Master 1 Informatique — Ducci 2
Mercredi 12 janvier 2022

Enseignants :
Oussama Messai / Serge Miguet
Université Lumière Lyon 2

Examen de Complexité

Durée : 1h30

Documents autorisés. Connexion à Internet interdite (smartphones et ordinateurs interdits)
Justifiez précisément chacune de vos réponses !

I. Analyse d'algorithme

Un programmeur débutant écrit l'algorithme suivant, pour vérifier si deux éléments d'un tableau A à n éléments sont égaux :

```
fonction DeuxEgaux(A[1..n])
    drapeau ← faux
    pour i allant de 1 à n faire
        pour j allant de 1 à n faire
            si A[i] = A[j] alors
                drapeau ← vrai
    renvoyer drapeau
```

1. Quelle est la complexité en temps de cet algorithme ?

Cet algorithme exécute deux boucles imbriquées dans lesquelles il passe n fois. Il effectue donc n^2 itérations du corps de la boucle la plus profonde, pour une complexité de $O(n^2)$ — quadratique.

2. Montrez que le programmeur s'est trompé, et que cet algorithme renvoie toujours `vrai` (sauf si le tableau est vide), même si tous les éléments du tableau sont différents.

Si le tableau est vide, on n'entre pas dans les boucles et le drapeau reste à faux. Sinon, le premier test effectué est : si A[1] = A[1], qui est toujours vrai, donc le drapeau passe à vrai et la fonction renvoie vrai.

3. Apportez une première modification (limitée à 2 à 4 caractères) à ce code pour qu'il ne renvoie `vrai` que si deux éléments du tableau sont égaux. Cette première modification diminue environ de moitié le nombre d'opérations effectuées (donnez le nombre exact de comparaisons effectuées), mais ne change pas sa complexité.

Une modification toute simple consiste à modifier la deuxième boucle en :

```
        pour j allant de i+1 à n faire
```

Ainsi, une case du tableau n'est jamais comparée à elle-même.
On peut par ailleurs se rendre compte que lorsque i vaut n, la boucle sur j n'est pas effectuée puisque n+1 est immédiatement supérieur à n. On peut faire une toute petite amélioration qui consiste à modifier aussi la première boucle de la manière suivante :

```
    pour i allant de 1 à n-1 faire
```

Cet algorithme fait bien ce qu'on lui demande et effectue n(n-1)/2 comparaisons, mais reste quadratique — $O(n^2)$.

4. En utilisant les résultats du cours, proposez un algorithme pour résoudre le même problème en complexité O(n.log n).

Il suffit de trier le tableau A avec un algorithme en O(n.log n) (par exemple HeapSort), puis de comparer en O(n) les n-1 paires d'éléments successifs dans le tableau. La complexité de cet algorithme est dominée par celle du tri, donc O(n.log n).

5. En supposant que les valeurs des éléments de A sont des nombres entiers compris entre 1 et n, proposez un algorithme qui résout le même problème en O(n).

L'algorithme suivant utilise un tableau de marques pour savoir si un élément du tableau a déjà été rencontré. Lorsqu'on est sur un élément, s'il est marqué, c'est qu'il est en double dans le tableau, sinon, on le marque. Cet algorithme effectue deux parcours en O(n), et est donc linéaire.

```
drapeau←faux
pour i allant de 1 à n faire
  | marque[i] ←faux
pour i allant de 1 à n faire
  | si marque[A[i]] alors drapeau←vrai
  | sinon
  |   | marque[A[i]]←vrai
renvoyer drapeau
```

6. En supposant que les valeurs des éléments de A sont des nombres entiers compris entre 1 et n-1, proposez un algorithme qui résout le même problème en O(1).

Si on a n nombres dont les valeurs sont comprises entre 1 et n-1, alors même si les n-1 premiers sont différents, le $n^{ème}$ est nécessairement égal à l'un des n-1 premiers. Il y a donc forcément une paire de nombres égaux. On peut renvoyer `vrai` en temps O(1), sans risque de se tromper.

II. Comparaison de temps d'exécution

1. Quelle est la valeur maximale de n pour laquelle un algorithme dont le temps d'exécution est $2n^2$ s'exécute plus vite qu'un algorithme dont le temps d'exécution est $100n$ sur la même machine ?

$2n^2 < 100n \Leftrightarrow n < 50$

Jusqu'à n=49, le premier algorithme va plus vite que le deuxième.

2. On veut comparer les implémentations du tri par insertion et du tri par fusion sur la même machine. Pour un nombre n d'éléments à trier, le tri par insertion demande $8n^2$ étapes alors que le tri par fusion en demande 64 $n.\log_2 n$. Quelles sont les valeurs de n pour lesquelles le tri par insertion l'emporte sur le tri par fusion (vous pouvez limiter n aux puissances de 2) ?

$8n^2 < 64n \log_2 n \Leftrightarrow n < 8 \log_2 n$

n	**1**	**2**	**4**	**8**	**16**	**32**	64	128
$\log_2 n$	0	1	2	3	4	5	6	7
$8 \log_2 n$	0	8	16	24	32	40	**48**	**56**

Pour les puissances de 2 inférieures ou égales à 32, le tri par insertion va plus vite que le tri fusion. Pour 64 et au-delà, c'est l'inverse (la résolution numérique de l'inéquation donne n < 43,559, mais il n'y a pas d'expression analytique simple pour obtenir cette valeur).

III. Analyse d'un compteur binaire

On dispose d'un compteur binaire de k bits, représenté à l'aide d'un tableau A[0..k-1]. Le bit de poids faible est stocké dans A[0] et le bit de poids fort est stocké dans A[k-1]. Par exemple, avec k=8 bits, le nombre décimal x=19 qui se décompose en $x=16 + 2 + 1 = 2^4 + 2^1 + 2^0$, est représenté à l'aide du tableau binaire suivant :

A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	1	0	0	1	1

On désire analyser la complexité de la fonction Incrémenter( ) dont le pseudo-code est donné ci-dessous[1] :

[1] la primitive *longueur*[A] renvoie en temps constant le nombre d'éléments du tableau A, c'est-à-dire k.

```
Incrémenter(A)
  i←0
  tant que i<longueur[A] et A[i]=1 faire
    A[i]←0
    i←i+1
  si i<longueur[A] alors
    A[i]←1
```

3. A partir du tableau A donné en introduction, illustrez cinq appels successifs à la fonction Incrémenter(A). Pour chacun de ces cinq appels, représentez le tableau A à la fin de l'exécution, indiquez le nombre d'itérations i_{max} effectuées dans la boucle **tant que**, et indiquez quel est le nombre décimal x représenté dans le tableau A.

n	A	i_{max}	x
	00010011		19
1	00010100	2	20
2	00010101	0	21
3	00010110	1	22
4	00010111	0	23
5	00011000	3	24

4. Quelle est la complexité en fonction de k, dans le meilleur des cas et dans le pire des cas, de l'appel à la fonction Incrémenter(A) ? donnez des exemples de valeurs de x pour lesquelles ces cas favorable et défavorable se produisent effectivement.

La complexité est égale (à une constante près) au nombre d'itérations effectuées dans la boucle **tant que**. Ce nombre est égal au nombre de 1 à droite de l'écriture binaire de x. Dans le meilleur des cas, ce nombre est à 0 et la complexité est O(1). Ce cas se produit pour tous les x pairs, dont l'écriture binaire se termine par un 0. Dans le pire des cas, ce nombre est égal à k et la complexité vaut O(k). Ce cas se produit lorsque les k bits sont égaux à 1, c'est à dire quand $x = 2^k - 1$.

5. Quelle est la plus grande valeur n de x, que l'on peut représenter dans le tableau A ?

La plus grande valeur n de x se produit lorsque les k bits de x sont à 1. On a donc $n = 2^k - 1$.

6. Reprenez la question 2 et montrez que si on exprime la complexité de la fonction Incrémenter( ) en fonction de n plutôt qu'en fonction de k, on obtient O(log n) dans le pire des cas.

On a établi que la complexité dans le pire des cas était égale à O(k). Or :

$2^k - 1 = n$ (d'après la question 3)
$2^k = n+1$ (en ajoutant 1)
$k = \log (n+1)$ (en prenant le log)

La complexité de la fonction Incrémenter( ) est donc égale à $O(\log (n+1)) = O(\log n)$

On désire maintenant évaluer la complexité T(n) de l'opération Énumérer(n) qui, à partir du nombre x=0, effectue n appels successifs à la fonction Incrémenter(A).

7. Montrez qu'une borne supérieure « pessimiste » permet d'affirmer que T(n) = O(n.log n).

Chacun des n appels à Incrémenter(A) peut être borné par O(log n). Comme on a n appels, cela donne une borne supérieure en O(n.log n).

Dans la suite de l'exercice, nous allons montrer que cette borne supérieure peut être affinée, et que l'on pourra établir que T(n) = O(n), ce qui est plus optimiste.

8. Montrer que pour estimer T(n), on peut compter le nombre total d'affectations effectuées à un élément de A[].

Le nombre d'opérations effectuées par la fonction incrémenter est égal à c1 + c2 * n, où n est le nombre d'itérations de la boucle. Le nombre d'affectations d'éléments de A est égal à 1 + n. Ces deux quantités sont du même ordre de grandeur.

9. Complétez le tableau suivant, dans lequel vous représenterez les 17 premiers appels à la fonction Incrémenter( ). t(x) est le nombre d'affectations à un élément de A[] effectuées lors de l'*exécution courante* de la fonction Incrémenter(x), et T(x) est le nombre total de ces affectations effectuées depuis le début de l'exécution de la fonction Énumérer(17), lors des *appels précédents* à la fonction Incrémenter( ).

x	A[7] … A[0]	t(x)	T(x)
0	00000000	1	0
1	00000001	2	1
2	00000010	1	3
3	00000011	3	4
4	00000100	1	7
5	00000101	2	8
6	00000110	1	10
7	00000111	4	11
8	00001000	1	15
9	00001001	2	16
10	00001010	1	18
11	00001011	3	19
12	00001100	1	22
13	00001101	2	23
14	00001110	1	25
15	00001111	5	26
16	00010000	1	31

10. Vérifiez expérimentalement que T(x) semble être majoré par une fonction linéaire de x.

Expérimentalement, il semble en effet que $t(x) < 2x$. Pour $x = 2^k$, on a l'impression que $t(x) = 2^{k+1}-1$

11. Pour quelles itérations le bit A[0] change-t-il de valeur ? Combien de fois cela se produit-il pour les n itérations ?

Le bit A[0] change à chaque itération, donc n fois en n itérations.

12. Pour quelles itérations le bit A[1] change-t-il de valeur ? Combien de fois cela se produit-il pour les n itérations ?

Le bit A[1] change une itération sur 2. Donc n/2 fois en n itérations.

13. En poursuivant ce raisonnement, montrez que le bit A[j] n'est affecté qu'une fois toutes les 2^j itérations.

Quand A[j] passe à 1 à une itération donnée, les j bits à sa droite viennent nécessairement de passer à 0. Il faut $2^j - 1$ itérations supplémentaires pour que ces j bits passent à 1, et ceci sans que A[j] ne soit modifié. C'est à l'itération suivante (donc au bout de 2^j itérations) que A[j] passera à 0. A la même itération, les j+1 bits allant de 0 à j sont passés à 0. Ce n'est qu'au bout de 2^j Nouvelles itérations que A[j] repasse à 1. A[j] est donc modifié toutes les 2^j itérations.

14. Montrez que le nombre total d'affectations à un bit de A[] dans la procédure Énumérer(n) est égal à :

$$T(n) = \sum_{j=0}^{k-1} \frac{n}{2^j}$$

Le bit numéro j est modifié une fois toutes les 2^j itérations. Comme on fait n itérations, le bit numéro j est modifié $n/2^j$ fois. Le nombre total de modifications de bits est donc égal à la somme pour toutes les valeurs de j, de $n/2^j$.

15. En déduire que $T(n) \leq 2n$.

$$T(n) = \sum_{j=0}^{k-1} \frac{n}{2^j} \leq n \sum_{j=0}^{k-1} \left(\frac{1}{2}\right)^j = n\left(\frac{\left(\frac{1}{2}\right)^k - 1}{\frac{1}{2} - 1}\right) = 2n\left(1 - \frac{1}{2^k}\right) = 2n - 2 \leq 2n$$

16. En supposant que l'ensemble des valeurs que peut prendre x sont équiprobables, donnez la complexité en moyenne de la fonction Incrémenter(x).

n itérations successives de l'opération Incrémenter( ) prennent un temps proportionnel à n. En moyenne, chacune prend donc un temps constant. Certaines d'entre elles prennent un temps logarithmique, mais elles sont tellement moins nombreuses que celles qui prennent un temps constant, que le temps est constant en moyenne.