Algorithmique et programmation procédurale - TD No 4

Tableaux - CORRIGE

Exercice 1

Écrire une fonction qui retourne toutes les valeurs >= le premier élément dans un tableau.

Exemple : plusGrandQuePremier ({7,4,6,8,3,7,4,9}) = {7,8,7,9}

Corrigé

```
Fonction plusGrandQuePremier (A[n]: Tableau de T) : Tableau de T
Variables i, j : Entier,
          B[ ] : Tableau de T
Début
        j ← 0
        Pour i ← 0 à n-1
                Si A[i] >= A[0] Alors
                        j ← j + 1
                FinSi
        FinPour
        B ← CréerTableau (j, T)
        j ← 0
        Pour i ← 0 à n-1
                Si A[i] >= A[0] Alors
                        B[j] ← i
                        j ← j + 1
                FinSi
        FinPour
        retourner B
Fin
```

Exercice 2

Écrire une fonction qui vérifie si les éléments d'un tableau donné sont triés ou pas (de façon croissante ou décroissante !).

Corrigé

Version 1 :

```
Fonction estTrie (A[n]: Tableau de T) : Booléen
Variables i : Entier
Début
        Si n > 1 Alors
          Si (A[0] <= A[n-1]) Alors
                Pour i ← 0 à n-2
                      Si (A[i] > A[i+1]) Alors
                                Retourner faux
                      FinSi
                FinPour
          Sinon
                Pour i ← 0 à n-2
                      Si (A[i] < A[i+1]) Alors
                                Retourner faux
                      FinSi
                FinPour
          FinSi
        FinSi
        Retourner vrai
Fin
```

Ou bien :

```
Fonction estTrie(A[n]: Tableau de T) : Booléen
Variables i : Entier, trie, ordreCroissant : Booléen
Début
        Si n > 1 Alors
          ordreCroissant ← (A[0] <= A[n-1])
          trie ← vrai
          i ← 1
```

```
        Tantque i < n-2 et trie
            Si (A[i]>A[i+1] et ordreCroissant) ou (A[i]<A[i+1] et non ordreCroissant) Alors
                trie ← faux
            FinSi
            i ← i+1
        FinTantque
        retourner trie
    FinSi
    Retourner vrai
Fin
```

Exercice 3

Écrire une fonction qui reçoit un tableau trié et une valeur, puis « insère » cette valeur (renvoie un nouveau tableau avec une case de plus). Évaluer le nombre d'affectations nécessaires à votre fonction dans le meilleur et le pire des cas, et en déduire sa complexité.

```
Fonction insérer (A[n] : Tableau de T, val : T) : Tableau de T
Variables i, pos : Entier, B[n+1] : Tableau de T
Début
    pos ← 0
    Tantque pos<n et A[pos] < val
        pos ← pos+1
    FinTantque
    Pour i ← 0 à (pos – 1)
        B[i] ← A[i]
    FinPour
    B[pos] ← val
    Si pos<n Alors
        Pour i ← pos+1 à n
            B[i] ← A[i-1]
        FinPour
    FinSi
    Retourner B
Fin
```

Nb d'affectations dans le meilleur des cas : 1 + 0 + (n+1) = n+2

Nb d'affectations dans le pire des cas : 1 + (n+1) + (n+1) = 2n+3 ➔ O(n)

Exercice 4

Écrire un algorithme qui permet de trouver l'intersection de deux tableaux des entiers triés dans l'ordre strictement croissant. Par exemple, l'intersection de {1, 3, 4, 6, 8} et de {4, 5, 6, 12} donne {4, 6}. Si les longueurs respectives de deux tableaux sont n et m, la complexité de votre algorithme doit être O(n+m) et non O(n*m).

```
Fonction intersection (A[n], B[m] : Tableau de T) : Tableau de T
Variables c, posA, posB : Entier, I[ ] : Tableau de T
Début
        // déterminer la taille de I
        posA ← 0 ;  posB ← 0 ;  c ← 0
        Tantque posA<n et posB <m
                Si A[posA] = B[posB] Alors
                        c ← c + 1 ;  posA ← posA + 1 ;  posB ← posB + 1
                SinonSi A[posA] < B[posB]
                        posA ← posA + 1
                Sinon { A[posA] > B[posB]}
                        posB ← posB + 1
                FinSi
        FinTantque
        I ← CréerTableau(c, T)
        // copier les valeurs
        posA ← 0 ;  posB ← 0;  c ← 0
        Tantque posA<n et posB <m
                Si A[posA] = B[posB] Alors
                        I[c] ← A[posA]; c ← c + 1;  posA ← posA + 1;  posB ← posB + 1
                SinonSi A[posA] < B[posB]
                        posA ← posA + 1
                Sinon { A[posA] > B[posB]}
                        posB ← posB + 1
                FinSi
        FinTantque
        Retourner I
Fin
```

Exercice 5

Proposer une version itérative de l'algorithme de recherche dichotomique dans un tableau.

Corrigé

```
Fonction RechDicIte (A[n] : Tableau de T, val : T) : Booléen
Variables min, max, mid : Entier
        min ← 0
        max ← n - 1
        Tantque (min <= max)
                mid ← (min + max) /2
                Si (val = A[mid]) Alors
                        Retourner vrai
                Sinon
                        Si (val < A[mid]) Alors
                                max ← mid-1
                        Sinon
                                min ← mid +1
                        FinSi
                FinSi
        FinTantque
        Retourner faux
Fin
```

Exercice 6. Tri par insertion

Réécrire le tri par insertion vu en cours pour trier dans l'ordre décroissant. Est-il possible d'améliorer le coût de cet algorithme en appliquant une recherche dichotomique pour l'insertion d'une valeur dans le sous tableau déjà trié ?

Corrigé

```
Procédure TriInsertion (ES A[n] : Tableau de T)
Variables i, j : Entier, temp : T
Début
        Pour i ← 1 à n-1
                j ← i
                Tantque j > 0 et (A[j] > A[j-1])
```

```
                    temp ← A[j]
                    A[j] ← A[j-1]
                    A[j-1] ← temp
                    j ← j-1
                FinTantque
        FinPour
Fin
```

Est-il possible d'améliorer le coût de cet algorithme en appliquant une recherche dichotomique pour l'insertion d'une valeur dans le sous tableau déjà trié ?

Dans le pire des cas cela ne change rien pour le coût car, même si la recherche de l'emplacement où il faut insérer est plus rapide (log(n) au lieu de n), l'insertion dans un tableau trié nécessite un décalage 1 qui lui est forcément linéaire (voir exercice 3 !). Avec une structure permettant une insertion en temps constant (exemple listes chaînées), on obtiendrait un gain de performances.