

TD sur la Complexité

Exercice 1

Soit la suite U_n défini 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 :

```
int function U (unsigned n ) {
    if (n <= 3 )
        return 1;
    else
        return U(n-1)* U(n-2)+ U(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 :

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

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

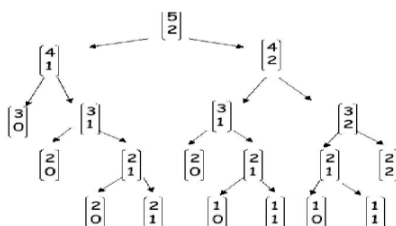
Question :

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

Correction :

```
int function Combinaison(int n,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	...	n-1	n
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
⋮	⋮	⋮	⋮				
n-1	1	n-1	$\binom{n-1}{2}$	$\binom{n-1}{3}$...	1	
n	1	n	$\binom{n}{2}$	$\binom{n}{3}$...	n	1

Exercice 3 :

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

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

```
int puissanceIterative (int 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 .

```
int puissanceRecursive (int 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 :

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

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

.....

$$a^1 = a$$

$$a^0 = 1$$

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

Sol 1 :

```
int puissanceDdynamique1(int a, unsigned n) {
    int temp = puissanceDdynamique1 (a, n/2);
    if (n == 0)
        return 1;
    if (n % 2 == 0)
        return temp * temp;
    else
        return temp * temp * a;
} /* O (log2 n) */
```

Sol 2 :

```

int puissanceDynamique2 (int a, unsigned n) {
    if (n == 0)
        return 1;
    if (n % 2 == 0)
        return puissanceDynamique2 (a, n/2) * puissanceDynamique2 (a, n/2);
    else
        return puissanceDynamique2 (a, n/2) * puissanceDynamique2 (a, n/2) * a;
} /* O(2log n) ==> O(n) */

```

Exercice 4 :

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) */

```