TD d'algorithmique avancée

Corrigé du TD 2 : récursivité

Jean-Michel Dischler et Frédéric Vivien

Suite de Fibonacci

La suite de Fibonacci est définie comme suit :

$$\text{Fib}(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{sinon.} \end{cases}$$

1. Écrivez un algorithme récursif calculant Fib(n).

   FIBONACCI(n)
   si $n = 0$ ou $n = 1$ **alors renvoyer** 1
   sinon renvoyer FIBONACCI($n-1$) + FIBONACCI($n-2$)

2. Montrez que la complexité (en nombre d'additions) de cet algorithme est en $\Omega(2^{\frac{n}{2}})$.
   On procède par récurrence. On veut montrer qu'il existe une constante c strictement positive telle que $T(n) \geq c.2^{\frac{n}{2}}$, pour des valeurs de n supérieures à une certaine borne n_0 (à déterminer). Supposons le résultat démontré jusqu'au rang $n-1$. Alors :

   $$T(n) = T(n-1) + T(n-2) + 1 \geq c.2^{\frac{n-1}{2}} + c2^{\frac{n-2}{2}} + 1 \geq c.2^{\frac{n-2}{2}} + c.2^{\frac{n-2}{2}} + 1 \geq 2 \times c.2^{\frac{n-2}{2}} = c.2^{\frac{n}{2}}$$

   Il nous reste juste à montrer que cette équation est vraie « au départ ». Nous ne pouvons bien évidemment pas partir des cas $n = 0$ et $n = 1$, puisque pour ces valeurs $T(n) = 0$. Nous partons donc des cas $n = 2$ et $n = 3$ (la récurrence nécessite deux *valeurs de départ) :*
   - *Cas $n = 2$:* FIBONACCI*(2) =* FIBONACCI*(1) +* FIBONACCI*(0), et $T(2) = 1$. Pour que la propriété désirée soit vraie, c doit donc vérifier :*

     $$1 \geq c.2^{\frac{2}{2}} = 2c \quad \Leftrightarrow \quad c \leq \frac{1}{2}$$

   - *Cas $n = 3$:* FIBONACCI*(3) =* FIBONACCI*(2) +* FIBONACCI*(1), et $T(3) = 2$. Pour que la propriété désirée soit vraie, c doit donc vérifier :*

     $$2 \geq c.2^{\frac{3}{2}} = 2\sqrt{2}c \quad \Leftrightarrow c \leq \frac{\sqrt{2}}{2}$$

   Donc si $c = \frac{1}{2}$, pour $n \geq 2$, on a $T(n) \geq c2^{\frac{n}{2}}$ et donc $T(n) = \Omega(2^{\frac{n}{2}})$.

3. Écrire un algorithme récursif qui calcule, pour $n > 0$, le couple (FIBONACCI(n), FIBONACCI($n-1$)).

   FIB-PAIRE(n)
   si $n = 1$ **alors renvoyer** (1, 1)
   sinon $(x, y) =$ FIB-PAIRE($n-1$)
   renvoyer $(x + y, x)$

4. Utilisez l'algorithme précédent pour écrire un nouvel algorithme calculant FIBONACCI(n).

FIBONACCI(n)
  si $n = 0$ **alors renvoyer** 1
    sinon (x, y) = FIB-PAIRE(n)
      renvoyer x

5. Qu'elle est la complexité (en nombre d'additions) de cet algorithme ?
   La complexité de l'algorithme FIB-PAIRE*, en nombre d'additions, est donnée par la récurrence* $T(n) = 1 + T(n-1)$*. On a donc* $T(n) = n - 1$ *pour* FIB-PAIRE*, et par extension pour la nouvelle version de* FIBONACCI*.*

Opérations ensemblistes

Dans cette partie on considère des ensembles représentés par des tableaux, certains ensembles seront triés et d'autres pas. **Toutes les solutions proposées doivent être récursives.**

1. Nous voulons un algorithme APPARTENANCE(A, x) qui recherche si un élément x appartient à l'ensemble A. Si x appartient effectivement à A, l'algorithme renverra VRAI, et FAUX sinon.
   (a) Cas des ensembles non triés :
      i. Écrivez un tel algorithme.

RECHERCHE(A, *rang*, x)
  si *rang* > *longueur*(A) **alors renvoyer** FAUX
  si A[*rang*] = x **alors renvoyer** VRAI
    sinon renvoyer RECHERCHE(A, *rang* + 1, x)

      L'appel initial de l'algorithme est alors RECHERCHE(A, 1, x).

      ii. Quelle est sa complexité en nombre de comparaisons ?
      Dans le pire cas, l'élément n'appartient pas à l'ensemble et tout le tableau est parcouru. La complexité au pire est donc en $\Theta(n)$*, où* n *est la longueur du tableau (et donc la taille de l'ensemble).*

   (b) Cas des ensembles triés (dans l'ordre croissant) :
      i. Écrivez un tel algorithme.

RECHERCHE(A, *rang*, x)
  si *rang* > *longueur*(A) ou A[*rang*] > x
    alors renvoyer FAUX
    sinon si A[*rang*] = x
      alors renvoyer VRAI
      sinon renvoyer RECHERCHE(A, *rang* + 1, x)

      L'appel initial de l'algorithme est alors RECHERCHE(A, 1, x).

      ii. Quelle est sa complexité en nombre de comparaisons ?
      Le pire cas est aussi en $\Theta(n)$ *: il intervient quand l'élément recherché n'appartient pas à l'ensemble mais est plus grand que tous les éléments de l'ensemble.*

      iii. Utilisez une recherche dichotomique pour améliorer votre algorithme.

RECHERCHE(A, x, *inf*, *sup*)
  milieu $\leftarrow \left\lfloor \frac{inf+sup}{2} \right\rfloor$
  si A[*milieu*] = x
    alors renvoyer VRAI
    sinon si A[*milieu*] > x **alors renvoyer** RECHERCHE(A, x, *inf*, *milieu* - 1)
      sinon renvoyer RECHERCHE(A, x, *milieu* + 1, *sup*)

iv. Quelle est la complexité de votre nouvel algorithme ?

Posons $n = sup - inf + 1$ le nombre d'éléments dans la partie du tableau à étudier. Considérons la taille du tableau lors de l'éventuel appel récursif. Nous avons deux cas à considérer :

- *L'appel effectué est :* RECHERCHE*(A, x, inf, milieu - 1). Le nombre d'éléments concernés est alors : $milieu - 1 - inf + 1 = \left\lfloor \frac{sup-inf}{2} \right\rfloor = \lfloor \frac{n-1}{2} \rfloor \leq \frac{n}{2}$.*
- *L'appel effectué est :* RECHERCHE*(A, x, milieu + 1, sup). Le nombre d'éléments concernés est alors : $sup - (milieu + 1) + 1 = \left\lceil \frac{sup-inf}{2} \right\rceil = \lceil \frac{n-1}{2} \rceil \leq \frac{n}{2}$.*

On passe donc d'un ensemble de taille n à un ensemble de taille au plus $\frac{n}{2}$. D'où $T(n) \leq 2 \times T(\frac{n}{2})$ (la fonction $T(n)$ étant croissante, on peut se permettre l'approximation). Par conséquent : $T(n) \leq 2 \times \log_2(n) T(\frac{n}{2^{\log_2 n}})$ et $T(n) = O(\log_2 n)$.

2. Nous voulons maintenant un algorithme UNION(A, B) qui nous renvoie l'union des deux ensembles qui lui sont passés en argument.

   (a) Cas des ensembles non triés :

   i. Écrivez un tel algorithme.

```
UNION(A, B, rang, C)
  si RECHERCHE(A, B[rang]) = FAUX
    alors longueur(C) ← longueur(C) + 1
          C[longueur(C)] ← B[rang]
  UNION(A, B, rang + 1, C)
```

   L'appel initial est UNION(A, B, 1, C) où C est un tableau de taille $longueur(A) + longueur(B)$, et dont les $longueur(A)$ premières cases contiennent les éléments de A.

   ii. Quelle est sa complexité ?

   La recopie de A dans C est de coût longueur(A).

   L'algorithme UNION *est appelé $longueur(B)$ fois, chacun de ces appels effectuant un appel à* RECHERCHE *sur A, dont le coût au pire est en longueur(A). La complexité au pire de* UNION *est donc en $\Theta(longueur(A) \times longueur(B))$ ou $\Theta(nm)$, n et m dénotant la taille des deux tableaux. Ce pire cas apparaît quand les tableaux A et B sont disjoints.*

   (b) Cas des ensembles triés (dans l'ordre croissant) :

   i. Écrivez un tel algorithme.

```
UNION(A, a, B, b, C)
  si a > longueur(A) et b > longueur(B) alors renvoyer C
  si b > longueur(B) ou B[b] > A[a]
    alors longueur(C) ← longueur(C) + 1
          C[longueur(C)] ← A[a]
          UNION(A, a + 1, B, b, C)
    sinon longueur(C) ← longueur(C) + 1
          C[longueur(C)] ← B[b]
          si a ≤ longueur(A) et A[a] = B[b] alors UNION(A, a + 1, B, b + 1, C)
                                            sinon UNION(A, a, B, b + 1, C)
```

   L'appel initial est UNION(A, 1, B, 1, C).

   ii. Quelle est sa complexité ?

   La complexité de cet algorithme est au pire en $\Theta(longueur(A) + longueur(B))$ ou $\Theta(n + m)$: à chaque appel on décrémente au moins de un l'ensemble des valeurs à considérer.

3. Nous voulons maintenant un algorithme INTERSECTION(A, B) qui nous renvoie l'intersection des deux ensembles qui lui sont passés en argument.

   (a) Cas des ensembles non triés :

   i. Écrivez un tel algorithme.

INTERSECTION(A, B, *rang*, C)
  si RECHERCHE(A, B[*rang*]) = VRAI
    alors *longueur*(C) ← *longueur*(C) + 1
      C[*longueur*(C)] ← B[*rang*]
  INTERSECTION(A, B, *rang* + 1, C)

L'appel initial est alors INTERSECTION*(A, B, 1, C) où C est un nouveau tableau, de taille* min(*longueur*(A), *longueur*(B))*, et ne contenant initialement aucun élément (*$longueur(C) = 0$*).*

ii. Quelle est sa complexité ?

L'algorithme INTERSECTION *est appelé longueur*(B) *fois, chacun de ces appels effectuant un appel à* RECHERCHE *sur A, dont le coût au pire est en longueur(A). La complexité au pire de* INTERSECTION *est donc en* $\Theta(longueur(A) \times longueur(B))$ *ou* $\Theta(nm)$*, n et m dénotant la taille des deux tableaux. Ce pire cas apparaît quand les tableaux A et B sont disjoints.*

(b) Cas des ensembles triés (dans l'ordre croissant) :

i. Écrivez un tel algorithme.

INTERSECTION(A, a, B, b, C)
  si $a > longueur(A)$ ou $b > longueur(B)$ **alors renvoyer** C
  si $A[a] = B[b]$ **alors** *longueur*(C) ← *longueur*(C) + 1
      C[*longueur*(C)] ← $A[a]$
      renvoyer INTERSECTION(A, $a + 1$, B, $b + 1$, C)
    sinon si $B[b] > A[a]$ **alors renvoyer** INTERSECTION(A, $a + 1$, B, b, C)
      sinon renvoyer INTERSECTION(A, a, B, $b + 1$, C)

L'appel initial est INTERSECTION(A, 1, B, 1, C, 0).

ii. Quelle est sa complexité ?

La complexité de cet algorithme est au pire en $\Theta(longueur(A) + longueur(B))$ *ou* $\Theta(n + m)$ *: à chaque appel on décrémente au moins de un l'ensemble des valeurs à considérer.*

4. Nous voulons maintenant un algorithme DIFFÉRENCE(A, B) qui nous renvoie la différence des deux ensembles qui lui sont passés en argument (La différence de A et de B, notée $A \setminus B$ est l'ensemble des éléments de A n'appartenant pas à B).

(a) Cas des ensembles non triés :

i. Écrivez un tel algorithme.

DIFFÉRENCE(A, *rang*, B, C)
  si RECHERCHE(B, A[*rang*]) = FAUX
    alors *longueur*(C) ← *longueur*(C) + 1
      C[*longueur*(C)] ← A[*rang*]
  DIFFÉRENCE(A, *rang* + 1, B, C)

L'appel initial est alors DIFFÉRENCE(A, 1, B, C) où C est un tableau de taille *longueur*(A), ne contenant initialement aucun élément (*longueur*(C) = 0).

ii. Quelle est sa complexité ?

L'algorithme DIFFÉRENCE *est appelé longueur*(A) *fois, chacun de ces appels effectuant un appel à* RECHERCHE *sur B, dont le coût au pire est en longueur(B). La complexité au pire de* DIFFÉRENCE *est donc en* $\Theta(longueur(A) \times longueur(B))$ *ou* $\Theta(nm)$*, n et m dénotant la taille des deux tableaux. Ce pire cas apparaît quand les tableaux A et B sont disjoints.*

(b) Cas des ensembles triés (dans l'ordre croissant) :

i. Écrivez un tel algorithme.

```
DIFFÉRENCE(A, a, B, b, C)
  si a > longueur(A) alors renvoyer C
  si A[a] = B[b] alors renvoyer DIFFÉRENCE(A, a + 1, B, b + 1, C)
                 sinon si A[a] < B[b] alors longueur(C) ← longueur(C) + 1
                                           C[longueur(C)] ← A[a]
                                           renvoyer DIFFÉRENCE(A, a + 1, B, b, C)
                                       sinon renvoyer DIFFÉRENCE(A, a, B, b + 1, C)
```

L'appel initial est DIFFÉRENCE(A, 1, B, 1, C, 0).

ii. Quelle est sa complexité ?

La complexité de cet algorithme est au pire en $\Theta(longueur(A) + longueur(B))$ ou $\Theta(n+m)$: à chaque appel on décrémente au moins de un l'ensemble des valeurs à considérer.