1 Tri à bulles

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 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. Ce résultats appliqué à $k = n$, le nombre total d'éléments du tableau, indique qu'au plus n passes sont nécessaires pour trier le tableau.

Exercice 2

Une application directe de l'exercice précédant fournit l'algorithme suivant pour trier un tableau A indicé de 1 à n :

Algorithme 1 TRI-BULLE(A, n)
1 **pour** $i \leftarrow 1$ **à** $n-1$ **faire**
2 **pour** $j \leftarrow 1$ **à** $n-i$ **faire**
3 **si** $A[j] > A[j+1]$ **alors**
4 échanger $A[j]$ et $A[j+1]$

Il est également possible de chercher à optimiser l'algorithme afin de s'arrêter si aucun échange n'a été réalisé au cours d'une passe. En pratique, ceci ne modifie que très peu la complexité.

Exercice 3

Remarquons tout d'abord que la complexité de l'algorithme tel que nous l'avons décrit est totalement indépendante du contenu du tableau. La complexité dans le cas le pire est donc identique à celle en moyenne.

Le nombre de comparaisons est égal à

$$\sum_{i=1}^{n-1}\sum_{j=1}^{n-i} 1 = \sum_{i=1}^{n-1} n-i = \sum_{i'=1}^{n-1} i' = \frac{n(n-1)}{2} = \Theta(n^2)$$

Le tri a bulle est donc un tri quadratique, tout comme le tri par insertion.

2 Tri fusion

Exercice 4

Afin de fusionner deux tableaux triés, il faut tout d'abord trouver le plus petit élément figurant dans les deux. Or, cet élément est nécessairement le premier élément d'un des deux tableaux. Par conséquent, nous allons devoir maintenir deux indices qui parcourent les deux

tableaux respectivement, par ordre croissant. La principale difficulté consiste à ne pas "déborder" d'un des deux tableaux, d'où l'algorithme suivant utilisant comme astuce l'inversion initiale du second tableau :

Algorithme 2 FUSION(A, p, q, r)

```
1  pour i ← p à q faire
2      B[i] = A[i]
3  pour i ← q + 1 à r faire
4      B[r + q + 1 − i] = A[i]
5  i ← p
6  j ← r
7  pour k ← p à r faire
8      si B[i] < B[j] alors
9          A[k] ← B[i]
10         i ← i + 1
11     sinon
12         A[k] ← B[j]
13         j ← j − 1
```

Cet algorithme commence par effecture $r - p + 1$ affectations puis exécute la boucle principale $r - p + 1$ fois. Sa complexité en temps et en espace est donc linéaire en $r - p$.

Exercice 5

Si $n = 2^k$, la complexité $C(n)$ du tri fusion d'un tableau de n éléments vérifie la relation de récurrence suivante :

$$C(n) = C(n/2) + C(n/2) + \Theta(n)$$

où les $C(n/2)$ indiquent la complexité des tris de chaque moitié et le $\Theta(n)$ la complexité de leur fusion.

On a donc

$$C(n) = 2C(n/2) + \Theta(n) = 2(2C(n/4) + \Theta(n/2)) + \Theta(n) = ... = \log_2 n \times \Theta(n)$$

La complexité de l'algorithme est donc en $\Theta(n \log n)$. De plus, cette complexité est totalement indépendante du contenu du tableau ; elle est donc également valable dans le cas le pire, ce qui en fait un avantage important par rapport au tri rapide ("quicksort").

Exercice 6

Dans le cas d'entiers n non puissance de 2, le découpage des tableaux est déséquilibré et les calculs de complexités peuvent être rendus très complexes. Nous pouvons cependant remarquer que tout entier n peut être encadré entre N et $2 \times N$ avec N une puissance de 2 ($N \leq n < 2 \times N$).

Il est de plus raisonnable d'admettre que $C(N) \leq C(n) < C(2 \times N)$. Par conséquent, $\Theta(N \log(N)) \leq C(n) < \Theta(2N \log(2N))$. Mais $\Theta(2N \log(2N)) = \Theta(N \log(N)) = \Theta(n \log(n))$ donc $C(n) = \Theta(n \log(n))$ pout tout n.

3 Exponentiation binaire

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 l'on considère une multiplication entre deux entiers comme étant 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 naïf en terme d'opérations bit-à-bit est donc

$$\theta\left(\log(a) \times \log(a) + \log(a^2) \times \log(a) + ... \log(a^{b-1}) \times \log(a))\right)$$

$$= \theta\left(\log(a)^2 \left(\sum_{i=1}^{b-1} i\right)\right) = \theta\left(\log(a)^2 \times \frac{b(b-1)}{2}\right)$$

$$= \theta\left(\log(a)^2 \times b^2\right)$$

En pratique, ce calcul est bien compliqué. Il est bien plus efficace de se dire que l'on a $b-1$ multiplications entre a et a^i, soit une complexité de $\theta(b) \times \theta(\log a) \times \theta(\log(a^b))$ opérations élémentaires bit à bit.

Exercice 8

Afin de décomposer un entier b en base 2, on peut commencer par calculer b_0. Si $b_0 = 0$, b est nécessairement pair et inversement si $b_0 = 1$, b est impair. Par conséquent, b_0 est égal au reste de la division euclidienne de b par 2. Ensuite, le même raisonnement appliqué au quotient de la division (b div 2) va permettre de calculer b_1 et ainsi de suite. On obtient donc l'algorithme suivant :

Algorithme 3 BINAIRE(b)
1 $i \leftarrow 0$
2 **tant que** $b \neq 0$ **faire**
3 $\quad b_i \leftarrow b \bmod 2$
4 $\quad b \leftarrow b$ div 2
5 $\quad i \leftarrow i + 1$
6 **renvoyer** $(b_0, ...b_{i-1})$

La boucle principale de cet algorithme est répétée tant que b n'est pas nul. Afin de déterminer la complexité temporelle il faut donc 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 égal à une puissance de 2, $b = 2^k$, on voit que l'on doit le diviser $k+1$ fois par 2 avant d'obtenir 0. Or k n'est autre que le logarithme en base 2 de b. la complexité du calcul est donc $\Theta(\log b)$.

Dans le cas général, il existe un entier k tel que $2^k \leq b < 2^{k+1}$. De plus, cet entier k vaut $\lfloor \log_2 b \rfloor$ ("partie entière de $\log_2 b$") soit approximativement $\log_2 b$. 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 :

Algorithme 4 EXPONENTATION(a, b)
1 $s_0 \leftarrow a$
2 **pour** $i \leftarrow 1$ **à** k **faire**
3 $\quad s_i \leftarrow (s_{i-1})^2$
4 $p \leftarrow 1$
5 **pour** $i \leftarrow 1$ **à** k **faire**
6 $\quad$ **si** $b_i = 1$ **alors**
7 $\quad\quad p \leftarrow p \times s_i$
8 **renvoyer** 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 du logarithme de 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 précédamment décrit, il est nécessaire de stocker les s_i. La complexité spatiale est donc également $\Theta(\log b)$ entiers.

Exercice 10

Il est cependant facile d'éviter le stockage de s_i en utilisant les valeurs au fure et à mesure. Il est également possible d'intégrer directement la décomposition en base 2. On obtient alors l'algorithme suivant :

Algorithme 5 EXPONENTATION(a, b)
1 $s \leftarrow a$
2 $p \leftarrow 1$
3 **tant que** $b \neq 0$ **faire**
4 $\quad$ **si** $b = 1 \bmod 2$ **alors**
5 $\quad\quad p \leftarrow p \times s$
6 $\quad s \leftarrow s^2$
7 $\quad b \leftarrow b \operatorname{div} 2$
8 **renvoyer** p

Ceci nous donne le programme suivant en C :

```
int puissance(int a, int b)
{
  int p=1;
  while (b>0)
```

```
    {
      if ((b%2)==1)
        p=p*a;
      a=a*a;
      b=b/2;
    }
  return p;
}
```

Il est également possible d'écrire ceci de manière récursive, peut-être plus lisible :

```
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 précédant s'adapte très simplement en effectauant des réduction modulaires dés 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.
3. $e = 65537 = 2^{16} + 1$; l'avantage de cet exposant est de ne nécessité que deux multiplication et non 8 comme c'est le cas en moyenne.