SOLUTIONS SÉRIE 4

Question # 1

A) Écrivez un algorithme diviser pour régner qui trouve le plus grand et le plus petit élément d'un tableau de n nombres.

Solution :

Algorithm *MinMax*($A[l...r]$) :
  IF $r = l$
    $minval \leftarrow A[l]$
    $maxval \leftarrow A[l]$
  ELSE
    $(minval, maxval) \leftarrow$ *MinMax*($A[l...\lfloor (l+r)/2 \rfloor]$)
    $(minval2, maxval2) \leftarrow$ *MinMax*($A[\lfloor (l+r)/2 \rfloor + 1...r]$)
    IF $minval2 < minval$
      $minval \leftarrow minval2$
    IF $maxval2 > maxval$
      $maxval \leftarrow maxval2$
  RETURN $(minval, maxval)$

B) Trouvez la récurrence qui décrit le nombre de comparaisons entre deux éléments du tableau fait par votre algorithme (on suppose ici que $n = 2^k$). Donnez l'ordre de croissance de cette récurrence.

Solution :

Nous comptons le nombre de comparaisons entre des éléments du tableaux. Dans notre algorithme, il s'agit des lignes

— IF $minval2 < minval$ et

— IF $maxval2 > maxval$.

Notez que $minval$, $maxval$, $minval2$ et $maxval2$ sont bien des éléments du tableau même s'il s'agit de valeurs de retour. Nous exprimerons par $C(n)$ le nombre de comparaisons entre des éléments du tableau fait par l'algorithme dans tous les cas :

$$C(n) = 2C(n/2) + 2 \text{ pour } n > 1, C(1) = 0$$

En utilisant le théorème général, puisque $r = b = 2$ et $d = 0$, nous obtenons que $C(n) \in \Theta(n^{\log_b r}) = \Theta(n^{\log_2 2}) = \Theta(n)$. Remarquez que tout autre choix valide d'opération de base mène au même résultat de classe de complexité.

C) *Comparez votre algorithme avec l'algorithme de force brute qui résout le même problème.*
Solution :
Les deux algorithmes sont dans le même ordre exact en ce qui concerne le nombre de comparaisons.

Question # 2

A) *Écrivez un algorithme diviser pour régner qui calcul la valeur de* a^n *où* $a > 0$ *et* $n \geq 0$.
Solution :

Algorithm *DivConqPower*(a, n) :
  IF $n = 0$
    RETURN 1
  IF $n = 1$
    RETURN a
  RETURN *DivConqPower*$(a, \lceil n/2 \rceil) \times$ *DivConqPower*$(a, \lfloor n/2 \rfloor)$

B) *Trouvez et résolvez la récurrence qui décrit le nombre de multiplications fait par votre algorithme (on suppose ici que* $n = 2^k$*).*
Solution :
Nous comptons le nombre de multiplications entre des éléments du tableaux. Il n'y en a q'une seule à la ligne 5. Nous exprimerons par $C(n)$ *le nombre de mulitiplications fait par l'algorithme dans tous les cas :*

$$C(n) = 2C(n/2) + 1 \text{ pour } n > 1, C(1) = 0$$

En utilisant le théorème général, puisque $r = b = 2$ *et* $d = 0$*, nous obtenons que* $C(n) \in \Theta(n^{\log_b r}) = \Theta(n^{\log_2 2}) = \Theta(n)$.
Remarquez que tout autre choix valide d'opération de base mène au même résultat de classe de complexité.

C) *Comparez votre algorithme avec l'algorithme de force brute qui résout le même problème.*
Solution :
Les deux algorithmes sont dans le même ordre exact en ce qui concerne le nombre de multiplications.

Question # 3

Des joueurs de Chess-Boxing se réunissent pour un tournoi. Chaque partie est constituée d'exactement deux joueurs et d'un unique gagnant. Le but est de faire le moins de parties possibles pour déterminer un unique grand gagnant.

A) Écrivez un algorithme diviser pour régner qui détermine le grand gagnant du tournoi de Chess-Boxing. Vous pouvez faire appel à l'algorithme ChessBoxing(joueur1, joueur2) qui retourne le gagnant parmi une paire de joueurs en $\Theta(1)$.

Solution :

Algorithme 1 : Tournoi($J[1..n]$)

```
1 // Déterminer le grand gagnant d'un tournoi de Chess-Boxing.
2 // Entrées : Un ensemble J de n joueurs.
3 // Sortie : Un unique grand gagnant.
4 si |J| = 1 alors
5     retourner J[1]
6 gagnant1 ← Tournoi(J[1, ..., ⌊n/2⌋])
7 gagnant2 ← Tournoi(J[⌊n/2⌋ + 1, ..., n])
8 retourner ChessBoxing(gagnant1, gagnant2)
```

B) Analysez l'efficacité de votre algorithme en A).

Solution :

Prenons comme opération de base l'appel à l'algorithme ChessBoxing(joueur1, joueur2). Alors, l'efficacité de l'algorithme en A) est donnée par la récurrence qui suit.

$$C(n) = \begin{cases} 0 & \text{si } n = 1 \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + 1 & \text{si } n > 1 \end{cases}$$

En supposant que $n = 2^k$ et en utilisant le théorème général, puisque $r = b = 2$ et $d = 0$, nous obtenons que $C(n) \in \Theta(n^{\log_b(r)}) = \Theta(n^{\log_2(2)}) = \Theta(n)$, $\forall n \in \mathbb{N}$.

C) Allez voir le pseudo-code de la solution en A). Quelle est l'efficacité de cet algorithme si les tableaux en paramètres sont passés par copie. Quelle est l'efficacité s'ils sont passés par référence ?

Solution :

L'analyse faite en B) considère que les tableaux sont passés par référence. Si toutefois les tableaux sont passés en copie, alors l'opération de base est plutôt la copie d'un élément du vecteur. Dans ce cas, on obtient plutôt la récurrence qui suit.

$$C(n) = \begin{cases} 0 & \text{si } n = 1 \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n & \text{si } n > 1 \end{cases}$$

En supposant que $n = 2^k$ et en utilisant le théorème général, puisque $r = b = 2$ et $d = 1$, nous obtenons que $C(n) \in \Theta(n^d \log(n)) = \Theta(n \log(n))$, $\forall n \in \mathbb{N}$.

Question # 4

Lorsque nous avons parlé de la conception et de l'analyse des algorithmes au chapitre 2, nous avons mentionné que la base d'un logarithme n'est pas importante dans la majorité des contextes qui découlent de l'analyse d'algorithmes. Est-ce vrai pour les deux parties du théorème général qui inclus des logarithmes ?

Solution :

Pour le second cas où la solution est donnée par $\Theta(n^d \log n)$, la base du logarithme ne fait que déterminer une constante qui est inutile ($n^d c_1 \log n$ vs $n^d c_2 \log n$). Par contre, dans le troisième cas où la solution est donnée par $\Theta(n^{\log_b a})$, la valeur de la base du logarithme influence la classe de complexité ($\Theta(n^\alpha)$ dépend de la valeur de α).

Question # 5

Trouvez l'ordre de croissance de la fonction $T(n)$ qui est exprimée selon la récurrence (utilisez le théorème général pour résoudre la récurrence) :

A) $T(n) = T(n/2) + c, T(1) = 0$

Solution :

Puisque $r = 1, b = 2$ *et* $d = 0$, *nous avons que* $r = b^d$ *et donc* $T(n) \in \Theta(\lg n)$.

B) $T(n) = 4T(n/2) + n, T(1) = 1$

Solution :

Puisque $r = 4, b = 2$ *et* $d = 1$, *nous avons que* $r > b^d$ *et donc* $T(n) \in \Theta(n^2)$.

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

Solution :

Puisque $r = 4, b = 2$ *et* $d = 2$, *nous avons que* $r = b^d$ *et donc* $T(n) \in \Theta(n^2 \log n)$.

D) $T(n) = 4T(n/2) + n^3, T(1) = 1$

Solution :

Puisque $r = 4, b = 2$ *et* $d = 3$, *nous avons que* $r < b^d$ *et donc* $T(n) \in \Theta(n^3)$.

Question # 6

Soit l'algorithme Quicksort($A[l \ldots r]$) présenté à la figure 1 (voir p. 128 de A. Levitin) :

```
ALGORITHME QuickSort(A[l..r])
//Entrée: le sous tableau A[l..r] de A[0..n-1]
//Sortie: le sous tableau A[l..r]  trié
if l < r
   s ← Partition(A[l..r])
   QuickSort(A[l..s-1])
   QuickSort(A[s+1..r])
```

FIGURE 1 – Tri rapide

A) Est-ce que les tableaux formés d'éléments tous égaux doivent être considérés comme des instances : du pire cas, du meilleur cas, ou aucun des deux ?

Solution :
Ce sont des instances du pire cas, car pour ces instances, le pivot choisit sera toujours le premier élément du tableau. On aura donc des sous-tableaux de taille 0 et $n-1$.

B) *Est-ce que les tableaux triés en ordre décroissant doivent être considérés comme des instances : du pire cas, du meilleur cas, ou aucun des deux ?*
Solution :
Ce sont des instances du pire cas, car pour ces instances, le pivot choisit sera toujours l'élément maximal du tableau. On aura donc un sous-tableau vide, et l'autre contenant $n-1$ éléments.

Question # 7

A) *En considérant Quicksort avec la technique de sélection du pivot "median-of-three" (qui consiste à choisir comme pivot l'élément médian parmi l'élément le plus à gauche, celui le plus à droite et celui au centre), est-ce que les tableaux triés en ordre croissant doivent être considérés comme des instances : du pire cas, du meilleur cas, ou aucun des deux ?*
Solution :
Ce sont des instances du meilleur cas, car pour ces instances, le pivot choisit sera toujours l'élément du milieu du tableau, qui se trouve à être la médianne du tableau. On aura donc des sous-tableau de tailles semblables.

B) *Répondez à la même question qu'en A mais pour des tableaux triés en ordre décroissant.*
Solution :
Même réponse qu'en A).

***Question # 8**

On vous donne une collection de n boulons de tailles différentes ainsi que les n écrous correspondants. Il est permis d'essayer un boulon avec un écrou dans le but de savoir si l'écrou est trop large pour le boulon, pas assez large pour le boulon ou exactement de la bonne taille pour le boulon. Cependant, il n'est pas possible de comparer deux boulons ensemble ni deux écrous ensemble. Le problème est de trouver, pour chaque boulon, l'écrou qui lui convient. Décrivez votre algorithme en français, puis en pseudo-code. Votre algorithme doit avoir une efficacité moyenne de $\Theta(n \log n)$.
Solution :

En français.

1. *S'il n'y a ni boulon ni écrou alors on ne fait rien. S'il y a seulement un boulon et un écrou alors b_1 correspond nécessairement à b_2. S'il y a seulement deux boulons b_1 et b_2 et deux écrous e_1 et e_2, on teste est-ce que e_1 va avec e_1 ou e_2. Ensuite, on déduit à quel boulon correspond l'écrou e_2.*

2. *On sélectionne un écrou e_1. Pour chaque boulon b, on teste avec l'écrou e s'il est trop petit, trop grand ou s'il est de la bonne taille. De cette façon, on construit deux ensembles de boulons $B_<$, l'ensemble des boulons qui sont plus petit que e_1 et $B_>$, l'ensemble des boulons qui sont plus grand que e_1. On retient aussi $b_=$, le boulon qui correspond à e_1.*

3. *Maintenant, pour chaque écrou, sauf e_1, on teste avec le boulon $b_=$ s'il est trop petit ou trop grand (aucun ne sera de la bonne taille puisque les écrous sont tous différents et e_1 n'est pas utilisé ici). De cette façon, on construit deux ensembles d'écrous $E_<$, l'ensemble des écrous qui sont plus*

petit que B et $E_>$, l'ensemble des écrous qui sont plus grand que B.

4. *On sait maintenant que e_1 et $b_=$ vont ensemble. On applique les étapes (1) à (4) sur les ensembles $E_<, B_<$ et $E_>, B_>$.*

En pseudo-code. *L'algorithme 3 présente le pseudo-code de l'algorithme. Les opérateurs $=$, $>$ et $<$ permettent, dans cet exemple, la comparaison entre un boulon et un écrou. Nous utiliserons une notation ensembliste pour représenter B et E.*

Notes supplémentaires sur l'algorithme. *L'algorithme 3 a trois cas de base. Ces cas de bases ne sont pas requis :*

- *Si $B = E = \emptyset$ alors les deux boucles ne seront pas exécutées et $P = \emptyset$ sera retourné.*
- *Si $|B| = |E| = 1$ alors le premier boulon b_1 sera nécessairement égal à e_1 lors de l'exécution de la première boucle. La seconde boucle n'est pas exécutée dans ce cas car $B \setminus \{e_1\} = \emptyset$. Les appels récursifs après la seconde boucle seront tous deux sur BoulonsEcrous$(\emptyset, \emptyset)$ et nous nous retrouvons dans le premier cas.*
- *S $|B| = |E| = 2$ alors il ne s'agit pas d'un cas particulier et il suffit d'exécuter les deux boucles.*

Toutefois, ces trois cas de basent ajoutent à la clarté de notre algorithme.

Notes sur l'analyse. *L'analyse en moyenne de cet algorithme est très semblable à celle de Quicksort. La seule différence est dans la partition. L'algorithme de Quicksort partitionne en visitant chaque élément du tableau une fois, alors qu'ici on a besoin de partitioner deux tableaux à chaque fois. Il faut donc visiter chaque élément des 2 tableaux une fois, ce qui requiert 2 fois plus de travail que dans le cas de Quicksort. Il est facile de voir que la constante multiplicative ne change pas de résultat final.*

Algorithme 3 : L'algorithme BoulonsEcrous

```
// Trouver, pour chaque boulon, l'écrou qui lui convient.
// Entrées : Un ensemble de B de n boulons différents, un ensemble E de n écrous
   correspondants, mais désordonné.
// Sortie : Un ensemble P de paires d'écrous et de boulons.
si |B| = 0 alors
    retourner ∅
si |B| = 1 alors
    retourner {(b_1, e_1)}
si |B| = 2 alors
    si b_1 = e_1 alors
        retourner {(b_1, e_1), (b_2, e_2)}
    retourner {(b_1, e_2), (b_2, e_1)}
P = ∅
B_< ← ∅
B_> ← ∅
pour b ∈ B faire
    si b > e_1 alors
        B_> ← {b} ∪ B_>
    sinon
        si b < e_1 alors
            B_< ← {b} ∪ B_<
        sinon
            b_= ← b
            P ← {(b, e_1)}
E_< ← ∅
E_> ← ∅
// Notez que E \ {e_1} correspond à {e_2, ..., e_n}
pour e ∈ E \ {e_1} faire
    si e > b_= alors
        E_> ← {e} ∪ E_>
    sinon
        E_< ← {e} ∪ E_<
P ← P ∪ BoulonsEcrous(B_<, E_<) ∪ BoulonsEcrous(B_>, E_>)
retourner P
```

Question # 9

Écrivez un algorithme qui est une version récursive de la recherche binaire présentée à la figure 2 (voir l'algorithme BinarySearch($A[0 \ldots n-1], K$) à la p. 134 de A. Levitin).

```
ALGORITHM BinarySearch(A[0..n-1], K)
  l ← 0; r ← n-1
  while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] then return m
    else if K < A[m] then r ← m – 1
    else l ← m + 1
  return -1
```

FIGURE 2 – Recherche binaire

Solution :

Algorithm *BSR*($A[l \ldots r], K$) :
  IF $l > r$
    RETURN -1
  ELSE
    $m \leftarrow \lfloor (l + r)/2 \rfloor$
    IF $K = A[m]$
      RETURN m
    ELSE IF $K < A[m]$
      RETURN *BSR*($A[l \ldots m-1], K$)
    ELSE
      RETURN *BSR*($A[m+1 \ldots r], K$)