# Solution Serie 2 :

## Exercice 1 :

Tableau de permutation d'ordre N :

- Tableau est de dimension N
- Chaque 1 <T[i] < N, pour 1< i < N
- Unicité des valeurs, pour tout i <> j, T[i] <> T[j].

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 8 | 5 | 7 | 1 | 3 | 2 | 6 | 4 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

1/- Algorithme naïf quadratique (complexité $O(N^2)$)

Algorithme permutation

```
Début
Booléen B :=vrai ;
i := 1
Tant que (i <= N) et (B=vrai)
Faire  Si T[i]<1 ou T[i] > N alors B :=faux ;
          Sinon
          j :=i+1
          Tant que (j <=N) et (B=vrai)
                    faire Si T[i]=T[j] alors B :=faux ;
                    sinon j=j+1 ;
                    fsi ;
          Fait ;
          i=i+1 ;
          Fsi ;
Fait ;
Retourner (B) ;
Fin.
```

Au pire cas : T est un tableau de permutations

Nombre d'unité de temps dépend des deux boucles imbriqués, la boucle interne dépend étroitement de l'indice de la boucle externe dans ce cas nous pouvons voir que :

Quand i = 1 la boucle interne s'exécute en N-1 fois
Quand i = 2 la boucle // // // N-2 fois
…….
Quand i = N-1 // // // 1 fois
Et quand i = N la boucle interne ne fait aucune itération.
Dans ce cas nous pouvons traduire le comportement des deux boucles par la somme suivante :

$$\sum_{i=1}^{N} N - i = (N - 1) + (N - 2) + \ldots + 1 = \sum_{i=1}^{N-1} i = \frac{N(N-1)}{2} = O(N^2).$$

Ce qui nous donne une complexité de l'ordre de $O(N^2)$

2- linéaire avec tableau auxiliaire.

Initialement, A[i] = 0, pour tout 1<=i <=N

Pour chaque valeur 1<=T[i] <= N, on va utiliser T[i] comme indice dans le tableau auxiliaire ; si A[T[i]] = 0, cela signifie que c'est la première version de la valeur de T[i]. Donc A[T[i]] doit être mis à 1.

Si A[T[i]] = 1 cela signifie qu'il y a répétition et donc il ne s'agit pas d'un tableau de permutations.

```
Début
B :=Vrai ;
Pour i :=1 à N faire Aux[i] :=0 fait ;
 i :=1
Tant que (i<=N) et (B= vrai)
Faire Si T[i]>=1 et T[i] <= N alors  Si Aux[T[i]]=1 alors B :=faux ;
                                             Sinon Aux[T[i]] :=1 ;
                                                   i=i+1 ;
                                     Fsi ;
        Sinon B :=faux ;
        Fsi ;
Fait ;
Retourner (B) ;
Fin.
```

Pire cas c'est toujours lorsque le tableau T est de permutations.

Nombre d'itérations dans le pire cas = N+N= 2*N = O(N).

3- L'Algorithme linéaire sans tableau auxiliaire :

```
Début

Booléen B =vrai ;

I=1 ;

Tant que (i<=N et B = vrai)

Faire si T[i]<1 ou T[i] > N alors B = faux ;

        Sinon Si T[i] <> i alors si T[T[i]] = T[i] alors B = faux ;

                                 Sinon // permuter

                                       Temp= T[T[i]] ; T[T[i]]= T[i] ; T[i]= Temp ;
```

```
                              Fsi ;
                Fsi ;
                i=i+1 ;
                Fsi ;
        Fait ;
I=1 ;
Tant que (i<=N et B= vrai)
Faire Si T[i] <> i et T[T[i]] = T[i] alors B = faux ;
        Sinon i= i+1 ;
        Fsi ;
Fait ;
Fin.
```

Pire cas c'est toujours le même : T est un tableau de permutations.

Le nombre d'itérations dans le pire cas = N+N= 2*N = O(N).

**Exercice 2 :** Produit Matricielle

```
Début
Pour i =1 à N
        Faire
                Pour j= 1 à M //N
                Faire
                        C[i,j]=0 ;
                        Pour k= 1 à P // N
                        Faire  C[i,j]= C[i, j] + A[i, k]* B[k, j] ;        Fait ;
                Fait ;
Fait ;
Fin.
```

C'est un algorithme basé sur des boucles imbriquées « Pour », donc la terminaison des boucles n'aura lieu que lorsque la borne supérieur est atteinte (c'est-à-dire i=N+1, j=M+1 et k= P+1) et aucune terminaison n'est possible avant donc le pire cas = meilleure as = moyen cas.

Trois boucles imbriquées indépendantes, donc :

Le nombre d'itérations = N*M*P = O(N*M*P).

Réponse à la question 1 : Cas de matrices carrées d'ordre N, Le nombre d'itérations = N*N*N = $O(N^3)$.

**Exercice 5 :**

- Algorithme A :

Tant que (i<=M) et (j <=N) faire …… fait ;

Condition d'arret : (i >M) ou bien (j > N) ; cela signifie que dès qu'un des deux indices atteint sa borne supérieur, la boucle s'arrete.

Ainsi, le nombre d'itération = Min (N, M)= O (Min (N, M)).

- Algorithme B :

Tant que (i<= M) ou (j <= N) faire …. Fait ;

Condition d'arret : (i>M et j >N), cela signifie que les deux indices doivent atteindre leurs bornes supérieurs pour que la boucle s'arrête.

Ainsi, la nombre d'itération = Max (N, M) = O(Max(N, M)).

- Algorithme C :

Simule deux boucles séquentielles ; c'est-à-dire que l'indice j reste à un jusqu'à ce que l'indice i atteint sa borne supérieure.

Nombre d'itérations = N+ M = O(N+M).

- Algorithme D :

Simule deux boucles imbriquées ; c'est-à-dire, que lorsque l'indice i atteint sa borne supérieur, l'indice j est incrémenté de 1 et l'indice i est remis à 1 pour reprendre à partir du début. Dans ce cas, l'indice i sera remis à 1 autant que fois que l'indice j sera incrémenté.

Ainsi, le nombre d'itération= N*M = O(N*M).

J=1, I=1

J=1, i=2 ;

…..

J=1, i=m ;

J=1, i= m+1    bloc sinon ici, j=2, i=1

J=2, i=1,

……

J=2, i=m+1    bloc sinon j=3, i=1

……..

J=n,i=1

……

J=n, i=m+1     bloc sinon, j=n+1, i=1

Fin de la boucle.

**Exercice 3 :** PGCD

1. Version Itérative :

```
PGCD (A, B : entier) ;     Entier ;
Début
        R= A mod B
        Tant que (R <> 0)
        Faire    A= B ;
        B= R ; // A mod B
        R= A mod B ;
        Fait ;
        PGCD = B ;
Fin ;
```

2. Complexité :

Analyse de l'algorithme :

- Boucle tant que avec initialisation qui dépend des valeurs passées en paramètre A et B.
- Condition d'arrêt connue : R = 0 (le dernier reste = 0)
- Le pas de la boucle dépend du reste de la division entre les valeurs actuelles de A et de B.
- On ne peut déterminer le nombre d'itérations directement, il est impératif de traduire le comportement de cette boucle en une série ou suite à déterminer.
- La suite ou la série correspondante devra être décroissante, parce qu'elle démarre d'une certaine valeur R et décroît progressivement jusqu'à atteindre la valeur de 0

R1, R2, R3, …… $R_{k-1}$=PGCD, $R_k = 0$, avec k le nombre d'itération de la boucle (à déterminer)

$R_1=A_1$ mod $B_1$, $R_2= A_2$ mod $B_2$, $R_3=A_3$ mod $B_3$ ………, PGCD, 0

A mod B, B mod R1, R1 mod R2, R2 mod R3, ……, PGCD, 0 ……. (I)

D'abord commençons par déterminer le pire cas (c-à-d, dans quel cas la boucle va devoir effectuer le maximum d'itérations possibles).

1. A et B sont premiers entre eux, donc PGCD(A, B) = 1. Ce qui signifie qu'aucun diviseur ne peut être en commun entre A et B.

(I) devient comme suit : A mod B, B mod R1, R1 mod R2, R2 mod R3, ……, 1, 0 --- (II)

Donc, dans la définition de notre suite on a : $U_0$= 0, $U_1$= 1. Un= R1

Le nombre d'itération de la boucle correspond à la langueur de la suite Un jusqu'au termine $U_0$.

2. A mod B = A- (A div B)*B, A div B = quotient de division de A par B.

A=100, B=15 PGCD (A, B )= 5

100 : 15 R1= 10, R2= 5, R3= 0

A=100, B=51 PGCD (A, B)= 1

A=100, B= 51, R1 = 49 ; A=51, B= 49, R=2 ; A=49, B=2, R= 1, A=2, B=1, R = 0.

Si à chaque itération A div B =1, alors R= A mod B R= A - B

Ainsi, (II) devient :

A –B, B – R1, R1 – R2, R2 – R3, …… , 1, 0

**A ,B , A – B** , **B- (A – B)**, **(A- B)** – **[B – (A- B)]**, B – (A-B) – [(A- B) – [B – (A- B)]], …. , 1, 0

3. A et B appartiennent à la même suite que les restes, ce qui signifie que les valeurs initiales de A et B suivent aussi la même suite que l'évolution des R.

Si on note A= Un et B = Un-1 on aura :

R1 = $U_{n-2} = U_n - U_{n-1}$ Un = $U_{n-1} + U_{n-2}$, U1= 1, U0= 0.

Deux possibilités :

- Soit vous avez identifié une suite de Fibonnachi et donc Un <= $2^k$
- Ou bien on va résoudre la suite :

On a :

Un = $U_{n-1}+ U_{n-2}$

Un-1= $U_{n-2} + U_{n-3,}$ on remplace dans l'expression des Un, on obtient :

$Un = 2^* U_{n-2} + U_{n-3}$, On sait que Un >= 0 pour tout N.

$Un <= 2^* U_{n-2} <= 2^*2\ U_{n-4} <= 2^3\ U_{n-6}, <=\ldots\ldots <= 2^k\ U_1$. Puisque U1 = 1

Donc, $Un <= 2^k$, K représente le nombre de termes de la suite pour atteindre $U_0$, donc il représente le nombre de reste qui correspond aussi au nombre d'itérations.

Donc $k <= Log_2 (Un)$ $k <= Log_2(A)$.

Ainsi, la complexité de l'algorithme est en $O(Log_2(A))$.

Pour la complexité spatiale : la version itérative à nécessiter 4 zones mémoires de type entier. Cette taille de dépend pas des données entrées dans la complexité spatiale est en O(1).

3. Version Récursive :

PGCD_R (A, B : entier) Entier ;
Début
Si B = 0 alors retourner(A) ;
Sinon PGCD_R (B, A mod B) ;
Fsi ;
Fin ;

Les mêmes conditions du pire cas sont valables pour la version récursive.
$Rn=An \bmod B_n$

Si on note Rn : le premier reste calculer en fonction des valeurs initiales de A et B donc on aura :

Rn = A mod B.

Rn-1 = B mod Rn

$R_{n-2} = R_n \bmod R_{n-1}, R_1= 1, R_0 = 0.$

Sous les mêmes hypothèses du pire cas vu précédemment dans la version itérative on aura :

$R_{n-2} = R_n - R_{n-1}, R_1= 1, R_0 = 0.$

$R_n= R_{n-1} + R_{n-2}, R_1=1, R_0= 0.$

On obtient donc une suite de Fibonnachi.

Ainsi, la version récursive du PGCD à le même ordre de complexité que la version itérative.

Pour la complexité spatiale : chaque appel récursif nécessite 3 zones mémoires. Au total, on a k appel récursif qui correspond à log (A). donc, on peut dire qu'au total, la version récursive nécessite 3* log (A) zone mémoire. Ainsi, la complexité spatiale est en O (log A)

4. Algorithme du PPCM :

A=4, B=7 PPCM (A, B) = 28

A=4, B=6 PPCM (A, B)= 12.

PPCM (A, B : entier)     entier ;

Début
PPCM = A ;
Tant que (PPCM Mod B <> 0)
Faire PPCM = PPCM +A ; Fait ;
Fin.

Complexité :

- Pire cas : PPCM (A, B) = A*B
- Nombre d'itérations dans le pire = B

On va rajouter (B-1) fois la valeur de A au PPCM pour atteindre une valeur qui soit en même temps multiple de A et de B.

Donc complexité = O(B).

Pour optimiser cette complexité, on peut utiliser la relation entre le PGCD et le PPCM :

PPCM = (A*B) / PGCD (A, B)

Dans ce cas, la complexité du PPCM = Complexité du PGCD = O($\text{Log}_2$ A).

**Exercice 4 :**

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 3 | 1 | 5 | 1 | 7 | 1 | 9 | 1 |
| | | 0 | 1 | 5 | 1 | 7 | 1 | 1 | 1 |
| | | | 1 | 5 | 1 | 7 | 1 | 1 | 1 |
| | | | | 0 | 1 | 7 | 1 | 1 | 1 |
| | | | | | 1 | 7 | 1 | 1 | 1 |
| | | | | | | 0 | 1 | 1 | 1 |
| | | | | | | | 1 | 1 | 1 |
| | | | | | | | | 1 | 1 |
| | | | | | | | | | 1 |

Deux versions de l'algorithme sont possibles, le premier naive et parcours le tableau élément par élément à la recherche des multiples d'un nombre donné.

La seconde version, plus optimisé, est basée sur la fréquence des multiples. Par exemple, les multiples de 2 se trouvent à un pas de 2. Les multiples de 3 à un pas de 3, les multiples de 5 à un pas de 5 et ainsi de suite pour éviter de parcours tous les éléments un par un à chaque étape d'élimination des multiples.

```
Algorithme version 1
Début
Pour i=2 à N-1 faire
Si T[i] <> 0 // non encore éliminer, donc n'a pas de diviseur, donc il est premier
          j= i+1 ;
          Tant que (j <= N) faire
                    Si T[j] mod T[i] = 0 alors T[j] = 1 fsi ;
                    Sinon j=j+1 ;
          Fait ;
          T[i] = 0 ;
          Fsi ;
Fait ;
Fin.
```

Nombre d'itération de cet algorithme
$= (N-2)+ (N-3)+ (N-5)+(N-7)+ \ldots +1 \le (N-2)+(N-3)+(N-4)+ \ldots + 1 \le (N-1)(N-2)/2$
Donc cette version de l'algorithme est en $O(N^2)$.

```
Algorithme version 2
Début
Pour i=2 à N-1 faire
Si T[i] <> 0 // non encore éliminer, donc n'a pas de diviseur, donc il est premier
          j= 2*i ; // le premier multiple de i
          Tant que (j <= N) faire
                    T[j] = 1 ;
                    j=j+i; // un pas de i à chaque fois
          Fait ;
          T[i] = 0 ;
          Fsi ;
Fait ;
Fin.
```

Nombre d'itérations $= N/2 + N/3+ N/5 + N/7 + .. + 1 \le N/2+ N/3+N/4+ N/5+ \ldots + 1$

$\le N( \frac{1}{2}+ 1/3+ \frac{1}{4}+ 1/5+\ldots + 1/N)$

C'est une série harmonique qui converge vers ln n .

Donc l'ordre de complexité de cette version de l'algorithme est $O(n \ln n)$.

**<u>Exercice 6 :</u>**

```
Int Recherche_séquentielle( int clé)
Début
Tant que (T[i] <> Clé) faire i = i+ 1 ; fait ;
Fin ;
```

L'invariant de boucle correspond à la position de la clé qui est 1 <= position <= N

Pire cas : T[N]=clé. La complexité est en O(N).

Meilleure cas : T[1] = clé. La complexité est en o(1).

```
Int Recherche_rec_Seq(int clé, int pos)
Début
Si T[pos] = clé alors return pos ;
Sinon return (Recherche_rec_Seq(clé, pos +1)) ;
Fsi ;
Fin ;
```

Si la valeur peut ne pas exister dans le tableau les deux algorithmes deviennent :

```
Int Recherche_séquentielle( int clé)
Début
Int pos = -1 ;
Int i =1 ;
Tant que (i<= N) et (pos = -1)
Si (T[i] = Clé) alors pos = i ;
Sinon i = i+ 1 ;
fsi ;
return pos
Fin ;
```

```
Int Recherche_rec_Seq(int clé, int pos)
Début
Si pos > N alors return -1
Sinon Si T[pos] = clé alors return pos ;
Sinon return (Recherche_rec_Seq(clé, pos +1)) ;
Fsi ;
Fin ;
```

Recherche dichotomique :

```
Int Recherche_dicho(int clé)
Début
Int d=1, f=n, m ;
M= (d+ f)/2 ;
Tant que (T[m] <> clé)
Faire Si T[m] > clé alors f= m-1 ;
        Sinon d = m+1 ;
        Fsi ;
        M= (d+f)/2 ;
Fait ;
```

Pire cas la clé existe sur le dernier milieu calculé, Complexité au pire cas = O(log n)

Le nombre d'itérations dans le pire cas correspond au nombre de milieu calculé pour atteindre la position de la valeur clé. Le principe est tel que, l'espace du tableau est à chaque fois diviser par deux jusqu'à ce que la recherche puisse cerner la position de la valeur recherché. En d'autres termes, jusqu'à ce qu'il y'ai qu'une seule position entre l'indice de début et de fin.

Donc, c'est une succession de division par deux jusqu'à atteindre une dimension de 1. D'où la complexité logarithmique.

Meilleur cas : la valeur recherchée existe sur la position du premier milieu. Complexité au meilleur cas = o(1).

```
Int Recherche_dicho-rec(int deb, int fin, int clé)
Début
Int milieu = (Deb + fin ) /2 ;
Si T[milieu] = clé alors return milieu ;
Sinon si T[milieu] < clé alors return Recherche_dicho_rec(milieu+1, fin, clé)) ;
Sinon return Recherche_dicho_rec(deb, milieu+1, clé))  ;
Fsi ;
Fin ;
```

L'invariant de boucle suit cette équation de récurrence :

T(1)=1.

T(N)=T(N/2)+1.

La résolution de cette équation de récurrence permettra de déduire que le nombre d'itérations = long n.
Le pire cas et le meilleur cas sont identiques à la version itérative.

Si la valeur clé peut ne pas appartenir au tableau, les deux algorithmes deviendront ainsi :

```
Int Recherche_dicho(int clé)
Début
Int d=1, f=n, m, pos =-1 ;
Tant que (d <=f) et (pos = -1)
Faire
        M= (d+ f)/2 ;
        Si T[m] = clé alors pos =m ;
        Sinon Si T[m] > clé alors f= m-1 ;
                Sinon d = m+1 ;
                Fsi ;
        Fsi ;
Fait ;
```

```
Return pos ;
Fin ;

Int Recherche_dicho-rec(int deb, int fin, int clé)
Début
Int milieu ;
Si deb > fin alors return -1 ;
Sinon milieu = (Deb + fin ) /2 ;
Si T[milieu] = clé alors return milieu ;
Sinon si T[milieu] < clé alors return Recherche_dicho_rec(milieu+1, fin, clé)) ;
Sinon return Recherche_dicho_rec(deb, milieu+1, clé))  ;
Fsi ;
Fin ;
```

**Exercice 8 :**

1. Algorithme de fusion dans le cas de tableau :

En entrée : deux tableaux T1 et T2 de dimension respectivement N1 et N2.
En sortie : Tableau T de dimension N = N1+N2.

```
Début
Int i=1,j=1, k =1 ;
Tant que (i<=N1) et (j<=N2)
Faire si (T1[i] < T2[j]) alors T[k] = T1[i] ; k=k+1 ; i=i+1 ;
                Sinon Si (T1[i] > T2[j]) alors T[k] = T2[j] ; k=k+1 ; j= j+1 ;
                        Sinon T[k]= T1[i] ;
                                T[k+1] = T2[j] ;
                                k=k+2 ; i=i+1 ; j= j+1 ;
                        fsi ;
        fsi ;
Fait ;
Tant que (i<=N1)
Faire T[k] = T1[i] ;
        i= i+1 ; k=k+1 ;
fait ;
Tant que (j<=N2)
Faire T[k]= T2[j] ;
        j=j+1 ; k=k+1 ;
fait ;
Fin.
```

Complexité au pire cas = complexité au meilleure cas = complexité au moyen cas = O(N1+N2)= O(N).

2. Algorithme de fusion cas de liste chainée

```
En entrée deux liste L1 et L2
En sortie liste L
Début
p1= L1 ; p2= L2 ;
si p1.elt < p2.elt alors L=L1 ; sinon L= L2 ;
                              fsi ;
P= L ;
Tant que (p1->svt <> nil) et (p2->svt <> nil)
Faire
Si (p1.elt < p2.elt)
Alors    Tant que (p1.elt < p2.elt) et (p1->svt <> nil)
                   faire p->svt=p1 ; p1=p1->svt ; fait ;
                   x= p2 ;
                   p2= p2->svt ;
                   p->svt=x ; //rompre le chainage avec la liste L1.
Sinon Si (p1.elt > p2.elt)
         Alors Tant que (p1.elt > p2.elt) et (p2->svt <> nil)
                              faire p->svt=p2 ; p2=p2->svt ; fait ;
                   x= p1 ;
                   p1= p1->svt ;
                   p->svt=x ; //rompre le chainage avec la liste L2.
                   Sinon p->svt=p1 ; p1=p1->svt ;
                   p=p->svt ; p->svt=p2 ; p2=p2->svt ;
                   p=p->svt ; p->svt=nil ;
                   fsi ;
Fait ;
//traitement du dernier élément de la liste
Si (p1->svt=nil et p1.elt <=p2.elt) alors p1->svt = p2 ; fsi ;
Si (p2->svt=nil et p2.elt <=p1.elt) alors p2->svt = p1 ; fsi ;
Fin.
```

Complexité au pire cas = complexité au meilleure cas = complexité au moyen cas = $O(N1+N2)= O(N)$.