Master 1 Informatique — Ducci 2
Mercredi 9 janvier 2019

Enseignants :
Mehdi AYADI / Carlos CRISPIM JUNIOR / Serge MIGUET
Université Lumière Lyon 2

Examen de Complexité

Durée : 1h30

Documents autorisés. Connexion à Internet interdite (smartphones et ordinateurs interdits)

Questions de cours

Vous pouvez justifier vos réponses en donnant les algorithmes qui résolvent les problèmes proposés, et en explicitant la raison pour laquelle on ne peut pas trouver mieux.

Étant donné un tableau T de n éléments arbitraires :

1) Quelle est la complexité, *dans le pire des cas*, de la recherche de l'élément x dans le tableau T :
    a. Si T est quelconque ?

Il faut nécessairement parcourir tout le tableau pour rechercher x. Le pire des cas se produit si x est absent du tableau (ou s'il correspond au dernier élément consulté). La complexité dans le pire des cas est donc O(n).

    b. Si T est trié du plus petit au plus grand élément ?

On peut mettre en place une recherche dichotomique, et diviser par deux à chaque itération le nombre d'éléments candidats susceptibles d'être égaux à x. Là aussi, le pire des cas se produit lorsque x est absent du tableau (ou trouvé à la dernière itération). Cet algorithme a une complexité en O(log n).

2) Mêmes questions pour la complexité de ce problème, *dans le meilleur des cas* ?

Si on a de la chance, on trouve x dès la première itération (aussi bien dans le cas a. que dans le cas b.). On a donc un algorithme en O(1) dans le meilleur des cas.

3) Quelle est la complexité, *dans le pire des cas*, de la recherche du plus petit élément y de T ?
    a. Si T est quelconque ?

Il faut là aussi parcourir tous les éléments de T, pour une complexité de O(n).

    b. Si T est trié du plus petit au plus grand élément ?

Le plus petit élément est le premier élément du tableau, qui peut être obtenu en temps constant. La complexité est donc O(1).

4) Mêmes questions pour la complexité de ce problème, *dans le meilleur des cas* ?

Même dans le cas le plus favorable, il faut parcourir tous les éléments du tableau (s'il est non trié) à la recherche du plus petit. La complexité du cas a. reste O(n). Celle du cas b. est également O(1).

5) Quelle est la complexité algorithmique du problème du tri du tableau T, en ne s'autorisant, pour seules opérations sur les éléments du tableau que la comparaison et l'échange de deux éléments du tableau ?

La complexité algorithmique du problème du tri par comparaisons-échanges est de O(n.log n). Une preuve consiste à considérer l'ensemble des permutations d'un ensemble à n éléments. Cet ensemble est de cardinal n ! (factorielle n). Le tableau initial est l'un de ses éléments. Le tableau final en est un autre. Tout algorithme de tri qui procède par comparaisons-échanges, diminue à chaque comparaison le nombre de permutations possibles, pour arriver à la fin de l'algorithme à une solution dans laquelle il n'y a plus qu'une seule permutation possible (le tableau trié). Dans le cas le plus favorable, le nombre de permutations possibles est divisé par deux à chaque comparaison. Le nombre minimal de

comparaisons à effectuer est k = $\log_2(n !)$ (combien de fois peut-on diviser n ! par 2 jusqu'à obtenir quelque chose de plus petit que 1 ?). La formule de Stirling établit que $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$. On en déduit que log(n !) = O(n.log n).

Nous avons étudié et comparé pendant le cours et les travaux dirigés, six algorithmes de tri qui n'effectuent que des comparaisons et des échanges : Tri par bulles, Tri fusion, Tri par insertion, Tri rapide, Tri par sélection, Tri par tas. Pour trier des tableaux de n éléments, certains de ces algorithmes ont une complexité algorithmique en $O(n^2)$, d'autres une complexité en O(n.log n). Certains ont une complexité mémoire de O(n), d'autres ont une complexité mémoire de O(1).

Saurez-vous identifier (en justifiant vos choix) ces six algorithmes par ce jeu de devinettes ?

a) J'ai une complexité algorithmique *en moyenne* de O(n.log n), mais *dans le pire des cas* de $\Theta(n^2)$. Qui suis-je ?

Je suis le tri rapide (*Quicksort*), qui se comporte bien (O(n.log n)) sur la plupart des tableaux, mais mal sur les tableaux triés ou presque triés (Θ (n^2)). Ces tableaux triés ou presque triés étant rares comparé à l'ensemble des permutations possibles, la complexité en moyenne est malgré tout de O(n.log n).

b) J'ai une complexité algorithmique de O(n.log n) et une complexité mémoire de O(1). Qui suis-je ?

Je suis le tri par tas (*Heapsort*), seul algorithme de tri étudié à avoir à la fois une complexité asymptotiquement optimale, et une complexité mémoire constante.

c) J'ai une complexité algorithmique de O(n.log n), mais j'ai besoin d'un tableau auxiliaire de taille O(n) pour pouvoir fonctionner. Qui suis-je ?

Je suis le tri-fusion (*Merge Sort*). La partition du tableau en deux sous-tableaux indépendants nécessite en effet un espace auxiliaire de O(n) éléments.

d) Je fais au plus n-1 échanges, quel que soit le tableau qui m'est donné en entrée. Qui suis-je ?

Je suis le tri par sélection. En effet, à chaque itération de la boucle externe, on recherche le plus petit élément du sous-tableau restant à trier, et on ne fait qu'un seul échange pour le mettre à sa place. Il y a donc $O(n^2)$ comparaisons, mais seulement O(n) échanges (n-1 au plus, car il n'y a pas d'échange à faire pour le sous-tableau à 1 élément).

e) Je fais systématiquement exactement n(n-1)/2 comparaisons, quel que soit le tableau qui m'est donné en entrée. Je fais un nombre d'échanges compris entre 0 et n(n-1)/2 en fonction du tableau qui m'est donné en entrée. Qui suis-je ?

Je suis le tri par bulles, qui compare systématiquement toutes les paires d'éléments successifs, dans des sous tableaux de taille décroissante allant de n à 2. Si le tableau est trié, il n'y a aucun échange à faire. Si le tableau est trié à l'envers, il y a un échange à chaque comparaison.

f) Si on me donne en entrée un tableau trié, je ne fais que n-1 comparaisons, et aucun échange. Qui suis-je ?

Je suis le tri par insertion, qui cherche à chaque itération la place où insérer l'élément courant dans la partie triée du tableau… et qui se rend compte en un seul test qu'il est déjà à sa place.

Tri par paquets

Lorsque l'on fait certaines hypothèses sur les données que l'on doit trier, et que l'on se donne d'autres instructions que les comparaisons et les échanges, on peut obtenir des algorithmes plus efficaces que les algorithmes de tri généralistes. C'est le cas de l'algorithme de tri par paquets, dans lequel on va faire une hypothèse sur la distribution des données. On utilisera ensuite la valeur d'une donnée source pour placer cette donnée à un endroit particulier d'un tableau destination (opération élémentaire qui

n'existe pas dans les algorithmes de tri par comparaison et échange). La suite de ce problème se propose d'étudier cet algorithme.

On suppose disposer d'un tableau A contenant n nombres aléatoires compris entre 0 et 1. On suppose que ces nombres suivent une distribution uniforme, c'est à dire que tous les intervalles de même longueur contiennent approximativement le même nombre de points. Dans l'exemple suivant, on a tiré n=10 nombres aléatoires, compris entre 0 inclus et 1 exclus. Le tableau A initial vaut <0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68>. On découpe l'intervalle [0, 1[en n=10 sous-intervalles de longueurs égales [0, 0.1[, [0.1, 0.2[, et ainsi de suite jusqu'à [0.9, 1[, et on place chaque nombre du tableau A dans le sous-intervalle qui lui est associé. On crée pour cela un tableau B dont les éléments sont des listes de nombres issus du tableau A, et tombant dans le même sous-intervalle.

	A
0	0.78
1	0.17
2	0.39
3	0.26
4	0.72
5	0.94
6	0.21
7	0.12
8	0.23
9	0.68

Étape 1 : remplir les listes de B

	B		
0			
1	0.17	0.12	
2	0.26	0.21	0.23
3	0.39		
4			
5			
6	0.68		
7	0.78	0.72	
8			
9	0.94		

Étape 2 : trier les listes de B

	B		
0			
1	0.12	0.17	
2	0.21	0.23	0.26
3	0.39		
4			
5			
6	0.68		
7	0.72	0.78	
8			
9	0.94		

Étape 3 : concaténer les listes de B

	B
0	0.12
1	0.17
2	0.21
3	0.23
4	0.36
5	0.39
6	0.68
7	0.72
8	0.78
9	0.94

1) Illustrez l'action du tri par paquets sur le tableau A=<0.79, 0.13, 0.16, 0.64, 0.39, 0.20, 0.89 0.53, 0.71, 0.42>

	A
0	0.79
1	0.13
2	0.16
3	0.64
4	0.39
5	0.20
6	0.89
7	0.53
8	0.71
9	0.42

Étape 1 : remplir les listes de B

	B		
0			
1	0.13	0.16	
2	0.20		
3	0.39		
4	0.42		
5	0.53		
6	0.64		
7	0.79	0.71	
8	0.89		
9			

Étape 2 : trier les listes de B

	B		
0			
1	0.13	0.16	
2	0.20		
3	0.39		
4	0.42		
5	0.53		
6	0.64		
7	0.71	0.79	
8	0.89		
9			

Étape 3 : concaténer les listes de B

	B
0	0.13
1	0.16
2	0.20
3	0.39
4	0.42
5	0.53
6	0.64
7	0.71
8	0.79
9	0.89

2) Donnez un algorithme s'exécutant en O(n) et permettant d'effectuer l'étape 1[1].

Pour i **allant de** 0 **à** n-1 **faire**
    B[i] ← Ø // *liste vide*
Pour i **allant de** 0 **à** n-1 **faire**
    Ajouter A[i] à la liste B[int(A[i] * n)]

[1] On pourra utiliser la fonction int(x) qui permet de calculer la partie entière du nombre à virgule x.

3) Pour l'étape 2, on suppose que B[i] contient une liste de taille n_i. L'hypothèse sur l'uniformité de la distribution permet en moyenne, d'assimiler n_i à une constante indépendante de n. On peut prendre n'importe quel algorithme de tri quadratique (par exemple, le tri par insertion, qui fonctionne très bien pour des petits tableaux). Sous cette hypothèse, quelle est la complexité du tri de l'une des lignes du tableau B ? Et de l'étape 2 dans son ensemble ?

La complexité du tri de la ligne i est $O(n_i^2)$, c'est à dire une constante indépendante de n, donc O(1). Comme il y a n ligne à trier, l'étape 2 a une complexité O(n).

4) Donnez un algorithme linéaire permettant d'effectuer l'étape 3.

k←0 // *indice parcourant le tableau C, qui contiendra le tableau final. On pourrait aussi réutiliser A*
pour i **allant de** 0 **à** n-1 // *indice parcourant les listes de B*
    pour j **allant de** 0 **à** $n_i - 1$ // *indice parcourant la liste B[i]*
        C[k] ← B[i][j]
        k← k+1

Cet algorithme est composé de deux boucles imbriquées, mais la boucle la plus profonde n'est effectuée en tout que $\sum_{i=0}^{n} n_i$ fois, c'est à dire n. On n'a donc bien un algorithme en O(n).

5) En conclure que l'algorithme complet s'exécute en moyenne en O(n).

Les étapes 1, 2 et 3 s'exécutent en O(n). On a donc un algorithme complet de complexité O(n).

6) Quel est le temps d'exécution de l'algorithme, dans le cas le plus défavorable ? quelle modification simple permettrait à l'algorithme de garder son temps d'exécution moyen linéaire, tout en obtenant un temps d'exécution de O(n.log n) dans le pire des cas ?

Le cas le plus défavorable se produit lorsque l'une des listes de B contient n éléments. Si on a effectivement utilisé un algorithme quadratique pour le tri des listes de B à l'étape 2, on obtient un algorithme en $O(n^2)$. Il suffit d'utiliser un algorithme en O(n.log n) pour le tri des listes de B de l'étape 2, pour que notre algorithme ait un pire des cas en O(n. log n).

[bonus] Si on gère les listes de B dans des tableaux créés au début de l'exécution de l'algorithme, et que l'on veut permettre de traiter le pire des cas, quelle est la complexité en mémoire de cette implémentation ? Pouvez-vous proposer une ou plusieurs autres structures de données, qui limiteraient à O(n) la complexité mémoire de l'algorithme ?

Si on gère les listes de B dans des tableaux, il faut allouer n tableaux de n éléments, ce qui donne une complexité mémoire de $O(n^2)$.

Une première solution permettant de limiter à O(n) la mémoire auxiliaire utilisée par l'algorithme est d'utiliser des listes chaînées. L'ajout d'un élément se fait en temps constant. En revanche, les listes chaînées ne sont pas réputées pour être très efficaces car elles génèrent un grand nombre d'allocations et de libérations de petits espaces mémoires, et l'implémentation d'un ramasse-miettes pour gérer efficacement la réutilisation des espaces libérés.

Une autre solution consiste à utiliser des tables à réallocation dynamique, dont la taille est multipliée par deux quand elles sont pleines. Elles présentent un surcoût lors de la réallocation, puisqu'il faut recopier l'ancienne table dans la nouvelle et libérer l'espace alloué à l'ancienne. Mais on peut montrer que ce surcoût est amorti lors d'une utilisation régulière, et que le coût moyen de l'ajout d'un élément est constant.