Année Spéciale 2013-2014

UNIVERSITÉ PARIS SUD
IUT D'ORSAY

TD : Complexité des algorithmes

Exercice 1

On considère deux manières de représenter ce que l'on appelle des « matrices creuses », c'est-à-dire des matrices d'entiers contenant environ 90% d'éléments nuls :

a) La matrice est représentée par un tableau à deux dimensions dont les cases contiennent les éléments.

b) La matrice est représentée par un tableau à une dimension. On ne s'intéresse qu'aux éléments de la matrice *qui ne sont pas nuls*. Chaque case du tableau contient un triplet (i, j, a) correspondant à l'indice de ligne, l'indice de colonne, et la valeur d'un élément non nul.

Le problème considéré consiste à calculer la somme des éléments d'une matrice. On demande d'écrire un algorithme permettant de calculer cette somme, pour chacune des deux représentations, puis de comparer leur complexité spatiale (espace mémoire occupé) et leur complexité temporelle (nombre d'opérations à effectuer). Que peut-on conclure de cette comparaison ? Montrer qu'il existe une valeur critique du nombre d'éléments non nuls à partir de laquelle une méthode l'emporte sur l'autre.

Exercice 2

On considère, pour effectuer la recherche d'un élément dans un tableau, la recherche séquentielle et la recherche dichotomique. On s'intéresse à leur complexité temporelle.
Pour cela, considérer un tableau ayant mille éléments (version trié, et version non trié). Pour chaque algorithme, et pour chaque version du tableau, combien de comparaisons sont à effectuer pour :

- trouver un élément qui y figure ?
- trouver un élément qui n'y figure pas ?

Quels sont les cas où le tableau est parcouru complètement et les cas où un parcours partiel est suffisant ?
Conclure en donnant la complexité temporelle pour chaque algorithme

Exercice 3

On considère un tableau à une dimension contenant des lettres majuscules. On désire compter la fréquence de chacune des 26 lettres de l'alphabet. Ecrire deux procédures qui donnent en sortie un tableau de fréquence: l'une où le tableau est parcouru 26 fois, et l'autre (plus performante !) où le calcul est fait en un seul parcours. On pourra supposer que l'on dispose d'une fonction auxiliaire *position(lettre)* qui pour chaque lettre donne sa position dans l'alphabet : *position('A') = 1, …, position('Z) = 26.*

Exercice 4

On considère trois tris élémentaires : le tri sélection, le tri par insertion (trois variantes), et le tri bulle. On considère pour un tableau les deux cas extrêmes où le tableau est déjà trié (dans l'ordre croissant), et celui où il est trié dans l'ordre décroissant. Décrire avec précision le comportement et la complexité de chacun des algorithmes dans ces deux cas. Quelles conséquences peut-on en tirer ?

PROPOSITION DE CORRIGE

Durée prévue : une séance

Exercice 1

a) tableau à deux dimensions

```
algo :  | somme := 0
        |  pour cptL := 1 à n faire              {pour chaque ligne}
        |        pour cptC := 1 à n  faire       {pour chaque colonne}
                       somme := somme + tab[cptL, cptC]
```

1. complexité spatiale : n * n
2. complexité temporelle : n* n sommes

b) tableau à une dimension

Donner un exemple de matrice presque vide et leur faire mettre sous forme de tableau à une dimension pour comprendre cette représentation de matrice.

```
algo :  | somme := tab[1].val
        | pour cpt := 2 à m faire somme := somme + tab[cpt].val
```

3. complexité spatiale : 3 * m
4. complexité temporelle : m-1

La complexité temporelle est toujours favorable à la représentation avec un tableau à 1 dimension.

La complexité spatiale l'est également tant que 3 * m < n * n, c'est à dire m < (n*n)/3.

Exercice 2 Revoir poly, transparents 33, 34, et 35. Les coefficients du polynôme sont mémorisés dans un tableau a.

a) Calcul de la valeur d'un polynôme en un point (1)

```
p := a[0]
pour  i :=1 à n faire
        xpi := puissance(x, i)
        p :=  p + a[i]* xpi
fpour
```

5. nombre d'additions : **n**
6. nombre de multiplications : pour calculer xpi : i-1 multiplications,
   pour calculer a[i]*xpi : 1 + (i-1) = i multiplications,
   donc, pour la boucle : 1 + 2 + 3 + … + i + … + n = **n(n+1)/2**

➔ **algorithme en O(n^2)**

b) Calcul de la valeur d'un polynôme en un point (2)

```
p := a[0]
```

```
xpi := 1
pour  i := 1 à n faire
        xpi := xpi * x faire p := p + a[i]* xpi
```

7. nombre d'additions : **n**
8. nombre de multiplications : **2n**

➔ **algorithme en O(n)**

c) Calcul de la valeur d'un polynôme en un point (3) (méthode de Homer)

```
p := a[n]
pour  i := n - 1 à 0, pas -1 faire p := p*x + a[i]
```

9. nombre d'additions : **n**
10. nombre de multiplications : **n**

➔ **algorithme en O(n)**

Exercice 3

Recherche d'un élément dans un tableau -- Revoir poly, transparents 36 et 37
Opérations élémentaires retenues: les **comparaisons**

1. Recherche séquentielle dans un tableau de 1000 éléments non trié
   11. élément ne s'y trouvant pas (complexité **au pire**) **:** 1000 comparaisons (tableau est parcouru complètement)
   12. élément qui s'y trouve (complexité **moyenne**) : n/2=500 comparaisons (parcours partiel suffisant)

➔ **algorithme en O(n)**

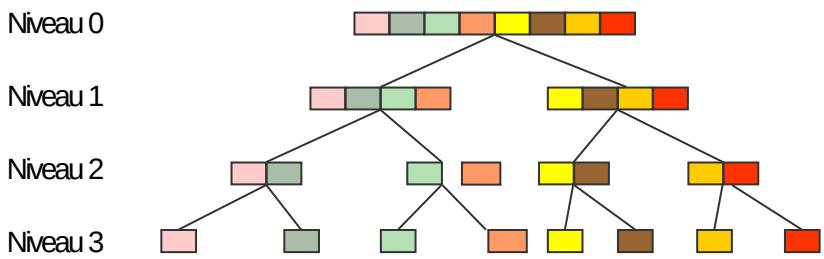
2. Recherche séquentielle dans un tableau trié
   13. élément ne s'y trouvant pas (complexité **au pire**) **:** 1000 comparaisons (parcours partiel suffisant)
   14. élément qui s'y trouve (complexité **moyenne**) : n/2=500 comparaisons (parcours partiel suffisant)

➔ **algorithme en O(n)**

3. Recherche dichotomique (dans un tableau non trié) : impossible

4. Recherche dichotomique (dans un tableau trié)
   15. complexité **au pire** = complexité **moyenne** = nombre p d'intervalles considérés

Exemple avec **n = 8** = 2^3 tableau de 8 éléments ➔ **3 comparaisons**



Profondeur de l'arbre de décision : de l'ordre de log2n
Donc pour un tableau de 1000 éléments, disons **1024** ➔ **10 comparaisons**

➔ **algorithme en O(log2n)**

Exercice 4 fréquence de chacune des 26 lettres dans un texte

La méthode qui parcourt le tableau 26 fois consiste à parcourir le texte pour compter d'abord tous les A, recommencer pour compter tous les B, etc. On utilise un tableau d'entiers de 26 cases, qui sert à mémoriser les occurrences.
La deuxième méthode demande de disposer de la conversion d'une lettre en un entier, qui correspond à sa place dans l'alphabet. (En c++, on ferait tout simplement appel à i = car – 'A').

```
char texte[TMAX]
int cpt, pos, occ[26]
pour cpt := 1 à 26 faire occ[cpt]:= 0 {initialisation à zéro}
cpt := 1
tant que (texte[cpt] ≠ CARFIN) faire
        pos :=  conversion(texte[cpt])
        occ[pos] := occ[pos] + 1
        cpt := cpt + 1
ftq
```

Exercice 5 Fibonacci

```
tab: tableau [0,n] d'entiers
tab[0] := 1 ; tab[1] := 1
pour i := 2 à n faire tab[i]=tab[i-1] + tab[i-2]
retourner tab[n]
```

16. complexité spatiale : n+1
17. complexité temporelle : n-2 additions

Variante :

```
tab: tableau [0,1] d'entiers
tab[0] := 1 ; tab[1] := 1
pour i := 1 à n/2   faire tab[0]=tab[0] + tab[1]
                          tab[1]=tab[0] + tab[1]
si n pair alors retourner tab[0] sinon retourner tab[1]
```

18. complexité spatiale : 2
19. complexité temporelle : n additions

Exercice 6 Les trois tris

Le tri sélection fait autant de comparaisons dans tous les cas. La seule différence tient au nombre de mises à jour du minimum.
Ce tri a cependant l'avantage de ne faire que peu d'échanges, ce qui peut être important pour des données de grande taille.

Le tri par insertion sous la dernière variante donnée en cours ne fait qu'un test par tour de boucle si le tableau est trié, d'où une complexité linéaire dans ce cas : avantage pour les tableaux presque triés.

Pour le tri bulle, le comportement dépend du fait que l'on ait ou pas modifié l'algorithme pour qu'il s'aperçoive qu'un tableau est trié (ce qui est le cas si aucune permutation n'a été faite au cours d'une passe).