
TD d'ULIN501

Année 2007

Version 1.0

Université de Montpellier
Place Eugène Bataillon
34095 Montpellier Cedex 5

RODOLPHE GIROUDEAU
161, RUE ADA
34392 MONTPELLIER CEDEX 5
TEL : 04-67-41-85-40
MAIL : RGIROU@LIRMM.FR

ULIN 501
TD – Séance n° 1

Exercice 1 – Complexité

Soit A et B , deux algorithmes pour résoudre un même problème. La complexité de A est en $\theta(n^2)$ et son exécution avec $n = 100$ dure 1s. La complexité de B est en $\theta(n \log n)$ et son exécution avec $n = 100$ dure 10s.

1. L'application prévoit une valeur de n égale à 1000. Quel algorithme faut-il a priori choisir ?
2. Même question si n égale à 10000.

Exercice 2 – Complexité

Soit un algorithme en $\theta(n^2)$. Un ordinateur X permet de traiter en 1 mn des problèmes de taille n_0 . Quelle est la taille des problèmes que l'on pourra traiter en 1 mn avec un ordinateur 100 fois plus rapide ?

Exercice 3 – Complexité

Lesquelles des assertions suivantes sont vraies ? Prouvez vos réponses.

1. $1000 \in O(1)$;
2. $n^2 \in O(n^3)$;
3. $n^3 \in O(n^2)$;
4. $2^{n+1} \in O(2^n)$;
5. $(n + 1)^2 \in O(n^2)$;
6. $n^3 + 3n^2 + n + 1996 \in O(n^3)$;
7. $n^2 * n^3 \in O(n^3)$;
8. $2^{2n} \in O(2^n)$;
9. $\frac{1}{2}n^2 - 3n \in \theta(n^2)$.

Exercice 4 – Complexité

Comparer deux à deux les fonctions suivantes :

1. $f_1(n) = n^2 + 100$
2. Soit f_2 défini de la manière suivante :

$$\begin{aligned} f_2(n) &= n \text{ si } n \text{ est impair} \\ &= n^3 \text{ si } n \text{ est pair} \end{aligned}$$

3. Soit f_3 défini de la manière suivante :

$$\begin{aligned} f_3(n) &= n \text{ si } n < 100 \\ &= n^3 \text{ si } n \geq 100 \end{aligned}$$

Exercice 5 – Complexité

1. Soient f_1 et f_2 deux fonctions telles que $f_1 = \theta(g)$ et $f_2 = \theta(g)$ et et $f_1 \geq f_2$. A-t'on $f_1 - f_2 = \theta(g)$?

Exercice 6 – Complexité

Quelle est la complexité de la boucle suivante :

```
for i:=1 to (n-1) do
  for j:=(i+1) to n do
    for k:=1 to j do
      {instruction}
```

Exercice 7 – Complexité

Montrer que $O(f + g) = O(\max(f, g))$.

Exercice 8 – Complexité

Regrouper en classes d'équivalence pour θ les fonctions suivantes. Trier les classes d'équivalence pour O .

$$n, 2^n, n \log n, n - n^3 + 7n^5, n^2, \log_2 n, n^3, \sqrt{n} + \log_2 n, \log_2 n^2, n!, \log n.$$

ULIN 501
TD – Séance n° 2

Exercice 1 – Étude d'un exemple

1. Soit l'algorithme suivant pour trouver le plus grand élément d'une liste d'éléments stockés dans un tableau :

```
M:=T[1];  
for j:=2 to n do  
  if T[j] > M then M:=T[j]
```

- (a) Etudier la complexité en nombre de comparaisons de cet algorithme.
- (b) Expliquer pourquoi, il ne peut pas y avoir d'algorithme de recherche du plus grand élément effectuant moins de comparaisons.
- (c) Calculer le nombre minimum et le nombre maximum d'affectations de M lors de l'exécution de l'algorithme. Donner à chaque fois la caractéristique des tableaux qui permettent d'atteindre ces valeurs et la probabilité d'avoir un tel tableau (on supposera que toutes les valeurs de T sont différentes et que toutes les permutations des éléments dans le tableau sont équiprobables).
- (d) Calculer le nombre moyen d'affectations de M (indication : pour un j donné quelle est la probabilité d'effectuer $M := T[j]$).
2. (a) En vous inspirant de l'exercice précédent, donner un algorithme qui calcule les 2 plus grands éléments (une seule boucle "for" avec "if" imbriqués).
- (b) Donner un autre algorithme en inversant les deux "if" et calculer le nombre de comparaisons au pire pour les deux algorithmes.
- (c) Calculer le nombre moyen de comparaisons pour chacun des deux algorithmes et déterminer le meilleur.
3. Pour déterminer les 2 plus grands éléments, on utilise la stratégie du tournoi. Les éléments sont comparés deux à deux et les plus grands éléments sont conservés. On réitère jusqu'à ce qu'il ne reste qu'un seul élément qui est donc le plus grand élément. Soit la liste $L = \{1, 7, 20, 12, 9, 1, 14, 17, 25, 29, 3\}$.
- (a) Combien de comparaisons sont effectuées pour trouver le plus grand élément ?
- (b) A combien d'éléments au pire, le plus grand élément a-t-il été comparé directement ?
- (c) Que peut-on dire sur le deuxième plus grand élément ?
- (d) En déduire un algorithme pour trouver les deux grands éléments et déterminer sa complexité au pire en nombre de comparaisons.
- (e) Question subsidiaire : montrer que cet algorithme est optimale en nombre de comparaisons au pire.

Ulin 501
TD – Séance n° 3

Exercice 1 – Programmes récursifs

Analyser la complexité des différents programmes en fonction du nombre d'appels récursifs effectués :

```
Rec1(n)
si n<2 alors retourner(1)
    sinon retourner(Rec1(n-1)+2)
```

```
Rec2(n)
si n<2 alors retourner(1)
    sinon retourner(Rec2(n div 2)+2)
```

```
Rec3(n)
si n<2 alors retourner(1)
    sinon retourner(Rec3(n-1)*(Rec3(n-1)+2))
```

```
Rec4(n)
si n<2 alors retourner(1)
    sinon retourner((Rec4(n div 2)+2)*(Rec4(n div 2)+3))
```

Exercice 2 – Calculs récursifs

Calculer x^n , pour des valeurs entières et positives de n , de plusieurs façons en utilisant les définitions récursives suivantes (qu'il faudra compléter) et évaluer l'ordre de grandeur de la complexité de chacune des méthodes :

1. $x^n = x * x^{n-1}$
2. $x^n = x^{ndiv2} * x^{ndiv2} * x^{nmod2}$
3. Même définition que 2 mais en utilisant une variable intermédiaire pour stocker x^{ndiv2} .

Exercice 3 – Calculs récursifs

On rappelle la définition des nombres de Fibonacci :

$$f_1 = 0, f_2 = 1, \forall n \geq 2 f_n = f_{n-1} + f_{n-2}$$

Programmer la fonction $f(n)$:

1. En utilisant la définition récursive (évaluer sa complexité).

2. En utilisant une version itérative efficace (évaluer sa complexité).

3. Montrer que si $n \geq 1$ alors

$$\begin{aligned} - f_{2n} &= f_n^2 + 2 * f_n * f_{n-1} \\ - f_{2n+1} &= f_n^2 + f_{n+1}^2 \end{aligned}$$

Évaluer approximativement l'ordre de grandeur de la complexité d'un programme pour calculer f_n qui utilise directement cette nouvelle définition récursive.

En utilisant des variables intermédiaires améliorer le programme précédent et calculer sa nouvelle complexité.

4. Montrer que

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

En déduire un programme en $\theta(\log n)$ pour calculer f_n .

Tris
TD – Séance n° 4

Exercice 1 – Tri bulle

Soit l'algorithme suivant qui trie un tableau t :

```
for i:=1 to n-1 do
  for j:=n downto i+1 do
    if t[j] < t[j-1] then echanger (t[j],t[j-1])
```

1. Quel est le nombre de comparaisons dans le pire des cas et en moyenne ?
2. Même question pour le nombre d'échanges.
3. Modifier l'algorithme pour trouver le k^{ieme} plus petit élément du tableau t . Quelle est la complexité de cet algorithme ?

Exercice 2 – Tri par sélection

Répondre aux mêmes questions que précédemment avec l'algorithme qui consiste pour trier les éléments dans les cases de 1 à n ($n > 1$) à chercher l'indice, max du plus grand élément dans les cases de 1 à n et échanger les cases d'indice max et n , puis trier les éléments dans les cases de 1 à $(n - 1)$. Ecrire l'algorithme.

Exercice 3 – Tri fusion

Nous souhaitons étudier le tri fusion.

1. Rappeler le principe.
2. Appliquer le sur le tableau suivant :

| | | | | | | | |
|----|---|---|----|----|----|---|---|
| 20 | 5 | 7 | 12 | 21 | 33 | 1 | 6 |
|----|---|---|----|----|----|---|---|

3. Calculer le nombre maximum de comparaisons. Pour simplifier, on supposera que n est une puissance de 2.
4. Calculer en moyenne le nombre de comparaisons.

Exercice 4 – Tri rapide

1. Soit T un tableau de n entiers distincts. Donner un algorithme efficace qui déplace les entiers de T de telle façon que les cases de plus petits indices contiennent les entiers inférieurs à l'entier qui était initialement en $T[1]$ (cet entier sera appelé pivot dans la suite), et les cases de plus grands indices les entiers plus grands que le pivot. Plus précisément, s'il y a dans T , $(p - 1)$ entiers inférieurs au pivot alors les cases de 1 à $(p - 1)$ contiennent ces entiers, la case p contient le pivot, et les cases de $(p + 1)$ à n contiennent les entiers plus grands que le pivot.
2. En déduire un algorithme de tri utilisant deux appels récursifs et analyser cet algorithme au pire et en moyenne.
3. Quelle est la taille maximum de la pile nécessaire pour gérer les appels récursifs de cet algorithme ?

4. Réécrire l'algorithme avec un seul appel récursif et une boucle **tant que** en décursifiant l'appel récursif terminal.
5. Modifier l'algorithme proposé en 4) pour que l'appel récursif porte sur la partie du tableau contenant le moins d'entiers et montrer que la pile nécessaire est alors de hauteur logarithmique au pire.
6. Modifier l'algorithme de tri rapide de façon à avoir le k^{ieme} plus petit élément du tableau.
7. Montrer que l'algorithme précédent est linéaire (c'est à dire en $\theta(n)$ en moyenne).

ULIN 501
TD – Séance n° 5

Exercice 1 – Arbres binaires : Application au tri par tas

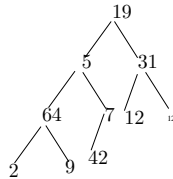


FIG. 1 – Arbre binaire

1. Un arbre parfait est un arbre binaire dont tous les niveaux sont complètement remplis sauf éventuellement le dernier niveau, et dans ce cas les sommets (feuilles) du dernier niveau sont groupés le plus à gauche possible.
 - (a) Comment peut-on représenter efficacement un arbre binaire parfait par un tableau et un entier (représentant la taille en nombre d'éléments du tas) ? Donner le tableau représentant l'arbre binaire proposé à la figure 1.
 - (b) Donner les fonctions permettant de savoir si un sommet est une feuille, de connaître le père d'un sommet, le fils gauche, le fils droit.
 - (c) Calculer la hauteur d'un arbre binaire parfait.
2. Un tas est un arbre binaire parfait tel que chaque élément d'un sommet est inférieur ou égal aux éléments de ses fils.
 - (a) Donner les fonctions permettant d'obtenir le minimum d'un tas,
 - (b) de supprimer le minimum d'un tas,
 - (c) d'ajouter un élément à un tas.
 - (d) Evaluer la complexité de chacune de ces fonctions
3. Donnez une fonction qui structure un tableau quelconque en tas en ajoutant successivement les éléments au tas déjà construit (initialement le tas est vide, la partie gauche du tableau est structurée en tas, les éléments du tableau sont ajoutés un à un de la gauche vers la droite). Evaluer la complexité de cette construction.
4. Même question mais en construisant le tas de la droite vers la gauche. On considère que les feuilles sont des tas et on fusionne les tas en ajoutant les sommets internes. Evaluer la complexité de cette construction.

5. En déduire un algorithme de tri par ordre décroissant d'un tableau (tri par tas). Evaluer sa complexité.
6. Adapter-le pour trouver le k^{ieme} plus petit élément d'un tableau. Evaluer la complexité de recherche de cet élément.

ULIN 501
TD – Séance n° 6

Exercice 1 – Arbres de recherches et arbres équilibrés

1. Ecrire une fonction qui imprime les éléments d'un arbre de recherche en ordre croissant.
2. A chaque sommet d'un arbre binaire de recherche, on ajoute le nombre d'éléments sous ce sommet ($nombre(A) = nombre(A.filsgauche) + nombre(A.filsdroit) + 1$). Ecrire une fonction de recherche du k^{ieme} plus petit élément.
3. Ecrire une fonction qui ajoute un élément à la racine d'un arbre binaire de recherche (on coupe l'ABR en 2 : un arbre contenant les éléments inférieurs à l'élément et un les éléments supérieurs).
4. Ecrire un algorithme de fusion de deux ABR A et B :
 - en ajoutant un par un les éléments de A dans B évaluer approximativement la complexité moyenne de cette stratégie),
 - en utilisant la procédure de coupure nécessaire pour la question précédente (évaluer approximativement la complexité moyenne de cette stratégie).
5. On considère la méthode de recherche auto-adaptative dans un ABR, si on recherche x on veut que x soit la racine de l'ABR parès la recherche (on fait une succession de rotation lors de la recherche qui permet d'obtenir ce résultat).
6. On appelle suite d'arbres de Fibonacci la suite d'arbres binaires définis de la façon suivante : $F_0 = \emptyset$, $F_1 = o$ (l'arbre avec un seul sommet), et pour $n \geq 2$, F_n est l'arbre binaire dont le sous arbre gauche est F_{n-1} et le sous arbre droit est F_{n-2} .
 - (a) Donner F_i avec $i = 1 \dots 5$ Quelle est la hauteur de F_n ? Que peut-on en déduire?
 - (b) Calculer le nombre de sommets de F_n .
 - (c) Calculer le nombre de sommets minimal que doit avoir un arbre équilibré de hauteur h . Conclure.

Exercice 2 – Arbre bicolore

Un arbre rouge et noir est un arbre binaire de recherche comportant un bit de stockage supplémentaire par noeud : sa couleur, qui peut valoir soit rouge, soit noir. En contrôlant la manière dont les noeuds sont colorés sur n'importe quel chemin allant de la racine à une feuille, les arbres rouge et noir garantissent qu'aucun de ces chemins n'est plus de deux fois plus long que n'importe quel autre, ce qui rend l'arbre approximativement équilibré.

Chaque noeud de l'arbre contient maintenant les champs couleur, clé, gauche, droit et p . Si un fils ou le père d'un noeud n'existe pas, le champ correspond contient la valeur nil. Un arbre binaire de recherche est un arbre rouge et noir s'il satisfait les propriétés suivantes :

1. chaque noeud est soit rouge, soit noir

2. Chaque feuille (nil) est noire
3. Si un noeud est rouge, alors ses deux fils sont noirs
4. Chaque chemin simple reliant un noeud à une feuille contient le même nombre de noeuds noirs

On appelle hauteur noire le nombre de noeuds noirs présents dans un chemin quelconque descendant d'un noeud x (sans l'inclure) à une feuille. Cette fonction est notée $hn(x)$.

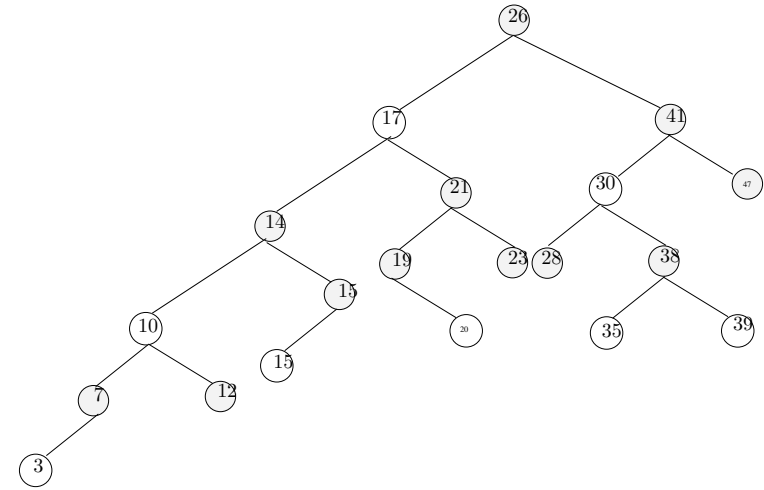


FIG. 1 –

1. Est-ce que cette fonction est bien définie ?
2. Donner la fonction $hn(x)$ pour l'arbre donné à la figure 1.
3. Dessiner l'arbre binaire de recherche complet de hauteur 3 à partir des clés $\{1, \dots, 15\}$. Ajouter les feuilles nil et colorier les noeuds de trois manières différentes, pour que les hauteurs noires des arbres des arbres rouge et noir résultants soient 2, 3 et 4.
4. On suppose que la racine d'un arbre rouge et noir est rouge. Si on la rend noire, l'arbre reste-t'il un arbre rouge et noir ?
5. Montrer qu'un arbre rouge et noir comportant n noeuds internes a une hauteur au plus égale à $2 \log(n + 1)$.
6. Montrer que le chemin simple le plus long reliant un noeud x d'un arbre rouge et noir à une feuille descendante a une longueur au plus égale à deux fois celle du plus court chemin simple reliant un noeud x à une feuille descendante.

7. Quel est le plus grand nombre possible de noeuds internes dans un arbre rouge et noir de hauteur noire k ? Quel est le plus petit nombre possible?
8. Décrire un arbre rouge et noir de n clé ayant le plus grand rapport possible entre les noeuds internes rouges et les noeuds internes noirs. Quel est ce rapport? Quel est l'arbre possédant le plus petit rapport possible, et quel est ce rapport?

ULIN 501
TD - Séance n° 7

Exercice 1 – Table de hachage en adressage ouvert

Cet exercice a pour but de se familiariser avec quelques fonctions de hachage en adressage ouvert. La taille de la table de hachage est notée par m . Si p et q sont deux entiers naturels, $p \bmod q$ est le reste de la division euclidienne de p par q .

Nous considérons les fonctions de hachage :

$$h_1(k) = k \bmod m \text{ et } h_2(k) = 1 + k \bmod (m - 1)$$

1. Insérer successivement les clés 10, 22, 31, 4, 15, 28, 17, 88 et 59 dans une table de hachage de taille $m = 11$ gérée en adressage ouvert en utilisant comme fonction de hachage principale $h(k, i) = (h_1(k) + i) \bmod m$.
2. Même question lorsque $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$.

Exercice 2 – Hachage avec chaînage

Cet exercice analyse le nombre moyen de clés examinées lors de la recherche d'une clé dans une table de hachage où les collisions sont gérées par chaînage.

1. Soit h la fonction de hachage définie par $h(k) = k \bmod 9$ et soit une table de hachage de taille $m = 9$. Donner l'état de la table après insertion successive des clés 5, 28, 19, 15, 20, 33, 12, 17 et 10 lorsque que les collisions sont gérées par chaînage. On considère une fonction de hachage h uniforme utilisée pour une table de taille m où les collisions sont résolues par la technique du chaînage.
2. Si n clés sont présentes dans la table, quelle est la longueur moyenne d'une liste? En déduire la durée moyenne (en nombre de clés testées) de recherche d'une clé non présente dans la table (recherche infructueuse).
3. On admet que si $r(x)$ est le rang d'insertion de la clé x présente dans la table les n valeurs possibles de $r(x)$ sont équiprobables. Déterminer la durée moyenne de recherche de x . En déduire la durée moyenne de recherche d'une clé présente dans la table.
4. Que peut-on conclure des deux questions précédentes lorsque $n = O(m)$?

Exercice 3 – Table de hachage avec résolution des collisions par chaînage

Vous programmerez les différentes opérations possibles sur une table de hachage avec résolution des collisions par chaînage :

- Insertion d'une clé
- Suppression d'une clé
- Recherche d'une clé

Vous préciserez pour chacune de ces opérations, la complexité en temps de celle-ci. Vous utiliserez les deux types de fonctions de hachage possible.

Nous allons maintenant considérer un exemple précis dans lequel la table a une longueur $m = 11$, et nous allons insérer dans cet ordre les clés : 10, 22, 31, 4, 15, 28, 17, 88, 59. Vous afficherez le contenu de la table de hachage pour chacune des fonctions de hachage possible.

Exercice 4 – Table de hachage avec adressage ouvert

Dans un premier temps vous écrirez les différentes opérations possibles sur une table de hachage :

- Insertion d'une clé
- Suppression d'une clé
- Recherche d'une clé

Vous utiliserez les trois fonctions de hachage possibles, en considérant :

- $h'(k) = k \bmod m$
- $c_1 = 1$ et $c_2 = 3$
- $h_1(k) = k \bmod m$ et $h_2(k) = 1 + (k \bmod (m - 1))$

Nous allons maintenant considérer un exemple précis dans lequel la table a une longueur $m = 11$, et nous allons insérer dans cet ordre les clés : 10, 22, 31, 4, 15, 28, 17, 88, 59. Vous afficherez le contenu de la table de hachage pour chacune des fonctions de hachage possible.

Exercice 5 – Recherche dans un ensemble constant

On vous demande d'implanter un ensemble de n éléments avec des clés numériques. L'ensemble est constant, c'est-à-dire que les opérations d'insertion et de destruction sont interdites. La seule opération à mettre en oeuvre est l'opération de recherche à partir de la clé. On se place dans le cas d'un hachage par adressage ouvert.

Complexité dans le cas d'une recherche infructueuse (le pire cas)

On suppose disposer d'une fonction h de hachage uniforme, c'est-à-dire que pour toute clé k , la séquence construite par $\langle h(k, 1), h(k, 2), \dots, h(k, m) \rangle$ est une permutation de $\langle 0, 1, \dots, m - 1 \rangle$. En clair, cela revient à dire que $h(k, i) = h(k, j) \rightarrow i = j$.

On note α la densité de remplissage, c'est à dire le rapport n/m en sachant que $n \leq m$ et donc $\alpha \leq 1$.

Très important : Pour notre problème nous imposerons que $\alpha < 1$.

On va essayer de calculer le nombre d'accès à la table dans le cas où la clé recherchée n'est pas dans la table et de démontrer le théorème suivant :

Théorème 1 *Etant donnée une table de hachage par adressage ouvert avec un taux de remplissage $\alpha = n/m < 1$ et en supposant une fonction de hachage uniforme, le nombre d'accès à la table lors d'une recherche infructueuse est au plus $1/(1 - \alpha)$.*

Dans le cas d'une recherche infructueuse, tous les accès sauf le dernier, se font sur des cases occupées et le dernier accès est fait sur une case vide.

Si l'on note $p_i = \text{Pr}\{\text{exactement } i \text{ essais sont tombés sur une case occupée}\}$ pour $i = 0, 1, 2, \dots$

1. Pourquoi pour $i > n$ a-t-on $p_i = 0$?
2. Quel est le nombre de sondages attendu?
3. Si on pose $q_i = \text{Pr}\{\text{au moins } i \text{ accès à une case occupée}\}$ pour $i = 0, 1, 2, \dots$, on sait que $\sum_{i=0}^{\infty} i \cdot p_i = \sum_{i=0}^{\infty} q_i$.

Calculer q_1, q_2, q_i .

Sachant que $(n-j)/(m-j) \leq n/m$ si $n \leq m$ et $j \geq 0$, majorer q_i et donner un majorant de

$$1 + \sum_{i=0}^{\infty} q_i.$$

Conclure la démonstration.