Programmation avancée

Examen écrit, première session, 6 janvier 2021

Livres fermés. Durée : 2h00.

Remarques

— Aides électroniques, documents, livres et notes interdits.
— Répondez à chaque question sur une feuille **séparée**, sur laquelle figurent vos noms, prénoms et matricules.
— Merci d'indiquer clairement le numéro de question avant votre réponse.
— Soyez bref et concis, mais précis.
— Sauf mention explicite, toutes les complexités sont à décrire par rapport au temps d'exécution des opérations concernées. Soyez toujours le plus précis possible dans le choix de votre notation (Ω, O ou Θ).
— Donnez une spécification claire et concise pour chaque fonction auxiliaire que vous définissez.

Question 1 : questions courtes diverses

Aucune des questions ci-dessous ne nécessite de réponse de plus de trois à quatre lignes.

1. Soit une table hash de taille 10 (indicée de 0 à 9) avec la fonction hash $h(k) = k \bmod 10$, adressage ouvert, et sondage linéaire. Donnez la table hash après ajout de 42, 22, 89, 23, 12 et 39 en supposant qu'il n'y a pas de rehashage. La table après insertion des trois premiers éléments est ci-dessous.

		42	22						89

***** Solution *****

La table après insertion des 6 éléments est :

39		42	22	23	12				89

2. Donnez la complexité la plus précise possible en fonction de `N` pour chaque extrait de code C suivant :

A
```
int count = 0;
for (int i = N; i > 0; i--)
  for (int j = i; j > 0; j--)
    for (int k = j; k > 0; k--)
      count++;
```

B
```
int count = 0;
int i = N;
while (i > 0) {
  for (int j = i - 1; j > 0; j--)
    count++;
  i /= 2; }
```

***** Solution *****

Comptons le nombre d'exécutions de `count++;` :

A. $\sum_{i=1}^{N}\sum_{j=1}^{i}\sum_{k=1}^{j} 1 = \sum_{i=1}^{N}\sum_{j=1}^{i} j = \sum_{i=1}^{N} i(i-1)/2 \in \Theta(N^3)$

B. $N - 1 + N/2 - 1 + N/4 - 1 + \ldots 1 - 1 = -\log_2 N - 1 + \sum_{k=0}^{\log_2 N} 2^k$
$= -\log_2 N - 1 + 2^{1+\log_2 N} - 1 = 2N - 2 - \log_2 N \in \Theta(N)$

3. Le nombre de comparaisons nécessaires, pour tout algorithme de tri comparatif, est $\Omega(n \log n)$ en pire cas (le nombre d'éléments à trier étant n). Cela se démontre en utilisant la notion d'arbre de décision.
   — À quoi correspondent les feuilles de l'arbre de décision ?
   — Quelles sont les contraintes sur le nombre de feuilles de l'arbre de décision ?
   — En quoi cela permet-il de dire qu'il y a $\Omega(n \log n)$ comparaisons en pire cas ? (Pas de calcul, donnez simplement l'intuition en quelques lignes)

***** Solution *****

   — Les feuilles correspondent aux permutations permettant de trier l'entrée, en fonction des résultats des comparaisons.
   — Il y en a $n!$
   — Pour qu'un arbre binaire ait $n!$ feuilles, il faut qu'il ait une hauteur dans $\Omega(n \log n)$. Comme la longueur d'une branche correspond au nombre de comparaisons, le nombre de comparaisons en pire cas est $\Omega(n \log n)$.

4. Considérons un problème pour lequel on a un algorithme qui le résout dont la complexité en pire cas est $\Theta(n^2)$. Que peut-on dire de la complexité du problème ?

***** Solution *****

La complexité du problème est bornée par le haut par n^2, ou encore la complexité du problème est dans $O(n^2)$.

5. Quand on parle de complexité $\Theta(\log n)$, en quelle base est $\log n$? (Expliquez en quelques lignes)

***** Solution *****

Cela n'a pas d'importance, car cela change simplement la constante en facteur d'une même fonction. Par exemple

$$\Theta(\log_a n) = \Theta(\ln n / \ln a) = \Theta(\ln n / \ln b) = \Theta(\log_b n)$$

6. Soit le bout de code suivant :

```
int f(int a) {
  if (g(a))
    return h1(a);
  else
    return f(h2(a));
}
```

Ce code est récursif terminal (tail recursive). Transformez-le en code itératif.

***** Solution *****

```
int f(int a) {
  while (!g(a))
    a = h2(a);
  return h1(a);
}
```

7. Si on souhaite stocker n éléments dans une structure de données, mais que n n'est pas connu a priori, peut-on utiliser un tableau ? Si oui, est-ce que cela a un effet négatif sur la

complexité théorique (par rapport à l'utilisation d'une liste, où l'ajout en tête est toujours en temps constant) ? Expliquez en quelques lignes (vous pouvez vous référer à une étude de complexité réalisée en cours théorique).

***** Solution *****

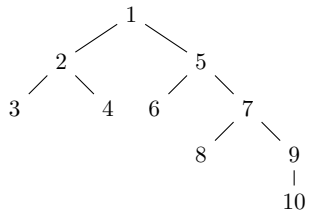
On peut utiliser un tableau extensible. Cela n'a aucun effet sur la complexité théorique car le coût amorti est constant, pour autant que la taille du tableau soit doublée à chaque fois qu'il est nécessaire d'allouer plus d'éléments.

8. Quelle est la différence entre pile et file ? En utilisant une liste doublement liée, que peut-on implémenter efficacement ?

***** Solution *****

Une pile a un comportement LIFO (last in, first out) : le premier élément retiré est l'élement inséré le plus récemment. Une file a un comportement FIFO (first in, first out) : le premier élément retiré est l'élement ajouté le plus tôt. Avec une liste doublement liée, on peut implémenter à la fois une pile et une file, en fait une deque (double ended queue).

9. Donnez dans l'ordre les numéros des nœuds quand l'arbre suivant est parcouru en largeur d'abord :



Expliquez l'idée de l'algorithme et donnez sa complexité en temps en pire cas par rapport au nombre de noeuds de l'arbre.

***** Solution *****

Les nœuds dans l'ordre sont 1, 2, 5, 3, 4, 6, 7, 8, 9, 10. Il suffit d'utiliser une queue, d'y mettre initialement la racine, visiter à chaque fois le premier nœud de la queue, et ajouter à la queue les enfants directs du nœud visité. L'algorithme est linéaire en meilleur et pire cas.

Question 2 : analyse d'algorithmes

Soit SELECT l'algorithme suivant permettant de trouver l'élément de $A[p\,..\,r]$ qui serait à la k-ème position si $A[p\,..\,r]$ était trié :

```
SELECT(A, p, r, k)
1  if p == r
2        return A[p]
3  q = PARTITION(A, p, r)
4  if q == k
5        return A[q]
6  else if q < k
7        return SELECT(A, q + 1, r, k)
8  else
9        return SELECT(A, p, q − 1, k)
```

L'appel de fonction $q = \textsc{Partition}(A, p, r)$ sélectionne un élément pivot dans $A[p, r]$, et réordonne les éléments dans A de façon à placer cet élément pivot en $A[q]$, les éléments d'indice p à $q-1$ étant tous inférieurs ou égaux au pivot, les éléments d'indice $q+1$ à r étant tous strictement supérieurs à $A[q]$.

1. De quel algorithme vu au cours est inspiré Select ?
2. Analysez la complexité de l'exécution de $\textsc{Select}(A, 1, n, k)$ en temps et en espace au pire et au meilleur cas en fonction de n, la taille de A. On suppose que l'appel à Partition est $\Theta(r-p)$.
3. Que pouvez-vous dire de la complexité en temps au pire cas pour $\textsc{Select}(A, 1, n, k)$ lorsque A est tel que $\textsc{Partition}(A, p, r) = \frac{p+r}{2}$ à chaque étape de la récursion ?

***** Solution *****

L'algorithme est inspiré du QuickSort, comme en atteste l'appel à Partition.

Soit n le nombre d'élément du tableau A. Le meilleur cas apparaît lorsque le pivot renvoyé à la ligne 3 est l'élément recherché (l'élément à la position k dans le tableau trié). Par exemple, si $k = n/2$ et A est un tableau tel que l'élément médian est la position n. Il n'y a alors pas d'appels récursifs. La complexité de Partition étant linéaire, on conclut que la complexité en temps de Select est $\Theta(n)$, puisque les autres opérations se font en temps constant. La complexité en espace est elle constante puisqu'il n'y a aucune mémoire auxiliaire allouée.

Le pire cas survient lorsqu'on cherche le minimum et que A est trié. Dans ce cas, q vaudra r à chaque étape de la récursion et Select sera rappelé sur un tableau de taille $n-1$. Soit $T(n)$ la complexité en temps au pire cas de Select, on a

$$T(n) = \begin{cases} c, & \text{si n=1} \\ T(n-1) + c'n + c'', & \text{sinon} \end{cases} \tag{1}$$

où $c'n$ est la complexité de Partition, et c'' est le coût des opérations en temps constant. On en déduit que la complexité en temps est $T(n) \in \Theta(n^2)$. La complexité en espace est due aux appels récursifs. Si on tient compte du fait que la récursion est terminale, la complexité en espace peut être constante. Sinon, elle est de l'ordre de $\Theta(n)$ (appels récursifs).

Enfin, lorsque le pivot partition le tableau en deux parties égales, la complexité au pire cas devient

$$T'(n) = \begin{cases} c, & \text{si n=1} \\ T'(n/2) + c'n + c'', & \text{sinon} \end{cases} \tag{2}$$

ce qui donne $T'(n) \in \Theta(n)$.

Question 3 : structures de données

1. Qu'est ce qu'un tas maximum ?
2. Écrivez une fonction efficace $\textsc{Heap-Replace}(H, k, v)$, remplaçant un élément à la position k dans le tas H par l'élément v et retourne l'élément remplacé. Les valeurs stockées dans le tas H sont des nombres entiers. $H.array$ permet d'accéder au tableau représentant le tas dont la capacité est donnée par $H.capacity$. $H.size$ est le nombre d'éléments stockés dans le tas.
3. Donnez le pire cas de la fonction $\textsc{Heap-Replace}(H, k, v)$ et sa complexité en temps en fonction de n, le nombre d'éléments stockés dans le tas ($= H.size$).

***** Solution *****

1. On attend arbre binaire complet + propriété d'ordre
2. Fonction :

```
HEAPIFY-UP(H, k)
1  while k > 1 ∧ H.array[PARENT(k)] < H.array[k]
2        SWAP(H.array[k], H.array[PARENT(k)])
3        k = PARENT(k)
```

```
HEAP-REPLACE(H, k, v)
1  old = H.array[k]
2  H.array[k] = v
3  MAX-HEAPIFY(H, k)
4  HEAPIFY-UP(H, k)
5  return old
```

3. Dans le pire cas, on remplace une feuille (par exemple le minimum du tas max) par v tel que v est la nouvelle valeur maximum du tas. Dans ce cas, on doit faire remonter le noeud concerné jusqu'à la racine du tas. Le pire cas est aussi rencontré lorsqu'on remplace la racine par la valeur minimum du tas. On a donc on complexité $\Theta(h)$ où h est la hauteur de l'arbre. Un tas étant un arbre équilibré, on a $h = \log n$ et donc une complexité $\Theta(\log n)$ au pire cas.

Question 4 : résolution de problème

L'algorithme glouton de la caissière ne fournit une solution optimale que pour certains ensembles de pièces. Fournissez une formulation récursive pour le cas général. Plus précisément, soit $C = \{c_1, \ldots, c_n\}$ l'ensemble des espèces disponibles, dont on supposera disposer en quantité suffisante, formulez récursivement CoinChange(V), le nombre minimum de pièces nécessaires pour égaler V avec les espèces de l'ensemble C. N'oubliez pas le (ou les) cas de base.

***** Solution *****

$$\textsc{CoinChange}(V) = \begin{cases} 0, & \text{si } V = 0 \\ +\infty, & \text{si } V < 0 \\ \min_{1 \le i \le n} \textsc{CoinChange}(V - c_i) + 1, & \text{sinon} \end{cases} \tag{3}$$