Algorithmique TD2
L3 Informatique – ENS Cachan

Guillaume Bury

25 septembre 2017

Exercice 1

Expliquer et prouver le comportement de la fonction 91 de McCarthy :

```
fonction f91(x: entier) {
  if x > 100 then
    return x - 10
  else
    return f91(f91(x + 11))
  }
```

Exercice 2

1. On considère un tableau $T[1..n]$ d'entiers naturels. On modifie T par le procédé itératif suivant : si on peut trouver deux indices $1 \leq i < j \leq n$ tels que $T[i] > T[j]$, alors on échange $T[i]$ et $T[j]$ et on essaye à nouveau, sinon on stoppe. Remarquons que ce procédé est non déterministe puisque dans le cas où existent plusieurs paires (i, j) « telles que... » on peut choisir l'une quelconque de ces paires. Que fait ce procédé ?
2. On modifie la procédure d'échange. Soient deux indices $i < j$ tels que $T[i] > T[j]$. L'échange consiste maintenant à choisir deux nouvelles valeurs a_i et a_j telles que $T[i] \geq a_j \geq a_i \geq T[j]$ et à effectuer $T[j] \leftarrow a_j$, $T[i] \leftarrow a_i$. Autrement dit, on peut rapprocher les valeurs des éléments échangés. Par exemple, on pourra passer de (0, 30, 20, 40) à (0, 22, 28, 40). Que peut-on dire de ce nouveau procédé ?
3. On modifie une dernière fois la procédure d'échange. Soient deux indices $i < j$ tels que $T[i] > T[j]$. L'échange consiste maintenant à choisir deux nouvelles valeurs a_i et a_j telles que $T[i] \geq a_j \geq a_i \geq T[j]$ et $a_j > T[j]$ puis à effectuer $T[j] \leftarrow a_j$, $T[i] \leftarrow a_i$. Autrement dit, $T[j]$ doit forcément croître après l'opération. Que peut-on dire de ce dernier procédé ?

Exercice 3

1. Donner la complexité en temps de l'algorithme vu à l'école primaire pour multiplier deux entiers.
2. En observant que

$$(a \times 10^k + b)(c \times 10^k + d) = ac \times 10^{2k} + (ac + bd - (a - b)(c - d)) \times 10^k + bd$$

   proposer un algorithme de type « diviser pour régner » et calculer sa complexité en temps.
3. Adapter l'algorithme à la multiplication de deux polynômes.

```
Select(T : int array,
       n : int, k : int)
  n_1 <- 0; n_2 <-0; n_3 <- 0;
  T_1 <- Array.make n 0; T_3 <- Array.make n 0;
  // On sélectionne une valeur du tableau
  valeur <- T[1];
  for i = 1 to n do
    if T[i] < valeur then
      n_1 <- n_1+1;
      T_1[n_1] <- T[i];
    else if T[i] = valeur then
      n_2 <- n_2+1;
    else
      n_3 <- n_3+1;
      T_3[n_3] <- T[i];
  done
  // On recherche l'élément dans un des tableaux par un appel récursif ou
  // on renvoie valeur suivant les valeurs de n_1, n_2 et n_3
  if n_1 >= k then
   return Select(T_1,n_1,k);
  else if n_1 + n_2 >= k then
   return valeur;
  else
    return Select(T_3,n_3, k - (n_1 + n_2 ) );
```

FIGURE 1 – Un algorithme par partition

Exercice 4

Soit T un tableau de n cellules contenant des valeurs numériques et $k \leq n$ un entier.

1. Ecrire un algorithme naïf qui renvoie la $k$ième plus petite valeur du tableau T et l'indice correspondant dans le tableau.
2. Ecrire un algorithme alternatif avec tri.
3. Comparer la complexité des deux algorithmes. Indiquez les cas favorables à l'un ou l'autre des algorithmes.
4. Soit l'algorithme 1. Démontrer que cet algorithme renvoie la $k$ième plus petite valeur du tableau T. Quel type de partition rend l'algorithme le plus efficace ?
5. En tenant compte de la question précédente, on propose l'algorithme 2 qui est une adaptation du précédent. Démontrer que n_1 et n_3 sont tous les deux inférieurs à $3n/4$.
6. Soit $Time(n)$, le pire temps d'exécution de cet algorithme pour un tableau de taille inférieure ou égale à n. Démontrer que pour une constante c bien choisie :

$$\forall n < 50, \quad Time(n) \leq c \cdot n$$

$$\forall n \geq 50, \quad Time(n) \leq c \cdot n + Time(\frac{n}{5}) + Time(\frac{3n}{4})$$

En déduire la complexité de cet algorithme.

```
Select(T : int array,
       n : int, k : int)
R <- Array.make n 0; S <- Array.make n 0;
T_1 <- Array.make n 0; T_3 <- Array.make n 0;
mediane <- 0; m <- 0; n_1 <- 0; n_2 <- 0; n_3 <- 0; i <- 0; j <- 0;
if n < 24 then
  // On applique l'algorithme de la question 3
  S <- Trie(T,1,n);
  return S[k];
else
  // On divise T en paquets de 5 éléments et
  // on range dans R, l'élément médian de chaque paquet
  m <- floor(n/5);
  for i = 0 to m -1 do
    for j = 1 to 5 do
      S[j] <- T[5*i+j]};
    done;
    S <- Trie(S,1,5);
    R[i+1] <- S[3];
  done
  // On calcule le médian des médians par un appel récursif
  mediane <- Select(R,m, ceil(m/2));
  for i = 1 to n do
    if T[i] < mediane then
      n_1 <- n_1+1;
      T_1[n_1] <- T[i];
    else if T[i] = mediane then
      n_2 <- n_2+1;
    else
      n_3 <- n_3+1;
      T_3[n_3] <- T[i];
  done;
  // On recherche l'élément dans un des tableaux par un appel récursif ou
  // on renvoie mediane suivant les valeurs de n_1, n_2 et n_3
  if n_1 >= k then
    return Select(T_1,n_1,k);
  else if n_1 + n_2 >= k then
    return mediane;
  else
    return Select(T_3,n_3, k - (n_1 + n_2) );
```

FIGURE 2 – Algorithme par partition optimisé