IF111 - Algorithmes et structures de données
EI2 - Récursivité et Diviser pour régner

Rohan Fossé

rfosse@labri.fr

1 Récursivité

Exercice 1.1

Pour chacune des fonctions suivantes, dire pour quelles valeurs du paramètre n elle termine et ce qu'elle fait ou calcule. Ces fonctions sont-elles récursives terminales ?

```
1: function f(n)
2:     if n == 0 then
3:         return 1
4:     else
5:         return f(n + 1)
```

```
1: function sommeBis(n)
2:     if n == 0 then
3:         return 0
4:     else
5:         result ← sommeBis(n − 1)
6:         result ← result + n
7:         return result
```

```
1: function g(n)
2:     if n <= 1 then
3:         return 1
4:     else
5:         return 1 + g(n − 2)
```

Exercice 1.2

Construire la forme récursive de la fonction $g()$ définie ci-dessous.

```
1: function g(n)
2:     i ← 0
3:     for i < n do
4:         afficher(i)
```

Exercice 1.3

Écrire une fonction récursive qui calcule la somme de nombres de 1 à n, si n > 0 et renvoie 0 sinon. Quelle est la complexité ?

Exercice 1.4

Donner un algorithme récursif pour calculer x^n , x et n positifs non nuls. Quelle est la complexité ? Peut-on calculer x^n avec moins de multiplications ?

Exercice 1.5

Écrire une fonction récursive $pgcd(m, n)$ qui calcule le plus grand diviseur commun des deux entiers (non-négatifs) m et n.

Exercice 1.6

Écrire un algorithme récursif qui prend un paramètre n et qui teste si n contient au moins un zéro dans son écriture en base 10. On fait ici la convention que l'écriture en base 10 de zéro est zéro. Quelle est la complexité de cet algorithme ?

Exercice 1.7

On considère les fonctions mutuellement récursives suivantes :

```
1: function u(n)
2:     if n == 0 then
3:         return 1
4:     else
5:         return u(n − 1) + v(n − 1)

6: function v(n)
7:     if n == 0 then
8:         return 0
9:     else
10:        return 2 ∗ u(n − 1) + v(n − 1)
```

1. Calculer u(0), u(1), u(2), u(3), v(0), v(1), v(2) et v(3).
2. Montrer que la suite (u_n) vérifie la récurrence, $u_{n+2} = 2u_{n+1} + u_n$.

Exercice 1.8

Décrire une fonction récursive qui, étant donné un entier x, détermine la valeur la plus proche de x dans un tableau d'entiers.

2 Diviser pour régner

Exercice 2.1

Un site internet cherche à regrouper ses membres en fonction des goûts musicaux de chacun. Pour cela, chaque membre doit classer par ordre de préférence une liste d'artistes 1.

On dit que deux membres, Arthur et Béatrice, ont des goûts musicaux proches lorsque qu'il y a peu d'inversions dans leurs clas-sements : une inversion est une paire d'artiste L, M telle qu'Arthur préfère L à M et Béatrice préfère M à L.

On cherche donc à compter le nombre d'inversion dans les classements d'Arthur et Béatrice.

1. Compter le nombre d'inversion les classements suivants :

   Arthur: Britney Spears, Lady Gaga, Michael Jackson, Madonna, Céline Dion ;

Béatrice: Lady Gaga, Madonna, Britney Spears, Michael Jackson, Céline Dion.

2. Proposer un algorithme naïf qui résout le problème. Quelle est sa complexité ?

   On cherche maintenant à améliorer l'algorithme précédent en utilisant le paradigme Diviser-Pour- Régner. Pour cela, on coupe le classement de chaque membre en deux sous-classements de même taille, celui des artistes préférés (classement supérieur) et celui des autres artistes (classement inférieur). On compte alors les inversions (L, M) qui peuvent être de deux types : soit L et M apparaissent dans le même sous-classement de Béatrice, soit L et M apparaissent dans deux différents sous-classements de Béatrice (inversions mixtes).

3. On suppose que les deux sous-classements de Béatrice sont triés en fonction du classement d'Arthur. Montrer qu'on peut alors compter les inversions mixtes en temps linéaire.

4. Donner un algorithme de type Diviser-Pour-Régner qui fonctionne en temps $\mathcal{O}$(n log n).

Exercice 2.2

Écrire un algorithme diviser-pour-regner pour calculer la position du plus grand élément d'un tableau d'entiers.
Écrire la récurrence pour le nombre de comparaisons faites par l'algorithme (on comptera seulement les comparaisons entre les éléments du tableau).

Exercice 2.3

Soit E une suite de n éléments rangés dans un tableau $E[1..n]$. On dit qu'un élément $x \in E$ est *majoritaire* si le cardinal de l'ensemble E_x=$\{y \in E$ tel que $y = x\}$ est strictement plus de $\frac{n}{2}$. On supposera que n est une puissance de 2. On suppose que la seule opération qu'on sait effectuer sur les éléments du tableau est de lire la valeur d'un élément et de vérifier si deux éléments sont égaux ou non.

1. Écrire un algorithme itératif pour vérifier si E possède un élément *majoritaire*. L'algorithme retourne la couple (x, c_x) si x est l' élément *majoritaire* avec c_x occurrences et la couple $(-, 0)$ autrement.

   Quel est le nombre de comparaisons à faire dans le pire des cas?

2. En s'appuyant sur la technique *diviser pour régner*, donner un algorithme récursif basé sur un découpage de E en deux sous-tableaux de même taille. L'algorithme $Majoritaire(E, i, j)$ retourne la couple (x, c_x) si x est l' élément *majoritaire* avec c_x occurrences dans le (sous)tableau $E[i..j]$ et la couple $(-, 0)$ autrement. L'appel initial sera $Majoritaire(E, 1, n)$. L'algorithme utilisera un algorithme $Occurence(E, x, i, j)$ qui retourne le nombre d'occurrence de x dans le (sous)tableau $E[i, j]$.

   Donner le nombre de comparaisons faites par l'algorithme pour un tableau de taille n. L'algorithme $Occurence(E, x, i, j)$ exécute un nombre de comparaisons linéaire dans la taille du tableau $E[i, j]$.

Exercice 2.4

On veut compter le nombre de 0 dans un tableau T de n cases. Chaque case de T contient 0 ou 1 et tous les 0 précédent les 1. Ecrire un algorithme récursif qui détermine le nombre de 0 en $O(logn)$ opérations élémentaires. Pour cet algorithme on s'inspirera à la recherche par dichotomie.

3 Complexité et master theorem

Exercice 3.1

On suppose qu'on dispose d'un tableau trié *tab* dans lequel on cherche la position d'une valeur x (avec la convention que la position est -1 si x n'apparaît pas dans *tab*). On propose la fonction

de recherche suivante dans laquelle l'opération $tab[a : b]$ construit un tableau constitué des cases de tab d'indices compris au sens large entre a et b :

```
1:  function search(x, tab)
2:      if tab.length == 0 then
3:          return -1
4:      if tab.length == 1 then
5:          if tab[0] == x then
6:              return 0
7:          else
8:              return -1
9:      else
10:         pos ← tab.length/2
11:         if tab[pos] == x then
12:             return pos
13:         if tab[pos] < x then
14:             subpos ← search(x, tab[(pos + 1) : (tab.length − 1)])
15:             if subpros >= 0 then
16:                 return subpos + pos + 1
17:             else
18:                 return -1
19:         else
20:             subpos ← search(x, tab[0 : (pos − 1)])
21:             if subpos >= 0 then
22:                 return subpos
23:             else
24:                 return -1
```

1. En supposant que l'opération $tab[a : b]$ peut être réalisée en temps constant, calculer la complexité en temps dans le cas le pire de ce programme en fonction de la longueur de tab.
2. En supposant que l'opération $tab[a : b]$ peut être réalisée en temps $\mathcal{O}(b - a + 1)$, calculer la complexité en temps dans le cas le pire de ce programme en fonction de la longueur de tab.

Exercice 3.2

On considère l'algorithme d'exponentiation rapide classique implémenté par la fonction suivante

```
1:  function fastpow(x, n)
2:      if n == 0 then
3:          return 1
4:      if n == 1 then
5:          return x
6:      if n % 2 == 0 then
7:          returnfastpow(x ∗ x, n/2)
8:      else
9:          returnx ∗ fastpow(x ∗ x, n/2)
```

A l'aide du théorème maître, calculer la complexité en temps de cette fonction.

Exercice 3.3

Donner la complexité des récurrences suivantes grace au *master theorem*.

1. $T(n) = 4T(n/2) + n$
2. $T(n) = 4T(n/2) + n^2$