ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Section d'Informatique et de Systèmes de Communication

Corrigé de la série 8 16 Novembre 2007

1. Sorting

a) Considérons l'algorithme suivant :

Input: Une suite de 3 éléments $a[0], a[1], a[2]$.
Output: La permutation (p_0, p_1, p_2) de $(0, 1, 2)$ telle que $a[p_0] \leq a[p_1] \leq a[p_2]$.

```
1:  if a[0].key < a[1].key then
2:    if a[1].key < a[2].key then
3:      return (0, 1, 2)
4:    else
5:      if a[2].key < a[0].key then
6:        return (2, 0, 1)
7:      else
8:        return (0, 2, 1)
9:  else
10:   if a[0].key < a[2].key then
11:     return (1, 0, 2)
12:   else
13:     if a[2].key < a[1].key then
14:       return (2, 1, 0)
15:     else
16:       return (1, 2, 0)
```

On rappelle qu'il y a $3! = 6$ permutations possibles de 3 éléments, nous avons donc 6 possibilités pour la permutation initiale (chacune ayant la même probabilité $\frac{1}{6}$) :

$$\begin{array}{lllll} a[0] & < & a[1] & < & a[2] \\ a[0] & < & a[2] & < & a[1] \\ a[1] & < & a[0] & < & a[2] \\ a[1] & < & a[2] & < & a[0] \\ a[2] & < & a[0] & < & a[1] \\ a[2] & < & a[1] & < & a[0] \end{array}$$

En analysant l'algorithme on voit que si $a[0] < a[1] < a[2]$ ou si $a[1] < a[0] < a[2]$ il faudra 2 comparaisons, et dans les 4 autres cas il en faudra 3. Il faudra donc en moyenne

$$\frac{2+2+3+3+3+3}{6} = \frac{16}{6}$$

comparaisons.

b) On a :

• Au début du premier passage, $a[0].key$ sera la clé la plus grande, cet élément va donc remonter jusqu'à la position $N-1$, (ce qui nécessitera $N-1$ échanges).

• Au début du deuxième passage, $a[0].key$ sera la deuxième plus grande clé, et cet élément va donc remonter jusqu'à la position $N-2$ (ce qui nécessitera $N-2$ échanges).

• De même, au début du $i^{\text{ème}}$ passage, $a[0].key$ sera la $i^{\text{ème}}$ plus grande clé, et cet élément va donc remonter jusqu'à la position $N-i$ (ce qui nécessitera $N-i$ échanges).

Au total il faudra donc

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

échanges.

c) Si un élément x se trouve à la position j au début d'un passage de BubbleSort (donc $a[j] = x$), après le passage il se trouvera à une position k avec $k \geq j-1$. En effet, lors d'un passage un élément peut "monter" jusqu'à la fin de la suite (i.e. la position $N-1$), mais ne ne peut "descendre" qu'au plus d'une position.

L'élément à la position p_i doit par définition se trouver à la position i à la fin de l'algorithme. Si $p_i > i$, cet élément doit donc "descendre" de $(p_i - i)$ position. Puisqu'il ne peut "descendre" que d'au plus une position par passage, il faudra au moins $(p_i - i)$ passages pour qu'il arrive à sa place.

Ainsi il faut à BubbleSort au moins

$$\max_{0 \leq i \leq N-1}(p_i - i)$$

passages pour trier la suite.

2. Shell Sort

a) Nous utilisons Shell Sort avec les incréments $2, 1$. Nous allons donc procéder en deux étapes : Nous allons d'abord réordonner la suite pour qu'elle soit 2-triée, puis nous allons réordonner cette suite 2 triée pour qu'elle soit 1-triée (c'est-à-dire triée au sens habituel du terme).

Informellement la première étape consiste à trier les deux sous suites $(a[0], a[2], a[4])$ et $(a[1], a[3], a[5])$ (pour que la suite totale soit bien 2-triée). Chacune de ces sous-suite est triée en utilisant Insertion Sort.

Etape de l'incrément 2 **:** Notre suite est dans son état initial :

$$(4, 3, 2, 6, 1, 5)$$

Nous commençons comme décrit dans l'algorithme avec $i = h$, donc ici $i = 2$. Nous comparons $a[2]$ (2) avec $a[0]$ (4), et puisque $a[2]$ est plus petit nous échangeons ces deux éléments :

$$(2, 3, 4, 6, 1, 5)$$

(nous avons en fait appliqué la première étape de Insertion Sort à la sous-suite $(a[0], a[2], a[4])$).

Ensuite nous passons à $a[3]$ (6), que nous comparons avec $a[1]$ (3) (c'est la première étape de Insertion Sort sur $(a[1], a[3], a[5])$. Puisque $a[1] < a[3]$ $(3 < 6)$ nous n'avons rien à faire.

Nous continuons avec l'élément $a[4]$ (1). Nous voulons faire la deuxième étape de Insertion Sort sur $(a[0], a[2], a[4])$ $((2, 4, 1))$, c'est à dire insérer $a[4]$ (1) à la bonne position dans $(a[0], a[2])$ $((2, 4))$. Nous le comparons à $a[2]$ (4), il est plus petit, donc nous continuons,

nous le comparons à $a[0]$ (2), il est encore plus petit et nous savons donc qu'il doit être le premier élément dans cette sous-suite. Pour le placer en premier nous devons décaler $a[0]$ et $a[2]$ vers la droite, on obtient donc :

$$(1, 3, 2, 6, 4, 5)$$

finalement nous regardons l'élément $a[5]$ (5). Nous voulons le placer à la bonne position dans $(a[1], a[3])$ $((3, 6))$. Après deux comparaisons nous voyons qu'il doit aller au milieu (puisque $a[1] < a[5] < a[3]$, i.e. $3 < 5 < 6$), on a donc :

$$(1, 3, 2, 5, 4, 6)$$

Et nous avons fini cette étape, la suite est bien 2-triée.

Etape de l'incrément 1 **:** Nous faisons maintenant un Insertion Sort normal. Nous commençons avec $a[1]$ (3) que nous comparons à $a[0]$ (1) et voyons qu'aucun mouvement n'est nécessaire. Ensuite nous regardons $a[2]$ (2) , il faut le placer à la bonne position dans $(a[0], a[1])$, c'est à dire au milieu :

$$(1, 2, 3, 5, 4, 6)$$

Nous continuons avec $a[3]$ (5) qu'il faut mettre à la bonne position dans $(a[0], a[1], a[2])$, il n'y a donc rien à faire. Nous regardons ensuite $a[4]$ (4), et voyons qu'il faut le mettre entre $a[2]$ (3) et $a[3]$ (5) :

$$(1, 2, 3, 4, 5, 6)$$

finalement nous regardons $a[5]$ (6) et voyons qu'il est à la bonne place. Nous avons terminé et notre suite est bien triée.

Pour le nombre de mouvements, nous les comptons en supposant qu'un échange nécessite 3 mouvements ($v[1]$ vers *temp*, $v[2]$ vers $v[1]$, *temp* vers $v[2]$).

Pour passer de $(2, 3, 4, 6, 1, 5)$ à $(1, 3, 2, 6, 4, 5)$, l'algorithme place d'abord $a[4]$ dans une variable temporaire (mouvement 1), puis il déplace $a[2]$ dans $a[4]$ (mouvement 2), puis il déplace $a[0]$ dans $a[2]$ (mouvement 3), et finalement il met la valeur temporaire dans $a[0]$ (mouvement 4). Pour ce passage il a donc fallu 4 mouvements.

Tous les autre changements sont de simples échanges d'éléments dans la liste. Comme il y a en a 4, on obtient un total de $4 \times 3 + 4 = 16$ mouvements.

b) Soit a une suite 2-triée et 3-triée. Elle peut être triée en utilisant l'algorithme suivant :

Call : SORT(a)
Input: Suite a 2-triée et 2-triée de N éléments avec des clés entières
Output: Transformation de a telle que $a[i].key \leq a[i+1].key$ pour $0 \leq i < N - 1$

```
1: for i = 1, ..., N − 1 do
2:     if a[i] < a[i − 1] then
3:         Echanger a[i] et a[i − 1]
```

Il s'agit d'un passage de Bubble Sort. Comme souvent, on voit intuitivement que cet algorithme devrait marcher, mais les détails d'une preuve formelle sont plus compliqués :

Preuve que cet algorithme marche : Remarquons d'abord que comme la suite est 2 triée, pout tout k on a

$$a[k] \leq a[k+2] \leq a[k+4] \leq \dots \tag{1}$$

De même, puisque la suite est 3-triée on a $a[k] \leq a[k+3]$, et comme elle est 2-triée on obtient :

$$a[k] \leq a[k+3] \leq a[k+5] \leq a[k+7] \leq \dots \tag{2}$$

En combinant (**??**) et (**??**) on obtient

$$a[k] \leq a[k+2], a[k+3] \dots, a[N-1] \tag{3}$$

Nous voyons dans l'algorithme que la variable i va de 1 a $N-1$. Nous allons montrer par induction qu'après l'itération i les éléments $a[0], \dots, a[i-1]$ sont tous a la bonne place.

base : $i = 1$. Si on pose $k = 0$ dans (**??**) on voit que $a[0]$ est plus petit que tous les éléments, sauf $a[1]$ (on ne sait pas lequel de $a[0]$ et $a[1]$ est le plus grand).

• Si $a[0] \leq a[1]$ alors $a[0]$ est bien le plus petit élément de la suite, et on voit aussi que la première iteration de l'algorithme ne va rien changer, le plus petit élément sera donc bien à la première place.

• Si $a[1] < a[0]$ alors on a $a[1] < a[0] \leq a[2], \dots, a[N-1]$, et donc $a[1]$ est le plus petit élément de la suite. On voit aussi que la première itération de l'algorithme va échanger $a[0]$ et $a[1]$, on aura donc bien le plus petit élément de la suite à la première place.

Pas : On suppose qu'après l'itération i, $a[0], \dots, a[i-1]$ sont tous à la bonne place, et on veut montrer qu'après l'itération $i+1$ $a[0], \dots, a[i]$ seront tous à la bonne place.

Il faut donc montrer que le plus petit élément de $\{a[i-1], \dots, a[N-1]\}$ va être mis à la position $a[i-1]$.

On avait avant de commencer l'algorithme $a[i-1] \leq a[i+1], \dots, a[N-1]$. Cependant, il se peut qu'après l'itération i un nouvel élément se trouve à la position $a[i-1]$ (par contre $a[i], \dots, a[N-1]$ sont les mêmes). Mais le nouvel élément qui se trouve à la position $a[i-1]$ est un des éléments qui se trouvaient avant le début de l'algorithme dans $a[0], \dots a[i-1]$, donc la proposition suivante reste vraie :

$$a[i-1] \leq a[i+1], a[i+2] \dots, a[N-1] \tag{4}$$

• Si $a[i-1] \leq a[i]$, on voit en utilisant (**??**) que $a[i-1]$ est le plus petit élément de $a[i-1], \dots, a[N-1]$ il est donc à la bonne place (puisque $a[0], \dots, a[i-2]$ sont tous à la bonne place il doit venir juste après). Et on voit on effet que l'algorithme ne modifie pas la liste.

• Si $a[i] < a[i-1]$, en utilisant (**??**) on voit que $a[i] < a[i-1] \leq a[i+1], a[i+2] \dots, a[N-1]$. Donc $a[i]$ est le plus petit élément de $a[i-1], \dots, a[N-1]$, il faut le mettre à la place de $a[i-1]$ (juste après $a[i-2]$), et c'est bien ce que fait l'algorithme.

Finalement, puisque l'algorithme fait $N-1$ itérations, et qu'il y a au plus un échange dans chaque itération, le nombre d'échanges sera au plus $N-1$. En fait il n'y a aucun input pour lequel $N-1$ échanges seront nécessaires, c'est une borne supérieure.

3. Tri par échanges adjacents

a) On fait par itération autant d'échanges que l'autre algorithme fait des mouvements. Comme un échange est plus cher, cet algorithme est moins vite que celui du cours, mais la différence n'est que d'un facteur constant ; en l'ordre les algorithmes sont pareils.

b) On a l'égalité suivante d'évenements :

$$\{|\sigma(i)-i|=j\} = \{\sigma(i)\in\{i+j,i-j\}\} = \{\sigma(i)=i+j\}\cup\{\sigma(i)=i-j\}.$$

Au moins l'une des quantités $i+j$ et $i-j$ appartient à $\{1,\dots,N\}$. Si par exemple $i+j\in\{1,\dots,N\}$ on a alors

$$P(|\sigma(i)-i|=j) \geq P(\sigma(i)=i+j).$$

Comme σ est uniformément choisi, on a $P(\sigma(i)=k)=1/N$ pour tout k tel que $1\leq k\leq N$, d'où le résultat.

c) En utilisant le résultat du point précédent, on calcule

$$\begin{aligned}
E[|\sigma_i - i|] &\geq \sum_{j=0}^{\lfloor N/2\rfloor} P(|\sigma_i-i|=j)\cdot j \\
&\geq \sum_{j=0}^{\lfloor N/2\rfloor} \frac{1}{N}\cdot j \\
&= \frac{1}{N}\cdot\frac{(\lfloor N/2\rfloor-1)\lfloor N/2\rfloor}{2} \\
&\geq \frac{1}{N}\cdot\frac{(N/2-2)(N/2-1)}{2} \\
&= \frac{1}{N}\cdot\frac{(N-4)(N-2)}{8} \\
&= \frac{N^2-6N+8}{8N} \\
&\geq \frac{N}{9},
\end{aligned}$$

où la dérnière inégalité n'est vraie que si N est assez grand.

d) Comme chaque transposition (échange adjacent) ne peut approcher un élément de sa position finale de 1, il faut pour chaque élément $\Omega(N)$ transpositions touchant l'élément en question. Une transposition touche au plus (en fait, exactement) deux éléments. Donc il faut effectuer

$$N\frac{\Omega(N)}{2} = \Omega(N^2)$$

transpositions en moyenne.

4. Dégénérescence de QuickSort

a) Si la suite est déjà triée, le pivot sera à chaque fois le plus grand élément. La suite sera donc coupée en une suite vide (à droite) et une suite consistant en tous les éléments sauf le pivot (à gauche). Il faudra donc faire $N-1$ étapes qui nécessiteront $1, \ldots, N-1$ comparaisons, donc au total $\Omega(N^2)$.

b) Il s'agit à nouveau de construire des suites de sorte que le pivot soit fréquemment un élément très grand (ou très petit) de la suite. Cette stratégie empêche qu'on puisse arranger la suite de sorte que le pivot soit le plus grand élément de la suite, mais c'est toujours possible que ça soit le deuxième plus grand élément.

Supposons que les clés sont les nombres $0, 1, \ldots, N-1$, et que N est pair. Alors l'arrangement

$$0, *, \cdots, *, (N-1), *, \cdots, *, a, (N-2),$$

(où le "$(N-1)$" se trouve à la position $N/2$, et les "$*$" dénotent n'importe quelle clé), fait que l'algorithme choisit $(N-2)$ comme pivot, puisque c'est la médiane de $(0, N-1, N-2)$. Après le premier pas de partitionnement de QuickSort on aura

$$\underline{0, *, \cdots, *, a, *, \cdots, *}, (N-2), (N-1),$$

où les $*$ n'ont pas changé de position. (Remarquons que QuickSort effectue $N-2$ comparaisons pour arriver à cette étape.)

On veut que le comportement pathologique ci-dessus se répète : pour y arriver, on peut disposer les éléments dans la sous-suite soulignée ci-dessus de manière analogue, i.e., positionner $(N-3)$ au milieu et $(N-4)$ tout à droite. Donc l'arrangement

$$0, *, \cdots, *, N-3, N-1, *, \cdots, *, N-4, *, N-2$$

fait que les deux premières itérations exhiberont un comportement pathologique.

En répétant la construction, on peut s'arranger pour que les premières $\lfloor N/4 \rfloor$ itérations seront de ce genre :

$$0, *, \cdots, N-\frac{2N}{4}-1, \cdots, N-3, N-1, \cdots, *, N-\frac{2N}{4}, *, N-\frac{2N}{4}+2, *, \cdots, N-4, *, N-2.$$

Pour effectuer chacune de ces itérations, $\geq N/2$ comparaisons sont nécessaires, donc au total $\geq N/4 \cdot N/2 = \Omega(N^2)$ comparaisons.

Remarque : Comme il n'était pas précisé que les éléments de la suite doivent être *distincts*, on peut trouver une solution bien plus simple : Si tous les éléments sont égaux ($a[0] = \ldots = a[N-1]$) il faudra $\Omega(N^2)$ comparaisons.

En effet, à chaque passage le pointeur p ne s'arrêtera que lorsqu'il arrivera au début de la suite (la position ℓ dans l'algorithme du cours), alors que q ne s'arrêtera qu'à la position $r-1$. Ainsi la sous-suite de droite sera toujours vide. Il faudra donc faire un passage de QuickSort sur une suite de taille N, puis $N-1$, puis $N-2$, etc... Comme chaque passage nécessite $\Omega(N)$ comparaisons, il faudra $\Omega(N^2)$ comparaisons au total.

5. Priority Queues

a) Ce sont les 15 arrangement suivants :

$$\begin{array}{lll} (4,2,5,1,3), & (4,1,5,3,2), & (4,3,5,1,2), \\ (5,3,4,1,2), & (5,2,4,1,3), & (5,1,4,3,2), \\ (3,5,4,1,2), & (1,5,4,3,2), & (2,5,4,1,3), \\ (1,3,4,5,2), & (2,1,4,5,3), & (3,1,4,5,2), \\ (1,2,4,3,5), & (3,2,4,1,5), & (2,3,4,1,5) \end{array}$$

b) L'idée est de mémoriser dans la queue de priorité les non-premiers et de ainsi simuler un crible d'Erastosthène. On peut alors successivement tester si un nombre donné est premier en testant s'il est le prochain élément dans la queue de priorité. Une première version de cet algorithme serait alors la suivante. (Dans l'algorithme suivant, H est une queue de priorité stockant des entiers, et tel que l'élément prioritaire $\top[H]$ est toujours l'élément le plus petit.

Input: Un entier N indiquant la fin de la queue de priorité
Output: Une suite croissante de tous les premiers entre 1 et N

```
H ← {4, 6, 8, ..., ⌊N/2⌋ · 2}
print 2
i ← 3
while i ≤ N do
  x ← ⊤[H]
  if x > i then
    for j = 2 · i, 3 · i, ..., ⌊N/i⌋ · i do
      H ← H ∪ {j}
    print i
  else
    while x = i do
      pop(H)
      x ← ⊤[H]
  i ← i + 1
```

Le désavantage de cette solution est qu'il est très couteux de stocker tous les multiples d'un premier dans la queue de priorité. Une petite refléxion montre qu'il suffit de stocker le prochain multiple d'un premier donné si l'on sait de quel premier c'est un multiple : à chaque fois qu'un tel nombre est enlevé de la queue de priorité, il suffit de rajouter le prochain multiple du même premier dans la queue. Pour pouvoir le faire, il faut stocker des couples (x, p), où p est le premier en question est x est son prochain multiple. L'algorithme suivant réalise cet approche avec deux modifications en plus : Les nombres pairs ne sont pas considérés, et les premiers $> \sqrt{N}$ ne sont pas inserés dans la queue puisque les nombres composés $\leq N$ ont toujours un facteur $\leq \sqrt{N}$.

Input: Un entier N indiquant la fin de la queue de priorité
Output: Une suite croissante de tous les premiers entre 1 et N

```
H ← {(9, 3)}
print 2
print 3
for i = 3, 5, 7, 9, ..., ⌊N/2⌋ · 2 do
```

```
(x, p) ← ⊤[H]
if x > i then
  print i
  if i ≤ √N then
    H ← H ∪ {(3 · i, i)}
else
  while x = i do
    pop(H)
    H ← H ∪ {(i + 2p, p)}
    (x, p) ← ⊤[H]
```

c) L'opération SiftUp réalise le cœur d'une insertion : il suffit d'ajouter l'élément à la fin et de faire un SiftUp par la suite :

Input: Un heap (a, N), où a est un tableau, et N le nombre d'éléments dans le heap. Un élément e à insérer.
Output: (a, N) modifiés.

```
a[N] ← e
N ← N + 1
SiftUp(a, N − 1)
```

De manière similaire, on peut aussi effacer un élément en utilisant les opérations SiftUp et SiftDown. Néanmoins, la procédure correcte est légèrement plus compliquée que pour une insertion.

Input: Un heap (a, N) et un indice k d'un élément à enlever.
Output: (a, N) modifiés.

```
a[k] ← a[N − 1]
N ← N − 1
if k < N then
  if a[⌊k/2⌋] < a[k] then
    SiftUp(a, k)
  else
    SiftDown(a, k)
```

6. Sac à dos 0/1 avec des poids rationnels

a) On fait n itérations, chacune à $\theta(W)$ opérations, donc en tout, c'est $\theta(nW)$.

b) On peut appliquer l'algorithme aussi si les valeurs sont des rationnels. Rien ne change, sauf que les entrées de la matrice C seront maintenant aussi des rationnels.

c) Si $W, (v_1, w_1), \ldots, (v_n, w_n)$ est un problème du sac à dos ayant une solution optimale, alors pour n'importe quelle constante $c > 0$, le problème du sac à dos de paramètres $c \cdot W, (v_1, c \cdot w_1), \ldots, (v_n, c \cdot w_n)$ est équivalent.

Si l'on multiplie W et les w_i par $q_1 q_2 \cdots q_n$, on obtient un problème avec des coefficients entiers (supposant que W était entier) ; et on peut donc appliquer l'algorithme 0/1 du cours à ce problème modifié.

Comme le temps de parcours est proportionnel à W, on a intérêt à avoir ce nombre aussi petit que possible. Le plus petit entier avec lequel on peut multiplier les poids du problème

original pour avoir un problème avec des coefficients entiers est le plus petit multiple des q_i, $\mathrm{lcm}(q_1, \ldots, q_n)$.

d) Si on prend les q_i premiers entre-eux (par exemple, des premiers distincts), on aura $\mathrm{lcm}(q_1, \ldots, q_n) = q_1 q_2 \cdots q_n$. Donc, si la valeur de W était initialement 1, on aura $W = q_1 q_2 \cdots q_n$ et, si les $w_i \leq 1$,

$$\begin{aligned} \mathrm{length}(w_1, \ldots, w_n) &= \sum_{i_1}^{n} \log(q_i) + \log(p_i) \\ &\leq 2 \sum_{i_1}^{n} \log(q_i) \\ &= 2 \log(W), \end{aligned}$$

donc $W \geq \sqrt{2}^{\mathrm{length}(w_1,\ldots,w_n)}$. L'algorithme du cours a donc un temps de parcours

$$\Omega(nW) = \Omega\big(n(\sqrt{2})^{\mathrm{length}(w_1,\ldots,w_n)}\big) = \Omega\big((\sqrt{2})^{\mathrm{length}(w_1,\ldots,w_n)}\big).$$

e) La fonction $\mathrm{length}(\cdot)$ représente la longueur de l'input (si on suppose qu'on prend toujours $W = 1$ et donc que W ne fait pas partie de l'input). En effet, on a besoin de $\log(p_i)$ bits pour représenter p_i, de $\log(q_i)$ bits pour représenter q_i et donc de $\log(p_i) + \log(q_i)$ bits pour w_i.