Calculs de complexité d'algorithmes

- Notations asymptotiques :
  0 et Θ
- Complexité des algorithmes
- Exemples de calcul de complexité

Complexités d'un algorithme

- Un algorithme à partir d'une donnée établit un résultat .
- La taille de la donnée est mesurée par un entier n.
  - complexité temporelle
    une fonction de n qui mesure le temps de calcul pour une donnée de taille n
  - complexité en mémoire
    une fonction de n qui mesure la place mémoire utilisée pour le calcul sur une donnée de taille n

Complexités temporelles

- Dans le pire des cas : donne une borne supérieure sur le temps de calcul pour toutes les données de taille n

- En moyenne : fait la moyenne des temps de calculs pour toutes les données de taille n

Mesure-t-on vraiment le temps de calcul ?

- Non, car le temps de calcul dépend de la machine
- On évalue le nombre d'opérations "élémentaires" faites (additions, multiplications, etc.)

- On obtient donc une estimation du temps de calcul à une constante multiplicative près (le temps mis par la machine pour faire une opération "élémentaire")

- Dans cette estimation, on ne considère que le *terme dominant*

Définitions

- On dit que f est **dominée** par g

(noté f= O (g)) lorsque

$$\exists n_0, \exists c > 0, \quad \forall n \geq n_0, \left| f(n) \right| \leq cg(n)$$

- On dit que f est **du même ordre de grandeur** que g et l'on note f = Θ (g) lorsque f=O(g) et g=O(f).

Définitions

- **f est négligeable** devant g, (noté f =o(g)) lorsque f(n)/g(n) tend vers 0 quand n tend vers l'infini

- On dit que f est **équivalente** à g lorsque f(n)/g(n) tend vers 1 lorsque n tend vers l'infini

Relations entre O, o et autres

- **f est négligeable** devant g implique
  f est **dominée** par g ?

- f est **équivalente** à g
  implique
  f est **du même ordre de grandeur** que g ?

Remarque

- Hormis pour l'équivalence, ces notions sont indépendantes des constantes multiplicatives non nulles.

  Par exemple :

  si **f est négligeable** devant g,
  alors c**f est négligeable** devant c'g pour tout réels c et c' non nuls.

Polynômes et notations O et Θ

- Soit P(n) un polynôme en n. Pour quelles valeurs de p a-t-on $P(n)=O(n^p)$?

- Pour quelles valeurs de p a-t-on $P(n)=\Theta(n^p)$?

- Montrer que pour tout entier k, on a

$$\sum_{i=0}^{n} i^{k} = \Theta\left(n^{k+1}\right)$$

Échelle De Comparaison

- Exercice

Soient les fonctions

$f_1(n)=n$, $f_2(n)=2^n$,

$f_3(n)=n^2$, $f_4(n)=2n$,

$f_5(n)=n^n$, $f_6(n)=\log n$, $f_7(n)=n!$, $f_8(n)= n\log n$

Pour chaque couple (i, j) dire si on a $f_i=o(f_j)$,

$f_i=O(f_j)$

$f_i=\Theta(f_j)$.

10^9 Instructions/secondes (1 gigaHertz)

n	5	10	15	20	100	1000
log n	$3\ 10^{-9}$ s	$4\ 10^{-9}$ s	$4\ 10^{-9}$ s	$5\ 10^{-9}$ s	$7\ 10^{-9}$ s	10^{-8} s
2n	$10\ 10^{-9}$ s	$2\ 10^{-8}$s	$3\ 10^{-8}$ s	$4\ 10^{-8}$ s	$2\ 10^{-7}$ s	$2\ 10^{-6}$ s
nlogn	$12\ 10^{-9}$ s	$3\ 10^{-8}$s	$6\ 10^{-8}$s	10^{-7}s	$7\ 10^{-7}$s	10^{-5}s
n^2	$25\ 10^{-9}$ s	10^{-7} s	$2{,}25\ 10^{-7}$ s	$4\ 10^{-7}$ s	10^{-5} s	10^{-3} s
n^5	$3\ 10^{-6}$ s	10^{-4} s	$7{,}59\ 10^{-4}$ s	$3\ 10^{-3}$ s	10 s	10^6 s = 11 jours
2^n	$32\ 10^{-9}$ s	10^{-6} s	$3{,}28\ 10^{-5}$ s	10^{-3} s	$1{,}2\ 10^{21}$ s $4\ 10^{11}$ siècles	10^{292} s $3\ 10^{282}$ siècles
n !	$120\ 10^{-9}$ s	$4\ 10^{-3}$ s	$1{,}4\ 10^3$s= 23 minutes	$2{,}4\ 10^9$ s = 77 ans	10^{147} s $3\ 10^{139}$ siècles	10^{500} s
n^n	$3\ 10^{-6}$ s	10 s	$4{,}37\ 10^8$s = 13 ans	10^{17}s = $3\ 10^7$siècles	10^{191}s $3\ 10^{181}$siècles	10^{3000} s

En une Journée, un an jusqu'où peut-on, aller ?

f(n)	**1 jour**	1 an
n	$9\ 10^{13}$	$31\ 10^{15}$
log(n)	10^{310}	$10^{10^{16}}$
2n	$4.5\ 10^{13}$	$15\ 10^{15}$
n log(n)	$2\ 10^{12}$	$5\ 10^{14}$
n^2	10^7	$1.7\ 10^8$
n^5	600	2 000
2^n	32	55
n!	16	18
n^n	12	13

Pourquoi Utiliser O Et Θ Pour Mesurer Des Complexités?

- Expressions à une constante multiplicative près, indépendante du temps de calcul d'une instruction de base
- Toute instruction de base prend le temps 1
- Terme dominant uniquement donc expression simple

n, c’est quoi?

- La complexité s’exprime en fonction de la taille de la donnée
- A vous de dire quelle fonction taille vous avez choisie
- Et la donnée c’est quoi ?

Règle 1
Composition Séquentielle

- I_1 complexité temporelle en $\Theta(f_1(n))$
- I_2 complexité temporelle en $\Theta(f_2(n))$

Le bloc d'instructions

I_1 ;

I_2

a une complexité temporelle en

$$\Theta(\max(f_1(n), f_2(n)))$$

Règle 2
if (C) I1 else I2

- Évaluation de la condition C est en $\Theta(f(n))$
- De I_1 en $\Theta(f_1(n))$,
  de I_2 en $\Theta(f_2(n))$

Alors la complexité de l'instruction

`if (C)` I_1 `else` I_2

est en $O(\max(f(n),f_1(n),f_2(n)))$

Règle 3
Boucle for

En supposant que :

- I_1 a une complexité temporelle en $\Theta(f_1(n))$
- I_1 n'a aucun effet sur les variables i et n,

la complexité temporelle de la boucle

```
for (int i=0 ; i < n ; i++) {
  I1
}
```

est en $\Theta(n*f_1(n))$

- Si une instruction I se trouve au cœur de k boucles for imbriquées, chacune d'elle de la forme
  `for (int` i_m`=0 ;` i_m `< n ;` i_m`++)`
  où 0 < m < (k+1)

  combien de fois l'instruction I est elle exécutée ?

- Si une instruction I se trouve au cœur de k boucles for imbriquées, chacune d'elle de la forme

  for (int i_m=0 ; i_m < i_{m-1} ; i_m++)

  où 0 < m < (k+1)

  avec i_0=n

  combien de fois l'instruction I est elle exécutée ?

Règle 4
Boucle While (C) {I}

- Évaluation de C en $\Theta(f(n))$
- I en $\Theta(f_1(n))$
- Boucle while est exécutée $\Theta(g(n))$ fois

```
while (C) {
  I
}
```

est en $\Theta(g(n)*max(f(n),f_1(n)))$

Estimer les complexités des morceaux de codes suivants, sachant que
l'instruction I1 est en $\Theta(1)$ et

I1 ne modifie pas les entiers i, j, k et n

```
for (int i=0 ; i < n ; i++) {
  for (int j=i ; j < n ; j++) {
        for (int k=0 ; k < j ;
  k++) {
        I1
        }
  }
}
```

Estimer les complexités des morceaux de codes suivants, sachant que et les instructions I1, I2, I3 sont en $\Theta(1)$ et ne modifie pas les entiers i, j, k et n

```
int i=1 ;
int j=1 ;
while (i <n) {
  i++ ;
  I1 ;
  while (( j < n) && Condition) {
     j++ ;
     I2
  }
  I3 ;
}
```

Règle 5
Composition de Méthodes

- $methode_1(Classe_1\ o_1)$ en $O(f_1(taille_1(o_1))$
- $methode_2(Classe_2\ o_2)$ en $O(f_2(taille_2(o_2))$
- $methode_2$ renvoie un objet de Classe1

La complexité de **`methode1(methode2(o2))`**

est en

$O(max(f_2(taille_2(o_2)),f_1(taille_1(methode_2(o_2)))))$

On connaît l'écriture d'un nombre en base b, et l'on veut convertir ce nombre en base usuelle (base dix).

1. On utilise la méthode "convertionDirecte". Quelle en est la complexité?

```
public int convertionDirecte(int[] a, int b) {
  int résultat =a[0] ;
  int auxiliaire ;
  for (int rang =1 ; rang < a.length ; rang++) {
     if (a[rang] != 0) {
          auxiliaire = a[rang] ;
          for (int indice =0 ; indice <rang ;
                              indice ++) {
               auxiliaire = auxiliaire *b ;
          }
          résultat = résultat + auxiliaire;
     }
  }
  return résultat ;
}
```

2. On utilise la méthode "convertionDirecteV2" dans laquelle on mémorise dans une variable monome b^{rang}. Ecrire cette méthode "convertionDirecteV2" et en donner la complexité ?

3. Prouvez que la méthode suivante dite de Horner, effectue bien le même travail. Quelle en est la complexité ?

```
public int horner(int[] a, int b) {
  int n = a.length ;
  int résultat =a[n-1] ;
  for (int rang = n-2 ; rang >= 0 ;
                    rang--){
       résultat = b* résultat +a[rang] ;
    }
  return résultat ;
}
```

On désire élever l'entier a à la puissance n.

Quelle est la complexité de la méthode suivante ?

```
public int puissance(int n, int a) {
  int résultat = a ;
  for(int i =1 ; i <n ;i++){
     résultat=résultat*a ;
  }
  return résultat ;
}
```

Montrez que le code suivant est correct. Quel en est la complexité ?

```
public int puissance(int n, int a) {
  int aux = n ;
  int puissanceDea = a ;
  int résultat=1 ;
  while ( aux  != 0 ) {
    if (aux mod 2 == 1) {
      résultat = résultat * puissanceDea ;
    }
    aux=aux/2 ;
    puissanceDea = puissanceDea * puissanceDea ;
  }
  return résultat ;
}
```

Programmation récursive

- Quelques exemples
- Equations de récurrences
- Quelques méthodes de résolution

Recherche dichotomique du plus grand élément

- L contient n éléments
- Algorithme (récursif)
  - Si L contient un seul élément : c'est fini
  - Sinon :
    - Couper L en deux listes L_1 et L_2 de taille "presque" identiques
    - Chercher m_1 le max de L_1
    - Chercher m_2 le max de L_1
    - Retourner le max de m_1 et m_2

Combien de comparaisons ?

On note c(n) le nombre de comparaisons nécessaires pour la recherche dichotomique du plus grand élément dans une liste de taille n

- $c(1) = 0$
- $c(n) = c(\lceil n/2 \rceil)+c(\lfloor n/2 \rfloor)+1$

Déterminez la complexité de la méthode suivante

```
int factorial(int n) {
  if (n == 0) {
     return 1;
  }
  else {
     return (n*factorial(n-1));
  }
}
```

Méthode factorielle

- Soit c(n) la nombre de multiplications effectuées dans le calcul de factorial(n).

- On a c(n)=c(n-1)+1, c(1)=0

Recherche du maximum dans une table de n éléments

- Si n=1, renvoyer l'unique élément
- Sinon calculer récursivement le maximum des n-1 premiers élements;

  Le comparer avec le dernier élément; renvoyer le plus grand des deux.

Analyse : nombre de comparaisons effectuées

- C(n)= complexité de la recherche du plus grand parmi n

- c(n)=c(n-1)+1

- c(1)=0

Trier une table de n éléments

- Si n=1 rien à faire
- Sinon
  - rechercher le maximum de la table
  - échanger le maximum et le dernier élément
  - trier la sous-table constituée des n-1 premiers éléments

- $c(n)=c(n-1)+an+b$
- $c(1)=1$

Tours de Hanoi

- Combien de mouvements au minimum pour déplacer une tour de n disques

Tour de Hanoi

```
public class Towers {
  static int nDisks=7;
  public static void main(String[]
                                 args){
    moveTowers(nDisks,'A','B','C');
  }
}
```

```
// Pré-condition : n > 0
public static void moveTowers(int n, char
  from,  char inter, char to) {
  if (n==1) {
     moveDisk(1,from,to);
  }
  else {
     moveTowers(n-1,from,to,inter);
     moveDisk(n,from,to);
     moveTowers(n-1,inter,from,to);
  }
}
```

où moveDisk(n,from,to) peut être par exemple :
System.out.println("Disk" + n +" from "+ from + " to " + to)

Complexité de moveTowers

- $c(n)=2c(n-1)+k$
  (ou $c(n)=2c(n-1)+1$)

- Donc $c(n)=a2^n+b$

- $C(n)=\Theta(2^n)$

- On considère deux versions modifiées des tours de Hanoi. Dans chacun des cas, on demande quel est le nombre minimum de déplacements de disques nécessaires.

- La pile contient initialement 2n disques, de n tailles différentes, il y a deux disques de chaque taille. Les disques de même taille sont indistinguables.

- La pile comporte n disques de taille différente, mais les 3 piquets sont sur un cercle et les mouvements élémentaires de disques se font du piquet où est le disque à son suivant dans le sens des aiguilles d'une montre.

Nombres De Fibonacci

La suite de Fibonacci

Leonardo de Pise, surnommé Fibonacci est un mystère de l'histoire des mathématiques. Il serait né vers 1175 et mort en 1240 (?), et aurait vécu toute sa vie à Pise. Il a publié un unique livre, *Liber Abaci* (une œuvre collective ?).

Nombres De Fibonacci

Reproduction des lapins : «Possédant au départ un couple de lapins , combien de couples de lapins obtient-on en douze mois si chaque couple engendre tous les mois un nouveau couple à compter du second mois de son existence ? »

- Janvier : 1 couple
- Février : 1 couple
- Mars : 1 + 1 = 2 couples
- Avril : 2 + 1 = 3 couples
- Mai : 3 + 2 = 5 couples
- Juin : 5 + 3 = 8 couples
- Juillet : 8 + 5 = 13 couples
- Août : 13 + 8 = 21 couples
- Septembre : 21 + 13 = 34 couples
- Octobre : 34 + 21 = 55 couples
- Novembre : 55 + 34 = 89 couples
- Décembre : 89 + 55 = 144 couples
- Janvier : 144 + 89 = 233 couples

Nombres De Fibonacci

- Janvier : 1 couple
- Février : 1 couple
- Mars : 1 + 1 = 2 couples
- Avril : 2 + 1 = 3 couples
- Mai : 3 + 2 = 5 couples
- Juin : 5 + 3 = 8 couples
- Juillet : 8 + 5 = 13 couples
- Août : 13 + 8 = 21 couples
- Septembre : 21 + 13 = 34 couples
- Octobre : 34 + 21 = 55 couples
- Novembre : 55 + 34 = 89 couples
- Décembre : 89 + 55 = 144 couples
- Janvier : 144 + 89 = 233 couples

Le tableau correspond à ce qu'on appelle la suite des nombres de Fibonacci.

On note Fn le nombre de couples de lapins au mois n.

F(n) = nombre de couples au mois (n-1)

+ nombre de couples nés au mois n

= nombre de couples au mois (n-1)

+ nombre de couples productifs au mois (n-1)

= nombre de couples au mois (n-1)

+ nombre de couples nés au mois (n-2)

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

Nombres De Fibonacci

```
public int fibonacci (int n) {
  if (n==0) return 0 ;
  else {
     if (n==1) return 1 ;
     else
   return fibonacci(n-1)+fibonacci(n-2);
  }
}
```

Analyse de la complexité

- c(n)=c(n-1)+c(n-2)+1
- c(1)=c(0)=1

Complexité d'une méthode récursive ……

……. résolution d'une équation de récurrence

- Avec un outil de calcul formel (type maple)
- Avec des théorèmes de maths

Récurrences linéaires

- Définition:

  Une relation de récurrence linéaire homogène d'ordre k , à coefficients constants est définie par une équation de la forme

$$u_n = a_1 u_{n-1} + \ldots + a_k u_{n-k} \qquad (R)$$

  Le polynôme caractéristique associé est

$$P(r) = r^k - a_1 r^{k-1} - \ldots - a_{k-1} r - a_k$$

Solutions d'une équation de récurrence linéaire d'ordre k

- L'ensemble des solutions forme un espace vectoriel de dimension k

- Si r est racine du polynôme caractéristique alors $u_n = \alpha r^n$ est solution de l'équation.
- Cas des racines multiples

Méthode du polynôme caractéristique

- Soit E l'équation de récurrence.
- Soient r_i , les q racines du polynôme caractéristique de (E), r_i ayant multiplicité m_i.
- Les solutions de (E) s'écrivent sous la forme

$$\sum_{i=1}^{q} P_i(n)\, r_i^{\,n}$$

où les $P_i(n)$sont des polynômes en n de degré m_i-1.

Exemple

- Déterminer en fonction de u_0 et u_1, la suite telle que
- $u_n=u_{n-1}-2u_{n-2}$

Réponse

$$u_n = \left(\frac{u_0}{2} - i\left(\frac{2u_1 - u_0}{2\sqrt{7}}\right)\right) r_1^n + \left(\frac{u_0}{2} + i\left(\frac{2u_1 - u_0}{2\sqrt{7}}\right)\right) r_2^n$$

$$r_1 = \frac{1 + i\sqrt{7}}{2} \qquad r_2 = \frac{1 - i\sqrt{7}}{2}$$

Exercice

- Utilisez la méthode du polynôme caractéristique pour résoudre l'équation de récurrence

$$u_n = 4u_{n-1} - 4u_{n-2}$$

$$u_0 = 1$$

$$u_1 = 6$$

Exercice

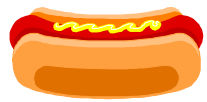
- Chaque jour, pour mon goûter , je m'achète

ou

2F 

ou

2F 

4F 

- Soit g_n le nombre de choix de goûters possibles si l'on a n Francs
  - Déterminer g_1, g_2, g_3 et g_4
  - Déterminer et résoudre l'équation de récurrence liant les g_n

- Donnez l’ensemble des solutions des équations de récurrences suivantes :
- $u_n=2u_{n-1} - u_{n-2}$
- $v_n=v_{n-1} + 6v_{n-2}$

- Déterminez la suite u_n , telle que :
- $un = 5u_{n-1} - 8u_{n-2} + 4u_{n-3}$
- $u1 = 3$, $u_2 = 11$, $u_3 = 31$

Equations non homogènes

- Soit R' l'équation non homogène

$$\forall\, n \geq k, u_n = a_1 u_{n-1} + \ldots + a_k u_{n-k} + b(n)$$

- On lui associe l'équation homogène R

$$\forall\, n \geq k, u_n = a_1 u_{n-1} + \ldots + a_k u_{n-k}$$

- La différence entre deux solutions de R' est une solution de R

Espace affine/ Espace vectoriel

- Soit s_n une solution particulière de R'.
- Toute solution de R' est obtenue à partir d'une solution de R en lui ajoutant s_n

Une recette de cuisine

- Si l'équation est de la forme

il existe une solution particulière de la forme

$$\forall n \geq k, u_n = a_1 u_{n-1} + \ldots + a_k u_{n-k} + \sum_{i=1}^{l} b_i^n P_i(n)$$

$$\sum_{i=1}^{l} b_i^n Q_i(n)$$

où $Q_i(n)$ est un polynôme de degré $d(P_i)+m_i$

avec $m_i=0$ si b_i n'est pas racine du polynôme caractéristique, et m_i = la multiplicité pour une racine

Exercices

$$u_n = 2u_{n-1} + 1, u_0 = 0$$

$$u_n = 2u_{n-1} + n + 2^n, u_1 = 0$$

- Donnez l'ensemble des solutions des équations de récurrence suivantes :
- $u_n = 3u_{n-1} - 2u_{n-2} + n$
- $v_n = v_{n-1} + 6v_{n-2} + 5^n$
- $w_n = w_{n-1} + 6w_{n-2} + 3^n$

- Résoudre l'équation de récurrence
- $u_n = 3u_{n-1} - 2u_{n-2} + n$, $u_0 = 0$, $u_1 = 0$

- Soit sommeFactoriel, la fonction définie par Evaluer la complexité en nombre de multiplications des méthodes récursives après

```
public int sommeFactoriel(int n) {
  int factorieln ;
  if (n<=1) {
    return n+1;
  }
  else {
    factorieln = n * ( sommeFactoriel (n-1)
          – sommeFactoriel (n-2) ) ;
    return sommeFactoriel (n-1) + factorieln ;
  }
}
```

```
public int sommeFactoriel(int n){
  int factorieln , somme ;
  if (n<=1) {
    return n+1;
  }
  else {
    somme = sommeFactoriel (n-1) ;
    factorieln= n*(somme–sommeFactoriel(n-2)) ;
    return somme + factorieln ;
  }
}
```

```
public class DeuxEntiers {
  int somme ;
  int factoriel;
  DeuxEntiers factorieletSommeFactoriel (int n) {
     DeuxEntiers resultat ;
     if (n==0) {
       resultat.somme = 1 ;
       resultat.factoriel = 1 ;
       return resultat ;
     }
     else {
       resultat = factorieletSommeFactoriel(n-1);
       resultat.factoriel = n*resultat.factoriel;
       resultat.somme = resultat.somme +
                        resultat.factoriel ;
       return resultat ;
     }
  }
}
```

```
public int sommeFactoriel (int n) {
    DeuxEntiers resultat ;
    resultat=factorieletSommeFactoriel(n) ;
    return resultat.somme ;
}
```

Parmi les méthodes récursives vues en exemple quelles sont celles dont on peut maintenant calculer la complexité?

- Factorielle
- Tri
- Tours de Hanoi
- Les nombres de Fibonnacci
- Mais pas la recherche dichotomique

Le cas de Fibonacci

- On obtient une complexité exponentielle pour la programmation récursive.

- Il existe des programmations plus efficaces du calcul du nième nombre de Fibonacci.

Fibonacci V2

```
public int fibonacci(int n ){
  int i =2;
  int fiMoins2 = 0 ;        // f(0) = 0
  int fiMoins1 = 1 ;        // f(1) = 1
  int fi = 1 ;              // f(2) = 1
     for (int i =3 ; i < n+1 ; i++) {
                  // mise à jour de fiMoins2 et de fiMoins1
        fiMoins2= fiMoins1;
        fiMoins1= fi;       // calcul de fi
        fi= fiMoins2 + fiMoins1;
                  // fi est égal au ième terme de la suite
     } ;
           // fi est le n^ième terme de la suite pour tout n > 0
     if (n==0) return 0 ;
     else  return fi ;
  }
```

Complexité de la V2

- Cette fois la complexité est linéaire

Méthode Rapide

- On utilise une autre relation d'induction

$$F_{2k} = F_k^2 + F_{k+1}^2$$

$$F_{2k+1} = (2F_k + F_{k+1})F_{k+1}$$

- On décompose n en base 2

$$n = \sum_{i=0}^{p-1} \text{décomposition}[i]2^i$$

- La suite $d_0=1$, $d_i=2d_{i1}+\text{decomposition}(p-i)$, est telle que $d_p=n$. On calcule les f_{d_i}.

Calcul des nombres de Fibonacci V3

```
public int[] decompose(int n) {
  int p = ⌈log2 n⌉  ;
  int auxiliaire = n ;
  int[p] decomposition ;
  for (int indice = 0 ; indice < p,
  indice ++) {
    decomposition[indice] =  auxiliaire
                          mod 2 ;
    auxiliaire = auxiliaire / 2;
  }
  return decomposition ;
}
```

Calcul Des Nombres De Fibonacci V3

```
public int fibonacci (int n) {
   int a =0 ;int b =1 ;
   int p = ⌈log2 n⌉  ;
   int auxiliaire ;
   int [p] decomposition = decompose (n) ;
   for (int indice =1 ; indice < = p ;
                    indice++){
      auxiliaire = a ;
      a = a*a + b*b;
      b = (2*auxiliaire+b)*b;
      if (decompose(p-indice)==1 ) {
         b = a+b ; a = b-a ;
       }
       if (n== 1) return 1;
      else return a ;
   }
}
```

Analyse de la version 3

- Cette fois la complexité est en log(n)

Et la recherche dichotomique ?

- On va considérer un cas plus général

Solutions de type *diviser pour régner*

- Pour résoudre un problème de taille n on divise le problème en a problèmes de taille n/b et chaque sous-problème est résolu récursivement
- La phase de division et combinaison des résultats partiels a une complexité en f(n)

L ’équation de récurrence des solutions *diviser pour régner*

- T(1) = constante
- T(n) = a T(n/b) + f(n)

Théorème

- T(n) peut alors être borné asymptotiquement comme suit :
- Si $f(n)= O(n^{\log_b a-e})$ pour une constante $e>0$, alors $T(n) = \Theta(n^{\log_b a})$.
- Si $f(n)= \Theta(n^{\log_b a})$, alors $T(n) = O(\log n\, n^{\log_b a})$.
- Si $f(n)=\Omega(n^{\log_b a+e})$ pour une constante $e>0$, et si $af(n/b) < cf(n)$ pour une constante $c<1$ alors $T(n) = \Theta(f(n))$

Lemme 1

- $T(n)=T(b^k)=\Theta(n^{\log_b a})+\sum_{j=0}^{k-1} a^j f(n/b^j)$

- Posons $g(n)=\sum_{j=0}^{k-1} a^j f(n/b^j)$

Lemme 2

- Si $f(n)= O(n^{\log_b a-e})$ pour une constante $e>0$, alors $g(n))= O(n^{\log_b a})$.
- Si $f(n)= \Theta(n^{\log_b a})$,

  alors $g(n) =O(\log n\, n^{\log_b a})$.
- Si $af(n/b) < cf(n)$ pour une constante $c<1$

  alors $g(n) = \Theta(f(n))$

Si f(n)= O($n^{\log_b a-e}$) pour une constante e>0, alors g(n)= O($n^{\log_b a}$).

- On a alors

$$g(n) = \mathrm{O}\left(\sum_{j=0}^{k-1} a^j \left(\frac{n}{b^j} \right)^{(\log_b a)-e} \right)$$

- Or

$$\sum_{j=0}^{k-1} a^j \left(\frac{n}{b^j} \right)^{(\log_b a)-e} = n^{\log_b a-e} \sum_{j=0}^{k-1} \frac{a^j}{b^{\log_b a-e}}$$

$$= n^{\log_b a-e} \sum_{j=0}^{k-1} \left(b^e\right)^j = n^{\log_b a-e} \frac{n^e-1}{b^e-1}$$

Exemple d’application du cas 1

- Recherche dichotomique du maximum

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

Si f(n)= $\Theta(n^{\log_b a})$, alors g(n) =O ($\log n\, n^{\log_b a}$).

On obtient cette fois

Or

$$g(n) = \Theta\left(\sum_{j=0}^{k-1} a^j \left(\frac{n}{b^j} \right)^{(\log_b a)} \right)$$

$$\sum_{j=0}^{k-1} a^j \left(\frac{n}{b^j} \right)^{(\log_b a)} = n^{\log_b a} \sum_{j=0}^{k-1} \frac{a^j}{b^{(\log_b a) j}}$$

$$= n^{\log_b a} \sum_{j=0}^{k-1} 1^j = n^{\log_b a} k = n^{\log_b a} \log_b(n)$$

Exemple de ce cas

- Le tri dichotomique

- $c(n)=2c(n/2)+n$

Si af(n/b) < cf(n) pour une constante c<1 alors g(n) = Θ (f(n))

$$g(n) \le \sum_{j=0}^{k-1} a^j f\left(\frac{n}{b^j}\right) \le \sum_{j=0}^{k-1} c^j f(n) \le \frac{f(n)}{1-c}$$

On se propose de multiplier entre eux des *grands nombres*.

a) Si l'on utilise la méthode naïve, combien de multiplications élémentaires sont effectuées ?

Soient U et V deux nombres de 2n chiffres en base B.

On peut donc écrire

$U=U_1B^n+U_2$ et

$V=V_1B^n+ V_2$ où

U_1, U_2, V_1, V_2 sont des nombres à n chiffres en base B.

b) On utilise l'égalité :

$$(U_1 B^n + U_2)(V_1 B^n + V_2) =$$
$$U_1 V_1 B^{2n} + (U_1 V_2 + U_2 V_1) B^n + U_2 V_2$$

pour calculer récursivement la multiplication.

C'est à dire que l'on ramène le problème d'une multiplication de deux nombres de 2n chiffres à celui de 4 multiplications de deux nombres de n chiffres, 4 décalages et trois additions.

On suppose qu'additions et décalages s'effectuent en $\Theta(n)$. Établir une relation de récurrence permettant d'évaluer la complexité de cet algorithme récursif de multiplications et la résoudre.

c) On utilise maintenant l'égalité

$(U_1 B^n + U_2)(V_1 B^n + V_2) =$
$U_1 V_1 B^{2n} + ((U_1 - U_2)(V_2 - V_1) + U_2 V_2 + U_1 V_1 B^n + U_2 V_2$

pour calculer récursivement la multiplication. C'est à dire que l'on ramène le problème d'une multiplication de deux nombres de 2n chiffres à celui de 3 multiplications de deux nombres de n chiffres, 5 décalages et 6 additions. On suppose qu'additions et décalages s'effectuent en $\Theta(n)$. Établir une relation de récurrence permettant d'évaluer la complexité de cet algorithme récursif de multiplications et la résoudre.

On se propose dans cet exercice de calculer la complexité de plusieurs algorithmes dont le but est de fusionner les p listes triées de longueur n contenues dans un tableau de listes en une seule liste triée de longueur np.

On suppose définie une classe Liste contenant entre autre une méthode permettant de fusionner une liste l1 triée de longueur n1 et un liste triée l2 de longueur n2 dont la signature est

public static Liste fusion (Liste l1, Liste l2)

et la complexité est en $\Theta(n1+n2)$.

Déterminer la complexité de la méthode suivante en fonction de n et de p.

```
public static Liste fusionMultiple(Liste[]
                        mesListes) {
  Liste L=mesListes[1];
  for (int i=2; i < mesListes.length; i++){
     L= Liste.fusion(L,mesListes[i]);
  }
  return L;
}
```

On suppose maintenant que p est une puissance de 2 et l'on propose maintenant d'utiliser l'algorithme de multifusion récursif suivant :

Pour multifusionner p listes de taille n

Si p=2 utiliser fusion

Sinon

Multifusionner (récursivement) les p/2 première listes

Multifusionner (récursivement) les p/2 dernières listes

Utiliser fusion pour fusionner le résultat des deux premières étapes.

Soit c(n,p) = la complexité de la fusion de p listes de taille n par cette méthode.

Déterminez la relation de récurrence suivie par cette suite, ainsi que c(n,2).

- Posez d(n,p)=c(n,p)/n.
- Déterminez la relation de récurrence suivie par cette suite.
- Montrez que d(n,p) ne dépend pas de p. On pourra montrer par induction sur p que pour tout p >=2, d(n,p)=d(1,p) pour tout n >0.
- Posez d(1,p)=f(p), et déterminez l'équation de récurrence suivie par f(p). Résoudre cette équation. En déduire c(n,p).

- On considère le programme java récursif suivant où b est une constante entière
- On suppose défini un objet *table* à partir d'une classe Table dérivée de la classe Vector en y ajoutant la méthode

  `table.echanger (int i, int j)`

  qui échange table.elementAt(i) et table.elementAt(j).

```
public void T(int debut, int fin){
   // opère sur la Table  table dans la tranche
   table[debut..fin]
   int n=fin-debut+1 ;  // la dimension de la tranche
   if (n>1) {
      if (n=2) {
   // tri par ordre croissant des deux éléments de la tranche
         if (table.elementAt(debut) >
           table.elementAt(fin)){
           table.echanger(debut, fin) ;
         }
      }
      else {
         T( debut, debut+n/b) ;
         T( fin-n/b, fin) ;
         T(debut, debut+n/b) ;
      }
   }
}
```

- Établir la relation de récurrence vérifiée par la complexité de cet algorithme
- Si b=3/2, (dans ce cas bien sûr l'algorithme utilise la partie entière de n/b) quelle en est la complexité ?
- Question bonus : démontrer que si b=3/2, T est un tri