Master 1 Informatique — Ducci 2
Vendredi 24 janvier 2020

Enseignants :
Carlos Crispim Junior / 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 !

Nous avons étudié pendant les CM, deux algorithmes de tri : le tri par bulles (*BubbleSort*), et le tri-fusion (*MergeSort*).

1. Rappelez les grands principes de ces deux algorithmes (sans nécessairement donner leur code ou leur pseudo-code), et explicitez leurs avantages et leurs inconvénients en termes de complexité en temps, et de complexité mémoire.

L'algorithme de tri par bulles parcourt l'ensemble des cases du tableau en comparant deux éléments successifs, et en les échangeant s'ils ne sont pas dans l'ordre. Après un premier passage, la seule certitude est que le plus grand élément se trouve à la fin du tableau. Il suffit de recommencer sur un tableau de taille n-1, puis n-2, … jusqu'à n'avoir plus qu'un tableau d'un élément sur lequel on n'a plus rien à faire. Sa complexité en temps est de $\Theta(n^2)$ et sa complexité en mémoire est de $O(1)$.
L'algorithme de tri fusion trie indépendamment l'une de l'autre les deux moitiés du tableau, grâce à deux appels récursifs. Les deux demi-tableaux sont ensuite interclassés, en recopiant dans un tableau auxiliaire, le plus petit élément non encore recopié de chacun des tableaux, jusqu'à ce que tous les éléments aient été recopiés. Sa complexité en temps est de $\Theta(n.log\,n)$, et sa complexité en mémoire est de $\Theta(n)$, à cause du tableau auxiliaire

Nous avons étudié pendant les TD, un autre algorithme de tri : le tri par tas (*HeapSort*).

2. Dans quelle mesure peut-on dire que ce troisième algorithme combine les avantages des deux algorithmes précédents, sans en avoir les inconvénients ?

La complexité en temps du tri par tas est de $\Theta(n.log\,n)$, ce qui est optimal pour un algorithme de tri par comparaisons et échanges, et sa complexité en mémoire est de $O(1)$: il trie sur place, sans nécessiter de tableau auxiliaire. Il présente donc les avantages de chacun des deux algorithmes précédents, sans en avoir les inconvénients.

L'objectif de la suite de ce problème est d'étudier en détails, un autre algorithme très célèbre, et peut-être même encore plus utilisé que le tri par tas : l'algorithme du Tri_Rapide (*QuickSort*), et ses variantes. On donne ci-dessous le pseudo-code de cet algorithme. `A` est un tableau de `n` nombres entiers :

```
Partition(A, p, r)
  x←A[r]
  i←p
  pour j de p à r-1 faire
    si A[j] ≤ x alors
      Echange(A, i, j)
      i←i+1
  Echange(A, i, r)
  Renvoyer i
```

```
Tri_Rapide_Recursif(A, p, r)
  si p < r alors
    q←Partition(A, p, r)
    Tri_Rapide_Recursif(A, p, q-1)
    Tri_Rapide_Recursif(A, q+1, r)


Tri_Rapide(A)
  Tri_Rapide_Recursif(A,0,Taille(A)-1)
```

3. Ecrivez le pseudo-code de la fonction `Echange(A, i, j)`, qui échange le contenu des éléments `i` et `j` du tableau `A`. Quelle est sa complexité ?

```
Echange(A, i, j)
  aux←A[i]
  A[i]←A[j]
  A[j]←aux
```

Ce code effectue un nombre constant d'opérations. Sa complexité est de $O(1)$.

4. Explicitez le rôle de la fonction `Partition()`. Illustrez chacune des étapes de son fonctionnement sur le tableau `A` représenté ci-dessous, à `n=9` éléments, lors de l'appel `q←Partition(A,0,8)`. Complétez le contenu du tableau `A` pour chaque valeur de `j`, ligne par ligne, à la fin de l'itération numéro `j`. La case numéro `j` est grisée. Vous indiquerez la valeur de `i` en entourant le chiffre de la case numéro `i`.

Comme son nom l'indique, la fonction Partition() partitionne le tableau en deux sous-parties : la première partie comprend les éléments inférieurs ou égaux au pivot (ici, la dernière case du tableau, qui vaut 6). La deuxième partie comprend les éléments supérieurs au pivot. Le fonctionnement des étapes 3 à 7 sont illustrées ci-dessous, et l'état final est représenté en dernière ligne.

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
État initial	8	5	10	2	3	4	12	4	6
j=0	8	5	10	2	3	4	12	4	6
j=1	5	8	10	2	3	4	12	4	6
j=2	5	8	10	2	3	4	12	4	6
j=3	5	2	10	8	3	4	12	4	6
j=4	5	2	3	8	10	4	12	4	6
j=5	5	2	3	4	10	8	12	4	6
j=6	5	2	3	4	10	8	12	4	6
j=7	5	2	3	4	4	8	12	10	6
Etat final	5	2	3	4	4	6	12	10	8

5. Dans l'exemple de la question 4, quelle est la valeur de retour de la fonction, qui est affectée à `q` et quel est le contenu de la case `A[q]` du tableau ? A la fin de l'exécution de la fonction, que peut-on dire des valeurs des éléments du tableau placés avant l'indice `q` ? et de ceux placés après l'indice `q` ? Quels sont les paramètres des deux appels récursifs `Tri_Rapide_Recursif()` qui suivent l'appel à la fonction `q←Partition(A,0,8)` ? Justifiez le fait que l'élément `A[q]` a trouvé sa place définitive dans le tableau, et ne changera plus de position dans la suite de l'algorithme.

La fonction renvoie la dernière valeur prise par i, c'est à dire 5. C'est la place où se range le pivot (choisi ici comme la dernière case du tableau) autour duquel on partitionne le tableau. On a donc A[q] = A[5] = 6. La première partie du tableau (avant q) contient les valeurs inférieures ou égales au pivot, et la deuxième partie (au-delà de q), les valeurs supérieures au pivot. Les deux appels récursifs qui suivent sont Tri_Rapide_Recursif(A,0,4) et Tri_Rapide_Recursif(A,6,8). Le premier va trier la première partie du tableau (cases allant de 0 à 4) le second va trier la deuxième partie du tableau (cases allant de 6 à 8). Dans les appels récursifs ultérieurs, les bornes de l'intervalle sur lequel agit Tri_Rapide_Recursif ne font que restreindre l'intervalle qui lui est transmis. Or ni l'un ni l'autre des deux intervalles gérés par les deux appels récursifs ne contiennent q. La case q du tableau ne sera donc plus jamais modifiée par les appels récursifs, et l'élément A[q] ne sera plus jamais déplacé.

6. Quelle est la complexité, en fonction de `n`, de l'appel de la fonction `q←Partition(A, 0, n-1)` ?

Outre les opérations en temps constant qui sont effectuées avant et après la boucle, la fonction effectue n-1 passages dans la boucle (pour j allant de 0 à n-2). Chacune des étapes effectue un test, et parfois un échange, qui s'effectuent en temps constant. La complexité de cet appel est donc en O(n) (complexité linéaire).

D'un point de vue de la complexité en temps, le scénario idéal se produit lorsque la fonction `Partition()` partage approximativement le tableau `A` de taille `n`, en deux sous-tableaux de taille `n/2`. Pour que cela se produise, il faut que l'élément `x`, que l'on appelle le *pivot*, initialisé ici à `A[r]`,

corresponde à la médiane du tableau, c'est à dire un élément qui comprendrait (à un près, si cela ne tombe pas juste) autant d'éléments qui lui sont inférieurs, que d'éléments qui lui sont supérieurs.

7. En supposant que ce scénario idéal se produit à chaque appel récursif, écrire l'équation récurrente qui exprime la complexité `T(n)` du tri rapide de `n` éléments en fonction de `T(n/2)`. Quelle est la solution de cette équation ? (vous pouvez démontrer ce résultat, ou utiliser le théorème central vu en cours pour répondre à cette question).

Si chacun des deux appels récursifs concerne un tableau de taille n/2, on a : $T(n) = 2\ T(n/2) + n\ c_1$, et $T(1) = c_2$, avec c_1 et c_2 des constantes.

On peut faire des substitutions successives dans cette équation récurrente et obtenir :

$T(n) = 2\ (2\ T(n/4) + n/2\ c_1) + n\ c_1 = 4\ T(n/4) + 2\ n\ c_1$

$= 4\ (2\ T(n/8) + n/4\ c_1) + 2n\ c_1 = 8\ T(n/8) + 3\ n\ c_1$

…

En poursuivant ces substitutions jusqu'au rang k, on obtient : $T(n) = 2^k\ T(n/2^k) + k\ n\ c1$. On continue ainsi jusqu'à atteindre une valeur de k pour laquelle on connaît $T(n/2^k)$. En effet, dès que $2^k >= n$, on tombe soit sur un tableau vide, soit sur un tableau à 1 élément, pour lequel le temps du tri est égal à la constante c_2. En prenant le logarithme de cette inégalité, on obtient $k>=log\ n$. En particulier, quand $k=log\ n$, on a $2^k = n$. L'équation s'écrit alors $T(n) = n\ c_2 + log\ n\ n\ c_1 = \Theta(n\ log\ n)$.

On obtient bien sûr le même résultat en utilisant le cas 2 du théorème central, qui s'applique quand $T(n)=aT(n/b) + f(n)$. Il suffit en effet de prendre a=2, b=2 et $f(n) = c_1\ n$. Le cas 2 peut en effet être utilisé quand $f(n) = \Theta(n^{log_b(a)})$. Or, $log_2(2) = 1$, et donc $f(n)= \Theta(n^1)= \Theta(n)$. Dans ce cas, le théorème affirme que $T(n) = \Theta(n^{log_b(a)} \times log\ n) = \Theta(n\ log\ n)$.

8. Le pire des cas se produit quand l'appel à la fonction `Partition()` partage à chaque étape, le tableau `A` en un tableau de taille `n-1`, et un autre tableau vide. Écrire alors `T(n)` en fonction de `T(n-1)`, et résoudre l'équation récurrente dans ce pire des cas.

Dans ce cas, on obtient : $T(n) = T(n-1) + c_1\ n$. En effectuant les substitutions successives, on obtient cette fois :

$T(n) = T(n-2) + c_1\ (n-1) + c_1\ n = T(n-2) + c_1\ (n-1 + n)$

$= T(n-3) + c_1\ (n-2) + c_1\ (n-1 + n) = T(n-3) + c_1\ (n-2 + n-1 + n)$

…

En poursuivant ainsi jusqu'au rang n-1, on obtient :

$T(n) = T(n-(n-1)) + c_1\ \sum_{i=1}^{n} i = T(1) + c_1\ n\ (n+1)/2$

$= \Theta(n^2)$

Même si l'appel à la fonction `Partition()` partage systématiquement le tableau en deux parties inégales (par exemple, 90% et 10% des cases du tableau), on peut montrer que la complexité de l'algorithme reste intéressante, et plus proche de la complexité de la question 7 que de celle de la question 8. En moyenne, sur un tableau quelconque, on estime que les proportions sont plutôt de l'ordre de 80% et 20%, pour la taille des deux sous-tableaux.

9. Montrez en revanche que si on donne en entrée à l'algorithme un tableau trié (qui n'est donc pas un tableau quelconque), on se trouve à chaque étape dans le cas défavorable évoqué en question 8.

Si on donne en entrée à l'algorithme un tableau trié, le pivot choisi par l'algorithme, qui est la dernière case du tableau, se trouve être la plus grande case du tableau. Tous les autres éléments du tableau sont plus petits que lui, et on se trouve dans le cas défavorable où la partition construit un tableau de taille n-1 et un tableau de taille vide.

Nous disposons donc d'une version de l'algorithme Tri-Rapide, dont la complexité est $\Theta(n.Log\ n)$ en moyenne, mais $\Theta(n^2)$ dans le pire des cas. Nous allons nous intéresser, dans la

suite de ce problème, à la sélection de la médiane en temps linéaire, qui permet d'obtenir un algorithme de complexité $\Theta(n.Log\ n)$ même dans le pire des cas.

10. Quelle est la médiane du tableau A de la question 4 ?

Le tableau A trié est [2, 3, 4, 4, 5, 6, 8, 10, 12]. Sa médiane est l'élément 5. Il y a en effet exactement 4 éléments qui sont plus petits que lui, et 4 éléments qui sont plus grands que lui.

11. Montrez que si l'on savait où se trouve la médiane dans le tableau, il suffirait d'un seul appel supplémentaire à la fonction `Echange()` pour se trouver dans la situation favorable de la question 7.

En effet, si on sait que la médiane se trouve en case k du tableau, il suffit, avant d'appeler la fonction q←Partition(A,p,r), d'un appel à la fonction Echange(A,k,r) pour échanger la médiane avec le dernier élément du tableau. Ce dernier élément est choisi comme pivot par la fonction Partition() et contient alors la médiane. Le partitionnement se passe donc de manière idéale, avec deux sous-tableaux de taille moitié.

12. Une première amélioration à l'algorithme consiste à le « randomiser », c'est à dire à choisir comme pivot, lors du partitionnement non pas forcément la dernière case du tableau, mais une case aléatoire à l'intérieur du tableau. Montrer comment modifier `Partition()` pour obtenir la fonction `Partition_Randomisee()` qui implémente cette idée (on pourra utiliser la fonction `Rand(a, b)` qui renvoie un nombre entier aléatoire compris entre `a` et `b` inclus.

```
Partition_Randomisee(A, p, r)
    k←Rand(p, r)                  # choisit un pivot aléatoire
    Echange(A, k, r)              # place ce pivot en dernière case
    renvoyer(Partition(A, p, r))  # utilise la fonction Partition() précédente
```

En cas de malchance, on obtient (très rarement) un algorithme en $\Theta(n^2)$, mais il n'existe plus *une entrée particulière* qui génère ce comportement, mais seulement *une séquence malchanceuse* de tirages au sort pour laquelle cela se produit. En moyenne, sur un grand nombre d'exécutions, la complexité est de $\Theta(n.Log\ n)$.

Par la suite, plutôt que traiter le problème de la détermination de la médiane, nous allons traiter le problème de la sélection de l'élément de rang `i`, c'est à dire de l'élément `x` tel qu'il existe exactement `i` éléments inférieurs ou égaux à `x` dans le tableau.

13. Pour quelle valeur de `i` faudra-t-il appeler cet algorithme pour obtenir la médiane du tableau ?

Si n est impair, alors, n-1 est pair, et il y a exactement (n-1)/2 éléments qui sont plus petits, et (n-1)/2 éléments qui sont plus grands que la médiane, qui se trouve donc au rang (n-1)/2 du tableau. Si n est pair, alors, il faut faire un choix entre les deux éléments qui sont « presque » au milieu du tableau : celui de rang n/2 ou celui de rang n/2 – 1 = ⌊(n-1)/2⌋, où ⌊x⌋ désigne la partie entière de x. Arbitrairement, on prend le plus petit des deux. Dans les deux cas (n pair ou n impair), on peut utiliser la formule i = ⌊(n-1)/2⌋ (mais le choix i = ⌊n/2⌋ est une réponse qui convient également).

14. Montrer qu'en utilisant un algorithme bien connu, on peut résoudre ce problème avec une complexité $\Theta(n\ Log\ n)$.

On peut bien sûr trier le tableau avec le meilleur algorithme connu (par exemple, le tri par tas), et sélectionner la case numéro `i` du tableau trié, qui répond au problème. Cet algorithme a une complexité $\Theta(n\ Log\ n)$.

On va maintenant chercher à résoudre le problème de la sélection de l'élément de rang i en temps linéaire.

15. Proposer un algorithme de complexité O(n) qui résolve le problème de la sélection pour i=0, (donc, trouver le plus petit élément de A) et pour i=1 (donc, trouver le deuxième plus petit élément de A). Indication : chercher simultanément les deux plus petits.

```
PlusPetit(A)
a←A[0]
pour i de 1 à n-1 faire
    si A[i] < a alors
        a←A[i]
renvoyer a
```

Recherche du plus petit élément

```
DeuxiemePlusPetit(A)
a←A[0]
b←A[1]
si b<a alors
    echanger(a,b)
pour i de 2 à n-1 faire
    si A[i] < a alors
        b←a
        a←A[i]
    sinon si A[i] < b alors
        b←A[i]
renvoyer (b)
```

Recherche du deuxième plus petit élément

Ces deux algorithmes parcourent le tableau à l'aide d'une unique boucle « pour ». Sur chaque case (pour chaque valeur de `i`) ils effectuent un nombre constant d'opérations. On obtient donc bien dans les deux cas un algorithme en $O(n)$, même si le deuxième algorithme effectue plus d'opérations que le premier.

Une autre solution consisterait à effectuer les deux premières étapes d'un tri par insertion, ou d'un tri par sélection, qui calculeraient d'abord le plus petit, puis le deuxième plus petit élément. Mais ces solutions effectueraient deux parcours complets du tableau, et seraient donc un peu plus coûteux que celui proposé.

Le même principe permettrait d'obtenir un algorithme en O(n) pour calculer l'élément de rang i, tant que i est une constante (par exemple, l'élément de rang 10, en effectuant 10 parcours du tableau). Mais si on essaye d'appliquer cet algorithme avec $i = \lfloor (n-1)/2 \rfloor$, on obtient un algorithme en $O(n^2)$.

Lorsque l'on appelle la fonction `Partition_Randomisee()`, on obtient en sortie le rang q du pivot. Si par hasard q égal à i, on a résolu le problème de la sélection de l'élément de rang i. Sinon, en comparant q et i, on peut savoir dans quel sous-tableau (à gauche ou à droite de q) rechercher le $i^{ème}$ plus petit élément.

16. En utilisant cette idée, proposer une version récursive de la fonction `Selection_Randomisee(A, p, r, i)`, qui renvoie le $i^{ème}$ plus petit élément du sous-tableau de A, dont les indices sont compris entre p et r. Attention au rang dans le sous-tableau qui n'est en général, pas égal au rang dans le tableau complet !

```
Selection_Randomisee(A, p, r, i) {
si (p = r) alors          // si un seul élément dans le tableau…
      renvoyer A[p];      // on le retourne
q ← Partition_Randomisee(A, p, r); // q=indice du pivot
      k ← q-p;            // k = rang de q dans le sous-tableau entre p et r
si (i=k) alors            // si on est sur celui que l'on recherche
      renvoyer A[q];      // on renvoie sa clef
sinon si (i<k) alors      // si on en a trouvé un qui était au-delà de i
      renvoyer Selection_Randomisee(A, p, q-1, i);
                          // on recherche en dessous de q, l'élément de rang i
Sinon
      renvoyer Selection_Randomisee(A, q+1, r, i-(k+1));
                          // on recherche au-dessus de q. Attention : le rang de i
                          // dans le sous-tableau qui commence à q+1 est i-(k+1)
                          // parce que tous les k+1 éléments compris entre p et q
                          // sont plus petit que i
```

17. En reprenant l'hypothèse de la question 7, écrivez l'équation récurrente, qui relie T(n) à T(n/2), et résolvez-là dans le cas le plus favorable (qui est également le plus proche du cas

moyen). Dans le cas le plus défavorable, reliez T(n) à T(n-1), et déduisez-en la complexité dans le pire des cas.

Dans le code de `Selection_Randomisee()`, on a deux appels récursifs. Mais contrairement à `Partition_Randomisee()` qui les exécute tous les deux, soit un seul des deux appels récursifs est réellement exécuté, soit aucun des deux. Dans ce cas favorable, on a donc $T(n) \leq T(n/2) + n\ c_1$. En effectuant les substitutions successives, on obtient :

$T(n) \leq T(n/4) + n/2\ c_1 + n\ c_1 = T(n/4) + (1 + ½)\ n\ c_1$

$\leq T(n/8) + n/4\ c_1 + (1 + ½)\ n\ c_1 = T(n/8) + (1 + ½ + ¼)\ n\ c_1$

En poursuivant ces substitutions jusqu'au rang k, on obtient : $T(n) \leq T(n/2^k) + \sum_{i=0}^{k-1} \frac{1}{2^i}\ n\ c_1$. En utilisant la formule vue en cours sur la somme des puissances de x (en prenant x = ½), on obtient que la somme vaut $(½^k - 1) / (½ - 1) = 2 - ½^{k-1}$, qui reste toujours plus petite que 2. Pour la même raison qu'en question 7, on s'arrête quand k = *log* n. On obtient donc que $T(n) \leq c_2 + 2\ n\ c_1 = O(n)$.
On pourrait également directement appliquer le cas 3 du théorème central, avec a=1, b=2 et ε=1 pour obtenir que T(n) = O(n).

En revanche, dans le cas défavorable où la partition crée un tableau de taille n-1, on a $T(n) = T(n-1) + c_1\ n$. C'est la même équation récurrente qu'en question 8, on obtient donc une complexité dans le pire des cas de $T(n) = O(n^2)$.

Pour les mêmes raisons que précédemment, on obtient maintenant un algorithme linéaire *en moyenne* pour résoudre le problème de la sélection du $i^{ème}$ élément d'un tableau. Mais il reste possible (quoique très improbable) que cet algorithme effectue un grand nombre d'opérations.

Sachez qu'il existe un algorithme déterministe et linéaire même dans le pire des cas, pour la sélection du $i^{ème}$ élément, mais ce problème est déjà bien assez long pour un examen de 1h30, et nous avons eu pitié de vous…