Algorithmique — IN102
PC 1

2 décembre 2005

Exercice 1 *Lors du premier parcours du tableau, l'élément le plus grand va «remonter» à la dernière position (tel une bulle remontant à la surface, d'où le nom du tri). Lors de la seconde passe, le suivant (le «deuxième plus grand») va faire de même et aller se placer en avant-dernière position, et ainsi de suite. Par conséquent, après k parcours, les k plus grands éléments sont placés. Si on applique ce résultat à n, on voit qu'au plus n passes sont nécessaires pour trier le tableau.*

Exercice 2 *L'algorithme du tri à bulles (tableau tab à n éléments) s'écrit :*

```
void triBulles(int *tab, int n) {
  for (int max=n-1; max > 0; max--)
    // positionner le maximum de tab[0..max] en tab[max]
    for (int i=0; i < max; i++)
      if (tab[i] > tab[i+1]) {
        // échange de tab[i] et tab[i+1] de sorte que tab[i+1] ≥ tab[i]
        int temp = tab[i]; tab[i] = tab[i+1]; tab[i+1] = temp;
      }
}
```

Exercice 3 *La complexité de l'algorithme est indépendante du contenu du tableau : elle est donc la même dans tous les cas (cas le pire, et en moyenne). En termes du nombre de comparaisons, pour un tableau de n éléments, la complexité est de :*

$$(n-1) + (n-2) \ldots + 2 + 1 = \frac{(n-1) \times (n-2)}{2} = \Theta(n^2)$$

En effet, quelle que soit la configuration des éléments dans le tableau, on effectue à chaque passage max comparaisons pour max variant de $n-1$ à 1.

Exercice 4 *La fusion parcourt les deux sous-tableaux en parallèle, sélectionnant à chaque étape l'élément le plus petit des deux tableaux, et s'arrête lorsqu'on a épuisé les 2 sous-tableaux. Lorsque le premier des deux sous-tableaux aura été épuisé, il faudra sélectionner tous les éléments restants du second. La fusion nécessitant un tableau auxiliaire, la solution consiste à placer les 2 sous-tableaux à fusionner « dos à dos » dans ce tableau intermédiaire, puis à parcourir ce tableau depuis chaque extrémité. De la sorte, lorsqu'un des sous-tableaux aura été épuisé, le « dos » du second bloquera la progression de l'indice parcourant le tableau épuisé en premier.*

On considère ici que le tableau intermédiaire est défini globalement à la procédure de tri, et que sa taille est la même que celle du tableau à trier :

```
void fusion(int *tab, int p, int q, int r) {
  int i, j, k;
  // Recopie des deux sous-tableaux « dos à dos » dans tmp
  for (i = p; i ≤ q; i++) tmp[i] = tab[i];
  for (i = q+1; i ≤ r; i++) tmp[r+q+1-i] = tab[i];
  // La fusion proprement dite
  i = p; j = r;
  for (k = p; k ≤ r; k++)
    if (tmp[i] < tmp[j]) {
```

```
    tab[k] = tmp[i]; i = i+1;
  } else {
    tab[k] = tmp[j]; j = j-1;
  }
}
```

La fusion des deux sous-tableaux (contenant au total $r-p+1$ éléments) nécessite exactement $r-p+1$ comparaisons, et une copie initiale des $r-p+1$ éléments. Sa complexité en temps et en espace est donc linéaire en $r-p$.

Exercice 5 *Le tri par fusion utilise une technique de résolution de problèmes appelée en Anglais «*divide and conquer*», ce qui correspond à notre « diviser pour régner ». Le tri d'un tableau à $n > 1$ éléments se décompose en 3 phases. La complexité $T(n)$ recherchée se décompose en celles de chacune de ces phases :*

1. *la* **division** *: calcul du milieu $q = (p+r)/2$, en temps constant $D(n) = 1$.*
2. *la* **conquête** *: traitement des deux sous-problèmes de taille $n/2$. Cette phase est donc de complexité $2 \times T(n/2)$.*
3. *la* **recombinaison** *: fusion des résultats du traitement des sous-problèmes, c'est-à-dire la fusion. Elle contribue à hauteur de $C(n) = n$ à la complexité globale.*

Par ailleurs, le traitement d'un tableau à un seul élément prend un temps constant.

La complexité $T(n)$ du tri par fusion d'un tableau de taille n est solution de l'équation récursive :

$$T(n) = \begin{cases} 1 & si\ n = 1 \\ n + 2 \times T(n/2) & si\ n = 2^k \qquad (k > 0) \end{cases}$$

Puisque $n = 2^k$, on a :

$$\begin{aligned} T(n) &= 2^k + 2 \times T(2^{k-1}) \\ &= 2^k + 2 \times 2^{k-1} + 2^2 \times 2^{k-2} + \ldots 2^{k-1} \times 2 + 2^k \times 1 \\ &= (k+1) \times n \\ &= \Theta(n \times \log(n)) \end{aligned}$$

Plus généralement, $T(n)$ vérifie la relation de récurrence :

$$T(n) = 2 \times T(n/2) + \Theta(n)$$

et, si $n = 2^k$, le développement de $T(n)$ produit la somme de k termes $\Theta(n)$, et donc $T(n) = \log(n) \times \Theta(n) = \Theta(n \times \log(n)$.

De plus, cette complexité est indépendante du contenu du tableau, et est donc valable dans le cas moyen comme dans le cas le pire.

Exercice 6 *Tout entier n peut être encadré entre deux puissances de 2 N et $2 \times N$ ($N \leq n < 2 \times N$). Il est de plus raisonnable d'admettre que T est croissante, c'est-à-dire que $T(N) \geq T(n) < T(2 \times N)$. Par conséquent, en utilisant le résultat de l'exercice précédent, $\Theta(N \log(N)) \geq T(n) < \Theta(2 \times N \times \log(N)) = \Theta(N\ log(N))$. Donc, $T(n) = \Theta(n \times \log(n))$.*

$$b = \sum_{i=0}^{k} b_i \times 2^i \quad \text{avec} \quad b_i \in \{0, 1\}$$

On peut ensuite remarquer

$$a^b = a^{\left(\sum_{i=0}^{k} b_i \times 2^i\right)} = \prod_{i=0}^{k} a^{b_i \times 2^i} = \prod_{i=0}^{k} \left(a^{(2^i)}\right)^{b_i}$$

Afin de calculer a^b, il suffit donc d'être capable de calculer les $a^{(2^i)}$ qui peuvent s'obtenir par élévation successive au carré et d'en multiplier ensuite certains entre eux (ceux pour lesquels b_i vaut 1).

Exercice 7 *L'algorithme naïf permettant de calculer a^b effectue $(b-1)$ multiplications successives par a. Sa complexité est donc $\Theta(b)$ multiplications, si on considère une multiplication entre deux entiers comme une opération élémentaire.*

Si maintenant on s'intéresse à des opérations «bit à bit», on voit que la complexité de la multiplication de x par y est en $\Theta(\log(x) \times \log(y))$ (par exemple en utilisant l'algorithme employé à la main). La complexité de l'algorithme naï en termes d'opérations bit à bit est donc :

$$\begin{aligned}
&\Theta(\log(a) \times \log(a) + \log(a^2) \times \log(a) + \ldots + \log(a^{b-1}) \times \log(a)) \\
&\quad = \Theta\left(log(a)^2 \times \sum_{i=1}^{b-1} i\right) \\
&\quad = \Theta\left(\log(a)^2 \times \frac{b(b-1)}{2}\right) \\
&\quad = \Theta\left(\log(a)^2 \times b^2\right)
\end{aligned}$$

Exercice 8 *Afin de décomposer un entier b en base 2, on peut commencer par calculer le bit b_0. Si $b_0 = 0$, alors b est pair et si $b_0 = 1$, b est impair. Par conséquent, b_0 est le reste de la division euclidienne de b par 2. Le même raisonnement appliqué au quotient de la division de b par 2 va permettre de claculer b_1 et ainsi de suite. On obtient l'algorithme suivant (en pseudo C) :*

```
binaire(int b) {
  int i = 0;
  while (b ≠ 0) {
    b_i = b/2;
    b = b % 2;
    i = i+1;
  }
  return (b_0, ..., b_{i-1});
}
```

La boucle principale de cet algorithme est répétée tant que b n'est pas nul. Afin de déterminer la complexité temporelle de cet algorithme, il faut être capable d'estimer le nombre de fois que l'on peut diviser un entier par 2 avant d'obtenir 0.

Dans le cas très particulier où b est une puissance de 2 ($b = 2^k$), on voit que l'on doit le diviser $k+1$ fois par 2 avant d'obtenir 0. Puisque $k = \log(n)$, la complexité est dans ce cas $\Theta(\log(b))$.

Dans le cas général, il existe un entier k tel que $2^k \leq b < 2^{k+1}$. Cet entier k vaut $\lfloor \log(b) \rfloor$ (la partie entière inférieure de $\log(b)$), soit approximativement $\log(b)$. Quelque soit b, la complexité de la décomposition en binaire est donc $\Theta(\log(b))$.

Exercice 9 *En utilisant l'équation $a^b = a^{\left(\sum_{i=0}^{k} b_i \times 2^i\right)} = \prod_{i=0}^{k} \left(a^{(2^i)}\right)^{b_i}$, et en remarquant que $k = \Theta(\log(b))$, on obtient l'algorithme d'exponentiation suivant :*

```
int exponentiation(int a, int b) {
  int s_0 = a;
  for (int i = 1; i ≤ k; i++)
     s_i = (s_{i-1})^2;
   int p = 1;
   for (int i = 1; i ≤ k; i++)
     if (b_i == 1)
       p = p×s_i;
   return p;
}
```

Cet algorithme commence par calculer les $s_i = a^{(2^i)}$ avant de multiplier ceux pour lesquels le bit associé b_i vaut 1. La complexité en fonction de k est facile à calculer. En effet, la première boucle effectue k élévations au carré et la seconde au plus k multiplications. De plus, k étant de l'ordre de $\log(b)$, la complexité temporelle de l'exponentiation est $\Theta(\log(b))$ multiplications. De plus, la première étape ne dépendant que de la taille de l'exposant, cette complexité est valable que ce soit en moyenne ou dans le cas le pire.

Enfin, dans l'algorithme décrit précédemment, il est nécessaire de stocker les s_i. La complexité spatiale est donc $\Theta(\log(b))$ entiers.

Exercice 10 *Il est facile d'éviter le stockage des s_i en utilisant les valeurs au fur et à mesure de leur calcul. Il est également possible d'intégrer directement la décomposition en base 2. On obtient alors l'algorithme suivant :*

```
int exponentiation(int a, int b) {
  int s = a; int p = 1;
  while (b ≠ 0) {
    if ((b % 2) == 1) p = p×s;
    s = s²;
    b = b/2;
  }
  return p;
}
```

On peut aussi écrire cet algorithme de façon récursive :

```
int puissance(int a, int b) {
  if (b == 0) return 1;
  if ((b % 2) == 1)
    return a × (puissance(a×a, b/2));
  else
    return puissance(a×a, b/2);
}
```

Exercice 11

1. *L'algorithme précédent s'adapte très simplement en effectuant des réductions modulaires dès que possible (et surtout pas à la fin du calcul uniquement).*

   ```
   int puissance (int a, int b, int n) {
     int p=1;
     while (b>0) {
       if ((b%2) == 1) p = (p×a)%n;
       a = (a×a)%n;
       b = b/2;
     }
     return p;
   }
   ```

2. *Avec des exposants de 1024 bits, on effectue 1024 élévations au carré et de l'ordre de 512 multiplications en moyenne. selon le nombre de bits valant 1 dans l'exposant. Un chiffrement nécessite donc de l'ordre de 1500 multiplications modulaires, ce qui est important, mais réalisable sur une carte à puce, par exemple.*

3. *$e = 65537 = 2^{16} + 1$; l'avantage de cet exposant est de ne nécessiter que deux multiplications et non 8 comme c'est le cas en moyenne.*