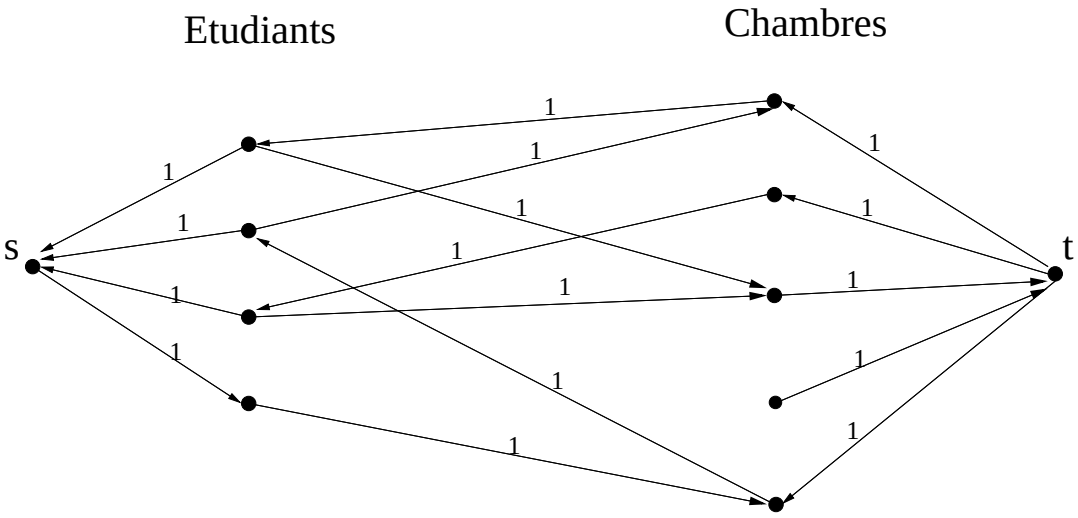
décrit ci-dessus. On obtient ainsi un réseau :



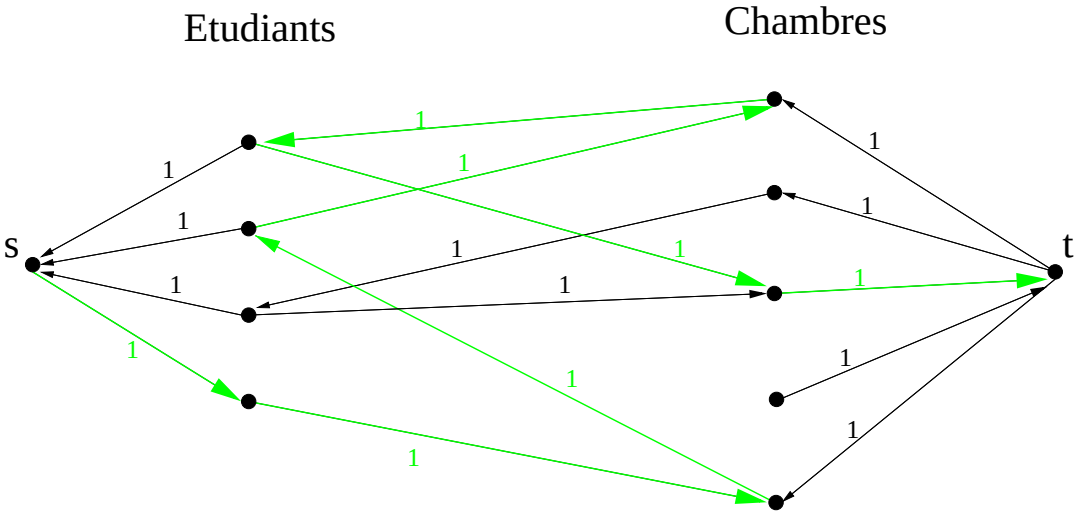
Nous construisons ensuite un flux f :



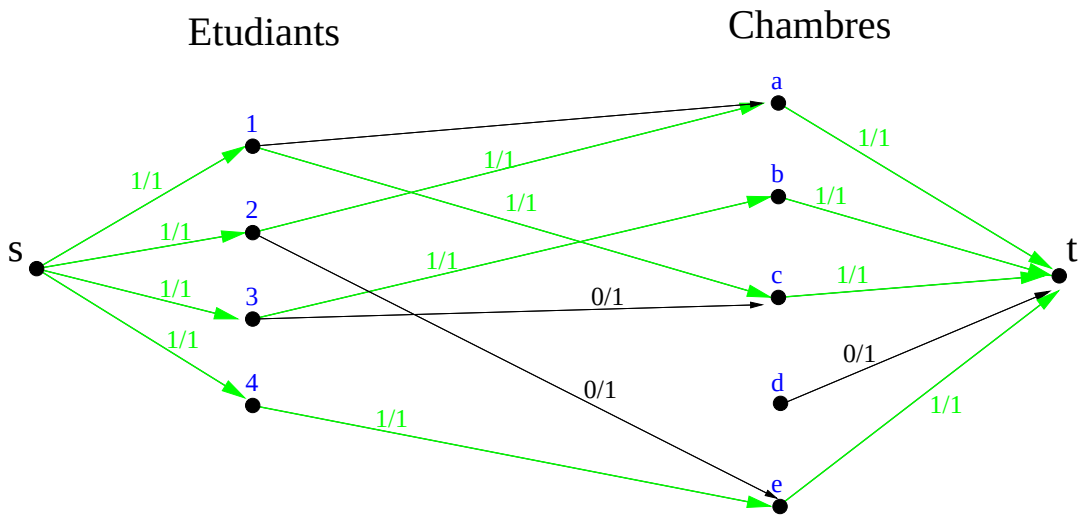
Nous construisons le graphe résiduel par rapport à f :



Nous trouvons un chemin f-augmentant :



Et obtenons ainsi un nouveau flux :



Ce flux est maximal : Le cut $S = \{s\}$, $T = V \setminus S$ le montre.

Le matching correspondant est le suivant :

élève	chambre
1	c
2	a
3	b
4	e

7.9 ARBRES COUVRANTS MINIMAUX

Supposons que nous aimerions connecter des sites A, B et C par cable de telle façon que tout le monde puisse communiquer avec tout le monde. Le coût des liens est comme suit :

lien	coût
$A \leftrightarrow B$	35
$B \leftrightarrow C$	127
$C \leftrightarrow A$	50

Une solution évidente est de relier chaque site à chaque site, ce qui revient dans notre cas à un coût de $35 + 50 + 127 = 212$.

On peut faire mieux. En ne liant que A et B, et A et C, on ne paye que $35 + 50 = 85$, et le problème est aussi résolu : si B veut parler à C, il suffit de passer le message à A qui le passera à C.

Le cas général du problème ci-dessus est un problème de graphe, il s'agit de trouver un graphe connexe qui lie tous les sommets, tout en minimisant les coûts. Voici un énoncé formel du problème :

Etant donné un graphe $G = (V, E)$ avec fonction de poids $w\colon E \to \mathbb{R}_{>0}$, trouver un arbre $T = (V, E')$ avec $E' \subseteq E$ et tel que

$$\sum_{e \in E'} w(e)$$

est minimal. Un tel arbre est appelé un *arbre couvrant minimal* (*minimum spanning tree, MST*). (Pourquoi cherche-t-on un arbre, et non pas un graphe quelconque connexe minimisant le coût ? C'est simplement parce que le meilleur graphe est toujours un arbre.)

Nous voulons trouver un arbre couvrant minimal pour un graphe G donné. Les arbres couvrants minimaux sont très importants en pratique, par exemple pour résoudre le problème de connecter un ensemble de sites en utilisant aussi peu de cable que possible, comme donné dans l'exemple ci-dessus.

Nous étudierons deux solutions à ce probleme : l'algorithme de Kruskal et l'algorithme de Prim.

7.9.1 L'algorithme de Kruskal

L'algorithme glouton suivant est dû à Kruskal (1956). Soit E l'ensemble des arêtes de G.

Algorithme 54 MinSpanKruskal(G, w)

Input: Graphe $G = (V, E)$ avec fonction de poids $w\colon E \to \mathbb{R}_{>0}$
Output: Ensemble E' d'arêtes qui forment un arbre couvrant minimal de G.

```
1: Trier E = {e_1, ..., e_m} de telle façon que w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_m).
2: E' ← ∅
3: for i = 1, ..., m do
4:    if le graphe (V, E' ∪ {e_i}) est acyclique then
5:       E' ← E' ∪ {e_i}
6:    end if
7: end for
```

Encore une fois, et d'ailleurs à chaque fois qu'un nouvel algorithme apparaît, les questions essentielles auxquelles il faut répondre sont : est-il juste ? Comment l'implémenter de manière efficace ? Quel est le temps de parcours ?

Lemme 4 *Soit $G = (V, E)$ un graphe avec une de fonction de poids et $E' \subseteq E$ un ensemble d'arêtes. S'il existe un arbre $T = (V, E_T)$ tel que $E' \subseteq E_T$, et $e \in E$ est la plus petite arête*

(c'est-à-dire l'arête de poids minimal) dans $E \setminus E'$ qui ne crée pas de cycle lorsqu'elle est ajoutée à E', alors il existe un arbre couvrant minimal $T' = (V, E_{T'})$ tel que $E' \cup \{e\} \subseteq E_{T'}$.

Notons que ce lemme prouve que l'algorithme de Kruskal est juste : l'algorithme utilise ce lemme en commençant avec $E' = \emptyset$.

Preuve. Si $e \in E_T$ alors on peut prendre $T' = T$ et la proposition est vérifiée. Supposons maintenant que $e \notin E_T$. Si on ajoute e à E_T on crée un cycle (puisqu'un arbre sur n sommets a toujours $n-1$ arêtes, donc en ajoutant une arète à T il ne sera plus un arbre et aura donc un cycle). Ce cycle a une arête e_1 qui appartient à $E \setminus E'$ (puisque le fait d'ajouter e à E' ne crée pas de cycle par supposition). Ajouter e_1 à E' n'aurait pas créé de cycle, et donc puisque e est la plus petite arête à avoir cette propriété, on a $w(e_1) \geq w(e)$. Si on remplace e_1 par e dans T on crée un autre arbre couvrant qui contient e.

Puisque $w(e_1) \geq w(e)$, le coût de cet arbre couvrant est au plus le coût de T, il est donc aussi minimal. ■

Quelle est l'implémentation la plus simple de l'algorithme de Kruskal ?

- Trier les m arêtes en temps $O(m \log(m))$ (nous avons vu comment faire dans le chapitre précédent),
- pour chaque arête en ordre, tester si elle crée un cycle dans la forêt que nous avons construite jusqu'à présent. Si oui on l'efface, sinon on l'ajoute a notre forêt. Avec un BFS/DFS, on peut faire le test en temps $O(n)$ (puisque l'arbre a au plus n sommets et au plus $n-1$ arêtes).

Le temps total est $O(mn)$.

Peut-on faire mieux ? *OUI*, en utilisant la *Structure de données Union Find*.

Que fait l'algorithme de Kruskal ?

Il commence avec une *forêt*[1] dans laquelle chaque élément est un sommet. Il connecte deux composantes de la forêt en utilisant une arête minimale, si les deux sommets de cette arête ne sont pas dans la même composante (i.e. aucun cycle n'est crée). Donc, si nous pouvons savoir rapidement quels sommets sont dans quelles composantes, ne n'avons pas besoin de tester la non-existence de cycles !

Si nous pouvons faire le test à la ligne 5 en temps $O(\log(n))$ et l'union à la ligne 7 en $O(1)$, alors un arbre couvrant minimal peut être trouvé en temps $O(m \log(n))$! (Il faut un temps $O(m)$ pour créer un heap, et $O(m \log(n))$ pour toutes les opérations à l'intérieur de la boucle).

Pour faire les tests et les unions en temps $O(\log(n))$, nous maintenons les sommets dans une *structure de données Union Find*.

[1] Une forêt est une collection d'arbres.

Algorithme 55 AbstractMinSpanKruskal(G, w)

```
1: Stocker les arêtes dans un heap
2: T ← ∅
3: while |T| < n − 1 do
4:    (v, w) ← l'arête minimale du heap
5:    if composante(v) ≠ composante(w) then
6:       ajouter (v, w) à T
7:       réunir les composantes de v et de w en une seule composante.
8:    end if
9: end while
```

7.9.2 La structure "Union Find"

Nous avons besoin d'une méthode rapide pour tester si une arête relie deux sommets dans une composante déjà créée. Pour cela, nous utilisons une structure de données qui permet de gérer une famille d'ensembles, i.e., de vérifier si deux éléments sont dans le même ensemble, et de fusionner deux ensembles en un autre ensemble.

On va munir chaque composante d'un (unique) représentant. On peut alors déterminer si deux sommets se trouvent dans la même composante en regardant s'ils ont le même représentant.

On stockera les éléments d'une composante dans un arbre, dont la racine est un représentant. L'opération Find cherche pour un sommet donné la racine de "son" arbre. L'opération Union forme l'union de deux arbres, en ajoutant l'arbre avec le plus petit nombre de sommets en tant que descendant de la racine de l'arbre avec le plus grand nombre de sommets. (Choix arbitraire, si les tailles sont les mêmes.)

Coût pour Find : hauteur de l'arbre.
Coût pour Union : hauteur de l'arbre.

Théorème 7.9 *Soit T l'arbre obtenu en formant des unions élément par élément dans un ordre arbitraire. Alors la hauteur de T est au plus égale à* $\log_2(|T|)$.

Preuve. Nous procédons par induction sur $|T|$ (le nombre de sommets dans T).

Base : Si $|T| = 1$ alors sa hauteur vaut 0, et on a bien $\log_2(|T|) = \log_2(1) = 0$.

Pas : Supposons que T est l'union de deux arbres T_1 et T_2 de hauteurs h_1 et h_2. Remarquons qu'on a $|T| = |T_1| + |T_2|$, donc (en supposant que T_1 et T_2 ont au moins un sommet chacun) $|T_1|, |T_2| < |T|$. Nous pouvons donc appliquer l'hypothèse d'induction pour obtenir $h_1 \leq \log_2(|T_1|)$ et $h_2 \leq \log_2(|T_2|)$.

Supposons sans perte de généralité que $|T_1| \leq |T_2|$, donc la racine de T_1 sera dans T un descendant de la racine de T_2 (par la construction de T). Notons h la hauteur de T, alors

$h = \max\{h_1 + 1, h_2\}$. Nous distinguons deux cas.

Cas 1. $h_1 < h_2$: alors $h = h_2 \leq \log_2(|T_2|) < \log_2(|T|)$, par hypothèse d'induction, et puisque $|T_2| < |T|$.

Cas 2. $h_2 \leq h_1$: $h = h_1 + 1 \leq \log_2(|T_1|) + \log_2(2) = \log_2(2 \cdot |T_1|) \leq \log_2(|T_1| + |T_2|) = \log_2(|T|)$.

Donc on a dans tous les cas $h \leq \log_2(|T|)$. ■

L'opération FIND peut être implémentée pour des composantes de graphes comme suit. Nous maintenons pour chaque sommet v du graphe les champs suivants :

• $v.pred$: pointeur au parent ou `NULL` si c'est une racine.

• $v.size$: la taille de la composante correspondante à v, si c'est le représentant. (S'il ne s'agit pas du représentant, ce champ ne sera pas utilisé.)

Algorithme 56 FIND(u)

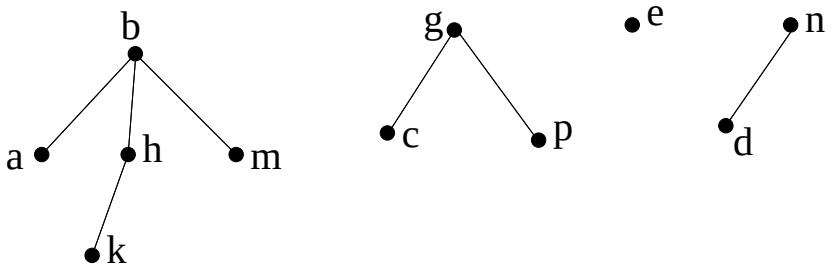
Input: Sommet u d'un graphe
Output: le représentant de la composante de u.

```
1: r ← u
2: while r.pred ≠ NULL do
3:     r ← r.pred
4: end while
5: return r
```

Exemple : Supposons que nous avons les composantes suivantes :

$$\{a, b, h, k, m\}, \{c, g, p\}, \{e\}, \{d, n\},$$

avec comme représentants respectifs b, g, e et n. Ils pourraient donc être stockés comme suit :

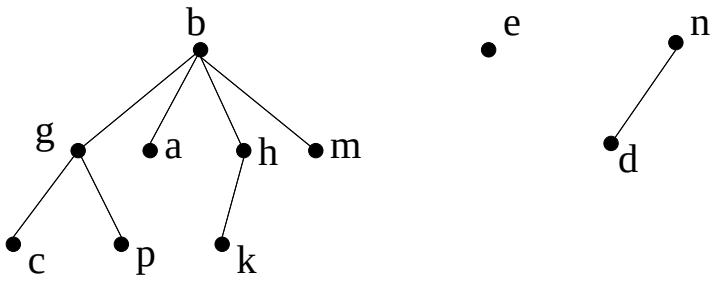


Donc par exemple FIND(k) retournerait b (la racine), FIND(g) retournerait g, FIND(e) retournerait e, et FIND(d) retournerait n. Si nous appelons UNION(a, m), l'algorithme ne toucherait pas aux arbres et retournerait `FALSE` (puisque a et m se trouvent dans la même composante). Par contre si nous appelons UNION(h, p), l'algorithme retournerait `TRUE` et nous aurions :

Algorithme 57 UNION(u, v)

Input: Sommets u et v d'un graphe.
Output: Fusionne les ensembles contenant u et v ; retourne `TRUE` ssi u et v étaient dans des composantes distinctes.

```
1: r ← FIND(u)
2: s ← FIND(v)
3: if r ≠ s then
4:    if s.size ≤ r.size then
5:       s.pred ← r
6:       r.size ← r.size + s.size
7:    else
8:       r.pred ← s
9:       s.size ← s.size + r.size
10:   end if
11:   return TRUE
12: else
13:   return FALSE
14: end if
```



En effet, puisque la composante de p (3 sommets) est plus petite que celle de h (5 sommets), sa racine g est ajoutée comme descendant de la racine de la composante de h (c'est à dire b). Nous avons donc fait l'union des deux composantes. Nos composantes sont donc maintenant : $\{a, b, c, g, h, k, m, p\}, \{e\}, \{d, n\}$.

7.9.3 Version finale de l'algorithme de Kruskal

Nous commençons par mettre toutes les arêtes dans un heap, afin de pouvoir rapidement retrouver la moins chère à chaque étape. Au fur et à mesure que nous choisissons quelles arêtes prendre pour notre arbre couvrant minimal, nous les plaçons dans l'ensemble T. Nous regardons les arêtes de la moins chère à la plus chère. Une arête est ajoutée à T si elle connecte deux composantes connexes (si UNION(u, v) retourne `TRUE`), c'est-à-dire si elle ne crée pas de cycle. On sait qu'un arbre sur n sommets doit avoir $n - 1$ arêtes, donc dès que $|T| = n - 1$ nous avons notre arbre T.

Algorithme 58 MinSpanKruskal(G, w)

Input: Graphe $G = (V, E)$ muni d'une fonction de poids $w\colon E \to \mathbb{R}_{>0}$
Output: Ensemble E' d'arêtes qui forment un arbre couvrant minimal de G.

```
1: BottomUpHeapCreate(E) (par rapport à w)
2: for v ∈ V do
3:    v.pred ← NULL
4:    v.size ← 1
5: end for
6: T ← ∅
7: while Nombre d'arêtes dans T < n − 1 do
8:    (u, v) ← DeleteMin(E)
9:    b ←Union(u, v)
10:   if b = TRUE then
11:      T ← T ∪ {(u, v)}
12:   end if
13: end while
14: return T
```

7.9.4 L'algorithme de Prim

Comme l'algorithme de Kruskal, l'algorithme de Prim trouve un arbre couvrant minimal dans un graphe G. Il commence à un sommet et grandit ensuite l'arbre arête par arête.

En tant qu'algorithme glouton, quelle arête devrions nous prendre ? L'arête la moins chère qui ajoute un sommet à l'arbre sans créer de cycle.

Pendant l'exécution, nous indexons chaque sommet soit avec "fringe" (anglais pour bord) s'il existe une arête que le relie à l'arbre déjà créé, et avec "unseen" sinon.

```
1: Choisir un sommet arbitraire pour commencer, ajouter ses voisins à fringe.
2: while il y a des sommets fringe do
3:    Choisir une arête e = (v, f), avec v appartenant à l'arbre et f à fringe.
4:    Ajouter le sommet v et l'arête e à l'arbre.
5:    Enlever f de fringe.
6:    Ajouter tous les voisins unseen de f dans fringe.
7: end while
```

Cette procédure crée clairement un arbre couvrant, puisqu'aucun cycle ne peut être introduit entre l'arbre et des sommets fringe. Mais s'agit-il d'un arbre minimal ?

Théorème 7.10 *Soit $G = (V, E)$ un graphe connexe et pondéré (avec poids). Soit $E_T \subseteq E$ un sous-ensemble d'arêtes tel que $T = (V, E_T)$ est un arbre couvrant minimal. Soit $E' \subseteq E_T$, et soit $V' \subseteq V$ l'ensemble des sommets incidents à une arête dans E'. Si (x, y) est une arête de poids minimal telle que $x \in V'$ et $y \notin V'$, alors $E' \cup \{(x, y)\}$ est un sous-ensemble d'un arbre couvrant minimal.*

Preuve. Nous pouvons supposer que $(x, y) \notin E_T$. Comme T est un arbre couvrant, il y a un chemin de x à y dans T. Soit (u, v) une arête sur ce chemin telle que $u \in V'$, $v \notin V'$. En effaçant (u, v) et en le remplaçant par (x, y), on obtient un autre arbre couvrant. Comme $w(u, v) \geq w(x, y)$, le coût de cet arbre couvrant est inférieur ou égal au coût de T. ■

Quel est le temps de parcours de l'algorithme de Prim ? Cela dépend des structures de données utilisées.

L'implémentation la plus simple : marquer chaque sommet comme appartenant à l'arbre ou non, et recommencer la recherche à chaque fois dès le début :

1: Choisir un sommet arbitraire pour commencer.
2: **while** il y a des sommets non-arbre **do**
3: Choisir une arête minimale avec une extremité dans l'arbre et l'autre hors l'arbre.
4: Ajouter l'arête choisie et le sommet correspondant à l'arbre.
5: **end while**

On peut réaliser ceci en temps $O(|V| \cdot |E|)$ en utilisant DFS/BFS pour parcourir toutes les arêtes avec un temps constant par arête et un total de n itérations, $n = |V|$. Peut-on faire mieux ? *OUI.*

L'algorithme amélioré résultant ressemble à l'algorithme de Dijkstra. Nous ajoutons à l'arbre T un sommet (et une arête) à chaque étape. Nous stockons l'ensemble Q des sommets qui ne sont pas encore dans T. Les sommets de T sont donc ceux de $V \setminus Q$. Pour chaque sommet $v \in Q$, nous stockons une **clé**, qui, à n'importe quel point de l'algorithme est la distance minimale de v à l'arbre actuel T. A la fin de l'algorithme, l'ensemble Q est vide, T contient tous les sommets du graphe, et est un arbre minimal couvrant. Nous implémentons Q sous forme de queue à priorité (heap) par rapport aux clés (les plus petits éléments étant prioritaires).

Remarquons que nous représentons les arêtes de T avec l'array p. Pour un sommet v, $p[v]$ est le "prédécesseur" de v, donc $(p[v], v)$ est une arête de T. Notons que T a bien $n-1$ arêtes puisque tous les sommets sauf s ont un prédécesseur.

Pour montrer que l'algorithme fonctionne correctement, nous devons montrer que $key[v]$ est toujours la distance de v à $V \setminus Q \cup \{u\}$, où u est choisi à l'étape 8. C'est une simple preuve par induction.

Pourquoi avons-nous besoin de l'étape 13 (SiftUp) ? Nous utilisons Q comme min-heap. Dans l'étape 12, la valeur de $key[v]$ a été réduite. Donc, v doit trouver sa juste place dans le heap, et il faut donc faire un SiftUp (c'est SiftUp plutôt que SiftDown, parce que l'arbre est un min-heap, i.e. la racine contient le plus *petit* élément).

Théorème 7.11 *Le temps de parcours de l'algorithme* MinSpanPrim *est* $O(m \log(n))$, *où* $m = |E|$, $n = |V|$.

Preuve. La queue est créée en temps $O(n)$ dans l'étape 6 (BottomUpHeapCreate, par

Algorithme 59 MinSpanPrim(G, w)

Input: Graphe $G = (V, E)$ avec fonction de poids $w\colon E \to \mathbb{R}_{>0}$
Output: Ensemble E' d'arêtes qui forment un arbre couvrant de G.

```
1: Choisir s ∈ V arbitrairement.
2: for u ∈ V \ {s} do
3:    key[u] ← ∞
4: end for
5: key[s] ← 0, p[s] ← NULL
6: Q queue de priorité (min-heap) sur V p.r. à key
7: while Q n'est pas vide do
8:    u ← ExtractMin(Q)
9:    Marquer u
10:   for v ∈ N[u] do
11:      if v n'est pas marqué ∧ w(u, v) < key[v] then
12:         p[v] ← u, key[v] ← w(u, v)
13:         SiftUp(Q, v)
14:      end if
15:   end for
16: end while
17: E' ← {(p[v], v) | v ∈ V ∧ v ≠ s}.
```

exemple). L'étape 8 utilise un temps $O(\log(n))$. La boucle while de la ligne 7 tourne n fois, et ainsi le coût du pas 8 dans l'algorithme entier est $O(n \log(n))$. La boucle for de la ligne 10 est parcourue au plus $2m$ fois, comme chaque arête est visitée au plus une fois pour chacun des sommets incidents. L'opération à la ligne 13 est effectuée en temps $O(\log(n))$, donc le coût total de cette opération pendant l'exécution complète de l'algorithme est $O(m \log(n))$. Toutes les autres opérations utilisent un temps $O(n)$. Ainsi, le temps de parcours complet est

$$O(n \log(n) + m \log(n) + n) = O(m \log(n)),$$

concluant la preuve. ■

7.10 DÉTECTER DES CYCLES

Pendant la discussion de l'algorithme Moore-Bellman-Ford nous avions supposé que le graphe ne contenait pas de cycles négatifs. Dans cette section nous verrons comment trouver de tels cycles. De plus, nous étudierons le problème de trouver un cycle avec le plus petit poids d'arête moyen (l'algorithme de Karp).

7.10.1 Détecter des cycles négatifs

L'algorithme suivant, d'une simplicité surprenante, détermine si un graphe $G = (V, E)$ muni d'une fonction de poids $w\colon E \to \mathbb{R}$ possède un cycle négatif.

Algorithme 60 NegativeCycle(G, w)

Input: Graphe $G = (V, E)$ avec fonction de poids $w\colon E \to \mathbb{R}$
Output: `TRUE` si G a un cycle négatif, `FALSE` sinon.

1: Choisir un $s \in V$ arbitraire.
2: Appliquer l'algorithme de Moore-Bellman-Ford à (G, s), pour obtenir pour tout $w \in V$ la distance la plus courte $\ell(w)$ entre s et w.
3: **for** $(u, v) \in E$ **do**
4: **if** $\ell(u) + w(u, v) < \ell(v)$ **then**
5: **return** `TRUE`
6: **end if**
7: **end for**
8: **return** `FALSE`

Théorème 7.12 *L'algorithme ci-dessus donne un résultat correct en $O(nm)$ opérations, où $n = |V|$ et $m = |E|$.*

Preuve. Pour le temps de parcours, nous savons qu'il faut $O(nm)$ opérations pour l'algorithme MBF, et voyons ensuite qu'il en faut $O(m)$ pour la boucle for aux lignes 3-7 (puisque la boucle est exécutée une fois pour chaque arête).

Montrons maintenant que l'algorithme est correct. S'il existe une arête (u, v) telle que $\ell(u) + w(u, v) < \ell(v)$, alors $\ell(v)$ n'est pas la longueur du plus court chemin de s à v, comme le chemin passant par u est encore plus court. Donc G contient nécessairement un cycle négatif, puisque nous savons que l'algorithme de Moore-Bellman-Ford marche sur tout graphe qui n'a pas de cycles négatifs. Supposons maintenant que l'algorithme termine avec l'output `FALSE`. Alors :

$$\forall\ (u, v) \in E\colon \quad \ell[u] + w(u, v) \geq \ell[v].$$

Soit $C = (v_0, v_1, \ldots, v_k)$ un cycle, i.e., $(v_i, v_{i+1}) \in E$ pour tout i, et $v_0 = v_k$. Nous aimerions montrer que C est positif.

$$\sum_{i=1}^{k} \big(\ell[v_{i-1}] + w(v_{i-1}, v_i)\big) \geq \sum_{i=1}^{k} \ell[v_i].$$

Comme $\sum_{i=1}^{k} \ell[v_{i-1}] = \sum_{i=1}^{k} \ell[v_i]$ $(v_0 = v_k)$, on peut enlever ces deux termes de la somme ci-dessus pour obtenir :

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0.$$

Donc, le cycle C est positif. Ainsi le graphe ne peut pas contenir de cycles négatifs. ■

7.10.2 L'algorithme de Karp pour trouver des cycles avec poids moyen minimal

Soit $C = (v_0, v_1, \ldots, v_k)$ un cycle dans le graphe G (donc $v_0 = v_k$). Nous supposons que G est orienté et connexe. Alors la quantité

$$\mu(C) := \frac{1}{k} \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

est le poids moyen de C. Dès lors,

$$\mu^* = \mu^*(G) = \min_{C \text{ cycle dans } G} \mu(C)$$

est le poids moyen minimal d'un cycle du graphe. Karp a construit en 1978 un algorithme pour calculer cette quantité.

Soit s un sommet arbitraire dans $G = (V, E)$. Pour $x \in V$ soit

$$F_k(x) := \min \left\{ \sum_{i=1}^{k} w(v_{i-1}, v_i) \;\middle|\; v_0 = s, v_k = x, (v_{i-1}, v_i) \in E \right\}$$

la longueur du plus court chemin de s à x utilisant exactement k arêtes. Pour tout $x \in V$, $0 \leq k < n$, nous pouvons calculer les valeurs $F_k(x)$ en utilisant la programmation dynamique :

- $F_0(s) = 0$
- $\forall\, x \in V \setminus \{s\}$: $\quad F_0(x) = \infty$
- $\forall\, 1 \leq k \leq n, x \in V$:

$$F_k(x) = \min_{u \in V \wedge (u,x) \in E} \{F_{k-1}(u) + w(u, x)\}.$$

Exercice : Montrer que les récursions sont correctes et que les valeurs $F_k(x)$ peuvent être calculées en temps $O(nm)$.

Théorème 7.13 *La quantité μ^* satisfait l'égalité suivante :*

$$\mu^* = \min_{x \in V} \max_{0 \leq k < n} \frac{F_n(x) - F_k(x)}{n - k}.$$

Preuve. Nous considérons d'abord le théorème dans le cas $\mu^* = 0$, i.e., nous montrons que dans ce cas

$$\min_{x \in V} \max_{0 \leq k < n} F_n(x) - F_k(x) = 0.$$

Soit $\ell(x)$ le coût minimal d'un chemin *simple* de s à x (i.e., un chemin qui ne passe pas deux fois par le même sommet).

Un tel chemin passe par au plus $n-1$ arêtes (sinon il ne serait pas simple). De plus, comme $\mu^* = 0$, il n'y a pas de cycles négatifs, les chemins simples sont donc les plus courts et on a donc

$$\forall x \in V : \quad \ell(x) = \min_{0 \le k < n} F_k(x) \le F_n(x).$$

Ceci implique que

$$\forall x \in V : \quad \max_{0 \le k < n} F_n(x) - F_k(x) \ge 0,$$

donc en particulier :

$$\min_{x \in V} \max_{0 \le k < n} F_n(x) - F_k(x) \ge 0.$$

Pour finir la preuve, nous devons montrer qu'il y a un sommet x tel que $F_n(x) = \ell(x)$. Soit C un cycle dans le graphe avec $\mu(C) = 0$, et soit $w \in C$ arbitraire (nous savons qu'un tel cycle existe puisque $\mu^* = 0$).

Soit P un chemin minimal de s à w suivi par n itérations du cycle C. P contient au moins n arêtes. Soit P' le chemin formé par les premières n arêtes de P. Le coût du chemin P est $\ell(w)$ (comme $\mu(C) = 0$ et il n'y a pas de cycles négatifs). Tout sous-chemin d'un chemin de coût minimal est aussi de coût minimal. Si E' dénote les arêtes de P', et x son sommet terminal, alors

$$\sum_{e \in E'} w(e) = \ell(x) = F_n(x).$$

Donc, x est un sommet avec $\ell(x) = F_n(x)$, ce qui montre l'affirmation du théorème pour $\mu^* = 0$.

Dans le cas général, en réduisant les poids des arêtes d'une constante t, on réduit

$$\min_{x \in V} \max_{0 \le k < n} \frac{F_n(x) - F_k(x)}{n-k}$$

par t. (Pourquoi ?) Choisir alors $t = -\mu^*$ et appliquer la partie précédente. ■

Théorème 7.14 *L' algorithme de Karp calcule son output en temps $O(nm)$.*

Preuve. Les valeurs $F_k(x)$ pour $x \in V$ peuvent être calculées en temps $O(nm)$. La même chose est vraie pour les valeurs

$$\frac{F_n(x) - F_k(x)}{n-k}.$$

Pour construire un cycle C avec $\mu(C) = \mu^*$, on doit trouver k et x tel que

$$\frac{F_n(x) - F_k(x)}{n-k} = \mu^*.$$

Ces valeurs peuvent être calculées est stockées pendant le calcul de μ^*. Cette identité dit que le cycle qui nous intéresse est formé des $n-k$ dernières arêtes d'un chemin minimal de s à x avec n arêtes. Il est donc facile de reconstruire le cycle si pendant le calcul de $F_k(x)$ nous stockons aussi le sommet prédécesseur $p_k(x)$:

$$p_k(x) := v \text{ avec } F_k(x) = F_{k-1}(x) + w(v,x).$$

Ceci conclut la preuve. ■

8
Les problèmes NP-complets

Nous avons jusqu'à présent surtout vu des problèmes pour lesquels il existe une solution algorithmique efficace, ce qui a peut-être suscité la fausse impression que tous les problèmes ont de telles solutions.

Ce n'est pas vrai du tout ! En fait, malgré de nombreux efforts des meilleurs experts du domaine, beaucoup de problèmes qui semblent similaires à ceux vus dans ce cours restent sans algorithme efficace.

La théorie de NP-complétude tente d'expliquer pourquoi nous n'avons pas réussi à trouver des bons algorithmes dans certains cas. Dans ce chapitre, nous verrons les aspects principaux de cette théorie.

8.1 LA CLASSE P

Une fonction $f\colon \mathbb{N} \to \mathbb{R}$ est dite *polynomiale* en fonction de $g\colon \mathbb{N} \to \mathbb{R}$ s'il existe un polynôme p et un entier N tels que pour tout $n \geq N$: $f(n) \leq p\big(g(n)\big)$.

Exemples :

- Décider si un graphe $G = (V, E)$ est connexe : la longueur d'input est polynomiale en $n = |V|$. La longueur d'output est $O(1)$.
- Trouver le flux maximal d'un réseau (G, c, s, t), où les coûts sont des entiers positifs $\leq M$: la longueur d'input est polynomiale en $|G|$ et $\log_2(M)$. La longueur d'output est aussi polynomiale en $|G|$ et $\log_2(M)$.
- Décider si un entier N est premier : la longueur d'input est $\lceil \log_2(N) \rceil$. La longueur d'output est $O(1)$.
- Le problème du sac à dos : Etant donné n objets de poids $\leq W$ et des valeurs $\leq V$, trouver la plus grande somme de sous-ensemble (*subset sum*) qui est $\leq M$: la longueur d'input est polynomiale en n, $\log_2(W)$, $\log_2(V)$, et $\log_2(M)$; la longueur d'output est polynomiale en n et $\log_2(W)$.

On dit qu'un problème *est dans P* ou *est en temps polynomial* s'il existe un algorithme pour le résoudre dont le temps de parcours est polynomial en la longueur de l'input.

Exemples :

- Décider si un graphe est connexe est dans P.
- Trouver le flux maximal d'un graphe est dans P.
- Est-ce que le problème du sac-à-dos est dans P ?
- Est-ce que le problème de décider la primalité d'un entier est dans P ?

Il a été récemment prouvé, après des années de recherche, voire des siécles (depuis Gauss en fait), que ce dernier problème est dans P. En effet un algorithme (l'algorithme AKS, 2002) résout ce problème en temps polynomial.

Le troisième problème n'est très probablement pas dans P : nous montrerons plus tard qu'il est NP**-complet**.

La classe P couvre, en un certain sens, la classe des problèmes que l'on peut *résoudre de manière efficace*, asymptotiquement.

8.2 RÉDUCTION POLYNOMIALE

But : Utiliser une solution d'un problème B pour résoudre un problème A.

Nous dirons que le problème A est *polynomialement réductible* à un problème B si n'importe quel input a de A, peut être transformé *en temps polynomial* en un input b de B tel que A est solvable sur a si et seulement si B est solvable sur b. De plus, nous exigeons que la longueur de b soit polynomiale en la longueur de a.

Nous notons ceci $A \leq_P B$. Cette relation entre problèmes est en fait une relation transitive : Si $A \leq_P B$ et $B \leq_P C$, alors $A \leq_P C$. En effet, deux réductions polynomiales forment une réduction polynomiale.

Exemple : Soit A le problème maximum bipartite matching, et B le problème du flux maximal. Nous avons vu que $A \leq_P B$.

Si $A \leq_P B$, alors "*A est au plus aussi dur que B*". Si $A \leq_P B$ et $B \leq_P A$, nous écrivons $A =_P B$. Dans ce cas, A et B sont "à peu près de même difficulté".

8.3 LA CLASSE NP

Pour de nombreux problèmes, aucun algorithme en temps polynomial n'est connu. Cependant, pour de nombreux *problèmes de décision*, on peut donner un algorithme qui vérifie une solution en temps polynomial.

Exemple : Considérons le problème de décision suivant : "Etant donné un entier n, est-t-il composé ?" (i.e. non-premier).

Supposons que nous ne sachions pas comment résoudre ce problème en temps polynomial (ce qui était encore le cas avant la decouverte de l'algorithme AKS en 2002). Il est

par contre facile de convaincre quelqu'un qu'un nombre n est composé en exhibant deux entiers $a > 1$ et $b > 1$ tels que $a \cdot b = n$. Ce test se fait en temps polynomial en $\log_2(n)$, i.e. en la longueur de l'input.

La classe NP consiste en tous les problèmes de décision pour lesquels il existe une *preuve* en temps polynomial. Par exemple, le problème ci-dessus de savoir si un nombre est composé est dans NP.

Tous les problèmes dans P sont aussi dans NP. (Pourquoi ?)

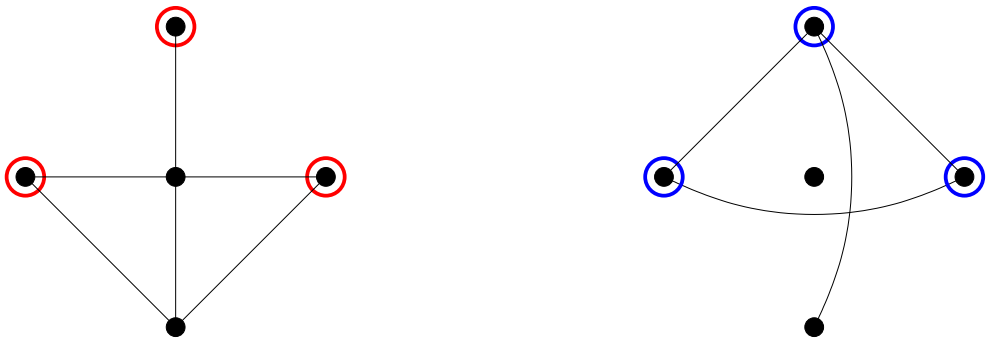
Nous donnons à présent une liste d'autres problèmes dans NP.

- IndependentSet : étant donné un graphe $G = (V, E)$, et un entier K, le graphe contient-il un ensemble indépendant de taille $\geq K$?
  Rappelons qu'un ensemble independant dans un graphe $G = (V, E)$ est un sous-ensemble I de V tel que deux éléments quelconques de I ne sont pas reliés par une arête de V.
- Clique : étant donné un graphe $G = (V, E)$ et un entier K, le graphe contient-il une clique de taille $\geq K$?
  Rappelons qu'une clique dans un graphe $G = (V, E)$ est un sous-ensemble C de V tel que deux éléments quelconque de C sont reliés par une arête de V.

Nous avons que

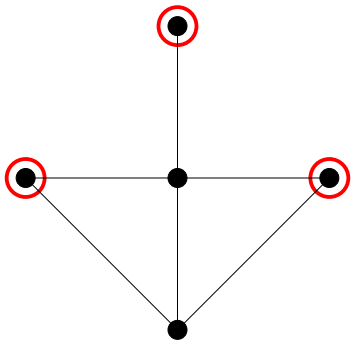
$$\text{Independent Set}(G, K) =_P \text{Clique}(\bar{G}, K).$$

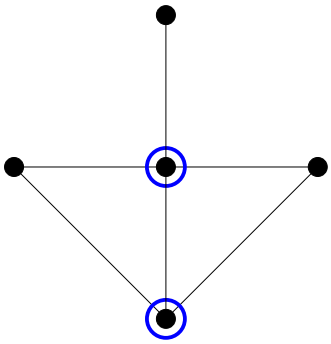
En effet, un ensemble indépendant est une clique dans le complément du graphe.



Un *node cover* dans un graphe $G = (V, E)$ est un sous-ensemble $S \subseteq V$ tel que chaque arête a au moins un point terminal dans S. Le problème Node-Cover est comme suit : étant donné un entier K et le graphe G, existe-t-il un node cover dans G de taille $\leq K$?

Remarquons que Node-Cover $=_P$ Independent-Set : Un sous-ensemble S de V est un node cover si et seulement si $I = V \setminus S$ est un ensemble indépendant !





8.4 NP-COMPLÉTUDE

Un problème A est appelé *NP-complet* si $A \in$ NP et pour tous les problèmes B dans NP on a $B \leq_P A$.

Ainsi, si on peut résoudre A en temps polynomial, alors on peut résoudre tous les problèmes dans NP en temps polynomial ! Donc, les problèmes NP-complets peuvent être considérés comme les problèmes les *plus difficiles* dans la classe NP. Nous avons le résultat suivant :

Si A est NP-complet, B est dans NP, et $A \leq_P B$, alors B est aussi NP-complet.

La preuve est en fait triviale et s'ensuit des définitions. Le premier problème a avoir été identifié comme étant NP-complet est le *satisfiability problem* (SAT).

8.5 PROBLÈMES DE SATISFIABILITÉ

Considérons les formules booléennes suivantes :

- Variables : $x_1, \ldots, x_n$;
- Littéraux λ_i : x_i ou $\neg x_i$;
- Clauses C_j : $\lambda_1 \vee \lambda_2 \vee \cdots \vee \lambda_{k_j}$ pour un certain $k_j > 0$;
- Formule F : $C_1 \wedge C_2 \wedge \cdots \wedge C_t$ pour un certain $t > 0$.

Exemple :

$$(x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_4).$$

Le probème SAT est alors le suivant :

SAT : Etant donné une formule F, peut-on donner aux variables des valeurs dans $\{0,1\}$ telles que F soit vraie ?

Théorème 8.1. (Théorème de Cook-Levin) SAT est NP-complet.

Le problème k-SAT est une variante de SAT dans laquelle chaque clause doit consister en exactement k littéraux.

Théorème 8.2. 3-SAT est NP-complet.

Preuve. Nous allons montrer que SAT $\leq_P$ 3-SAT. Etant donné une formule F comme input pour SAT, nous introduisons pour chaque clause

$$C_i = \lambda_1 \vee \cdots \vee \lambda_s$$

une nouvelle variable x_i, et divisons C_i en deux clauses :

$$F_i = (\lambda_1 \vee \cdots \vee \lambda_{s-2} \vee x_i) \wedge (\lambda_{s-1} \vee \lambda_s \vee \neg x_i).$$

Clairement, F_i est satisfaisable si et seulement si C_i est satisfaisable.

Nous procédons récursivement en coupant la clause la plus grande jusqu'à ce que toutes les clauses aient trois littéraux. Si F contient une clause avec deux littéraux λ_1 et λ_2, nous ajoutons une nouvelle variable X et divisons la clause en

$$(\lambda_1 \vee \lambda_2 \vee x) \wedge (\lambda_1 \vee \lambda_2 \vee \neg x).$$

Ceci termine la preuve. ■

On peut montrer de la même façon : k-SAT est NP-complet pour $k \geq 3$. Cependant : 2-SAT est dans P ! C'est-à-dire, il existe un algorithme pour résoudre 2-SAT en temps polynomial.

Not-all-equal-3-SAT (NAE-3-SAT) : Toutes les clauses ont trois littéraux. Une clause est satisfaite si et seulement s'il existe deux littéraux dans la clause avec des valeurs différentes.

Par exemple :

$$(x, y, z) \notin \{(1,0,1), (0,1,0)\}$$

satisfait la clause

$$x \vee \neg y \vee z.$$

Théorème 8.3. NAE-3-SAT est NP-complet.

Preuve. Nous montrons que 3-SAT $\leq_P$ NAE-SAT. Soit $F = C_1 \wedge \cdots \wedge C_m$ une formule dans laquelle chaque clause contient exactement 3 littéraux (donc F est un input pour le problème 3-SAT). Nous allons construire à partir de F une formule G de telle façon que F soit satisfaisable si et seulement si G est NAE-satisfaisable.

Nous allons utiliser dans G les mêmes variables $x_1, \ldots, x_n$ que dans F, en ajoutant en plus $m+1$ variables : $w_1, \ldots, w_m$ (donc un w_j pour chaque clause C_j), et une variable z. Ensuite chaque clause

$$C_j = \lambda_{j1} \vee \lambda_{j2} \vee \lambda_{j3}$$

dans F est transformée en 2 nouvelles clauses :

$$\overbrace{(\lambda_{j1} \vee \lambda_{j2} \vee w_j)}^{C_j^1} \wedge \overbrace{(\neg w_j \vee \lambda_{j3} \vee z)}^{C_j^2}.$$

dans G (G contient donc $2m$ clauses).

Si une attribution satisfait F, alors nous pouvons construire une attribution qui NAE-satisfait G : Nous gardons les mêmes valeurs pour $x_1, \ldots, x_n$, puis nous posons $z = 0$, et $\forall j = 1, \ldots, m : \ w_j = \lambda_{j3} \wedge \neg\lambda_{j1}$. En effet :
• Si $\lambda_{j3} = 1$ alors C_j^2 est NAE-satisfaite (puisque $z = 0$), et comme $w_j \neq \lambda_{j1}$, C_j^1 est aussi NAE-satisfaite par notre attribution.
• Si $\lambda_{j3} = 0$ alors $w_j = 0$, et donc $\neg w_j = 1$, ce qui implique que C_j^2 est NAE-satisfaite. De plus, nous devons avoir $\lambda_{j1} = 1$ ou $\lambda_{j2} = 1$ (puisque $\lambda_{j3} = 0$ et l'attribution doit satisfaire C_j) et donc C_j^1 est aussi NAE-satisfaite.

Réciproquement, si une attribution NAE-satisfait G, alors nous allons montrer que cette même attribution (restrainte aux variables $x_1, \ldots, x_n$) satisfait aussi F. Supposons sans perdre de généralité que $z = 0$ (sinon nous prenons le complément de l'attribution, qui NAE-satisfait aussi G). Nous voyons que pour chaque clause C_j :
• Si $\lambda_{j3} = 1$ alors C_j est clairement satisfaite.
• Si $\lambda_{j3} = 0$ alors nous devons avoir $\neg w_j = 1$ (pour que C_j^2 soit NAE-satisfaite), et donc $w_j = 0$ ce qui implique que soit $\lambda_{j1} = 1$ ou $\lambda_{j2} = 1$ (puisque C_j^1 est NAE-satisfaite) et donc $C_j = \lambda_{j1} \vee \lambda_{j2} \vee \lambda_{j3}$ est aussi satisfaite. ■

Un autre problème de satisfiabilité NP-complet est

Max-2-SAT : Etant donné une formule dont toutes les clauses ont au plus deux littéraux, trouver une attribution des variables qui maximise le nombre de clauses satisfaites.

Exemple fondamental :

$$x \wedge y \wedge z \wedge w \wedge$$
$$(\neg x \vee \neg y) \wedge (\neg y \vee \neg z) \wedge (\neg z \vee \neg x) \wedge$$
$$(x \vee \neg w) \wedge (y \vee \neg w) \wedge (z \vee \neg w).$$

Lemme 8.1. Si $x \vee y \vee z = 1$, alors il existe une attribution pour w qui satisfait sept des clauses ci-dessus. Si $x = y = z = 0$, alors pour n'importe quelle valeur de w, la formule ci-dessus aura au plus 6 clauses satisfaites.

Preuve. Exercice.

En utilisant cette construction, nous pouvons montrer que Max-2-SAT est NP-complet.

Théorème 8.4. Max-2-SAT est NP-complet.

Preuve. Nous réduisons 3-SAT à ce problème. Etant donné la formule 3-SAT $C_1 \wedge \cdots \wedge C_m$, nous introduisons pour chaque clause une nouvelle variable w_i et remplaçons la clause C_i

par les 10 clauses ci-dessus (avec w remplacé par w_i et x, y et z remplacé par les litéraux de C_i).

La nouvelle formule a $10m$ clauses. Nous montrerons que la formule a $\geq 7m$ attributions satisfaisantes si et seulement si la formule originale est satisfaisable. Dans chaque groupe de 10 nouvelles clauses, il peut y en avoir au plus 7 qui sont satisfaisantes. Ainsi, s'il y en a $\geq 7m$ qui sont satisfaites, le nombre de clauses satisfaites est exactement $7m$, et par le lemme ci-dessus toutes les clauses dans la formule 3-SAT originale sont satisfaites. S'il y a moins que $7m$ clauses satisfaites, alors il y a un groupe qui a moins que 7 clauses satisfaites, et donc la formule originale n'est pas satisfaisable. ■

8.6 QUELQUES PROBLÈMES DE GRAPHE NP-COMPLETS

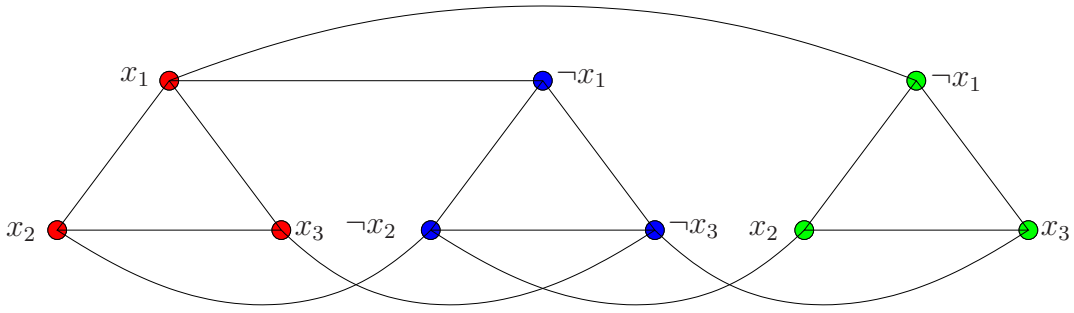
On rappelle d'abord que INDEPENDENT-SET est le problème suivant : Etant donné un graphe G et un nombre $t \in \mathbb{N}$, déterminer s'il existe dans G un ensemble indépendant de taille $\geq t$

Théorème 8.5. INDEPENDENT-SET est NP-complet.

Preuve. Nous allons montrer que 3-SAT $\leq_P$ INDEPENDENT-SET. Etant donné une formule 3-SAT avec t clauses, nous construisons un graphe qui a un ensemble indépendant de taille $\geq t$ si et seulement si la formule est satisfaisable. La construction de ce graphe se fait comme suit :

- Chaque clause correspond à un triangle.
- Les littéraux complémentaires sont reliés par une arrête.

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$$



Une attribution valable dans une formule 3-SAT avec t clauses donne un ensemble indépendant de taille $\geq t$:

Nous commençons par mettre dans notre ensemble tous les sommets correspondant à des littéraux qui sont vrais dans l'attribution. Dans chaque triangle, s'il y a plus que deux sommets dans notre l'ensemble, nous en enlevons un ou deux pour réduire le nombre à un.

L'ensemble résultant est indépendant : Un sommet dans un triangle ne peut pas être connecté a un sommet dans un autre triangle, puisque le littéral et son complément ne peuvent pas tous les deux valoir un. Deux sommets dans l'ensemble indépendant ne peuvent pas être dans un triangle, puisque le procédé s'en est assuré.

De plus, comme toutes les clauses sont vraies, il y a au moins un littéral par clause qui est vrai, et donc un moins un sommet par triangle qui est dans notre ensemble. Comme il y a t triangles, la taille de l'ensemble est $\geq t$.

Un ensemble indépendant de taille $\geq t$ dans le graphe mène à une attribution valable de la formule originale :

Clairement, si la formule originale a t clauses, alors le graphe a t triangles, et l'ensemble indépendant maximal est donc de taille t. Nous donnons à chaque littéral dans l'ensemble indépendant la valeur 1, et aux autres littéraux la valeur 0.

Alors chaque clause dans la formule est satisfaisable, et il n'y a pas de contradiction entre les littéraux et leurs compléments puisqu'ils sont connectés dans le graphe (donc les deux ne peuvent pas être dans l'ensemble indépendant). ■

Corollaire 8.1. CLIQUE et NODE-COVER sont NP-complets.

C'est parce que

$$\text{INDEPENDENT-SET} =_P \text{CLIQUE} =_P \text{NODE-COVER}.$$

Un cut dans un graphe $G = (V, E)$ est une paire $(S, V \setminus S)$, où $S \subset V$. La taille d'un cut est le nombre d'arrêtes entre S et $V \setminus S$.

MAX-CUT : Etant donné un graphe G et un entier n, existe-t-il un cut de taille $\geq n$?

Théorème 8.6. MAX-CUT est NP-complet.

Preuve. Nous allons montrer que NAE-3-SAT $\leq_P$ MAX-CUT. Etant donné une formule NAE-3-SAT F, nous construisons un graphe G et une valeur K tels que F est NAE-3-satisfaisable ssi G a un cut de taille $\geq K$. Soit m le nombre de clauses dans F, et soient $x_1, \ldots, x_n$ les variables.

On commence par poser $K = 5m$. Ensuite, notre graphe G a comme sommets $x_1, \ldots, x_n$, $\neg x_1, \ldots, \neg x_n$. Pour chaque clause $C_i = \alpha \vee \beta \vee \gamma$, on ajoute le triangle $[\alpha, \beta, \gamma]$ aux arêtes de G. Pour chaque i, soit n_i nombre d'occurrences de x_i ou $\neg x_i$ dans F. Ajouter n_i arêtes entre x_i et $\neg x_i$ dans G. G devient un graphe avec plusieurs arêtes entre ses sommets (un tel graphe est appelé un *multigraphe*). Remarquons que G a $2n$ sommets, et $6m$ arêtes.

Supposons qu'il y a un cut $(S, V \setminus S)$ de taille $\geq 5m$ dans G. Sans perte de généralité, les variables et leurs négations ont des côtés différents d'un cut. Autrement, si x_i et $\neg x_i$ se trouvent du même côté, ils contribuent ensemble au plus $2n_i$ arêtes au cut (nombre d'occurrences fois 2), et donc les mettre de deux côtés différents du cut ne diminuera pas la valeur du cut.

Nous considérons les littéraux de S comme *vrai* et ceux dans $V \setminus S$ comme *faux*.

Les arêtes entre des variables et leurs négations contribuent au plus $3m$ au cut (autant qu'il y a d'occurrences de littéraux). Les $2m$ arêtes restantes doivent venir des triangles. Un triangle contribue soit 0 (si les 3 sommets sont du même coté) soit 2 (s'il y a 2 sommets d'un coté et un de l'autre) au cut. Comme il y a m triangles qui doivent contribuer au moins $2m$, chaque triangle doit contribuer 2 au cut, et donc dans chaque triangle il y a 2 sommets d'un coté du cut et un de l'autre). Ainsi, dans chaque triangle, au moins un litéral est vrai et un est faux, i.e., la formule originale est NAE-satisfaisable.

Inversement, il est facile de traduire l'attribution de vérité qui satisfait toutes les clauses d'un cut de taille $5m$. ■

Remarquons que le problème de minimiser le cut est dans P ! Donc, *les problèmes de minimisation et de maximisation sont complètement différents.*

8.7 QUELQUES AUTRES PROBLEMES DE GRAPHE NP-COMPLETS

Un *cycle Hamiltonien* dans un graphe $G = (V, E)$ est un chemin $(v_0, v_1, \ldots, v_n)$ tel que $v_n = v_0$ et tel que $V = \{v_0, \ldots, v_{n-1}\}$. Le problème HAMILTONIAN-CYCLE demande si un graphe donné possède un cycle Hamiltonien.

Théorème 8.7. HAMILTONIAN-CYCLE est NP-complet.

Preuve. Nous ne la verrons pas ici. Il y a une preuve qui procède en montrant que 3-SAT $\leq_P$ HAMILTONIAN-CYCLE. ■

Notons que le problème HAMILTONIAN-PATH, qui demande s'il existe un *chemin Hamiltonien* dans un graphe donné (i.e. la condition $v_n = v_0$ est relaxée) est aussi NP-complet.

Voyons maintenant le problème du *Voyageur de Commerce* (*Travelling Salesman Problem*) TSP. Etant donné n villes $1, \ldots, n$, des distances entières et non négatives d_{ij} entre chaque paire de villes telles que $d_{ij} = d_{ji}$ et une borne entière B. On se demande s'il y a un tour des villes tel que la distance totale parcourue est inférieure à B. En d'autres mots, existe-t-il une permutation π telle que $\sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} + d_{\pi(n),\pi(i)} \leq B$.

Théorème 8.8. TSP est NP-complet.

Preuve. Nous montrons que HAMILTONIAN-CYCLE$\leq_P$ TSP. Pour un graphe donné G avec n sommets, nous construisons la matrice des distances d_{ij} et un budget B tel que G a un chemin Hamiltonien si et seulement s'il existe un tour de longueur $\leq B$.

Il y a n villes correspondant aux n sommets du graphe et

$$d_{ij} = \begin{cases} 1 & \text{si } (i,j) \in E, \\ n+1 & \text{sinon.} \end{cases}$$

De plus, $B = n + 1$. Clairement, s'il y a un tour de longueur $\leq B$, alors le tour doit utiliser les arêtes dans le graphe et doit visiter chaque ville une fois, donc c'est un parcours Hamiltonien.

Réciproquement, un chemin Hamiltonien dans le graphe mènera à un tour de coût $\leq B$. ■

Supposons que nous devons "colorier" les sommets d'un graphe donné avec k couleurs de telle façon qu'il n'y ait pas deux sommets adjacents de même couleur. Ce problème s'appelle le problème de k-coloriage (k-*coloring*). Pour $k = 2$, ce problème est assez facile à résoudre. Pour $k = 3$, c'est différent.

Théorème 8.9. 3-Coloring est NP-complet.

Preuve. Nous ne la verrons pas ici. On peut utiliser une réduction de NAE-SAT. ■

Il y a de nombreux autres problèmes de théorie des graphes qui sont NP-complets. Pour en savoir plus sur les problèmes NP-complets en général :

- M.R. Garey and D.S. Johnson, *Computers and Interactibility : A Guide to the Theory of NP-Completeness*, Freeman Publishers, 1979.
- C.H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994

8.8 PROBLÈMES DE THÉORIE DES ENSEMBLES

Etant donné des ensembles finis B, G, H (garçons, filles, maisons), chacun contenant n éléments, et un ensemble $X \subseteq B \times G \times H$, le problème Tripartite Matching demande s'il y a un sous-ensemble $M \subseteq X$ de taille n tel que $(b_1, g_1, h_1), (b_2, g_2, h_2) \in M \implies b_1 \neq b_2, h_1 \neq h_2$ et $h_1 \neq h_2$. C'est-à-dire, chaque garçon est connecté à une fille différente, et chaque couple possède sa propre maison.

Théorème 8.10. Tripartite Matching est NP-complet.

Preuve. Nous ne la verrons pas ici.

Le problème Exact Cover by 3-Sets : Etant donné une famille $F = \{S_1, \ldots, S_n\}$ de sous-ensembles d'un ensemble U telle que $|U| = 3m$ pour un entier m, et telle que $|S_I| = 3$ pour tout i, existe-t-il m ensembles dans F dont l'union est égale à U.

Théorème 8.11. Exact Cover by 3-Sets est NP-complet.

Preuve. Tripartite Matching est un cas spécial de Exact Cover by 3-Sets : dans ce cas, l'ensemble U est partitionné en trois ensembles égaux B, G, H tels que chaque ensemble dans F contient un élément de chacun des trois. ■

Rappelons le problème du sac-à-dos Knapsack : Etant donné n éléments de valeurs

positives entières v_i, des poids positifs w_i, une limite W, et une valeur T entière. Nous voulons savoir s'il y a un sous-ensemble $S \subseteq \{1, \ldots, n\}$ tel que

$$\sum_{i \in S} w_i \leq W$$

$$\text{and } \sum_{i \in S} v_i \geq T.$$

Chapitre 4 : Solution de programmation dynamique en temps $O(nW)$. Or, ce temps de parcours n'est pas polynomial en la longueur de l'input $O(n \log(W))$. Existe-t-il un algorithme en temps polynomial pour KNAPSACK ? C'est peu probable.

Théorème 8.12. KNAPSACK est NP-complet.

Preuve. Nous montrons que

$$\text{EXACT COVER BY 3-SETS} \leq_P \text{KNAPSACK}.$$

Soit $\{S_1, \ldots, S_n\}$ et $U = \{1, \ldots, 3m\}$ un input pour EXACT COVER BY 3-SETS. Nous voulons le transformer en un input de KNAPSACK

Soit

$$w_i = v_i = \sum_{j \in S_i} (n+1)^{3m-j}$$

et

$$W = T = \sum_{j=0}^{3m-1} (n+1)^j.$$

On peut voir

Alors le problème KNAPSACK est solvable si et seulement s'il existe un sous-ensemble I tel que $\sum_{i \in I} v_i = T$, ce qui signifie que $\cup_{i \in I} S_i = U$. ■