# LAB 4

## Exercice 1: Tower de Hanio

**1) Écrire un algorithme récursif qui résout le problème des Tours de Hanoi. On suppose qu'il y a n disques à déplacer (n est un nombre naturel, n>=1).**

```
1. Procedure Hanoi (n, A, B, C)
2. Begin
3. if (n >= 1) then
4. Hanoi(n-1, A, C, B);
5. Move disk from A to C;
6. Hanoi(n-1, B, A, C);
7. endif;
```

Explication :
n : number de disks
A : La tige de départ (source).
B : La tige intermédiaire (qui sera utilisée temporairement pour déplacer les disques).
C : La tige de destination (où tous les disques doivent être déplacés).

**2) Calculez la complexité de cet algorithme**

On a la relation de récurrence de ce algorithme :

$$\begin{cases} T(0) = 0 \\ T(n) = 2 * T(n-1) + 1 \end{cases}$$

$T(0) = 0$

$T(n) = 2\big(T(n-1)\big) + 1$

$= 2(2T(n-2) + 1) + 1 = 2^2 * T(n-2) + 2 + 1 = 2^2 * T(n-2) + 2^1 + 2^0$
$= 2^2 * (2T(n-3) + 1) + 2^1 + 2^0 = 2^3 * T(n-3) + 2^2 + 2^1 + 2^0$
….

$= 2^n T(n-n) + 2^{n-1} + \cdots + 2^2 + 2^1 + 2^0$

$$= \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

D'où la complexite de tower of hanio = $O(2^n)$

**3) Écrivez le programme correspondant en langage C.**

```c
void hanoi(int n, char A, char B, char C) {
    if (n == 1) {
        printf("Deplacer le disque 1 de %c a %c\n", A, C);
        return;
    }

    hanoi(n - 1, A, C, B);

    printf("Deplacer le disque %d de %c a %c\n", n, A, C);

    hanoi(n - 1, B, A, C);
}
```

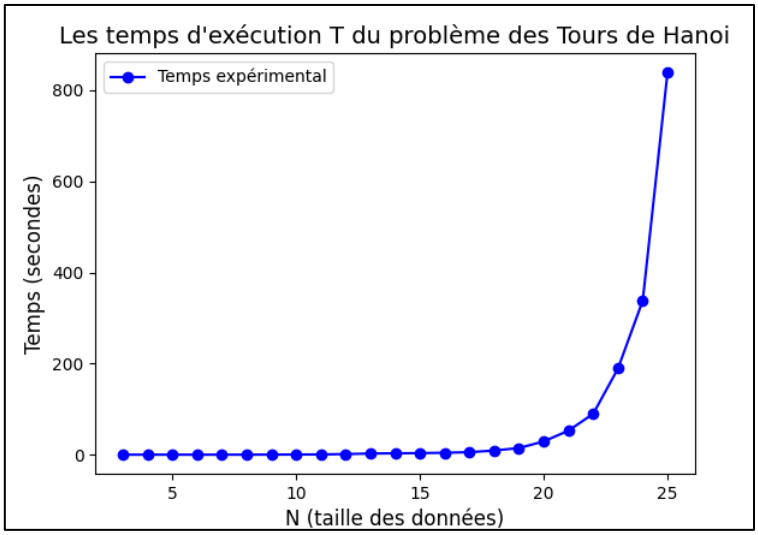
**4) Mesurer les temps d'exécution T (en secondes) pour un échantillon de données de variable n et représenter les résultats sous forme de tableau :**

| n | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|
| T(n) | 0.000 | 0.000 | 0.015 | 0.016 | 0.016 | 0.062 | 0.172 | 0.328 | 0.578 |

| 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|
| 1.312 | 2.546 | 3.015 | 3.505 | 4.252 | 5.520 | 9.116 | 14.582 | 29.135 | 52.756 |

| 22 | 23 | 24 | 25 |
|---|---|---|---|
| 89.808 | 189.152 | 338.056 | 838.233 |

**5) Représenter par un graphique les variations du temps T en fonction des valeurs :**




**6) Comparer la complexité théorique et les mesures expérimentales.**

| n | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|
| T(n) | 0.000 | 0.000 | 0.015 | 0.016 | 0.016 | 0.062 | 0.172 | 0.328 | 0.578 |
| Ttheo | 0.0001 | 0.0003 | 0.0007 | 0.001 | 0.0031 | 0.006 | 0.012 | 0.02 | 0.05 |

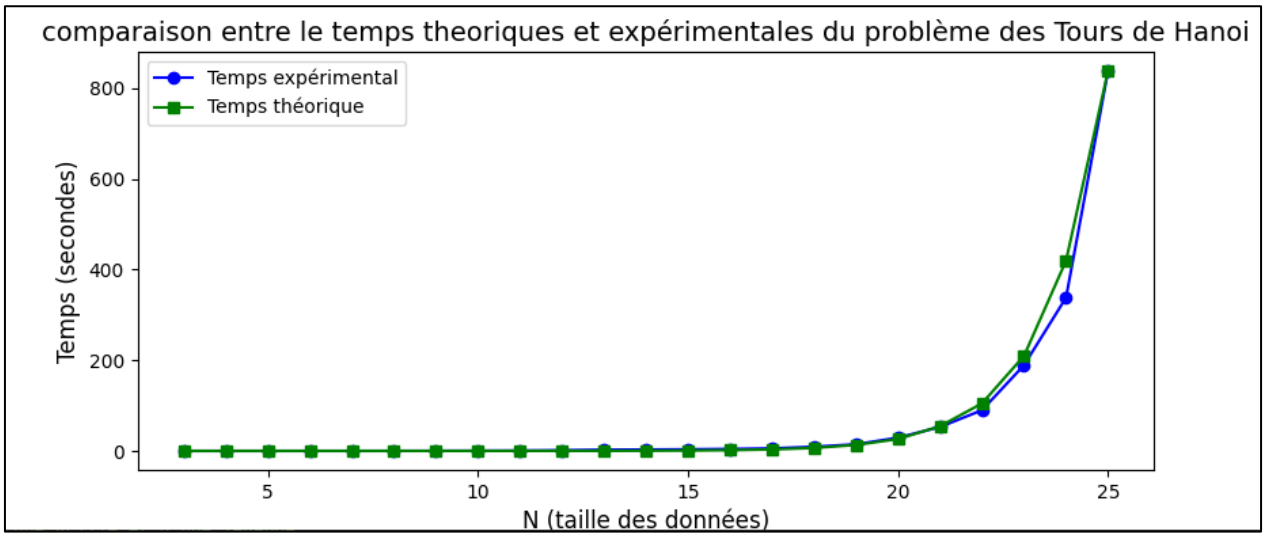
| 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|
| 1.312 | 2.546 | 3.015 | 3.505 | 4.252 | 5.520 | 9.116 | 14.582 | 29.135 | 52.756 |
| 0.10 | 0.20 | 0.41 | 0.81 | 1.63 | 3.27 | 6.55 | 13.10 | 26.21 | 54.42 |

| 22 | 23 | 24 | 25 |
|---|---|---|---|
| 89.808 | 189.152 | 338.056 | 838.233 |
| 104.85 | 209.71 | 419.43 | 838.233 |

$$\boldsymbol{T(n) = f(n) * \Delta t}$$

$$\boldsymbol{\Delta t = \frac{T(n)}{2^n - 1} = \frac{838.233}{2^{25} - 1} = 2.5 * 10^{-5}}$$

**7) Les prédictions théoriques sont-elles compatibles avec les mesures expérimentales ?**




Oui les prédictions théoriques sont compatibles avec les mesures expérimentales, car les deux suivent une tendance exponentielle similaire.

**8) Considérons l'algorithme itératif du problème des Tours de Hanoi. Écrivons-le Programme correspondant puis calculons les temps d'exécution (même tableau que la question 4. à compléter) et représentons les variations de temps par un graphique.**

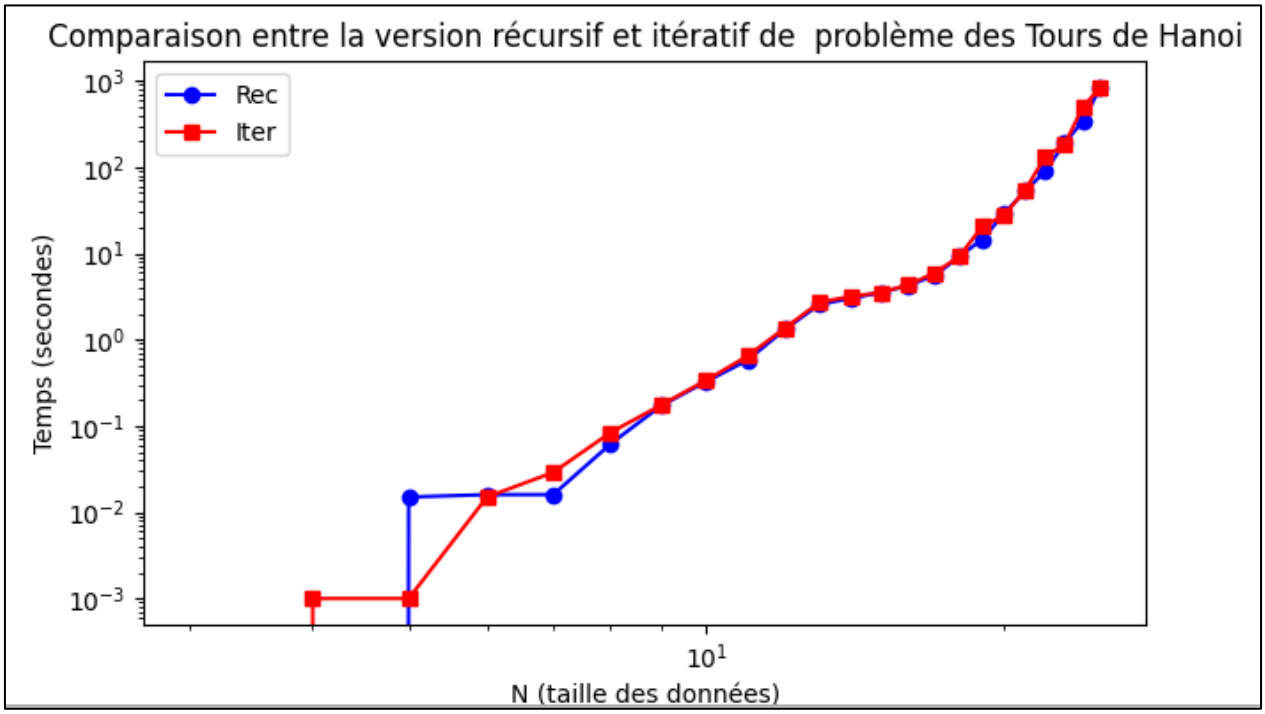
```
void moveDisk(int d, int ar) {
    printf("Deplacer le disque de %d a %d\n", d, ar);
}
void hanoiIterative(int n) {
    int total = pow(2, n) - 1;
    int tours[3];
 for (int i = 0; i < 3; i++) {
        tours[i] = i + 1;
    }
 if (n % 2 == 0) {
        int temp = tours[1];
        tours[1] = tours[2];
        tours[2] = temp;
    }
  for (int i = 1; i <= total; i++) {
        int d = (i % 3);
        int ar = ((i + 1) % 3);
        if (i % 3 == 0) {
            d = (d + 1) % 3;
        }
        moveDisk(tours[d], tours[ar]);
    }}
```

| n | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|
| T(n) | 0.000 | 0.001 | 0.001 | 0.015 | 0.029 | 0.084 | 0.175 |

| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|
| 0.340 | 0.645 | 1.363 | 2.696 | 3.178 | 3.503 | 4.326 | 5.908 | 9.165 | 20.508 |

| 20 | 21 | 22 | 23 | 24 | 25 |
|---|---|---|---|---|---|
| 27.689 | 54.430 | 133.343 | 179.799 | 496.823 | 826.836 |

**9) Comparez les versions récursive et itérative. Que remarquez-vous ?**




- **Petites valeurs de n** : Les deux méthodes sont similaires, mais la récursive est plus élégante.
- **Valeurs moyennes de n** : La méthode itérative commence à montrer un léger avantage.
- **Grandes valeurs de n** : La méthode itérative est meilleure car elle évite les limitations liées à la profondeur de récursion et à la surcharge mémoire.

La méthode itérative est globalement plus optimale pour des cas d'usage à grande échelle.

# Exercice 2 : Fibonacci Sequence

1) **Écrivez une fonction Fibo_Rec qui calcule de manière récursive le n-ième terme de la suite de Fibonacci. Donnez sa complexité.**

```
long Fibo_Rec(int n)
{
    if(n == 1 || n == 0)
        return n;
    return Fibo_Rec(n-1) + Fibo_Rec(n-2);
}
```

La relation de récurrence :

$T(0) = 0$

$T(1) = 1$

$T(n) = T(n-1) + T(n-2)$

$$T(n) = \sum_{i=0}^{n-2} 2^i + 2 = 2^{n-1} - 1 + 2 = 2^{n-1} + 1$$

Ce qui donne une complexité exponentielle O($2^n$)

2) **Écrivez une fonction Fibo_iter qui calcule de manière itérative le n-ième terme de la suite de Fibonacci de complexité linéaire.**

```
int Fibo_iter(int n)
{   if (n==0){
        return 0;
    }
    if (n==1){
        return 1;
    }
    int x=0;
    int y=1;
    int z=0;
    for (int i = 2; i <= n; i++) {
        z=x+y;
        x=y;
        y=z;
    }
    return z;
```

La complexité est linéaire O(n)

3) **Mesurer les temps d'exécution T pour l'échantillon de nombres n ci-dessous et compléter le tableau :**

| **n** | **5** | **10** | **20** | **30** | **40** | **45** | **50** | **55** |
|---|---|---|---|---|---|---|---|---|
| **Fibo_Rec** | 0.000 | 0.000 | 0.000 | 0.016 | 0.587 | 6.694 | 87.347 | 881.386 |
| **Fibo_iter** | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |

Les temps d'exécution pour Fibo_Rec augmentent exponentiellement, tandis que ceux pour Fibo_iter restent négligeables.

4) **Comparer les mesures théoriques et expérimentales des deux fonctions. Les prédictions théoriques sont-elles compatibles avec les mesures expérimentales ?**

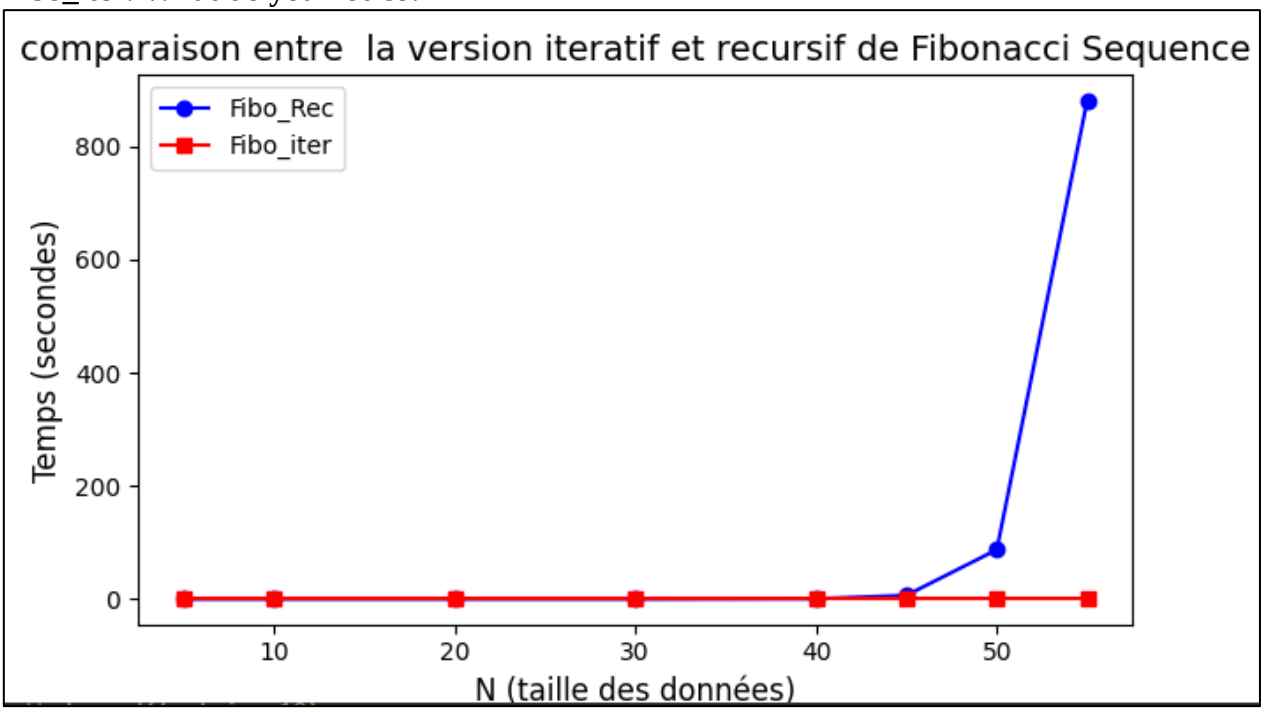
**Fonction Fibo_Rec**

Les mesures expérimentales confirment cette prédiction théorique. Les temps d'exécution augmentent rapidement de manière exponentielle

**Fonction Fibo_iter**

La version itérative du calcul de Fibonacci a une complexité temporelle linéaire,les mesures expérimentales confirment également cette prédiction théorique.

5) **Represent with a graph the variations of the running time T(n) of Fibo_Rec and Fibo_iter. What do you notice?**



Le graphique montre clairement que la version itérative est nettement supérieure à la version récursive pour calculer les nombres de Fibonacci.

La version récursive, avec une complexité exponentielle , devient rapidement inefficace à mesure que la taille de n augmente, rendant son utilisation impraticable pour de grandes valeurs. En revanche, la version itérative, grâce à sa complexité linéaire O(n), reste adaptée même pour des entrées de grande taille.

**6) Faites une copie de la fonction Fibo_iter et modifiez-la pour tester cette propriété (utilisez le type double).**

```
void FiboGoldenRatio(int n) {
    double a = 0, b = 1, fib, ratio;

    for (int i = 1; i <= n; i++) {
        fib = a + b;

        if (i == 1) {
            printf("%d: Un = %.15f, Ratio = Division by zero\n", i, b);
        } else {
            ratio = b / a;
            printf("%d: Un = %.15f, Ratio = %.15f\n", i, b, ratio);
        }
        a = b;
        b = fib;
    }
}
```

```
Enter the number of terms (n): 10
1: Un = 1.000000000000000, Ratio = Division by zero
2: Un = 1.000000000000000, Ratio = 1.000000000000000
3: Un = 2.000000000000000, Ratio = 2.000000000000000
4: Un = 3.000000000000000, Ratio = 1.500000000000000
5: Un = 5.000000000000000, Ratio = 1.666666666666667
6: Un = 8.000000000000000, Ratio = 1.600000000000000
7: Un = 13.000000000000000, Ratio = 1.625000000000000
8: Un = 21.000000000000000, Ratio = 1.615384615384615
9: Un = 34.000000000000000, Ratio = 1.619047619047619
10: Un = 55.000000000000000, Ratio = 1.617647058823529
```

Nous remarquons que lorsque $n \to \infty$

$\frac{U_n}{U_{n-1}} \to \phi \approx 1.618033988749894$. Cela signifie que la propriété est vérifiée.