TD 2 sur la Complexité

Exercice 1

Soit la suite U_n définit par :

$$\begin{cases} U_n = U_{n-1} \times U_{n-2} + U_{n-3} \\ U_0=1 \\ U_1=1 \\ U_2=1 \end{cases}$$

Question :

- Donner un algorithme récursif qui calcule U_n
- Évaluer sa complexité.

Correction :

```
unsigned  SuiteU (unsigned n ) {
    if (n <=3 )
        return 1;
    else
        return SuiteU (n-1)* SuiteU (n-2)+ SuiteU (n-3) ;
    }
```

➔La complexité de la solution est exponentielle : O (3^n)

Exercice 2 : Triangle de Pascal

On veut calculer les coefficients binomiaux $C_n^k = \binom{n}{k} = \frac{n!}{k!(n-k)!}$. Rappelons les propriétés suivantes :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \; pour \; 0 < k < n$$

$$\binom{n}{n} = 1 \; et \; \binom{n}{0} = 1$$

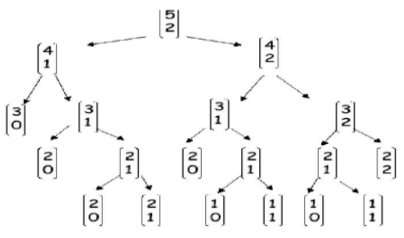
Question :

- Donner un algorithme récursif qui calcul de $\binom{n}{k}$
- Evaluer sa complexité.

Correction :

```
unsigned Combinaison(unsigned n, unsigned k) {
        if (k = = 0) || (k = = n)
                return 1;
        else
                return (Combinaison(n-1, k-1) + Combinaison(n-1, k));
}
```

➔La complexité de la solution est exponentielle : O (2^n)



	0	1	2	3	$\cdots$	$n-1$	n
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\ddots$		
$n-1$	1	$n-1$	$\binom{n-1}{2}$	$\binom{n-1}{3}$	$\cdots$	1	
n	1	n	$\binom{n}{2}$	$\binom{n}{3}$	$\cdots$	n	1

Exercice 3 :

1. Ecrire une fonction qui permet de calculer la somme des éléments d'une matrice carée

```
const unsigned n=10;
float M[n][n];
float somme () {
    unsigned i, j;
    float s=0;
    for (i=0; i<n; i++)
              for (j=0; j<n; j++)
                        s+= M[i][j];
     return s;
} /* O(n^2)  */
```

Exercice 4 :

1. Ecrire une fonction itérative puissanceIterative (a, n) qui permet de calculer a^n.

Rq. En utilisant seulement les opérateurs simples $(+, -, *, /)$

```
float puissanceIterative (float a, unsigned n) {
   unsigned i,
    int resultat = 1;
    for (i=0; i<n; i++)
           resultat *= a;
     return resultat;
}   /* O(n) */
```

2. Ecrire une fonction récursive puissanceRecursive (a, n) qui permet de calculer a^n.

```
float puissanceRecursive (float a, unsigned n) {
   if (n==0)
         return 1;
   else
         return  a* puissanceRecursive (a, n-1);
}   /* O(n) */
```

3. Supposant qu'on peut écrire la fonction puissance de la manière suivante :

$a^{n} = a^{n \text{ div } 2} \times a^{n \text{ div } 2}$ si n est pair, sinon $a^{n} = (a^{n \text{ div } 2} \times a^{n \text{ div } 2} \times a)$

$a^{n/2} = a^{n/4} \times a^{n/4}$

…………

$a^{1} = a$

$a^{0} = 1$

Proposer une fonction puissanceDdynamique (a, n) en utilisant le principe de la programmation dynamique.

Sol 1 :

```
float puissanceDdynamique1 (float a, unsigned n) {
   if (n == 0)
      return 1;
   if (n % 2 == 0)
      return puissanceDdynamique1 (a, n/2) * puissanceDdynamique1 (a, n/2);
   else
      return puissanceDdynamique1 (a, n/2) * puissanceDdynamique2 (a, n/2) * a;
   }  /* O(2 log n) ==> O(n) */
```

Sol 2 :

```
float puissanceDdynamique2 (float a, unsigned n) {
   int temp = puissanceDdynamique2 (a, n/2);
   if (n == 0)
      return 1;
   if (n % 2 == 0)
      return temp * temp;
   else
      return temp * temp * a;
   }  /* O (log2 n)  */
```

Exercice 5 :

Les nombres de Fibonacci sont définis par la récurrence :

- $F_0 = 1$
- $F_1 = 1$
- $F_n = F_{n-1}+F_{n-2}$ pour $n\geq 2$

On peut programmer le calcul de la valeur du nombre de Fibonacci au rang n de plusieurs façons :

- Solution récursif :

```
unsigned fibonR( unsigned n ){
   if((n==0)||(n==1))
        return 1;
   else return fibonR(n-1)+fibonR(n-2);
}/*Complexite est O(2^n)*/
```

- Solution itératif :

```
unsigned fibonI( unsigned n ){
   unsigned F0 = 1, F1 = 1, F = 1;
   for(int i = 2; i<= n; ++i){
      F = F0+F1;
      F0 = F1;
      F1 = F;
   }
   return F1;
} /* Complexite est O(n)*/
```

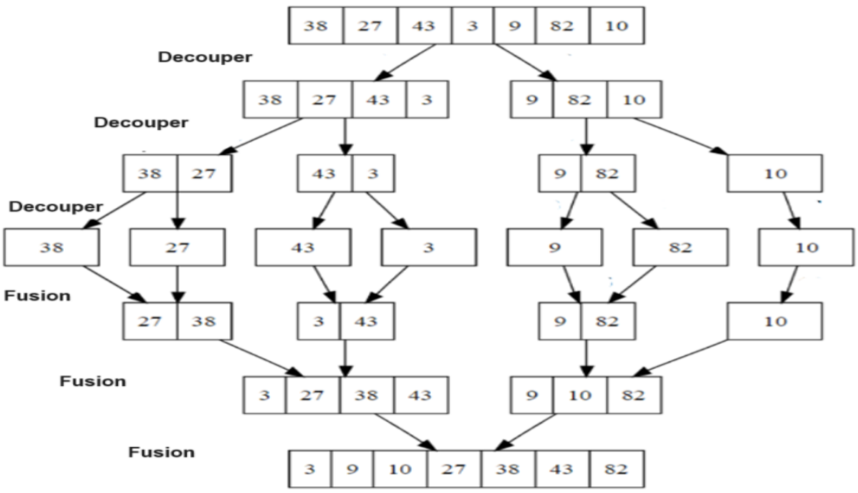
EXERCICE 6 :

Le principe de trie par fusion est :

- Découper le tableau T[1, .. n] à trier en deux sous-tableaux T[1, .. n/2] et T[n/2 +1,..n]
- Trier les deux sous-tableaux T[1, .. n/2] et T[n/2 +1,..n] (récursivement, ou on ne fait rien s'ils sont de taille 1)
- Fusionner les deux sous-tableaux triés T[1, .. n/2] et T[n/2 +1,..n] de sorte à ce que le tableau final soit trié.

1) Proposer une implémentation récursive pour cette technique

2) Déterminer sa complexité.



```
#include <stdio.h>
void fusion (int T [ ], int g, int m, int d);  /* O(n) */

void TriFusion(int T [ ], int g, int d){  /*O(n log2 n) */
       int m;
       if (g<d){
            m=(g+d)/2;
            TriFusion(T, g, m) ;
            TriFusion(T, m+1, d) ;
            Fusion(T, g, m, d) ;
      }
}


void fusion (int tab [ ], int g, int m, int d) {
     int n1 = m - g + 1, n2 = d - m;
     int i, j, k;
     int T1 [n1],  T2 [n2] ;

      for (i = 0; i < n1; i++)
            T1[i] = tab[g + i];

      for (j = 0; j < n2; j++)
            T2[j] = tab[m + 1 + j];

  /* maintient trois conteurs, un pour chacun des deux tableaux et
un pour maintenir l'index actuel du tableau trié final     */
       i = 0;         j = 0;         k = g;

      while (i < n1 && j < n2)
           if (T1[i] <= T2[j])
                tab[k++] = T1[i++];
           else
                tab[k++] = T2[j++];
       /* tab[k++]= (T1[i] <= T2[j]) ? T1[i++]:T2[j++]; */

  // Copiez tous les éléments restants du tableau non vide
      while (i < n1)
            tab[k++] = T1[i++];
      while (j < n2)
            tab[k++] = T2[j++];
 }
```