Programmation fonctionnelle - TD5

23 février 2022

1 Croissance de fonction [★]

Soit la fonction f définie par $f(n) = x^2 + 2x + 3$. Prouvez

1. $f \in \mathcal{O}(n^2)$.
2. $f \in \mathcal{O}(n^3)$.
3. $f \notin \mathcal{O}(n)$.

2 Comparaison asymptotique et composition [★★]

Prouvez les énoncés suivants :

1. Soient les fonctions f, f', g et g' telles que $f \in \mathcal{O}(g)$ et $f \in \mathcal{O}(g')$, alors $f + f' \in \mathcal{O}(\max(g, g'))$, et $f \times f' \in \mathcal{O}(g \times g')$.
2. Soient les fonction f, g et h telles que $f \in \mathcal{O}(g)$ et $g \in \mathcal{O}(h)$, alors $f \in \mathcal{O}(h)$.

3 Complexité d'un algorithme récursif [★★]

Une liste d'entiers est dîte *superdécroissante* si chacun de ses éléments est strictement plus grand que la somme de tous ceux qui le suivent. Par exemple la liste $[21, 10, 5, 3, 1]$ est superdécroissante, tandis que la liste $[8, 5, 3, 1]$ ne l'est pas. La fonction `superdecreasing` teste si une liste est superdécroissante :

```
def sum(l: List[Int]): Int = l match
  case Nil => 0
  case n :: xs => n + sum(xs)

def superdecreasing(l: List[Int]) : Boolean = l match
  case Nil => true
  case n :: xs => n > sum(xs) && superdecreasing(xs)
```

1. Donnez une expression récursive de $r_{\texttt{sum}}(n)$, le nombre de réductions nécessaires pour évaluer la fonction `sum` sur une liste de taille n.
2. Donnez une expression explicite (non récursive) de $r_{\texttt{sum}}(n)$.
3. Faite de même pour la fonction `superdecreasing`, et vérifiez qu'elle a une complexité temporelle quadratique en la longueur de la liste .
4. Écrivez une fonction équivalente à `superdecreasing` de complexité $\mathcal{O}(n)$ (indication : vous pouvez utilisez la fonction `reverse`, qui a une complexité $\mathcal{O}(n)$)

4 Exponentiation rapide [★]

L'algorithme suivant utilise le principe *diviser pour régner* afin de calculer x^y (on suppose que x et y sont suffisamment petits pour éviter des dépassement de capacité) :

```
def pow(x: Int, y: Int): Int =
  if y == 0 then 1 else
    val p = pow(x, y / 2)
    if y % 2 == 0 then p * p else p * p * x
```

1. Donnez une expression récursive de $r_{\texttt{pow}}(n)$, le nombre de réductions nécessaires pour évaluer `pow(x,n)`.
2. Déterminez la complexité temporelle de `pow(x,n)` en fonction de la valeur de n, et de la taille de l'entrée.

5 Suite de Padovan [★★]

Le programme suivant calcule le $n$ième nombre de la suite de Padovan :

```
def padovan(n: Int) : Int =
  if n < 0 then throw new IllegalArgumentException
  else if n < 3 then 1 else padovan(n-2) + padovan(n-3)
```

1. Le master theorem permet-il de déterminer la complexité de ce programme ? Expliquez.
2. Déterminez la complexité temporelle de cette fonction (indication : dessinez l'arbre des appels récursifs et déterminez sa taille en fonction de n).
3. Donnez une fonction qui calcule le même résultat en $\mathcal{O}(n)$.

6 Propriétés de la croissance asymptotique [★★★]

Prouvez les énoncés suivants :

1. Pour toutes valeurs $1 < a$ et $1 < b$, $log_a(n) \in \mathcal{O}(\log_b(n))$ (notez que cette propriété est vraie même si $a < b$).
2. Pour toutes valeurs a et b telles que $1 < a < b$, $x^a \in \mathcal{O}(x^b)$ et $x^b \notin \mathcal{O}(x^a)$.
3. Pour toutes valeurs a et b telles que $1 < a < b$, $a^x \in \mathcal{O}(b^x)$ et $b^x \notin \mathcal{O}(a^x)$.
4. Pour toutes valeurs $0 < a$ et $1 < b$, $x^a \in \mathcal{O}(b^x)$ et $b^x \notin \mathcal{O}(x^a)$.