Algorithmique avancée et preuve de programme

Algorithmique

Sommaire

Chapitre I

Chapitre I
Algorithmique avancée

1/ Rappel

Définition: Un algorithme est ensemble d'opérations permettant de faire un calcul ou de résoudre un problème.

Problématique: Deux problèmes majeurs que l'algorithme doit traiter sont:

- Trouver une **méthode** de résolution d'un problème.
- La méthode trouvée doit être **efficace**.

Exemple: On veut calculer x^n

Tel que: x est réel et n est naturel.

Algorithme simple:

```
y = x
Pour i ← 2 à n faire
    y = y.x
retourner y
```

Le nombre maximal d'opérations élémentaires de cet algorithme est :

$$T = (n\text{-}1)$$

Exemple (suite):

Algorithme binaire:

– Écriture en binaire du nombre n : $n = \sum_{0}^{p} a_i \cdot 2^i$

– Le bit **j** est le dernier bit de la représentation binaire de n et soit $\mathbf{y_j}$ le dernier résultat obtenu. Initialement, **j = p** et $y_p = x = x^{a_p}$

– Deux cas sont possibles pour $\mathbf{a_{j-1}}$:

$$\begin{cases} a_{j-1} = 1 \Rightarrow y_{j-1} = y_j^{\,2} \cdot x \\ a_{j-1} = 0 \Rightarrow y_{j-1} = y_j^{\,2} \end{cases}$$

Dans tous les cas nous avons : $y_{j-1} = y_j^{\,2} \cdot \left(x^{a_{j-1}}\right)$

Le nombre maximal d'opérations élémentaires de cet algorithme est : T = (2.p)

Exemple (suite):

Pour n=23 par exemple nous aurons:

L'algorithme simple utilise T = (n-1) = 22 opérations élémentaire.

Exemple (suite):

Pour n=23 par exemple nous aurons:

n=10111 avec p=4 et

(a_0=1, a_1=1, a_2=1, a_3=1, a_4=1)

Donc y_4=x

et comme :

$$y_{j-1} = y_j^{\,2} \cdot x^{a_{j-1}}$$

L'algorithme binaire utilise

T = 7 opérations élémentaires

$$\begin{aligned}
&y_0 = y_1^{\,2} \cdot x \\
\Rightarrow\ &y_0 = \left(y_2^{\,2} \cdot x\right)^2 \cdot x \\
\Rightarrow\ &y_0 = \left(\left(y_3^{\,2} \cdot x\right)^2 \cdot x\right)^2 \cdot x \\
\Rightarrow\ &y_0 = \left(\left(\left(y_4^{\,2}\right)^2 \cdot x\right)^2 \cdot x\right)^2 \cdot x \\
\Rightarrow\ &y_0 = \left(\left(\left(x^2\right)^2 \cdot x\right)^2 \cdot x\right)^2 \cdot x \\
\Rightarrow\ &y_0 = x^{23}
\end{aligned}$$

Chapitre I
Algorithmique avancée

Comment calculer le coût d'un algorithme (ou la complexité)?

Définition (Complexité): La complexité d'un algorithme est la **mesure du nombre d'opérations fondamentales** qu'il effectue sur un jeu de données. La complexité est exprimée comme une fonction de la taille du jeu de données.

Notons $\mathbf{D}_n$ l'ensemble des données de taille ***n*** et ***C(d)*** le coût de l'algorithme sur la donnée **d**.

Chapitre I
Algorithmique avancée

- **Complexité minimale** :

$$T_{\min}(n) = \min C(d),\ \forall d \in D_n$$

C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à ***n***. C'est une borne inférieure de la complexité de l'algorithme sur un jeu de données de taille ***n***.

Chapitre I
Algorithmique avancée

- **Complexité maximale** :

$$T_{\max}(n) = \max C(d),\ \forall d \in D_n$$

C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à ***n***.

Chapitre I
Algorithmique avancée

- **Complexité moyenne** :

$$T_{\text{moy}}(n) = \Sigma C(d) / |D_n|,\ \forall d \in D_n$$

C'est la moyenne des complexités de l'algorithme sur des jeux de données de taille ***n***.

Exemple

Dans l'exemple précédent, calculons le coût de l'algorithme binaire de x^n

Note : Les nombres avec **p** chiffres dans la représentation binaire sont dans l'intervalle $[2^{p-1};2^p-1]$.

Pour p=4 ➔ [1000;1111]

$1000 = 2^{4-1}$ et $1111 = 10000-1=2^4-1$

Exemple (suite)

Le nombre de chiffres dans l'écriture binaire de n : $1+[\log_2 n]$.

tel que: $[x]$ est 1 + la partie entière de x exp.

Pour n=23, nous aurons $1+[\log_2 23]=1+[4.532]=5$ chiffres en binaire)

Notons $v(n)$ le nombre de «1» dans l'écriture binaire de n.

Exemple (suite)

Le coût ou le nombre d'opérations effectuées est:

- $(1+[\log_2 n])-1$ opérations de carré;
- $v(n)$ - 1 multiplications par x.

Donc $T(n) = [\log_2 n] + v(n) - 1$

Sachant que $1 \le v(n) \le [\log_2 n] + 1$

$\Rightarrow [\log_2 n] \le T(n) \le 2.[\log_2 n]$

Exemple (suite)

Donc pour n=23 nous aurons:

$[\log_2 23] \le T(n) \le 2.[\log_2 23]$

$\Rightarrow 5 \le T(n) \le 10$

Premier algorithme de tri (tri par insertion)

Problématique:

Soit une séquence de ***n*** nombres $a_1, \ldots, a_n$

Résultat souhaité: Permuter les nombres jusqu'à ce qu'on aura la séquence:

$b_1 \leq b_2 \leq \ldots \leq b_n$

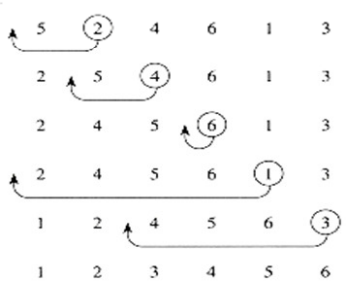
Tri par insertion (suite):

Algorithme:

```
Pour j ← 2 à n faire
   clé ← A[j]
   i ← j-1
   tant que i > 0 et A[i] > clé faire
   debut
     A[i+1] ← A[i]
     i ← i-1
   fin Tq
   A[i+1] ← clé
```


Tri par insertion (suite):

Exemple:



Tri par insertion (suite):

Calcul de la complexité:

Attribuons un coût en temps c_i à chaque instruction, et comptons le nombre d'exécutions de chacune des instructions.

Pour chaque valeur de $j \in [2..n]$, notons t_j le nombre d'exécutions de la boucle `tant que` pour cette valeur de j.

Tri par insertion (suite):

Calcul de la complexité (suite):

Algorithme	Coût	Nombre d'executions
`Pour j ← 2 à n faire`	c_1	n
`  clé ← A[j]`	c_2	n-1
`  i ← j-1`	c_3	n-1
`  tant que i>0 et A[i]>clé faire`	c_4	$\sum_{j=2}^{j=n} t_j$
`  debut`		
`    A[i+1] ← A[i]`	c_5	$\sum_{j=2}^{j=n} (t_j - 1)$
`    i ← i-1`	c_6	$\sum_{j=2}^{j=n} (t_j - 1)$
`  fin Tq`		
`  A[i+1] ← clé`	c_7	n-1

Tri par insertion (suite):

Calcul de la complexité (suite):

Le temps d'exécution total de l'algorithme est:

$$T(n) = \begin{cases} c_1 \cdot n + \\ c_2 \cdot (n-1) + \\ c_3 \cdot (n-1) + \\ c_4 \cdot \sum_{j=2}^{j=n} t_j + \\ c_5 \cdot \sum_{j=2}^{j=n} (t_j - 1) + \\ c_6 \cdot \sum_{j=2}^{j=n} (t_j - 1) + \\ c_7 \cdot (n-1) \end{cases}$$

Tri par insertion (suite):

Calcul de la complexité (suite):

La complexité maximale de notre algorithme est dans le cas où le tableau A est déjà **trié dans l'ordre inverse**.

Remarquez que nous avons $t_j = j$ pour tout j

Et sachant que: $\sum_{j=1}^{j=n} j = \frac{n \cdot (n+1)}{2}$

Donc: $\sum_{j=2}^{j=n} j = \frac{n \cdot (n+1)}{2} - 1$

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

Tri par insertion (suite):

Calcul de la complexité (suite):

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{j=n} j + c_5 \cdot \sum_{j=2}^{j=n} (j-1) + c_6 \cdot \sum_{j=2}^{j=n} (j-1) + c_7 \cdot (n-1)$$

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \left(\frac{n \cdot (n+1)}{2} - 1 \right) + c_5 \cdot \frac{n \cdot (n-1)}{2} + c_6 \cdot \frac{n \cdot (n-1)}{2} + c_7 \cdot (n-1)$$

$$T(n) = \left(\frac{c_4 + c_5 + c_6}{2} \right) \cdot n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4 - c_5 - c_6}{2} + c_7 \right) \cdot n - (c_2 + c_3 + c_4 + c_7)$$

$$T(n) = a \cdot n^2 + b \cdot n + c$$

Comme a, b et c sont des constantes, la complexité est de l'ordre de $\mathbf{O(n^2)}$.

Classes de la complexité:

Il existe 7 grande classes de complexité:

- Algorithme constant: O(1) cas où toutes les **instructions** sont exécutées **une seule fois** quelque soit la taille des données (pas de boucle);
- Algorithme sub-linéaire: O(log ***n***) cas où la **taille** du problème est **divisée par une entité** constante a chaque itération (ex : recherche dichotomique);

Classes de la complexité (suite):

- Algorithme linéaire: O(***n***) cas d'une boucle de 1 a ***n*** et chaque **itération** de la boucle **effectue** un travail de **durée constante et indépendante** de ***n***.
- Algorithme quasi-linéaire: O(***n***.log ***n***) cas d'une boucle où a chaque **itération** la taille du problème est **divisée** par une constante.

Classes de la complexité (suite):

- Algorithme polynomial: O(n^k) cas de **boucles imbriquées** allant de 1 a ***n***. Ces algorithmes sont considérés comme **lents** lorsque ***k*>3**.
- Algorithme exponentiel: O(k^n) cas de problèmes **complexe**. Ces algorithmes sont dit **impraticables**. (ex. Les tours de Hanoi)
- Algorithme factoriel: O(***n***!) cas d'une fonction **récursive** qui contient une **boucle** de 1 à ***n*** et elle s'appelle elle-même avec le paramètre (***n***-1). Trop **lent** comme algorithme.

2/ La récursivité

Définition: Un algorithme récursif est un algorithme qui est **défini** en **fonction** de **lui même**.

Il existe plusieurs types de récursivité:

2.1/ La récursivité simple

Exp. Calculez la puissance x^n récursivement.

```
PUISSANCE(x, n)
   Si n = 0 alors retourner 1
   sinon retourner x*PUISSANCE(x, n-1)
```

2.2/ La récursivité multiple

Plusieurs appels récursives.

Exp. Calculez les combinaisons C_n^p

tel que:

$$C_n^p = \begin{cases} 1 & \text{si p = 0 ou p = n} \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon} \end{cases}$$

```
COMBINAISON(n,p)
   si p=0 ou p=n alors retourner 1
   sinon retourner COMBINAISON(n-1,p)+COMBINAISON (n-1,p-1)
```


2.3/ La récursivité mutuelle

Deux fonctions qui s'appelles entre elles.

Exp. Définissez la parité d'un nombre ***n***.

```
PAIR(n)
   Si n=0 alors retourner vrai
   sinon retourner IMPAIR(n-1)
IMPAIR (n)
   Si n=0 alors retourner faux
   sinon retourner PAIR(n-1)
```


2.4/ La récursivité imbriquée

La fonction s'appelle elle-même avec comme paramêtre un appel à elle-même.

Exp. La fonction d'Ackermann.

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m,n-1)) & \text{sinon} \end{cases}$$

```
ACKERMANN(m,n)
   si m=0 alors retourner n+1
   sinon si n=0 alors retourner ACKERMANN(m-1,1)
   sinon retourner ACKERMANN(m-1,ACKERMANN(m,n-1))
```


2.5/ Problème de la récursivité

Le problème majeur de la récursivité est la terminaison. On peut tomber facilement sur une boucle infinie. Il faut surveiller le critère d'arrêt.

Le problème de la terminaison est indécidable. C'est-à-dire qu'on ne peut pas montrer qu'un algorithme récursif va se terminer un jour.

2.6/ Importance de l'ordre dans les appels récursifs

exp.

```
F(n)
   si n<>0 alors
      afficher n
      F(n-1)
```

pour n=2, nous aurons l'affichage suivant: 2, 1

```
F(n)
   si n<>0 alors
      F(n-1)
      afficher n
```

pour n=2, nous aurons l'affichage suivant: 1, 2

3/ Algorithmes de tri

3.1/ Tri par fusion

On divise la séquence de ***n*** nombres en deux sous-séquences de taille ***n***/2.

Puis on trie récursivement les deux sous-séquences.

Et enfin, on fusionne les deux sous-séquences triées pour produire la séquence complète triée.

3.1/ Tri par fusion (suite)

```
FUSIONNER(A, d, m, f)
        i←d; j←m+1; k←1;
        C tableau de taille (f-d+1)
        tant que i<=m et j<=f faire     // boucle de fusion de A[d..m] avec A[m+1..f]
                si A[i]<A[j] alors C[k]←A[i]
                                   i←i+1
                sinon    C[k]←A[j]
                         j←j+1
                k←k+1
        tant que i<=m faire       C[k]←A[i]
                                  i←i+1
                                  k←k+1
        tant que j<=f faire       C[k]←A[j]
                                  j←j+1
                                  k←k+1
        pour k←1 à f-d+1 faire    //on recopie le résultat dans le tableau original
                A[d+k-1]←C[k]
```


3.1/ Tri par fusion (suite)

```
TRI_FUSION(A, d, f)
      si d<f alors  m←[(d+f)/2]
                    TRI_FUSION(A, d, m)
                    TRI_FUSION(A, m+1, f)
                    FUSIONNER(A, d, m, f)
```

3.1/ Tri par fusion (suite)

Calcul de la complexité:

La 1ère boucle « tant que » s'exécute au plus (f-d) fois, d'où un coût total en O(f-d);

La 2ème boucles « tant que » a une complexité au pire de m-d+1

La 3ème boucle « tant que » a une complexité au pire de f-m, ces deux complexités (des 2 boucles ensemble) étant en O(f-d);

La 4ème boucle coûte O(f-d+1).

La somme des 4 boucle: (f-d) + (f-d) + (f-d+1) = (3(f-d)+1)

Donc, l'algorithme de fusion a une complexité en **O(f-d)**.

3.2/ Tri par tas (Heap sort)

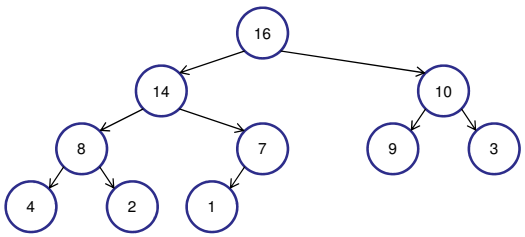
Définition

Un **tas** est un arbre binaire dont tous les niveaux sont complets sauf le dernier qui est rempli de la gauche vers la droite. Dans un tas, un père est toujours plus grand que ses deux fils.

3.2/ Tri par tas (suite)

Exp. d'un tas:

16	14	10	8	7	9	3	4	2	1
1	2	3	4	5	6	7	8	9	10



3.2/ Tri par tas (suite)

Un **tas** est représenté par un **tableau.** La racine est stocké dans la **1ère** case. Les éléments sont rangés de gauche à droite dans un niveau tel que:

$$A[PÈRE(i)] \geq A[i]$$

Les fonctions d'accès aux éléments du tableau sont:

```
PÈRE(i)
   renvoyer [i/2]
FILS_GAUCHE(i)
   renvoyer 2*i
FILS_DROIT(i)
   renvoyer 2*i+1
```

Chapitre I
Algorithmique avancée

3.2/ Tri par tas (suite)

Le processus de tri **construit** le tas initial, puis on **retire** la racine de l'arbre et on la met dans la **dernière** case du tableau, puis on **tamise** à première case (racine) vers le bas pour trouver sa place, puis on **réitère** jusqu'à ce qu'il ne reste dans l'arbre qu'un seul nœud.

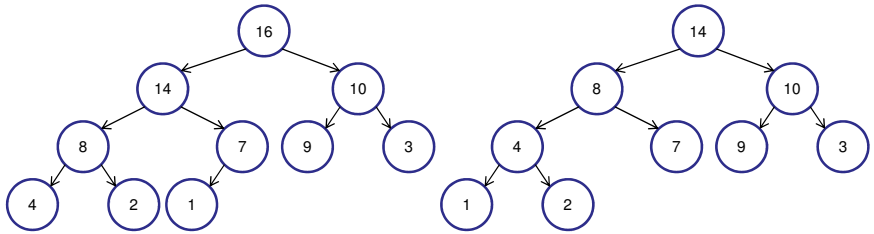
Chapitre I
Algorithmique avancée

3.2/ Tri par tas (suite)

Exp:

16	14	10	8	7	9	3	4	2	1
1	2	3	4	5	6	7	8	9	10

14	8	10	4	7	9	3	1	2	16
1	2	3	4	5	6	7	8	9	10



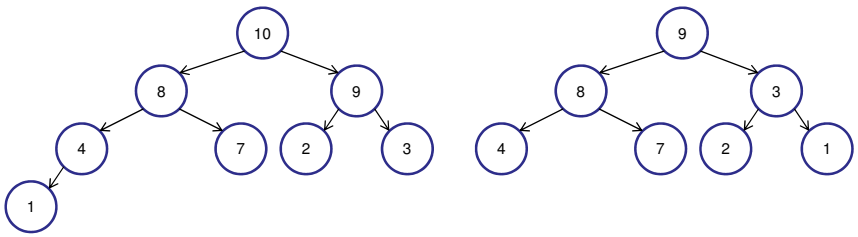
Chapitre I
Algorithmique avancée

3.2/ Tri par tas (suite)

Exp:

10	8	9	4	7	2	3	1	14	16
1	2	3	4	5	6	7	8	9	10

9	8	3	4	7	2	1	10	14	16
1	2	3	4	5	6	7	8	9	10



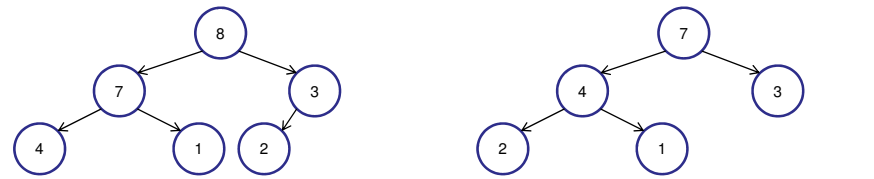
Chapitre I
Algorithmique avancée

3.2/ Tri par tas (suite)

Exp:

8	7	3	4	1	2	9	10	14	16
1	2	3	4	5	6	7	8	9	10

7	4	3	2	1	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

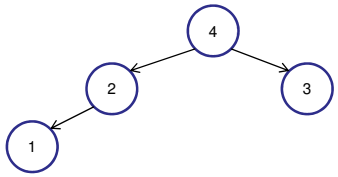


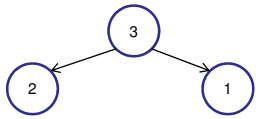
3.2/ Tri par tas (suite)

Exp:

4	2	3	1	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

3	2	1	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

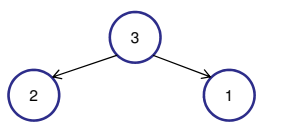


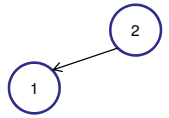


3.2/ Tri par tas (suite)

3	2	1	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

2	1	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10





1	2	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

3.2/ Tri par tas (suite)

Algorithme:

```
TAMISSER(A, idx, n)//{descend A[idx] à sa place
    k ← idx
    j ← FILS_GAUCHE(k)
    tant que j<=n
           si j<n et A[j]<A[j+1]
                  j ← FILS_DROIT(k)
           si A[k]<A[j]
                  échanger A[k] et A[j]
                  k ← j
                  j ← FILS_GAUCHE(k)
           sinon
                  j ← n+1 //pour sortir de "tant que"
```


3.2/ Tri par tas (suite)

Algorithme (suite):

```
TRI_PAR_TAS(A, n)
    pour i ← PÈRE(n) à 1        //Cette boucle organise le tas
           TAMISSER(A, i, n)
    pour i ← n à 2 //Enlève le max du tas et le met à la fin du tableau
           échanger A[i] et A[1]
           TAMISSER(A, 1, i-1)
```

3.2/ Tri par tas (suite)

Calcul de la complexité en nombre d'échanges:

La fonction « TAMISSER » fait au max $[\log_2 n]$ échanges.

La 1ère boucle de la fonction TRI_PAR_TAS() appel TAMISSER() au max $[n/2]$ fois, et la 2ème boucle fait $(n-1)$ échanges et appel TAMISSER() $(n-1)$ fois.

Donc: la complexité en nombre d'échanges est de l'ordre $\mathbf{O(n.\log_2 n) = ([n/2]+n-1).[\log_2 n]+(n-1)}$

3.3/ Tri rapide (Quick sort)

L'algorithme de tri rapide est décomposé en 3 phases:

- Le tableau A[d..f] est partitionné en 2 petits tableaux B[d..q] et C[q+1..f] tel que les éléments de B sont inférieurs ou égaux aux éléments de C;
- Les tableaux B et C sont triés d'une manière récursive;
- Combiner les deux tableaux B et C.

3.3/ Tri rapide (suite)

Algorithme:

```
TRI-RAPIDE(A, d, f)
    si d < f alors     q ← PARTITIONNEMENT(A, d, f)
                       TRI-RAPIDE(A, d, q)
                       TRI-RAPIDE(A, q+1, f)
```


3.3/ Tri rapide (suite)

Algorithme:

```
PARTITIONNEMENT(A, d, f)
    x ← A[d]
    i ← d-1
    j ← f+1
    tant que VRAI faire
      répéter j ← j-1 jusqu'à A[j] ≤ x
      répéter i ← i+1 jusqu'à A[i] ≥ x
      si i < j alors échanger A[i] ↔ A[ j]
      sinon retourner j
```

3.3/ Tri rapide (suite)

Exp:

4	3	6	2	1	5	7

après l'exécution on arrive à:

1	2	3	4	5	6	7

3.3/ Tri rapide (suite)

La complexité en nombre de permutations:

La cas maximal est quand le tableau est trié dans l'ordre inverse.

O(n) = [n/2]

4/ Algorithmes gloutons

Définition:

Un **algorithme glouton** fait un choix **optimal localement**, dans le but que ce choix mènera à la solution optimale globalement.

Les algorithmes gloutons **n'aboutissent** pas toujours à des **solutions optimales**.

Exp: Location d'une voiture.

Des clients déposent des demandes. Le but est d'affecter la voiture d'une manière à satisfaire le maximum de clients (petites périodes) et de ne pas laisser la voiture libre.

Soit l'ensemble de demandes $\boldsymbol{A}$ où chaque élément $\boldsymbol{a_i}$ possède une date début de location $\boldsymbol{d(a_i)}$ et date de fin $\boldsymbol{f(a_i)}$.

Exp: Location d'une voiture (suite)

La contrainte d'acceptation de demandes est:

$\forall a_i \in A, \forall a_j \in A, d(a_i) \leq d(a_j) \Rightarrow f(a_i) \leq d(a_j)$

Algorithme

```
LOCATION_VOITURE(A)
     Tri des éléments de A par date de fin croissante.
     On obtient donc une suite a1;a2; …;an telle que f(a1)≤f(a2)≤...≤f(an).
     F[1] ← A[1]
     j ← 1
     pour i ← 1 à n faire
         si f(F[j]) ≤ d(A[i]) alors        j ← j+1
                                           F[j] ← A[i]
     retourner F
```


Exp: Location d'une voiture (suite)

Cet algorithme est **glouton** car à chaque étape il prend la location « **la moins coûteuse** » : celle qui finit le plus tôt.

4.1/ Éléments de la stratégie gloutonne

Un algorithme glouton effectue plusieurs choix avant de déterminer une solution.

Cette stratégie ne produit pas toujours une solution optimale.

On peut citer 2 grands éléments de cette stratégie gloutonne:

4.1.1/ Propriétés du choix glouton

On faisant un choix localement optimal (ou choix glouton) on peut arriver à une solution globalement optimale.

4.1.2/ Sous-structure optimale

La structure d'un problème est **divisée** en **sous-structure optimale** de sous-problèmes pour pouvoir donner une solution optimale qui contient la solution optimale des sous-problèmes.

Les algorithmes gloutons sont basés sur la **théorie des matroïdes**.

Le problème est **modélisé** sous forme d'un matroïde est résolu avec un algorithme **glouton**.

4.2/ Matroïdes

Un matroïde est un couple M=(E, I) tel que:

1. $E \neq \emptyset$ un ensemble fini non vide
2. I est un ensemble de sous-ensembles indépendants de E, tel que: si $H \in I$ et $F \subset H$ alors $F \in I$. Remarque: $\emptyset \in I$
3. Si $F \in I$ et $H \in I$ et $|F| < |H|$, alors il existe $x \in H\text{-}F$ tel que $F \cup \{x\} \in I$

Exp:

Un sac peut contenir 2 éléments d'un ensemble de 3 éléments. Les solutions possibles sont les éléments de l'ensemble I du matroïde suivant:

Soit le matroïde M=(E, I) tel que:

E={a,b,c}, I={∅, {a}, {b}, {c}, {a,b}, {a,c}, {b,c}}

Théorème:

Tous les sous-ensembles indépendants maximaux d'un matroïde ont la même taille.

Exp:

Les solutions maximales du derniers exemple sont les éléments de I suivant: {a,b}, {a,c}, {b,c} . Il sont de même taille comme le prouve le théorème.

4.3/ Matroïdes pondérés

Un matroïde M=(E, I) est pondéré si chaque élément de E possède un poids suivant une fonction de pondération w: E $\rightarrow \Re$

Donc le poids d'un élément F de I est calculé comme suit:

$$w(F) = \sum_{x \in F} w(x)$$

Exp:

Un sac peut contenir un poids 6.5kg d'un ensemble de 3 éléments pondérés comme suit: (a,b,c) = (2,3,4)

Les solutions possibles sont les éléments de l'ensemble I du matroïde suivant:

Soit le matroïde M=(E, I) tel que:

E={a,b,c}, I={∅, {a}, {b}, {c}, {a,b}, {a,c}, {b,c}}

La solution optimale contient **l'élément optimal** {a,c} de poids 6kg

4.4/ Algorithme glouton sur matroïde pondéré

L'algorithme prend en entrée le matroïde et la fonction de pondération et retourne un sous-ensemble optimal F:

```
ALGO_GLOUTON(M(E,I),w)
    F ← ∅
    trier E d'une manière décroissante de poids
    pour tous x∈E faire
        si F∪{x}∈I alors F←F∪{x}
    retourner F
```

La complexité: O($n\log_2 n + n f(n)$) tel que f(n) le coût du teste de l'indépendance et $n\log_2 n$ est le coût du trie par tas.

Cet algorithme est glouton parce que:

- **Le choix glouton**: La **1ère partie** avait le choix entre utiliser E comme il est, le trier d'une manière croissante, le trier d'une manière décroissante… et le **choix local optimal** semble celui du **tri décroissante**. La 2ème partie dépend de la première.
- **La sous-structure optimale**: à chaque itération, l'algo. utilise E' à la place de E tel que **E'={y∈ E: {x,y}∈ I}**

5/ Graphes et arbres

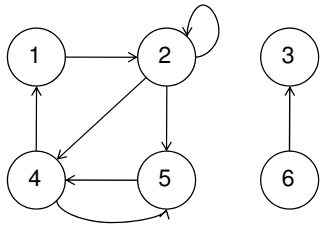
5.1/ Les graphes

Il existe deux types de graphes: orienté et non orienté.

➢ Un **graphe orienté** est un couple **G(S,A)** tel que **A** est **l'ensemble des sommets** et **S** est **l'ensemble des arcs.**

Exp: Soit G(S,A) un graphe orienté avec S={1,2,3,4,5,6}, et A={(1,2),(2,2),(2,4),(2,5),(4,1),(4,5),(5,4),(6,3)}

5.1/ Les graphes (suite)



• Dans un graphe **orienté**, le **degré sortant** d'un sommet est le **nombre d'arcs** qui en **partent**, le **degré rentrant** est le **nombre d'arcs** qui y **arrivent**.

Le degré d'un sommet = le degré entrant + le degré sortant.

5.1/ Les graphes (suite)

• Dans un graphe **orienté** G = (S,A), un **chemin de longueur *k*** d'un sommet ***a*** à un sommet ***b*** est une séquence $(a,u_1,\ldots,b)$ de sommets telle que:

$\forall i \in \{1,..,\boldsymbol{k}\}$, $(u_{i-1},u_i) \in A$.

• Un **chemin** est dit **élémentaire** si ces **sommets** sont tous **distincts**.

Exp:

Le chemin (1,2,4,5) est élémentaire de longueur 3.

Le chemin (2,5,4,5) n'est pas élémentaire.

5.1/ Les graphes (suite)

• Dans un graphe **orienté**, un **chemin** $(u_0,u_1,\ldots,u_k)$ forme un **circuit** si $\boldsymbol{u_0=u_k}$ et si le chemin contient au moins un arc.

• Un **circuit** est **élémentaire** si ses sommets sont **distincts**.

• Une **boucle** est un **circuit** de longueur **1**.

• Un graphe **orienté** est **fortement connexe** si chaque sommet est accessible à partir de tous les autres sommets.

• Les **composantes fortement connexes** d'un graphe orienté sont les **sous-graphes fortement connexes** $G_i(S_i,A_i)$ tel que $S_i \subset S$ et $A_i \subset A$.

Exp: Le graphe de l'exemple précédent contient trois composantes fortement connexes: {1,2,4,5}, {3} et {6}.

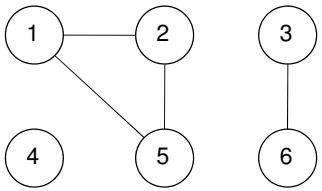
5.1/ Les graphes (suite)

➢ Un **graphe non orienté** est un couple **G(S,A)** tel que **A** est **l'ensemble des sommets** et **S** est **l'ensemble des arêtes**, avec l'arête (u,v)= l'arête(v,u).

• Dans un graphe **non orienté** les **boucles** sont **interdites**, et chaque arête (u,v) est formée par 2 sommets distincts u≠v.

Exp: Soit G(S,A) un graphe non orienté avec S={1,2,3,4,5,6}, et A={(1,2), (2,5),(5,1),(6,3)}

5.1/ Les graphes (suite)



• Le **degré** d'un sommet dans un graphe **non orienté** est le **nombre d'arêtes** qui lui sont attachées.

• Si un sommet est de **degré 0**, comme le sommet 4, il est donc **isolé**.

5.1/ Les graphes (suite)

• Dans un graphe **non orienté** G = (S,A), une **chaîne** d'un sommet ***a*** à un sommet ***b*** est une séquence $(a,u_1,\ldots,b)$ de sommets telle que:

$\forall i \in \{1,..,\boldsymbol{k}\}$, $(u_{i-1},u_i) \in A$.

• Dans un graphe **non orienté**, une **chaîne** $(u_0,u_1,\ldots,u_k)$ forme un **cycle** si $\boldsymbol{u_0=u_k}$ et si sa longueur≥**3**.

• Un **cycle** est **élémentaire** si ses sommets sont **distincts**.

• Un graphe **sans cycle** est dit **acyclique**.

5.1/ Les graphes (suite)

• Un graphe **non orienté** est **connexe** si chaque **paire de sommets** est **reliée** par une **chaîne**.

• Les **composantes connexes** d'un graphe G(S,A) non connexe sont les **sous graphes connexes** de $G_i(S_i,A_i)$ tel que $S_i \subset S$, $A_i \subset A$.

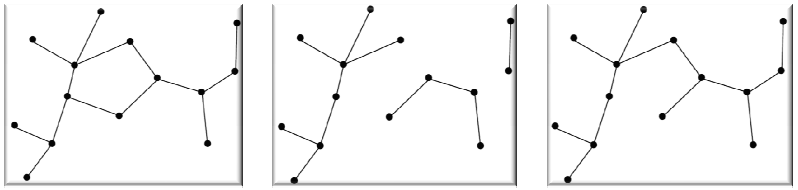
Exp:

Le graphe de l'exemple précédent contient trois composantes connexes: {1,2,5}, {3,6} et {4}.

5.2/ Les arbres

• Une **forêt** est un **graphe non orienté acyclique** et un **arbre** est un graphe **non orienté connexe acyclique**.

Exp:



Ni arbre ni foret | Foret | Arbre

5.2/ Les arbres (suite)

• Un **arbre enraciné** est un arbre contenant un sommet particulier choisi pour être une **racine**. L'arbre sera **parcouru** à partir de cette racine.

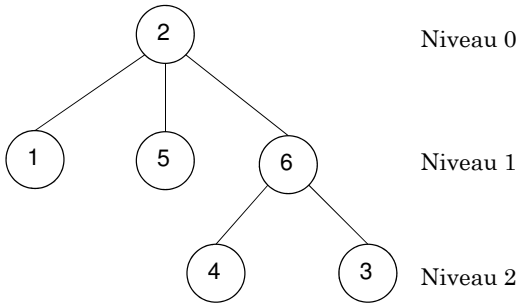
• Dans un arbre de racine ***r***, un nœud ***y*** sur le chemin de ***r*** à ***x*** est appelé **ancêtre** de ***x***. Si l'arbre contient l'arête (***x***,***y***) alors ***x*** et **fils** de ***y***, et ***y*** est **père** de ***x***.

• La **racine** est le seule nœud qui n'a **pas de père**.

• Le **degré** du nœud ***x*** est le **nombre de fils de *x*** (**+ 1** s'il n'est pas racine).

5.2/ Les arbres (suite)

Exp:



• La **hauteur** de cet arbre est 2 (le dernier niveau).

5.2/ Les arbres (suite)

• Un **arbre ordonné** est un arbre enraciné dans lequel les fils de chaque nœud sont ordonnés entre eux (exp. de gauche à droite ou de droite à gauche).

• Un arbre ***k*-aire** est une généralisation de la notion d'arbre où chaque nœud est de degré au plus ***k*+1**. (exp: arbre 2-aire=binaire).

• Un arbre ***k*-aire complet** est un arbre ***k*-aire** où chaque nœud est de degré ***k***.

5.3/ Les parcours

5.3.1/ Parcours des graphes

5.3.1.1/ Parcours en profondeur

Dans un parcours en **profondeur**, on descend le **plus profondément** possible puis on **remonte** pour explorer les autres branches en commençant par la branche **la plus basse** parmi celles non encore parcourues.

Les graphes peuvent contenir des **cycles** où il faut **éviter** de les parcourir **indéfiniment**. Pour éviter cela, les sommets sont **initialement** colorés par du **blanc**, puis dès que le sommet sera rencontré pour la **première fois** il sera coloré par du **gris**, et finalement, lorsque tous ses **successeurs** sont **parcourus** il sera coloré en **noir**.

5.3.1.1/ Parcours en profondeur (suite)

Algorithme:

```
PARCOURS_PROFONDEUR(G)
     pour chaque sommet u de G faire couleur[u] ← BLANC
     pour chaque sommet u de G faire
         si couleur[u] = BLANC alors VISITER(G, u, couleur)

VISITER(G, s, couleur)
     couleur[s] ← GRIS
     pour chaque voisin v de s faire
         si couleur[v] = BLANC alors VISITER(G, v, couleur)
     couleur[s] ← NOIR
```


5.3.1.2/ Parcours en largeur

Tous les noeuds à un niveau ***i*** doivent être visités avant les noeuds du niveau ***i*+1**.

Pour cela, nous avons besoin de garder l'ensemble des branches qu'il reste à visiter. On utilise pour ça une file **F**.

5.3.1.2/ Parcours en largeur (suite)

Algorithme:

```
PARCOURS_LARGEUR(G, s)
   couleur[s] ← GRIS
   pour chaque sommet u de G, u ≠ s faire couleur[u] ← BLANC
   F ← {s}
   tant que F ≠ ∅ faire
       u ← DEFILER(F)
       pour chaque voisin v de u faire
         si couleur[v] = BLANC alors
             couleur[v] ← GRIS
             INSERTION(F,v)
       couleur[u] ← NOIR
```

5.3.2/ Parcours des arbres

5.3.2.1/ Parcours en profondeur

On descend au niveau le **plus bas** dans l'arbre jusqu'à une feuille, puis on **remonte** pour **explorer** les autres **branches** qui n'ont pas encore été parcourus en commençant par la branche la **plus basse**.

5.3.2.1/ Parcours en profondeur (suite)

Algorithme:

```
PARCOURS_PROFONDEUR(A)
    si taille(A) > 1 faire
      pour tous les fils u de racine(A) faire dans l'ordre
        PARCOURS_PROFONDEUR(u)
```


5.3.2.2/ Parcours en largeur

Comme dans les graphes, les nœuds du niveau ***i*** seront visités avant ceux du niveau ***i*+1**.

Algorithme:

```
PARCOURS_LARGEUR(A)
    F ← {racine(A)}
    tant que F ≠ ∅ faire
      u ← DEFILER(F)
      pour tous les fils v de u faire dans l'ordre
        INSERTION(F,v)
```


Bibliographie

1. Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction à l'algorithmique*. Dunod, 1994.
2. Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of The Art of Computer Programming. Addison Wesley, 1969.
3. Donald E. Knuth. *Sorting and searching*, volume 3 of The Art of Computer Programming. AddisonWesley, 1973.