Complexité

Fabrice Rossi

1er mars 2012

Dans ces exercices, sauf mention contraire, on réalisera toutes les analyses dans le cas le pire et on donnera toujours les complexités demandées sous la forme d'un équivalent simple en Θ.

1 Programmes itératifs

Exercice 1.1
Donner la complexité en temps du programme suivant en fonction du paramètre `n` :

```
i = 0
j = 0
while(i < n) {
  if( i % 2 == 0) {
    j = j + 1
  } else {
    j = j / 2
  }
  i = i + 1
}
```

Exercice 1.2
On rappelle que si la variable `tab` contient un tableau, alors `tab.length` donne la longueur du tableau et `tab[i]` permet d'accéder à sa case numéro `i` (la numérotation commence à 0). Donner la complexité en temps du programme suivant en fonction de la longueur du paramètre `tab` :

```
x = 0
for(int i = 1 ; i < tab.length-1 ; i++) {
  for(int j = 0 ; j < 3 ; j++) {
     x = x + tab[i - 1 + j] * (j+1)
  }
}
```

Exercice 1.3
Soit le programme suivant, dépendant des tableaux `tab` et `filter` :

```
 x = 0
for(int i = 0 ; i <= tab.length - filter.length ; i++) {
  for(int j = 0 ; j < filter.length ; j++) {
     x = x + tab[i + j] * filter[j]
  }
}
```

Calculer la complexité en temps du programme en fonction des longueurs des deux tableaux.

Exercice 1.4
On cherche à déterminer le minimum et le maximum d'un tableau. On propose d'abord la solution suivante :

v1
```
min = tab[0]
max = tab[0]
for(int i = 1; i < tab.length; i++) {
  if(tab[i] < min) {
    min = tab[i]
  } else if(tab[i] > max) {
    max = tab[i]
  }
}
```

1. Calculer le nombre exact de comparaisons effectuées dans le cas le pire, en fonction de la longueur du tableau. On ne comptera que les comparaisons entre les éléments du tableau et le contenu des variables `min` et `max`.
2. Pour toute longueur n, donner un exemple de tableau de longueur n correspondant au cas le pire.

On propose la variante suivante :

v2
```
if(tab.length % 2 == 1)
  min = tab[0]
  max = tab[0]
  start = 1
} else {
  if(tab[0] < tab[1]) {
    min = tab[0]
    max = tab[1]
  } else {
    min = tab[1]
    max = tab[0]
  }
  start = 2
}
for(int i = start; i < tab.length; i += 2) {
  if(tab[i] < tab[i+1]) {
    if(tab[i] < min) {
      min = tab[i]
    }
    if(tab[i+1] > max) {
      max = tab[i+1]
    }
  } else {
    if(tab[i+1] < min) {
      min = tab[i+1]
    }
    if(tab[i] > max) {
      max = tab[i]
    }
  }
}
```

Calculer le nombre exact de comparaisons effectuées en fonction de la longueur du tableau. On ne comptera que les comparaisons entre les éléments du tableau entre eux et avec le contenu des variables `min` et `max`.

2 Appels de sous-programmes

Exercice 2.1
On suppose que l'exécution de la fonction `f(n)` prend un temps en $\Theta(n)$. En déduire la complexité du programme suivant en fonction de `n` :

```
for(int i = 0 ; i < n ; i++) {
   f(n-i)
}
```

Exercice 2.2
On suppose que l'exécution de la fonction `f(n)` prend un temps en $\Theta(n)$. En déduire la complexité du programme suivant en fonction de `n` :

```
for(int i = 1 ; i <= n ; i = i * 2) {
   f(i)
}
```

Exercice 2.3
On considère la fonction suivante :

pow
```
pow(x,n) {
  if(n == 0) {
    return 1
  }
  y = x
  for(int i = 2 ; i <= n ; i++) {
    y = y * x
  }
  return x
}
```

Calculer le nombre exact de multiplications réalisées par un appel à `pow(x,n)` en fonction de `n`, un entier positif ou nul. On utilise la fonction dans le programme suivant :

v1
```
val = 0
for(int i = 0; i < tab.length; i++) {
  val = val + tab[i] * pow(x,i)
}
```

Déduire de l'analyse précédente le nombre exact de multiplications réalisées par le programme en fonction de la longueur du tableau `tab`. On propose la variante suivante :

v2
```
val = 0
y = 1
for(int i = 0; i < tab.length; i++) {
  val = val + tab[i] * y
  y = y * x
}
```

Calculer le nombre exact de multiplications réalisées par le programme en fonction de la longueur du tableau `tab`.

On propose enfin le programme suivant :

v3

```
val = 0
for(int i = tab.length-1; i >= 0 ; i--) {
  val = tab[i] + x * val;
}
```

Calculer le nombre exact de multiplications réalisées par le programme en fonction de la longueur du tableau `tab`. Commenter les résultats.

3 Programmes récursifs

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` :

search

```
search(x,tab) {
  if(tab.length == 0) {
    return -1
  } else if(tab.length == 1) {
     if(tab[0] == x) {
       return 0
     } else {
       return -1
     }
  } else {
    pos = tab.length/2
    if(tab[pos] == x) {
      return pos
    } else if(tab[pos] < x) {
      subpos = search(x,tab[(pos + 1):(tab.length - 1)])
      if(subpos >= 0) {
        return subpos + pos + 1
      } else {
        return -1
      }
   } else {
      subpos = search(x,tab[0:(pos - 1)])
      if(subpos >= 0) {
        return subpos
      } else {
        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 $\Theta(b-a+1)$, 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

`fastpow`

```
fastpow(x,n) {
  if(n == 0) {
    return 1
  } else if(n == 1) {
    return x
  } else if (n % 2 == 0) {
    return fastpow(x * x, n / 2)
  } else {
    return x * fastpow(x * x, n / 2)
  }
}
```

En utilisant le théorème maître, calculer la complexité en temps de cette fonction.