Corrigé

Examen d'algorithmique

EPITA ING1 2016 S1; A. DURET-LUTZ

Durée : 1h30

Janvier 2013

1 Dénombrement (5 pts)

Donnez vos réponses en fonction de N.

1. **(2 pts)** Combien de fois le programme ci-dessous affiche-t-il `"x"` ?

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

Réponse :

$$\sum_{i=1}^{N}\sum_{j=0}^{i-1} 1 = \sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

Beaucoup trop d'entre-vous croient qu'on peut décaler les indices d'une somme sans changer son contenu et écrivent des horreurs du style

$$\sum_{i=1}^{N} i \overset{\textbf{faux}}{=} \sum_{i=0}^{N-1} i = \frac{N(N-1)}{2}.$$

En général $\sum_{i=1}^{n} f(i)$ n'est pas égal à $\sum_{i=0}^{n-1} f(i)$: le nombre de termes dans la somme est bien le même mais la fonction f est évaluée en des points différentes. On a par contre $\sum_{i=1}^{n} f(i) = \sum_{i=0}^{n-1} f(i+1)$

2. **(3 pts)** Et celui-ci ?

```
for (int i = 1; i <= N; ++i)
  for (int j = 1; j < i; ++j)
    for (int k = N; k > N - 2; --k)
      puts("x");
```

Réponse :

Curieusement la boucle la plus interne en a troublé plus d'un. La variable k ne peut pourtant prendre que deux valeurs (N et $N-1$) indépendamment de i et j. On peut donc voir les deux dernières lignes comme s'il s'agissait de deux appels à `puts("x")`.
D'autre part, la boucle j ne fait aucune itération lorsque $i = 1$, on peut donc faire commencer la première boucle à $i = 2$ sans changer le nombre de lignes affichées.

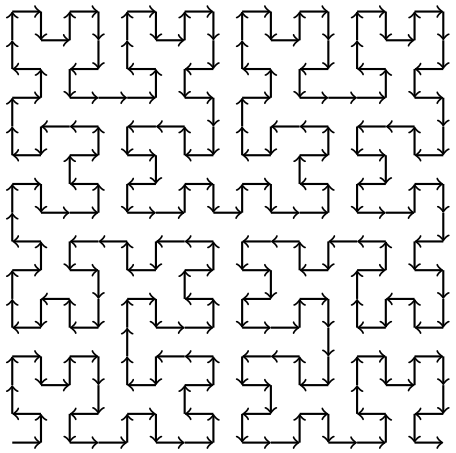
$$\sum_{i=2}^{N}\sum_{j=1}^{i-1}\sum_{k=N-1}^{N} 1 = \sum_{i=2}^{N}\sum_{j=1}^{i-1} 2 = 2\sum_{i=2}^{N}(i-1) = 2\frac{(1+N-1)(N-1)}{2} = N(N-1)$$

2 Complexité d'une fonction recursive (7 pts)

On considère la fonction recursive suivante pour dessiner une *courbe de Hilbert*. On suppose que chaque appel à `line()` trace un segment dans la direction indiquée, en temps constant.

```
void
hilbert(int n, int dir)
{
  if (n == 0)
    return;
  --n;
  if (dir == UP)
    {
      hilbert(n, RIGHT);
      line(UP);
      hilbert(n, UP);
      line(RIGHT);
      hilbert(n, UP);
      line(DOWN);
      hilbert(n, LEFT);
    }
  else if (dir == LEFT)
    {
      hilbert(n, DOWN);
      line(LEFT);
      hilbert(n, LEFT);
      line(DOWN);
      hilbert(n, LEFT);
      line(RIGHT);
      hilbert(n, UP);
    }
  else if (dir == RIGHT)
    {
      hilbert(n, UP);
      line(RIGHT);
      hilbert(n, RIGHT);
      line(UP);
      hilbert(n, RIGHT);
      line(LEFT);
      hilbert(n, DOWN);
    }
  else // if (dir == DOWN)
    {
      hilbert(n, LEFT);
      line(DOWN);
      hilbert(n, DOWN);
      line(LEFT);
      hilbert(n, DOWN);
      line(UP);
      hilbert(n, RIGHT);
    }
}
```

Par exemple `hilbert(4, UP)` dessine la courbe suivante :



1. **(3 pts)** Si j'exécute `hilbert(`n`, UP)` pour un n donné, combien de fois la fonction `hilbert` sera-t-elle appelée ? Donnez votre réponse en fonction de n. (L'appel initial doit être compté.)

Réponse :

Notons $h(n)$ le nombre d'appels à `hilbert` effectués pour un n donné. Évidement $h(0) = 1$ et pour tout $n > 0$ on a $h(n) = 1 + 4h(n-1)$. En remplaçant $h(n-1)$ par sa définition on trouve $h(n) = 1 + 4 + 4^2 h(n-2)$, et en continuant ces substitutions $h(n) = 1 + 4 + 4^2 + \cdots + 4^{n-1} + 4^n h(0)$. Autrement dit

$$h(n) = \sum_{i=0}^{n} 4^i = \frac{4^{n+1} - 1}{3}$$

La formule pour la somme des puissances était dans l'annexe.

2. **(2 pts)** Si j'exécute `hilbert(`n`, UP)` pour un n donné, combien de fois la fonction `line()` sera-t-elle appelée ? Donnez votre réponse en fonction de n.

Réponse :

Même raisonnement : si $\ell(n)$ est le nombre d'appels à `line()`, on a $\ell(n) = 3 + 4\ell(n-1)$ avec $\ell(0) = 0$. On développe la définition sous la forme $\ell(n) = 3 + 4 \times 3 + 4^2 \times 3 + \cdots + 4^{n-1} \times 3 + 4^n \ell(0)$, soit :

$$\ell(n) = 3 \sum_{i=0}^{n-1} 4^i = 3\frac{4^n - 1}{3} = 4^n - 1$$

3. **(2 pts)** Quelle est complexité de `hilbert(`n`, UP)` en fonction de n ?

Réponse :

Comme les appels à `line` sont en temps constant, la complexité est $\Theta(4^n)$.
Notez que $\Theta(2^n)$ ou $\Theta(3^n)$ ne sont pas des classes de complexité équivalentes : on ne peut pas passer d'une exponentielle à l'autre en multipliant par une constante.

3 Tri postal (8 pts)

On considère une série de codes postaux de 5 chiffres, représentés dans un tableau (0). L'objectif est de trier ce tableau en plusieurs étapes. Dans la première étape (1), nous trions le tableau par rapport au dernier chiffre de chaque code postal. Dans la seconde étape (2), le tableau est trié par rapport à son avant dernier chiffre. Etc. On effectue cinq étapes en triant à chaque fois par rapport au chiffre précédent. Après la cinquième étape (5), on espère que le tableau est trié.

(0)	(1)	(2)	(3)	(4)	(5)
58170	58170	79100	75013	75013	38700
79100	79100	46800	79100	65120	45800
87160	87160	45800	47120	45800	46800
47250	47250	38700	65120	46800	47120
49530	49530	75013	87160	47120	47250
75013	46800	47120	58170	87160	49530
46800	47120	65120	47250	47250	58170
47120	45800	49530	49530	58170	65120
45800	65120	47250	38700	38700	75013
65120	38700	87160	46800	79100	79100
38700	75013	58170	45800	49530	87160

1. Quel algorithme puis-je utiliser pour réaliser le tri de chaque étape et être sûr que le tableau sera trié après la cinquième étape ? (Cochez toutes les réponses correctes.)

   - ☐ le tri rapide (quick sort)
   - ☒ le tri par insertion
   - ☒ le tri fusion (merge sort)
   - ☐ le tri par sélection
   - ☐ le tri par tas (heap sort)

2. Indépendemment du fait que le résultat soit effectivement trié ou non, supposons que ce soit le tri par insertion qui soit utilisé à chaque étape. S'il y a n codes postaux à trier, quelle est la complexité totale du tri (les 5 étapes) ?

Réponse :

Le tri par insertion est un tri en $O(n^2)$, le répéter 5 fois ne change pas sa complexité.

3. On considère maintenant un facteur qui cherche à trier une pile d'enveloppes en fonction de leur code postal, cela à l'aide de dix casiers numérotés de 0 à 9. Le facteur prend chaque enveloppe en commençant par le haut de la pile, et la dépose, face vers le bas, dans le casier étiqueté par le dernier numéro du code postal. Une fois que toutes les enveloppes ont été réparties dans les casiers, ils forme une nouvelle pile en prenant le contenu de chaque casier dans l'ordre (casier 0 en haut, casier 9 en bas). Il a alors une pile d'enveloppes triées par rapport au dernier chiffre du code postal comme après l'étape (1) de l'exemple. Le facteur procède alors à 4 nouveaux tris, comme précédemment, pour trier sa pile par rapport aux 4 autres chiffres (toujours de la droite vers la gauche). Après ces cinq itérations la pile est triée.

   Cet algorithme se transpose naturellement sur une machine pour trier un ensemble d'entiers.

   Quelle structure de donnée vous semble la plus appropriée pour représenter l'un des 10 casiers.

   - ☐ une chaîne de caractères
   - ☒ un tableau
   - ☒ une liste chaînée
   - ☐ un table de hachage
   - ☐ un tas

   (Je n'attendais qu'une réponse parmis ces deux choix.)

4. Quelle est la complexité de ce tri par casiers pour trier n codes postaux ?

Réponse :

$\Theta(n)$

En effet, chaque passe est en $\Theta(n)$ et il y en a un nombre constant.

4 Chaîne de multiplication de matrices (7 points)

Soient $A_1, A_2, \dots A_n$, n matrices de dimensions respectives $p_0 \times p_1$, $p_1 \times p_2$, $\dots$, $p_{n-1} \times p_n$.

On souhaite calculer le produit $A_1 \cdot A_{i+1} \cdots A_n$ avec des multiplication matricielles classiques, mais en posant les parenthèses (pour fixer l'ordre d'évaluation des produits) de façon à minimiser le nombre de multiplications scalaires.

Notons $m[i,j]$ le nombre minimum de multiplications scalaires nécessaires pour multiplier $A_i \cdot A_{i+1} \cdots A_j$. Comme nous l'avons vu en cours, on a :

$$m[i,j] = \begin{cases} 0 & \text{si } i = j \\ \min\left\{ m[i,k] + m[k+1,j] + p_{i-1}p_k p_j \mid k \in \{i, \dots, j-1\} \right\} & \text{si } i < j \end{cases}$$

1. **(2 pts)** Que représente k dans la formule ci-dessus ?

Réponse :

Le min considère tous les parenthèsages possibles de la forme

$$(A_i A_{i+1} \dots A_k)(A_{k+1} A_{k+2} \dots A_j)$$

Le k désigne donc un parenthèsage unique en repérant le numéro de la dernière matrice du bloc de gauche.

2. **(2 pts)** Donnez, en fonction de n, la complexité de calculer $m[1,n]$ avec un algorithme de programmation dynamique qui implémente la définition ci-dessus.

Réponse :

Le calcul de $m[1,n]$ demande de remplire les $n(n+1)/2$ cases $m[i,j]$ telles que $i \leq j$. Mais chaque case demande $O(n)$ opérations à cause du calcul du min. On a donc $O(n^3)$ opérations.

En cours nous avions calculé une complexité plus précise en comptant le nombre d'opérations effectuées par le min pour chaque case (en fonction de la diagonale de m contenant la case). Si l'on note $d = j - i$ le numéro de la diagonale, il y a $n - d$ cases sur cette diagonale et pour chacune le min est calculé entre d valeurs.

Le nombre total de valeurs calculées par les min est donc

$$\sum_{d=1}^{n-1}(n-d)d = n\sum_{d=1}^{n-1} d - \sum_{d=1}^{n-1} d^2 = \frac{n^2(n-1)}{2} - \frac{(n-1)n(2n-1)}{6} = \Theta(n^3)$$

3. **(3 pts)** Dans le cas particulier où quatre matrices A_1, A_2, A_3 et A_4 sont de tailles respectives 10×20, 20×5, 5×50 et 50×4, complétez le tableau m :

	j=1	j=2	j=3	j=4
i=1	0	1000	3500	2200
i=2		0	5000	1400
i=3			0	1000
i=4				0

Notations asymptotiques

$$\mathrm{O}(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$
$$\Omega(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$
$$\Theta(g(n)) = \{f(n) \mid \exists c_1 \in \mathbb{R}^{+\star}, \exists c_2 \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \iff g(n) \in \mathrm{O}(f(n))$ et $f(n) \notin \mathrm{O}(g(n))$ $\quad f(n) \in \mathrm{O}(g(n)) \iff g(n) \in \Omega(f(n))$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \iff f(n) \in \mathrm{O}(g(n))$ et $g(n) \notin \mathrm{O}(f(n))$ $\quad f(n) \in \Theta(g(n)) \iff \begin{cases} f(n) \in \Omega(g(n)) \\ g(n) \in \Omega(f(n)) \end{cases}$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in \mathbb{R}^{+\star} \iff f(n) \in \Theta(g(n))$

Ordres de grandeurs

constante	$\Theta(1)$	
logarithmique	$\Theta(\log n)$	
polylogarith.	$\Theta((\log n)^c)$	$c > 1$
	$\Theta(\sqrt{n})$	
linéaire	$\Theta(n)$	
	$\Theta(n \log n)$	
quadratique	$\Theta(n^2)$	
	$\Theta(n^c)$	$c > 2$
exponentielle	$\Theta(c^n)$	$c > 1$
factorielle	$\Theta(n!)$	
	$\Theta(n^n)$	

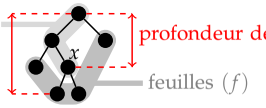
Théorème général

Soit à résoudre $T(n) = aT(n/b) + f(n)$ avec $a \geq 1$, $b > 1$

- Si $f(n) = \mathrm{O}(n^{\log_b a - \varepsilon})$ pour un $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.
- Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$.
- Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pour un $\varepsilon > 0$, et si $af(n/b) \leq cf(n)$ pour un $c < 1$ et tous les n suffisamment grands, alors $T(n) = \Theta(f(n))$.

(Note : il est possible de n'être dans aucun de ces trois cas.)

Arbres



Pour tout arbre binaire :

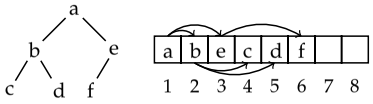
$n \leq 2^{h+1} - 1$ $\quad h \geq \lceil \log_2(n+1) - 1 \rceil = \lfloor \log_2 n \rfloor$ si $n > 0$

$f \leq 2^h$ $\quad h \geq \lceil \log_2 f \rceil$ si $f > 0$

$f = ni + 1$ (si l'arbre est *complet* = les nœud internes ont tous 2 fils)

Dans un arbre binaire équilibré une feuille est soit à la profondeur $\lfloor \log_2(n+1) - 1 \rfloor$ soit à la profondeur $\lceil \log_2(n+1) - 1 \rceil = \lfloor \log_2 n \rfloor$. Pour ces arbres $h = \lfloor \log_2 n \rfloor$.

Un arbre parfait (= complet, équilibré, avec toutes les feuilles du dernier niveau à gauche) **étiqueté** peut être représenté par un tableau.



Les indices sont reliés par :

$\text{Père}(y) = \lfloor y/2 \rfloor$

$\text{FilsG}(y) = y \times 2$

$\text{FilsD}(y) = y \times 2 + 1$

Identités utiles

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \quad \text{si } x \neq 1$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad \text{si } |x| < 1$$

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \quad \text{si } |x| < 1$$

$$\sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n)$$

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Définitions diverses

La complexité d'un problème est celle de l'algorithme le plus efficace pour le résoudre.

Un tri stable préserve l'ordre relatif de deux éléments égaux (au sens de la relation de comparaison utilisée pour le tri).

Un tri en place utilise une mémoire temporaire de taille constante (indépendante de n).

Rappels de probabilités

Espérance d'une variable aléatoire X **:** C'est sa valeur attendue, ou moyenne. $\mathrm{E}[X] = \sum_x \Pr\{X = x\}$

Variance : $\mathrm{Var}[X] = \mathrm{E}[(X - \mathrm{E}[X])^2] = \mathrm{E}[X^2] - \mathrm{E}^2[X]$

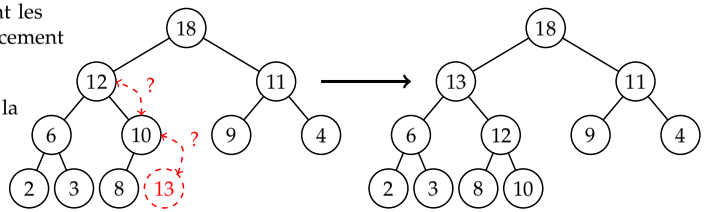
Loi binomiale : On lance n ballons dans r paniers. Les chutes dans les paniers sont équiprobables ($p = 1/r$). On note X_i le nombre de ballons dans le panier i. On a $\Pr\{X_i = k\} = C_n^k p^k (1-p)^{n-k}$. On peut montrer $E[X_i] = np$ et $\mathrm{Var}[X_i] = np(1-p)$.

Tas

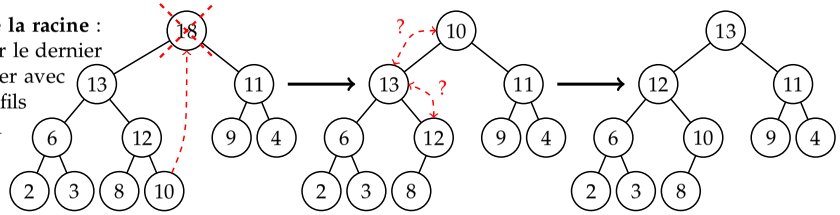
Un tas est un arbre parfait partiellement ordonné : l'étiquette d'un nœud est supérieure à celles de ses fils.

Dans les opérations qui suivent les arbres parfaits sont plus efficacement représentés par des tableaux.

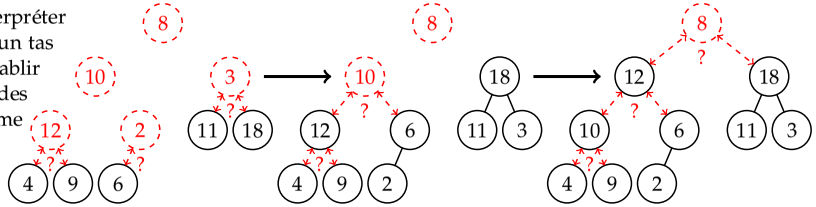
Insertion : ajouter l'élément à la fin du tas, l'échanger avec son père tant qu'il lui est inférieur (en remontant vers la racine).
$T_{\text{insert}} = \mathrm{O}(\log n)$



Suppression de la racine : la remplacer par le dernier nœud ; l'échanger avec son plus grand fils tant que celui-ci est plus grand.
$T_{\text{rem}} = \mathrm{O}(\log n)$



Construction : interpréter le tableau comme un tas (incorrect) puis rétablir l'ordre en partant des feuilles (vues comme des tas corrects).
$T_{\text{build}} = \Theta(n)$



Arbres Rouge et Noir

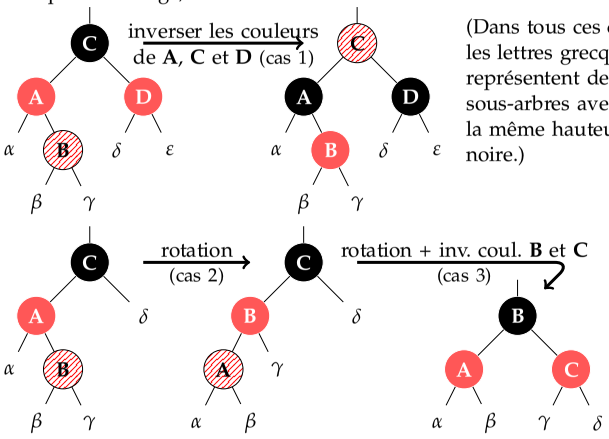
Les ARN sont des arbres binaires de recherche dans lesquels : (1) un nœud est rouge ou noir, (2) racine et feuilles (NIL) sont noires, (3) les fils d'un nœud rouge sont noirs, et (4) tous les chemins reliant un nœud à une feuille (de ses descendants) contiennent le même nombre de nœuds noirs (= la *hauteur noire*). Ces propriétés interdisent un trop fort déséquilibre de l'arbre, sa hauteur reste en $\Theta(\log n)$.

Insertion d'une valeur : insérer le nœud avec la couleur rouge à la position qu'il aurait dans un arbre binaire de recherche classique, puis, si le père est rouge, considérer les trois cas suivants dans l'ordre.

Cas 1 : Le père et l'oncle du nœud considéré sont tous les deux rouges. **Répéter** cette transformation à partir du grand-père si l'arrière grand-père est aussi rouge.

Cas 2 : Si le père est rouge, l'oncle noir, et que le nœud courant n'est pas dans l'axe père–grand-père, une rotation permet d'aligner fils, père, et grand-père.

Cas 3 : Si le père est rouge, l'oncle noir, et que le nœud courant est dans l'axe père–grand-père, un rotation et une inversion de couleurs rétablissent les propriétés des ARN.



(Dans tous ces cas, les lettres grecques représentent des sous-arbres avec la même hauteur noire.)