Corrigé

Examen d'algorithmique

EPITA ING1 2013 S1; A. DURET-LUTZ

Durée : 1 heure 30

Janvier 2011

1 Dénombrement (2 pts)

```
for (int i = 0; i <= N; i += 2)
  for (int j = i; j > 0; --j)
    puts("x");
```

Combien de fois le programme ci-dessus affiche-t-il `"x"` ? Donnez votre réponse en fonction de N en distinguant le cas où N est pair de celui où N est impair.

Réponse :

La boucle interne s'exécute toujours i fois pour des valeurs de i qui sont paires. On peut écrire

$$\sum_{\substack{0\le i\le N\\ i \text{ pair}}} i \quad = \quad \sum_{0\le i\le \lfloor N/2\rfloor} 2i \quad = \begin{cases} 2\sum\limits_{0\le i\le N/2} i = \dfrac{N}{2}\left(\dfrac{N}{2}+1\right) = \dfrac{N^2+2N}{4} & \text{si } i \text{ est pair} \\ 2\sum\limits_{0\le i\le (N-1)/2} i = \dfrac{N-1}{2}\left(\dfrac{N-1}{2}+1\right) = \dfrac{N^2-1}{4} & \text{si } i \text{ est impair} \end{cases}$$

Une formule correcte dans les deux cas est $\left\lfloor \frac{N}{2} \right\rfloor \cdot \left\lfloor \frac{N}{2}+1 \right\rfloor$.

2 Ordres de grandeur (2 pts)

Lesquelles de ces affirmations sont vraies ?

☐ $(\sqrt{n})^n \in O(n^{\sqrt{n}})$	☒ $(\sqrt{n})^n \in \Omega(n^{\sqrt{n}})$	☐ $n^{\sqrt{n}} \in \Theta((\sqrt{n})^n)$	☒ $\log_2(n!) \in O(n \log n)$
☒ $n^{\sqrt{n}} \in O((\sqrt{n})^n)$	☐ $n^{\sqrt{n}} \in \Omega((\sqrt{n})^n)$	☒ $\log_2(n!) \in \Theta(n \ln n)$	☐ $\log_2(n!) \in \Omega(n^n)$

3 File de Priorité à double entrée (6 pts)

Une file de priorité à double entrée est un type de données abstrait représentant un ensemble de valeurs supportant les opérations suivantes :
- $x \leftarrow$ FINDMAX(S) retourne la valeur maximale de S
- DELETEMAX(S) retire la valeur maximale de S (cela inclut la recherche de cette valeur)
- $x \leftarrow$ FINDMIN(S) retourne la valeur minimale S
- DELETEMIN(S) retire la valeur minimale de S (cela inclut la recherche de cette valeur)
- INSERT(S,x) insère la valeur x dans S

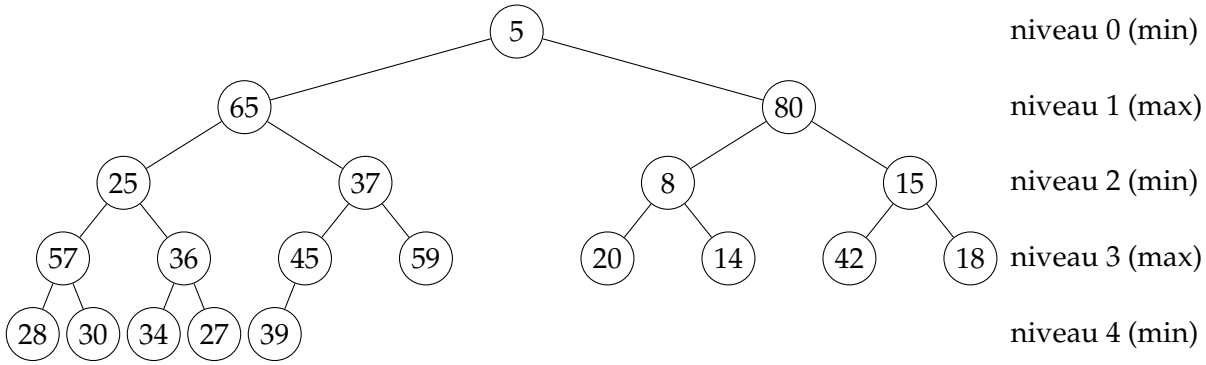
Une telle interface peut être implémentée sur plusieurs structures de données. Indiquez la complexité de ces cinq opérations sur chacune des structures de donnéees proposées dans le tableau suivant. Utilisez les notations Θ et O de façon précise, et en fonction du nombre n d'éléments présents dans la file de priorité.

	FINDMAX	DELETEMAX	FINDMIN	DELETEMIN	INSERT
Un tableau non trié	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Un tableau circulaire trié (valeurs croissantes)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$O(n)$
Une liste simplement chaînée triée (valeurs croissantes) sans pointeur sur la queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$
Une liste doublement chaînée, circulaire et triée	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$O(n)$
Un tas "max" (c'est-à-dire dont chaque nœud est plus grand que ses fils)	$\Theta(1)$	$O(\log n)$	$\Theta(n)$	$\Theta(n)$	$O(\log n)$
Un tas "min" (c'est-à-dire dont chaque nœud est plus petit que ses fils)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(\log n)$	$O(\log n)$
Un tas "min" et un tas "max" maintenus en parallèle (insertion et suppression des valeurs se font dans les deux tas)	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$O(\log n)$
Une arbre rouge et noir	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

4 Tas Min-Max (10 pts)

Un *Tas Min-Max* est une structure de donnée hybride entre le *Tas Min* et le *Tas Max*. Plus précisément, il s'agit d'un arbre binaire parfait dans lequel :
– les nœuds à un niveau **pair** sont plus **petits** que leurs descendants (directs ou indirects), et
– les nœuds à un niveau **impair** sont plus **grands** que leurs descendants (directs ou indirects).

Voici un exemple de tas min-max :



Constatez que la racine possède forcément la valeur minimale du tas, et que la valeur maximale est forcément l'un des deux fils de la racine (ou la racine elle-même si elle n'a pas de fils).

L'arbre étant parfait, il sera représenté et manipulé sous la forme d'un tableau :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
5	65	80	25	37	8	15	57	36	45	59	20	14	42	18	28	30	34	27	39

Les fils du nœud d'indice i se trouvent aux positions $\text{LEFTCHILD}(i) = 2i$ et $\text{RIGHTCHILD}(i) = 2i + 1$ lorsqu'elles existent. Le père est à la position $\text{PARENT}(i) = \lfloor i/2 \rfloor$ si elle existe.

1. **(2 pts)** Écrivez une fonction ISONMINLEVEL(A,i) qui décide en temps constant si l'indice i correspond à un nœud se trouvant sur un niveau pair (ou niveau "min") du tas min-max représenté par le tableau A. Les indices dans A commencent à 1 et vous pouvez supposer que le calcul d'un logarithme se fait un temps constant.

Réponse :

Le niveau du nœud d'indice i est $\lfloor \log i \rfloor$. On a donc :

ISONMINLEVEL(A,i)
  return $((\lfloor \log i \rfloor \mod 2) = 0)$

Beaucoup de personnes ont choisi de déterminer la nature du niveau avec une procédure comme celle-ci :

ISONMINLEVEL(A,i)
  return $(i = 0)$ or $A[i] > A[i/2]$.

Cette procédure, à laquelle j'ai quand même attribué 1.5 points, ne marche malheureusement que si $A[i]$ existe et si A respecte déjà la structure d'un tas min-max (ce qui n'est pas le cas lors que ISONMINLEVEL est appelé dans HEAPIFY).

Les algorithmes sur les *tas min-max* sont similaires aux algorithmes sur les *tas max* vus en cours.

La procédure HEAPIFY(A,i), prend un nœud d'indice i dont les deux sous-arbres vérifient la propriété de tas, et fait redescendre la valeur $A[i]$ dans l'un des sous-arbres afin que la propriété du tas soit vérifiée à partir de l'indice i. Dans le cas des *tas min-max*, cette procédure est obligée de distinguer entre les niveaux *min* et les niveaux *max* comme suit :

```
HEAPIFY(A,i)
  if ISONMINLEVEL(A,i)
    then HEAPIFYMIN(A,i)
    else HEAPIFYMAX(A,i)
HEAPIFYMIN(A,i)
  if A[i] has children then
    m ← index of the smallest of the children and grandchildren (if any) of A[i]
    if A[m] < A[i] then
      A[m] ↔ A[i]
      if A[m] is a grandchild of A[i] then
        if A[m] > A[PARENT(m)] then A[m] ↔ A[PARENT(m)]
        HEAPIFYMIN(A,m)
```

La procédure HEAPIFYMAX est identique à HEAPIFYMIN en changeant le sens des comparaisons et en définissant m comme l'indice du plus grand des fils et petits-fils.

2. **(2 pts)** Donnez la complexité en pire cas de HEAPIFY en fonction du nombre n de valeurs contenues dans A. Justifiez votre réponse.

Réponse :

HEAPIFYMIN et HEAPIFYMAX travaillent l'une est l'autre le long d'une branche de l'arbre que représente A. Toutes les branches sont de hauteur $\Theta(\log n)$. La progression se fait deux niveaux à la fois, mais cela fait donc toujours un $\Theta(\log n)/2 = \Theta(\log n)$ étapes. Le nombre d'opérations effectuées à chaque niveau est constant. La complexité est donc au pire $\Theta(\log n)$ si la branche est suivie de la racine jusqu'à une feuille. La complexité est moindre si l'algorithme s'arrête avant.
La complexité de HEAPIFY est donc $O(\log n)$.

Supprimer la plus petite valeur (la racine) ou la plus grande valeur (l'un de ses fils s'ils existent) d'un *tas min-max* se fait comme sur les tas classiques en remplaçant cette valeur par la dernière du tableau (dont on réduit la taille de 1) et en appelant HEAPIFY sur le nœud dont la valeur vient d'être modifiée.

3. **(3 pts)** Écrivez la procédure DELETEMAX(A) qui supprime la valeur maximale du *tas min-max* représenté par le tableau A. Vous pouvez supposer que le tableau possède un champ $A.size$ donnant sa taille, et que cette procédure n'est jamais appelée sur un tableau vide.

Réponse :

```
DELETEMAX(A)
  if A.size = 1 then
    A.size ← 0
  else
    if A.size > 2 and A[3] > A[2] then m ← 3 else m ← 2
    A[m] ← A[A.size]
    A.size ← A.size − 1
    if m < A.size then HEAPIFY(A, m)
```

4. **(1 pt)** Quelle est la complexité de la procédure précédente en fonction de la taille n du tableau A ? (Vous pouvez exprimer cette complexité en fonction de la complexité $H(n)$ de HEAPIFY si vous n'avez pas répondu à la question 2.)

Réponse :

C'est au pire la même complexité que HEAPIFY puisqu'on ajoute une nombre constant d'opérations : $O(H(n)) = O(\log n)$.

5. **(2 pts)** Donnez le contenu du tableau A de l'exemple après **deux** appels à DELETEMAX(A).

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5	59	42	25	27	8	15	57	36	45	37	20	14	39	18	28	30	34

Nous ne traitons pas l'insertion dans cet examen, mais la procédure est semblable à celle d'un tas classique : on ajoute en feuille puis on fait remonter la valeur pour respecter les contraintes.

5 Diviser pour régner (5 pts)

Soit un gros problème de taille n, assez compliqué pour que vous n'ayez pas envie d'en connaître les détails. Pour résoudre ce problème, on vous propose trois algorithmes :

L'algorithme A résout le problème en le divisant en 5 sous-problèmes de taille $n/2$, en résolvant ces cinq sous problèmes récursivement, puis en combinant les solutions en temps linéaire.

L'algorithme B résout le problème en résolvant récursivement deux problème de taille $n - 1$ puis en combinant leurs solutions en temps constant.

L'algorithme C résout le problème en le divisant en 9 sous-problèmes de taille $n/3$, en résolvant ces neuf sous-problèmes récursivement, puis en combinant les solutions en $\Theta(n^2)$.

Dans les trois cas, le découpage du problème en sous-problèmes se fait en temps constant.

Calculez la complexité de chacun de ces algorithmes.

Réponse :

- $T_A(n) = 5T_A(n/2) + \Theta(n)$
  On a $\Theta(n) = O(n^{\log_2(5)-\varepsilon})$ en prenant par exemple $\varepsilon = 1$. D'après le théorème général la solution est donc $T_A(n) = \Theta(n^{\log_2(5)})$.
  (Note : comme $2 < \log_2(5) < 3$ cette complexité est comprise entre $\Theta(n^2)$ et $\Theta(n^3)$.)
- $T_B(n) = 2T_B(n-1) + \Theta(1)$
  Remplaçons 1 par une constante c est résolvons par substitution :

$$T_B(n) = c + 2T_B(n-1) = c + 2c + 4T_B(n-2) = c + 2c + 4c + 8T_B(n-3)$$
$$= \left(c \sum_{i=0}^{k-1} 2^i \right) + 2^k T_B(n-k)$$

  On peut continuer jusqu'à $k = n-1$, alors $2^k T_B(n-k) = 2^{n-1} T_B(1) = 2^{n-1}\Theta(1)$

$$= \left(c \sum_{i=0}^{n-2} 2^i \right) + 2^{n-1}\Theta(1) = c(2^{n-1} - 1) + 2^{n-1}\Theta(1) = \Theta(2^{n-1}) = \Theta(2^n)$$

- $T_C(n) = 9T_C(n/3) + \Theta(n^2)$
  Ici $a = 9$ et $b = 3$, donc $n^{\log_b(9)} = n^2$. On a évidement $\Theta(n^2) = \Theta(n^2)$. D'après le théorème général la solution est donc $T_C(n) = \Theta(n^2 \log n)$.

Conclusion, c'est l'algorithme C le plus rapide.