

Résumé d'algorithmique

Auteurs : Samuel Robyr, Daniel Peppicelli, 3 juillet 2005

Merci à tous pour l'aide donnée, en particulier à Antoine Junod !

Dernière version disponible à l'url <http://epfl.neoch.net/>

Rappels

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$
$$\sum_{i=1}^n i = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$
$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{2n^3+3n^2+n}{6}$$
$$\sum_{i=1}^n i^3 = \frac{n^4+2n^3+n^2}{4}$$

Théorie des ensembles et définitions :

- $A \setminus B = \{x \in A \mid x \notin B\}$: la différence de A et B
- $A \times B$ (produit cartésien de A et B) :
tous les couples (a, b) où $a \in A$ et $b \in B$
- $A \triangle B = (A \setminus B) \cup (B \setminus A)$
- $A^n = A \times A \times \dots \times A$
- $\text{Pot}\{1, 2\} = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$ (ensemble des parties)
- $\text{Pot}^*(X)$ (ensembles des parties finies de X) :
ensemble de tous les sous ensembles finis de X .
Si X est fini $\text{Pot}(X) = \text{Pot}^*(X)$
- $R \subseteq A \times A$ une relation sur A
- R symétrique $\Rightarrow R^{-1} = R$
- $F : A \rightarrow B$ est *surjective* si $\forall b \in B \exists a \in A \mid b = F(a)$
- $F : A \rightarrow B$ est *injective* si $F(a) = F(a') \Rightarrow a = a'$
- Un *problème computationnel* est spécifié de façon abstraite par un triplet d'ensemble $P = (I, O, R)$ où I (inputs) et O (outputs) sont des ensembles non vides et $R \subseteq I \times O$ est une relation.

1 Induction

Induction simple. Une affirmation $A(n)$ est vraie ssi les deux conditions suivantes sont vraies :

- $A(k)$ (normalement on pose $k = 1$) ;
- $\forall n \geq k : A(n) \Rightarrow A(n+1)$.

Induction descendente. Une affirmation $A(n)$ est vraie $\forall n$ ssi les deux conditions suivantes sont vraies :

- $A(n)$ est vraie pour une infinité de n ;
- $\forall n \geq 2 : A(n) \Rightarrow A(n-1)$.

Induction forte. L'affirmation $A(n)$ est vraie ssi les deux conditions suivantes sont vraies :

- $A(k)$ (normalement on pose $k = 1$) ;
- $\forall n \in \mathbb{N} : A(k) \wedge \dots \wedge A(n) \Rightarrow A(n+1)$.

2 Analyse d'algorithmes

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

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

Alors f croît au plus aussi vite que g . Les constantes multiplicatives ne jouent pas de rôle : $a \cdot f(n) = O(b \cdot f(n))$.

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

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

Alors f croît plus lentement que g . Si $f = o(g)$ alors on a aussi $f = O(g)$.

Ω et θ . Pour des fonctions f et g nous disons que

- $f = \Omega(g)$ si $g = O(f)$
- $f = \theta(g)$ si $f = O(g)$ et $f = \Omega(g)$

Si $f = \theta(g)$ alors f et g ont le même ordre de magnitude¹.

Exemples

- $\log n = O(\log n)$
 - $n^2 = O(n^3)$
 - $an = O(bn), \forall a, b \in \mathbb{R}^+$
 - $1000 \cdot n^{1000} = O(2^n)$
 - $100 \log n = O(n)$
 - $O(m+n) = O(\max\{m, n\})$
 - $p(n)$ (degré $< d$) : $p(n) = o(n^d)$
 - $p(n)$ (degré d) : $p(n) \neq o(n^d)$
 - $p(n)$ (degré d) : $p(n) = \theta(n^d)$
- $-\log(n) + n^2 = O(n^2)$
 - $-n^7 = \Omega(n^6)$
 - $-100n^3 = \theta(10n^3)$
 - $-n + n^2 = o(n^3) \neq o(n^2)$
 - $-n + n^2 = O(n^2) = \theta(n^2)$
 - $-n^{1001} = o(1.001^n)$

Fonctions classées dans leur ordre selon O : $144, \log_2(1000n), 100\ln^3(n), 99n + 101, \frac{n}{1000} \log_{10}(n), n^6, \frac{1}{3}3^n, n!$

L'algorithme de Karatsuba. Soit $f(x) = \sum_{i=0}^{2n-1} a_i x^i$ et $g(x) = \sum_{i=0}^{2n-1} b_i x^i$ écrits comme suit :

$$f(x) = \underbrace{(a_0 + \dots + a_{n-1}x^{n-1})}_{=f_0(x)} + x^n \underbrace{(a_n + \dots + a_{2n-1}x^{n-1})}_{=f_1(x)}$$
$$g(x) = \underbrace{(b_0 + \dots + b_{n-1}x^{n-1})}_{=g_0(x)} + x^n \underbrace{(b_n + \dots + b_{2n-1}x^{n-1})}_{=g_1(x)}$$

Algorithme 1 KARATSUBA($f(x), g(x)$) rec : $O(n^{\log_2(3)})$

- $h_0(x) \leftarrow f_0(x) \cdot g_0(x)$
 - $h_2(x) \leftarrow f_1(x) \cdot g_1(x)$
 - $u(x) \leftarrow f_0(x) + f_1(x)$
 - $v(x) \leftarrow g_0(x) + g_1(x)$
 - $A(x) \leftarrow u(x) \cdot v(x)$
 - $h_1(x) \leftarrow A(x) - h_0(x) - h_2(x)$
 - $h(x) \leftarrow h_0(x) + x^n h_1(x) + x^{2n} h_2(x)$
 - return** $h(x)$
-

L'algorithme a besoin de $6n^2 + 1$ opérations (l'algorithme naïf en utilise $8n^2 - 4n + 1$). Pour les puissances de 2 on a les récurrences suivantes :

- $M(2n) = 3M(n), M(1) = 1$
- $A(2n) = 3A(n) + 8n - 4, A(1) = 0$
- $T(2n) = M(2n) + A(2n) = 3T(n) + 8n - 4, T(1) = 1$

Relations de récurrence. Soit $T : \mathbb{N} \rightarrow \mathbb{R}$ une fonction telle qu'il existe $b, c, d \in \mathbb{R}_{>0}$ et $a \in \mathbb{N}$ avec

- $T(n) \leq T(n+1) \forall n \geq 1$
- $T(an) \leq cT(n) + dn^b$

On a alors

- Si $a^b < c$, alors $T(n) = O(n^{\log_a(c)})$
- Si $a^b = c$, alors $T(n) = O(n^{\log_a(c)} \cdot \log_a(n))$
- Si $a^b > c$, alors $T(n) = O(n^b)$

¹ f et g croissent à la même vitesse.

3 Graphes et Arbres

Définitions Un graphe (V, E) est un ensemble fini de sommets V et une relation $E \subseteq V \times V$ (appelée ensemble d'arcs). Il est *connexe* si tout les sommets sont atteignables entre eux. Il est *non-orienté* si E est symétrique.

Quelques définitions liées aux arbres et aux graphes

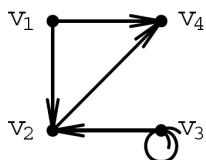
- le **indeg** d'un sommet a est le nombre d'arêtes qui *vont* vers a . le **outdeg** le nombre d'arêtes qui *sortent*.
- la longueur du chemin d'un sommet v à la racine est la **profondeur** de v .
- le **deg** d'un sommet est son nombre de fils.
- la **hauteur** d'un arbre est sa plus grande profondeur.
- le maximum $\max_{v \in V} \deg v$ s'appelle le **deg** de l'arbre.
- bal** (v_k) = hauteur de l'arbre de gauche - hauteur de l'arbre de droite.
- un graphe est dit **connexe** si tout les sommets sont reliés.
- un arbre est un graphe acyclique et connexe.

Parcourir des arbres Il y a trois méthodes de parcourir des arbres binaires :

- Parcours *preorder* : racine, arbre gauche, arbre droite.
- Parcours *inorder* : arbre gauche, racine, arbre droite ;
- Parcours *postorder* : arbre gauche, arbre droite, racine.

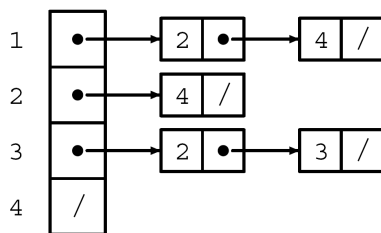
Représentation des graphes Pour représenter un graphe $G = (V, E)$ où $V = \{v_1, \dots, v_n\}$ on peut utiliser une matrice d'adjacence $A = (a_{ij})_{i \leq i, j \leq n}$ telle que pour tout i et j nous avons $a_{ij} \in \{0, 1\}$, définis comme suit :

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



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

On peut aussi utiliser des listes d'adjacences :



Représentation d'arbres Il y a plusieurs manières de représenter les arbres :

- par Structure de pointeurs** n : chaque n contient la valeur du noeud, un pointeur sur chaque fils et un sur son père.
- par Tableaux** : 4 entrée dans le tableaux : l'indice du noeud (i), sa valeur (val) et les indices des sous arbres de gauche et de droite.
- implicite** : Les sommets sont stockés dans un array, les arêtes s'obtiennent de manière implicite par la position des sommets dans l'array : si x est stocké dans $A[i]$, alors les enfants y_1, \dots, y_d de x sont stockés dans $A[di], A[di + 1], \dots, A[di + d - 1]$, où d est le degre ($d \geq 2$).

3.1 Arbres binaires de recherche

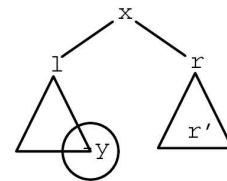
Search. $O(N)$

- 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 noeud 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.

Insert. $O(N)$ 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, 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 .

Delete. $O(N)$ On commence comme pour *Search*(x). Si la recherche échoue, x ne se trouve pas dans l'arbre, nous n'avons 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. Dans le cas général si x a deux fils, on remplace x par y et on remplace y par son sous-arbre de gauche.



Temps de parcours

- meilleur : $\lfloor \log_2(N) \rfloor$;
- moyen : $O(\log(N))$;
- pire : $N - 1$,

3.2 Arbres AVL

Un arbre binaire est dit *équilibré* (ou *arbre AVL*) si pour tout sommet k **bal** $(k) < 2$

La hauteur d'un arbre AVL avec n sommets vérifie l'inégalité : $h < 1.4404 \log(n) + 0.672$. L'hauteur est donc $O(\log(N))$.

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, alors une rotation est effectuée.

Effacement. On commence par effacer le sommet, on le remplace ensuite par le plus petit élément de son sous-arbre de droite ou par le plus grand élément de son sous-arbre de gauche, et pour finir on rééquilibre un utilisant des rotations.

4 Construction d'algorithmes

4.1 Les algorithmes diviser pour régner

Algorithme 2 DPR

Input: input de taille n et de taille maximale n_0

- 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**

4.2 Multiplication de matrices

L'algorithme **MATRIXCHAINORDER**(p) prend comme entrée un tableau décrivant la taille des matrices. (e.g $A_{30 \times 35} \cdot B_{35 \times 15} \cdot C_{15 \times 5}$ donne $p = \{30, 35, 15, 5\}$). S_{ij} est l'indice de la matrice qui précède la séparation du groupe de matrice en un produit optimal et M_{ij} le nombre de multiplications minimal nécessaires pour multiplier les matrices de i à j .

Algorithme 3 MATRIXCHAINORDER(p)

$\Theta(n^3)$

Input: $p = (p_0, p_1, \dots, p_n)$ décrivant les tailles des matrices

Output: Matrices M et S comme décrites ci-dessus

- 1: $n \leftarrow \text{length}(p) - 1$
- 2: **for** $i = 1 \dots n$ **do**
- 3: $M[i, i] = 0$
- 4: **end for**
- 5: **for** $l = 2 \dots n$ **do**
- 6: **for** $i = 1 \dots n - l + 1$ **do**
- 7: $j \leftarrow i + l - 1$
- 8: $M[i, j] = \infty$
- 9: **for** $k = i \dots j - 1$ **do**
- 10: $q \leftarrow M[i, k] + M[k + 1, j] + p_{i-1}p_kp_j$
- 11: **if** $q < M[i, j]$ **then**
- 12: $M[i, j] \leftarrow q$
- 13: $S[i, j] \leftarrow k$
- 14: **end if**
- 15: **end for**
- 16: **end for**
- 17: **end for**
- 18: **return** M et S

4.3 La méthode de horner

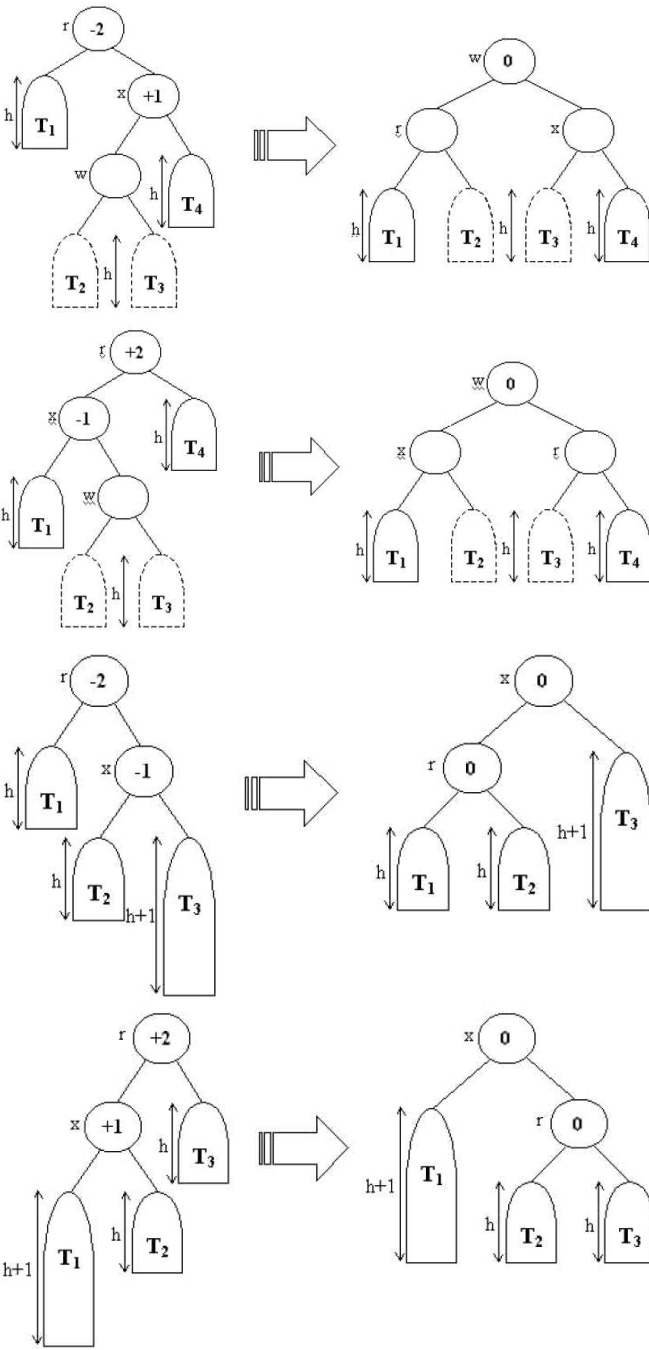
Permet d'évaluer efficacement un polynôme f . Elle est optimale pour des polynômes génériques.

Algorithme 4 HORNER($f(x), \alpha$)

$O(nW)$

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

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



Rotations RL, LR, RR et LL.

3.3 Hashing

Définition La fonction de hachage est une application $f: \mathbb{K} \rightarrow \{0, 1, \dots, m-1\}$ qui fait correspondre à chaque clé k un indice $h(k)$, appelé l'adresse de hachage. Si, pour $k, k' \in \mathbb{K}$, $k \neq k'$ et $h(k) = h(k')$ on a une collision d'adresse. k et k' sont synonymes.

La méthode de division avec reste Si m est la taille de la table de hachage alors la fonction de hachage sera :

$$h(k) := k \bmod m \quad h(k) \in \{0, 1, \dots, m-1\}$$

La méthode multiplicative La clé est multipliée par un nombre irrationnel ϕ , puis on regarde la partie fractionnaire (après la virgule) du résultat. Le nombre d'or $\frac{\sqrt{5}-1}{2}$ est un bon facteur de multiplication. $h(k) := \lfloor m(k\phi^{-1} - \lfloor k\phi^{-1} \rfloor) \rfloor$

Birthday Lemma Si $q \gtrsim 1.78\sqrt{M}$, alors la probabilité qu'une fonction aléatoire uniforme $f: \{1, 2, \dots, q\} \rightarrow M$ soit injective est inférieur à $1/2$.

4.4 L'algorithme du Knapsack 0/1

KNAPSACK(w, v, W) prend une liste de poids w , une liste de valeurs v associées et un poids maximum W comme entrées. Il renvoie une matrice C_{nW} qui indique le poids max que l'on peut avoir avec les n premiers objets en ne dépassant pas le poids max W .

Algorithme 5 KNAPSACK(w, v, W) $O(nW)$

Input: $w = (w_1, \dots, w_n), v = (v_1, \dots, v_n), W \in \mathbb{N}$

Output: Valeur V telle que $V = \max \sum_{i \in S} v_i$ pour tout $S \subseteq \{1, \dots, n\}$

avec $\sum_{i \in S} w_i \leq W$

```

1: for  $w = 0 \dots W$  do
2:    $c_{0,w} \leftarrow 0$ 
3: end for
4: for  $i = 1 \dots n$  do
5:    $c_{i,0} \leftarrow 0$ 
6:   for  $w = 1 \dots W$  do
7:     if  $w_i \leq w$  then
8:       if  $v_i + c_{i-1, w-w_i} > c_{i-1, w}$  then
9:          $c_{i,w} = v_i + c_{i-1, w-w_i}$ 
10:      else
11:         $c_{i,w} = c_{i-1, w}$ 
12:      end if
13:    else
14:       $c_{i,w} = c_{i-1, w}$ 
15:    end if
16:  end for
17: end for
18: return  $c_{nW}$ 
```

L'algorithme suivant permet de retrouver une solution optimale :

Algorithme 6 GETOPTIMALCHOICE(W, n, v_i, C)

Input: L'input de KNAPSACK et le C calculé

Output: Un choix optimal

```

1: for  $i = n, \dots, 1$  do
2:   if  $c_{i-1, W} = c_{i, W}$  then
3:     print "On ne prend pas l'objet"  $i$ 
4:   else
5:     print "On prend l'objet"  $i$ 
6:      $W \leftarrow W - v_i$ 
7:   end if
8: end for
```

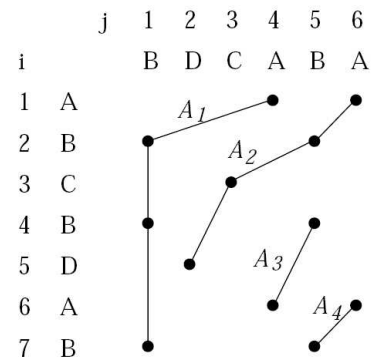
4.5 Le problème LCS

Définitions 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.

Soit $X(x_1, \dots, x_m)$ une suite. Alors $Z = (z_1, \dots, z_k)$ est une sous-suite de X s'il existe des indices $1 \leq i_1 < \dots < i_k \leq m$ tq pour tout $j \in [1, k]$ nous avons $z_j = x_{i_j}$. Le problème LCS (*Longuest Common Subsequence*) consiste à trouver la plus longue sous-suite commune à deux suites X et Y .

Dans un ensemble fini et partiellement ordonné M , la longueur p d'une chaîne maximale est égale au nombre minimal d'antichaînes d'une décomposition de M .



Algorithme 7 LCS(X, Y)

$O(mp)$

Input: Suites X et Y

Output: Une décomposition de M en antichaînes

```

1: for  $j = 1 \dots m$  do
2:    $s[j] \leftarrow n + 1$ 
3:    $A[j] \leftarrow \emptyset$ 
4: end for
5: for  $i = 1 \dots m$  do
6:    $l \leftarrow \min\{k \mid (i, k) \in M\}$ 
7:    $j \leftarrow l$ 
8:   while  $l \leq n$  do
9:     if  $l \leq s[j]$  then
10:       $A[j] \leftarrow A[j] \cup \{(i, k) \in M \mid l \leq k \leq s[j]\}$ 
11:       $temp \leftarrow s[j]$ 
12:       $s[j] \leftarrow l$ 
13:       $l \leftarrow \min\{k \mid (i, j) \in M \text{ AND } k > temp\}$ 
14:    end if
15:     $j \leftarrow j + 1$ 
16:  end while
17: end for
18: return  $A[j]$  où les antichaînes sont stockées et  $p = \max\{j \mid s[j] \leq n\}$  la longueur de la lcs( $X, Y$ )
```

4.6 Floyd-Warshall

L'algorithme de Floyd-Warshall permet de trouver pour toutes les paires de sommets du graphe $G = (G, E)$ la longueur de chemin la plus courte entre eux. Soit $V = \{1, \dots, n\}$, alors l'algorithme se base sur l'idée suivante : Fixons $k \leq n$. Pour toute paire de points i et j , trouver le plus court chemin entre i et j qui n'utilise que les sommets $1, \dots, k$. Commencer avec $k = 1$ et continuer jusqu'à ce que $k = n$.

$$C_{ij}^0 = \begin{cases} w(i, j) & \text{si } (i, j) \in E, \\ \infty & \text{sinon} \end{cases}$$

$$C_{ij}^k = \min \{C_{ij}^{k-1}, C_{ik}^{k-1} + C_{kj}^{k-1}\}$$

5 Algorithmes gloutons

Définition Un algorithme glouton est un algorithme qui maximise son profit à chaque étape.

5.1 L'algorithme Greedy Selector

GREEDYSELECTOR(s, f) prend en entrée un tableau s contenant les horaires de début de mobilisation d'une ressource par une personne i et f un tableau contenant les horaires de fin. Les tableaux sont triés selon f . Il renvoie A une listes d'indices des ressources tel que le nombre maximal de préférences soit satisfait.

Algorithme 8 GREEDYSELECTOR(s, f) $O(n \log(n))$

```

1:  $n \leftarrow \text{length}[s]$ 
2:  $A \leftarrow \{1\}$ 
3:  $j \leftarrow 1$ 
4: for  $i = 2, \dots, n$  do
5:   if  $s_i \geq f_i$  then
6:      $A \leftarrow A \cup \{i\}$ 
7:      $j \leftarrow i$ 
8:   end if
9: end for
10: return  $A$ 

```

5.2 Codes de Huffman

Définition Le codage de Huffman est une méthode de construction d'arbre de compression optimal pour un alphabet avec des fréquences données

Algorithme 9 HUFFMAN(C, f) $O(n \log(n))$

Input: C Ensemble de caractères avec f les fréquences correspondantes et $n = |C|$. (Q est une queue à priorité ordonnée par fréquences décroissantes, initialement vide).

Output: Arbre de compression Huffman T

```

1: for  $c \in C$  do
2:   Enqueue( $(c, f(c)), Q$ )
3: end for
4: for  $i = 1, \dots, n$  do
5:   Créer un nouveau noeud  $z$ 
6:    $left\_child = \text{deletemin}(Q)$ 
7:    $right\_child = \text{deletemin}(Q)$ 
8:    $f(z) = f(left\_child) + f(right\_child)$ 
9:   Les fils de  $z$  deviennent  $left\_child$  et  $right\_child$ 
10:  Enqueue( $(z, f(z)), Q$ )
11: end for

```

$$\text{Longueur moyenne} : \sum_{i=1}^n |\text{Longueur du code}| \cdot f_i$$

6 Algorithmes de tri

6.1 Algorithmes élémentaires

Algorithme 10 MERGE(S_1, S_2)

Input: Suites triées S_1 et S_2 de tailles n et m

```

1:  $i \leftarrow 1, i_1 \leftarrow 1, i_2 \leftarrow 1$ 
2: while  $i \leq n + m$  do
3:   if  $i_2 = m$  OR  $S_1[i_1] < S_2[i_2]$  then
4:      $S[i] = S_1[i_1]$ 
5:      $i_1 \leftarrow i_1 + 1$ 
6:   else
7:      $S[i] = S_2[i_2]$ 
8:      $i_2 \leftarrow i_2 + 1$ 
9:   end if
10:   $i \leftarrow i + 1$ 
11: end while

```

Algorithme 11 MERGESORT(S)

Input: Suites S de n entiers

```

1: if  $n = 1$  then
2:   return  $S$ 
3: else
4:    $m \leftarrow \lceil n/2 \rceil$ 
5:   Soit  $S_1$  la sous-suite de  $S$  composée des  $m - 1$  premiers éléments de  $S$ , et  $S_2$  celle formée des autres éléments.
6:    $S'_1 \leftarrow \text{MERGESORT}(S_1)$ 
7:    $S'_2 \leftarrow \text{MERGESORT}(S_2)$ 
8:   return MERGE( $S'_1, S'_2$ )
9: end if

```

Tout les algorithmes suivants prennent en entrée un tableau T d'objet dont chacun est muni d'une clé k . Ils renvoient une permutation de T tel que les clés soient triées dans l'ordre croissant.

Algorithme 12 SELECTIONSORT(a)

```

1: for  $i = 0, \dots, N - 2$  do
2:    $\min \leftarrow i$ 
3:   for  $j = i + 1, \dots, N - 1$  do
4:     if  $a[j].key < a[\min].key$  then
5:        $\min \leftarrow j$ 
6:     end if
7:   end for
8:   Echanger  $a[\min]$  et  $a[i]$  (si  $\min \neq i$ )
9: end for

```

SelectionSort n'effectue aucun mouvement si la liste est déjà triée. On a $N - 1$ mouvement si on a la liste $(2, 3, \dots, N, 1)$.

Algorithme 13 INSERTIONSORT(a)

```

1: for  $i = 1, \dots, N - 1$  do
2:    $j \leftarrow i - 1$ 
3:    $t \leftarrow a[i]$ 
4:   while  $a[j].key > t.key$  AND  $j \geq 0$  do
5:      $a[j+1] \leftarrow a[j]$ 
6:      $j \leftarrow j - 1$ 
7:   end while
8:    $a[j+1] \leftarrow t$ 
9: end for

```

Algorithme 14 SHELLSORT(a, h)

Input: Suite a , suite d'incrémentes $h_t > h_{t-1} > h_1 = 1$

```

1: for  $k = t, \dots, 1$  do
2:    $h \leftarrow h_k$ 
3:   for  $i = h, \dots, N - 1$  do
4:      $v \leftarrow a[i]$ 
5:      $j \leftarrow i$ 
6:     while  $j \geq h$  AND  $a[j-h].key > v.key$  do
7:        $a[j] \leftarrow a[j-h]$ 
8:        $j \leftarrow j - h$ 
9:     end while
10:     $a[j] \leftarrow v$ 
11:   end for
12: end for

```

Algorithme 15 BUBBLESORT(a)

```
1: Continue  $\leftarrow$  TRUE
2:  $r \leftarrow N - 1$ 
3: while Continue = TRUE do
4:   Continue  $\leftarrow$  FALSE
5:   for  $i = 0, \dots, r - 1$  do
6:     if  $a[i].key > a[i+1].key$  then
7:       Echanger  $a[i]$  et  $a[i+1]$ 
8:       Continue  $\leftarrow$  TRUE
9:   end if
10: end for
11:  $r \leftarrow r - 1$ 
12: end while
```

Algorithme 16 QUICKSORT(a, l, r)

Au 1er appel : r = indice du dernier élément + 1

Input: Suite a d'éléments, indices l et r

Output: Transformation de $a[l], \dots, a[r-1]$ en une suite triée

```
1: if  $r \leq l + 1$  then
2:   return Fini
3: end if
4:  $q \leftarrow l, p \leftarrow r - 1$ 
5:  $v \leftarrow a[r-1].key$ 
6: while  $p > q$  do
7:   while  $a[p].key \geq v$  AND  $p > l$  do
8:      $p \leftarrow p - 1$ 
9:   end while
10: while  $a[q].key \leq v$  AND  $q < r - 1$  do
11:    $q \leftarrow q + 1$ 
12: end while
13: if  $p > q$  then
14:   Echanger  $a[p]$  et  $a[q]$ 
15: end if
16: end while
17: Echanger  $a[q]$  et  $a[r-1]$ 
18: QuickSort( $a, l, q$ )
19: QuickSort( $a, q + 1, r$ )
```

Choix du pivot : Les 2 méthodes les plus connues sont : *aléatoire*, *3-median Strategy* (on choisit par exemple la médiane des 2 extrémités et du milieu)

6.2 HeapSort

Un heap est un tableau tel que :

$$\begin{aligned} a[i].key &\geq a[2i+1].key & \forall i \mid 2i+1 < N \\ a[i].key &\geq a[2i+2].key & \forall i \mid 2i+2 < N \end{aligned}$$

Priority Queues : Les heaps implémentent des queues à priorité 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 au heap et DELETEMIN enlève l'élément avec la clé la plus petite.

Sifting : La procédure de *sift* est une procédure efficace pour insérer un élément i dans un heap a SIFTDOWN utilise au plus $O(\log(N/i))$ comparaisons et échanges.

Algorithme 17 SIFTDOWN(a, i)

Input: Suite a d'éléments avec N clés et avec $a[i+1], \dots, a[N-1]$ vérifiant la propriété du heap.

Output: Transformation de a tel que $a[i], \dots, a[N-1]$ vérifie la propriété du heap.

```
1: Swapped  $\leftarrow$  TRUE
2: while Swapped = TRUE AND  $2i+1 < N$  do
3:   Swapped  $\leftarrow$  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:      $i \leftarrow j$ 
8:     Swapped  $\leftarrow$  TRUE
9:   end if
10: end while
```

L'algorithme *BottomUpHeapCreate* transforme une liste a en un heap :

Algorithme 18 BOTTOMUPHEAPCREATE(a)

```
1: for  $i = \lfloor (N-2)/2 \rfloor, \dots, 0$  do
2:   SIFTDOWN( $a, i$ )
3: end for
```

Algorithme 19 HEAPSORT(a)

```
1: BOTTOMUPHEAPCREATE( $a$ )
2: for  $i = N - 1, \dots, 1$  do
3:   Echanger  $a[0]$  et  $a[i]$ 
4:   Swapped  $\leftarrow$  TRUE
5:    $j \leftarrow 0$ 
6:   while Swapped = TRUE AND  $j < i$  do
7:     Swapped  $\leftarrow$  FALSE
8:     Trouver le plus grand descendant  $a[l]$  de  $a[j]$  tel que  $l < i$ .
      STOP s'il n'existe pas
9:     if  $a[l].key > a[j].key$  then
10:      Echanger  $a[j]$  et  $a[l]$ 
11:       $j \leftarrow l$ 
12:     Swapped  $\leftarrow$  TRUE
13:   end if
14: end while
15: end for
```

6.3 Temps de parcours

	$C_{min}(N)$	$C_{avg}(N)$	$C_{max}(N)$
SELECTIONSORT	$O(N^2)$	$O(N^2)$	$O(N^2)$
INSERTIONSORT	$N - 1$	$O(N^2)$	$O(N^2)$
BUBBLESORT	$N - 1$	$O(N^2)$	$O(N^2)$
QUICKSORT	$O(N \log N)$	$O(N \log N)$	$\Omega(N^2)$
HEAPSORT	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
	$M_{min}(N)$	$M_{avg}(N)$	$M_{max}(N)$
SELECTIONSORT	0	$O(N)$	$N - 1$
INSERTIONSORT	$2(N - 1)$	$O(N^2)$	$O(N^2)$
BUBBLESORT	0	$O(N^2)$	$O(N^2)$
QUICKSORT	0	$O(N \log N)$	$\Omega(N^2)$
HEAPSORT	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$

7 Algorithmes de graphes

Convention : V représente un ensemble de sommets et E un ensemble d'arêtes.

7.1 Parcourir les graphes

Savoir quels sommets sont atteignables depuis un certain sommet $s \in V$

Algorithme 20 ABSTRACTTRANVERSAL(G, s) $O(|E|)$

Input: Graphe $G = (V, E)$, sommet $s \in V$

Output: Marquer les sommets de V qui peuvent être atteints depuis s

```

1:  $Q \leftarrow \{s\}$ 
2: Marquer  $s$ .
3: while  $Q \neq \emptyset$  do
4:   Choisir  $v \in Q$ 
5:   while  $N[v] \neq \emptyset$  do
6:     Choisir  $v' \in N[v]$ 
7:      $N[v] \leftarrow N[v] \setminus \{v'\}$ 
8:     if  $v'$  n'est pas marqué then
9:        $Q \leftarrow Q \cup \{v'\}$ 
10:      Marquer  $v'$ 
11:   end if
12: end while
13:  $Q \leftarrow Q \setminus \{v\}$ 
14: end while
```

DFS et BFS. L'algorithme DFS (depth first search) est obtenu en utilisant un stack pour Q . L'algorithme BFS (breadth first search) est obtenu en utilisant une queue pour Q .

7.2 Tri topologique

Le tri topologique d'un graphe orienté acyclique $G = (V, E)$ consiste à donner un ordre aux sommets v_1, \dots, v_n de G tel que $(v_i, v_j) \in E$ implique que $i < j$.

Algorithme 21 TOPSORT(G) $O(|V| + |E|)$, mémoire : $O(|V|)$

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  $v \in V$  (e.g. via DFS)
2:  $t \leftarrow 0$ 
3: for  $i = 1, \dots, n$  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:    $v = \text{Head}(Q)$ 
10:  Dequeue( $Q$ )
11:   $t \leftarrow t + 1$ 
12:   $v.label = t$ 
13:  for  $w \in N[v]$  do
14:     $w.Indegree \leftarrow w.Indegree - 1$ 
15:    if  $w.Indegree = 0$  then
16:      Enqueue( $Q, w$ )
17:    end if
18:  end for
19: end while
```

7.3 Chemin les plus courts

Les algorithmes Moore-Bellman-Ford et Dijkstra permettent de trouver la distance la plus courte d'un point d'un graphe à tout les autres. MBF permet des arêtes négatives (contrairement à Dijkstra), la seule exigence étant la non existence de cycles de longueur négative. Les deux algorithmes ont pour entrée un graphe orienté $G = (V, E)$ avec fonction de poids $c : E \rightarrow \mathbb{R}$ et un sommet $s \in V$. Ils retournent tout deux, pour tout v appartenant à V la distance la plus courte $l(v)$ entre s et v . Soient $n = |V|$ et $m = |E|$.

Algorithme 22 DIJKSTRA(G, s) $O(n^2)$, mémoire : $O(n)$

```

1: for  $v \in V \setminus \{s\}$  do
2:    $l(v) \leftarrow \infty$ 
3: end for
4:  $l(s) \leftarrow 0, T \leftarrow \emptyset, v \leftarrow s$ 
5: while  $l(v) \neq \infty$  do
6:    $T \leftarrow T \cup \{v\}$ 
7:   for  $w \in V \setminus T$  do
8:     if  $(v, w) \in E$  then
9:        $d \leftarrow l(v) + c(v, w)$ 
10:      if  $d < l(w)$  then
11:         $l(w) = d$ 
12:      end if
13:    end if
14:  end for
15:   $v \leftarrow \arg \min_{w \in V \setminus T} l(w)$ 
16: end while
```

Algorithme 23 MBF(G, s) $O(mn)$

```

1: for  $v \in V \setminus \{s\}$  do
2:    $l(v) \leftarrow \infty$ 
3: end for
4:  $l(s) \leftarrow 0$ 
5: for  $i = 1, \dots, n - 1$  do
6:   for  $(v, w) \in E$  do
7:      $d \leftarrow l(v) + c(v, w)$ 
8:     if  $d < l(w)$  then
9:        $l(w) = d$ 
10:    end if
11:  end for
12: end for
```

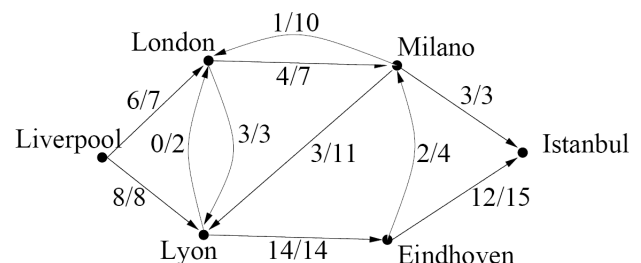
7.4 Flux de réseau

Réseau Un réseau (G, c, s, t) (*network*) est un graphe orienté $G = (V, E)$ avec une fonction de capacité $c : E \rightarrow \mathbb{R}_{\geq 0}$ et 2 sommets $s \in V$ et $t \in V$ (source et sink).

Cut Un cut (S, T) avec $S, T \in V$ est une partition de $V : S \cap T = \emptyset, S \cup T = V$, avec $s \in S$ et $t \in T$.

Flux Un flux (*flow*) est une application $f : E \rightarrow \mathbb{R}_{\geq 0}$ telle que :

- $\forall e \in E$ on a $f(e) \leq c(e)$ (contrainte de capacité)
- tout ce qui rentre dans un sommet v doit aussi en sortir



L'algorithme MAXFLOWMINCUT ($O(nm^2)$) permet de trouver le flux maximal :

- Construire le graphe des résidus.
- Trouver un chemin augmentant.
- Reporter ce chemin sur le graphe de flux
- Recommencer jusqu'à ce qu'il n'y ait plus de chemins augmentant.

On peut aussi utiliser le théorème suivant : la valeur de flux maximal est égal à la valeur du cut minimal.

7.5 Arbre couvrant minimaux

Etant donné un graphe $G = (V, E)$ avec fonction de poids $w : E \rightarrow \mathbb{R}_{>0}$. Trouver un arbre $T = (V, E')$ avec $E' \subseteq E$ et tel que $\sum_{e \in E'} w(e)$ est minimal. L'algorithme de Kruskal permet de le trouver :

- Trier E du plus petit au plus grand
- Pour tout les e_i croissants dans E : Si $E' \cup e_i$ n'est pas acyclique ajouter e_i à E

La structure Union find L'opération FIND cherche pour un sommet donné la racine de son arbre. 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'autre.

Algorithme 24 FIND(u)

Input: Sommet u d'un graphe

Output: la racine de la composante de u

```

1:  $r \leftarrow u$ 
2: while  $r.pred \neq \text{NULL}$  do
3:    $r \leftarrow r.pred$ 
4: end while
5: return  $r$ 
```

Algorithme 25 UNION(u, v)

Input: Sommet u et v d'un graphe

Output: Fusionne les ensembles contenant u et v ; retourne TRUE ssi u et v étaient dans les composantes distinctes.

```

1:  $r \leftarrow \text{FIND}(u)$ 
2:  $s \leftarrow \text{FIND}(v)$ 
3: if  $r \neq s$  then
4:   if  $s.size \leq r.size$  then
5:      $s.pred \leftarrow r$ 
6:      $r.size \leftarrow r.size + s.size$ 
7:   else
8:      $r.pred \leftarrow s$ 
9:      $s.size \leftarrow s.size + r.size$ 
10:  end if
11:  return TRUE
12: else
13:  return FALSE
14: end if
```

Algorithme de Kruskal Il permet de trouver l'arbre couvrant minimal en $O(m \log(n))$ où $n = |V|$ et $m = |E|$ si on utilise la structure UNIONFIND pour faire les tests et les unions. Le test de la ligne 5 peut être remplacée par :

"*if aucun cycle n'est créé quand on ajoute a_{min} à T then*".

Algorithme 26 MINSPANKRUSKAL(G, w)

$O(m \log(n))$

```

1: mettre toutes les arêtes dans un heap
2:  $\text{count} = 0$ ,  $T = \text{emptyset}$ 
3: while  $|T| < n - 1$  do
4:   trouver l'arête minimal  $a_{min} = (v, w)$ 
5:   if  $\text{component}(v) \neq \text{component}(w)$  then
6:     ajouter  $(v, w)$  à  $T$ 
7:      $\text{component}(v) = \text{component}(w)$ 
8:   end if
9: end while
10: return  $T$ 
```

Algorithme de Prim Il permet aussi de trouver l'arbre couvrant minimal en $O(m \log(n))$ mais d'une manière gloutonne. Pendant l'exécution, nous indexons chaque sommet avec *fringe* s'il existe une arête qui le relie à l'arbre déjà créé ou avec *unseen* sinon.

Algorithme 27 MINSPANPRIM(G, w)

$O(m \log(n))$

```

1: choisir un sommet arbitraire pour commencer
2: while il y a des sommets fringe do
3:   choisir une arête  $e = (v, f)$  de poids minimal entre l'arbre et fringe ( $v$  est un sommet dans l'arbre,  $f$  est dans fringe)
4:   ajouter  $e$  et  $f$  à l'arbre
5:   enlever  $f$  de fringe
6:   ajouter tous les voisins unseen de  $f$  dans fringe
7: end while
```

7.6 Détection de cycles

L'algorithme suivant détermine si un graphe $G = (V, E)$ muni d'une fonction de poids $w : E \Rightarrow \mathbb{R}$ possède un cycle négatif.

Algorithme 28 NEGATIVECYCLE(G, w)

$O(|V| \cdot |E|)$

Input: Graphe $G = (V, E)$ avec fonction de poids $w : E \rightarrow \mathbb{R}$

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  $l(w)$  entre  $s$  et  $w$ .
3: for  $(u, v) \in E$  do
4:   if  $l(u) + w(u, v) < l(v)$  then
5:     return TRUE
6:   end if
7: end for
8: return FALSE
```
