LAB 1

Part1 (Sum of the first N natural numbers)

1. Développer un algorithme itératif, noté Somme_1, qui permet le calcul de la somme, notée S, des n premiers nombres naturels :

$$S = \sum_{i}^{n} i = 1 + 2 + 3 + \cdots + n$$

L'entier naturel n doit être lu en entrée (n>=1). Utilisez la boucle while

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i, n, s;

    printf("Give the number n: ");
    scanf("%d", &n);
    s = 0;
    i = 1;
    while (i <= n) {
        s += i;
        i++;
    }
    printf("The sum is %d\n", s);
    return 0;
}
```

2. Calculez la complexité temporelle de cet algorithme.

printf("Give the number n: "); →1
scanf("%d", &n); →1
s = 0; →1
i = 1; →1
while (i <= n) { →n+1
s += i; →2n
i++; →2n
}
printf("The sum is %d\n", s); →1

f(n)=1+1+1+1+n+1+2n+2n+1 = 5n+6 →O(n)
L'algorithme a une complexité linéaire O(n).

$$\begin{aligned} T(n) &= f(n) * \Delta t \\ &= 5n + 6 * \Delta t \\ &= 5 * \Delta t * n + 6 * \Delta t \end{aligned}$$

3. Calculer la complexité spatiale de cet algorithme

l'algorithme utilise un espace mémoire fixe car l'espace requis par les variables (i,n,s) est égale =3
Le nombre d'instruction = 8
Le total=8+3=11 octet
Ce qui donne une complexité spatiale constante→ O(1).

4. Développer le programme itératif correspondant, noté PSum_1, avec le langage C.

```c
#include <stdio.h>
#include <stdlib.h>

int main2() {
    double i, n, s;
    printf("Give the number n: ");
    scanf("%lf", &n);
    s = 0;
    i = 1;
        while (i <= n) {
        s += i;
        i++;
    }

    printf("The sum is %lf\n", s);
    return 0;
}
```

Part II (Measuring Execution Time)

5. Reprenez le programme précédent et incluez les instructions qui permettent de mesurer les temps d'exécution T pour l'échantillon de valeurs de n donné dans le tableau ci-dessous (le nouveau programme est noté PSum_2).

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main3() {
    double i, n, s;
    clock_t T1, T2;
    double D;
    printf("Give the number n: ");
    scanf("%lf", &n);
    T1 = clock();
    s = 0;
    i = 1;
    while (i <= n) {
        s += i;
        i++;
    }
    T2 = clock();
    D = ((double)(T2 - T1)) / CLOCKS_PER_SEC;
    printf("Time taken: %f seconds\n", D);
    printf("The sum is %.0lf\n", s);
    return 0;
}
```

n	10^7	$2*10^7$	10^8	$2*10^8$	10^9	$2*10^9$	10^{10}	$2*10^{10}$	10^{11}	$2*10^{11}$
T Experimental	0.022	0.047	0.178	0.379	1.888	3.757	26.444	50.721	245.392	518.009
T Théorique	0.022	0.044	0.22	0.44	2.2	4.4	22	44	220	440

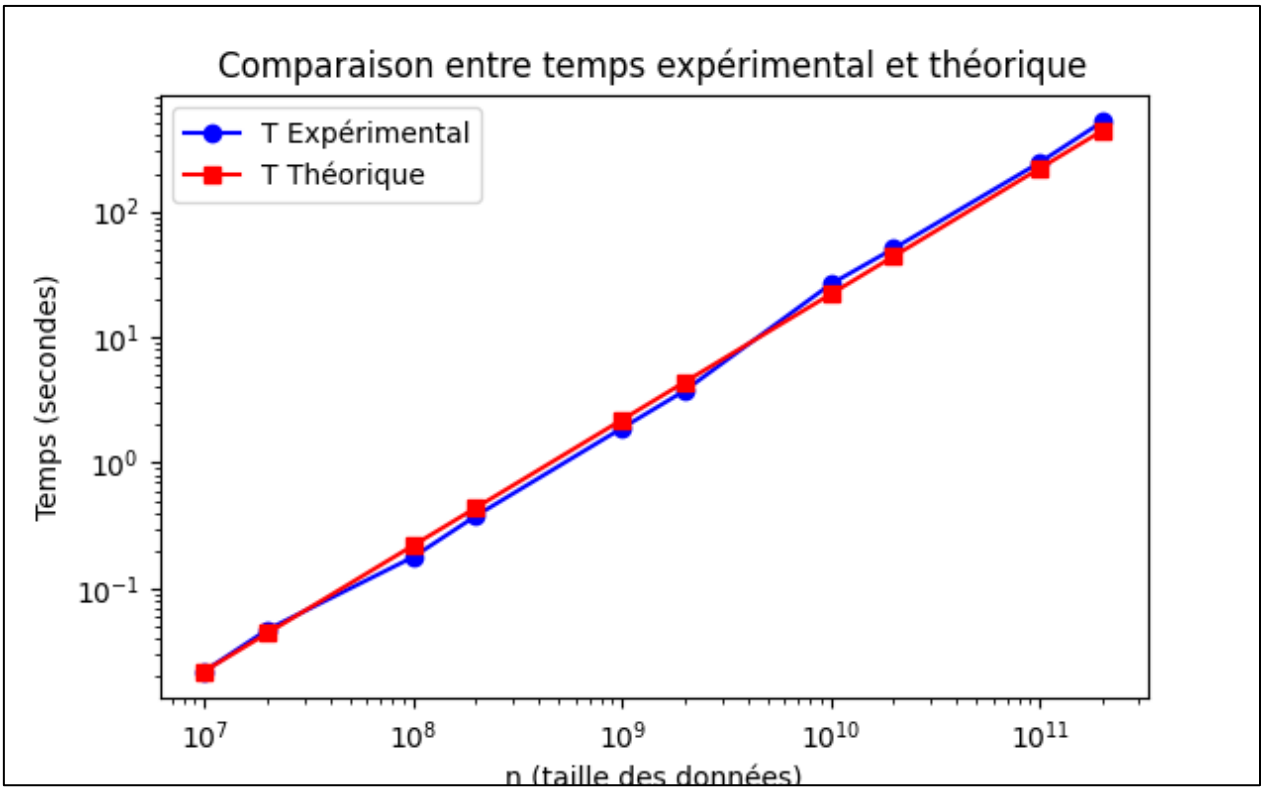
$$f(n) \times \Delta t = T(n)$$

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

$$\Delta t = \frac{T(n)}{5n + 6} = \frac{0.022}{5(10^7) + 6} = 4.4 * 10 - 10$$

6. Tracer les courbes d'évolution du temps d'exécution théorique et expérimental dans Excel.

7.



8. Comparez les courbes et interprétez les résultats obtenus.

La courbe expérimentale suit parfaitement la même tendance exponentielle que les valeurs théoriques. Cela confirme que la complexité temporelle de l'algorithme Somme_1 est bien en O(n)

LAB 2

Algorithm 1 (A1) : Naive approach

1. Écrivez l'algorithme correspondant

1. Début
2. Print(("Algorithme 1 : Entrez un nombre pour tester s'il est premier : »)
3. Lire n ;
4. Cmp=0 ;
5. Pour chaque entier iii allant de 1 à n
6. If(n mod i=0)alors
7. cmp ++ ;
8. faire ;
9. Si cmp=2 alors :
10. Print("Oui, n est un nombre premier.") ;
11. Sinon
12. Print("Non, n n'est pas un nombre premier.")
13. Fin

2. Calculate the best and worst case theoretical complexity of this algorithm in Big Oh

Meilleur cas:

Le meilleur cas se produit lorsque n est égal à 1:

Si n=1, la condition $n <= 1$ est vraie, donc la boucle ne s'exécute pas.

Complexité asymptotique dans le meilleur cas : O(1)

Pire cas:

Se produit pour des grandes valeurs de n, où la boucle s'exécute complètement jusqu'à i=n.

1. **Début**
2. **Print(**("Algorithme 1 : Entrez un nombre pour tester s'il est premier : ») →1
3. Lire n ; →1
4. Cmp=0 ; →1
5. Pour chaque entier iii allant de 1 à n → n
6. If(n mod i=0)alors →2
7. cmp ++ ; →2
8. faire ;
9. Si cmp=2 alors : → 2

10. Print("Oui, n est un nombre premier.") ; →1
11. Sinon
12. Print("Non, n n'est pas un nombre premier."
13. **Fin**

f(n)=1+1+1+n(4)+3=4n+6
alors, la complexité dans le pire cas est O(n)

3. Écrire le programme correspondant.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n, cmp = 0;

    printf("Algorithme 1 : Entrez un nombre pour tester s'il est premier : ");
    scanf("%d", &n);
        for (int i = 1; i <= n; i++) {
            if (n % i == 0) cmp++;
        }
        if (cmp == 2) {
            printf("N = %d\nEst-ce un nombre premier ?\n Oui", n);
        } else {
            printf("N = %d\nEst-ce un nombre premier ?\n Non", n);
        }
    return 0;
}
```

n	1000003	2000003	4000037	8000009	16000057	32000011	64000031
T(s)	0.003	0.005	0.010	0.017	0.034	0.069	0.132
Premier ?	oui	oui	oui	oui	oui	oui	oui

n	128000003	256000001	16000057	512000009	1024000009	2048000011
T	0.269	0.536	0.047	1.065	2.125	4.257
Premier ?	oui	oui	oui	oui	oui	oui

c- Que remarquons-nous à propos des données et des mesures obtenues ?

Nous remarquons que d'après les données et mesures obtenues, la croissance de la complexité temporelle est linéaire ~ O(n)

d- Comparer la complexité théorique et les mesures expérimentales.

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

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

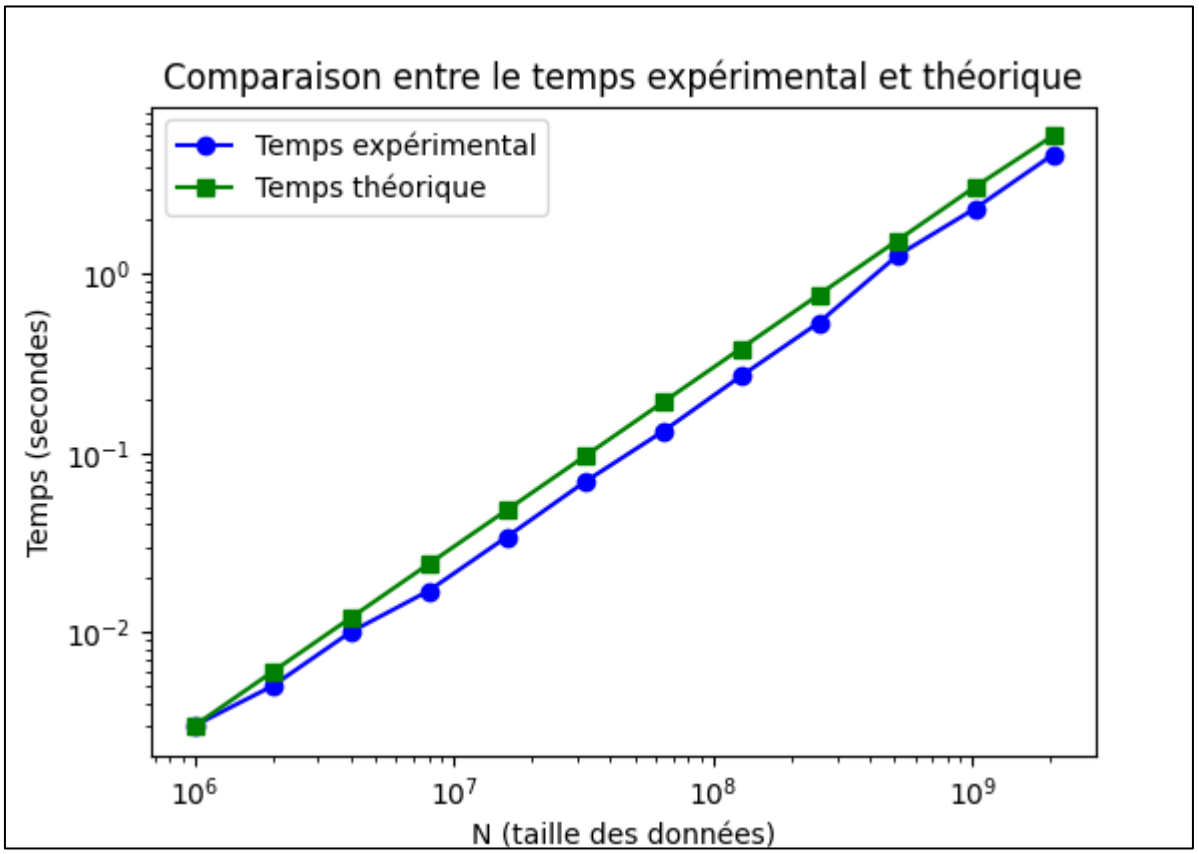
$$\Delta t = \frac{T(n)}{4n+6} = \frac{0.003}{4 * 1000003 + 6} = 7.5 * 10^{-10}$$

n	1000003	2000003	4000037	8000009	16000057	32000011	64000031
T(s)	0.003	0.005	0.010	0.017	0.034	0.069	0.132
T theori	0.003	0.006	0.012	0.024	0.048	0.096	0.192

n	128000003	256000001	512000009	1024000009	2048000011
T	0.269	0.536	1.265	2.325	4.657
T theori	0.384	0.768	1.536	3.072	5.93

Les prédictions théoriques et les mesures expérimentales sont très proches.

e- Represent (in the same graph) with 2 curves the variations of the experimental and theoretical execution time T(N) for best case and 1 curve for experimental worst case.



Algorithm 2 (A2)

1) Écrivez l'algorithme correspondant

```
1.  n, cmpt, i integer;
2.  Begin
3.  cmp ← 0;
4.  i ← 1;
5.  print('entrez un nombre pour tester s'il est premier ')
6.  lire(n);
7.  Pour i de 1 a n/2 faire
8.  if ( n%i == 0 ) then
9.  cmp ← cmp + 1;
10. end if
11. i ← i + 1;
12. fait
13. if ( cmp == 1 ) then
14. print('Oui, n est un nombre premier.' );
15. else
16. Print("Non, n n'est pas un nombre premier.")
17. end if
18. End
```

Calculate the best and worst case theoretical complexity of this algorithm in Big Oh

Meilleur cas:

Le meilleur cas se produit lorsque n est égal à 1:

Si n=1, la condition $n <= 1$ est vraie, donc la boucle ne s'exécute pas.

Complexité asymptotique dans le meilleur cas : O(1)

Pire cas:

Se produit pour des grandes valeurs de n, où la boucle s'exécute complètement jusqu'à i=n/2.

```
1.  cmp ← 0; →1
2.  print('entrez un nombre pour tester s'il est premier ') →1
3.  lire(n); →1
4.  Pour i de 1 a n/2 faire →n/2
5.  if ( n%i == 0 ) then  →2
6.  cmp ← cmp + 1;  →2
7.  end if
8.  fait
```

```
9.  if ( cmp == 1 ) then →2
10. print('Oui, n est un nombre premier.' ); →1
11. else
12. Print("Non, n n'est pas un nombre premier.")
13. end if
```

f(n)=1+1+1+$\frac{n}{2}$ (4)+3=4$\frac{n}{2}$+6=2n+6

alors, la complexité dans le pire cas est O(n)

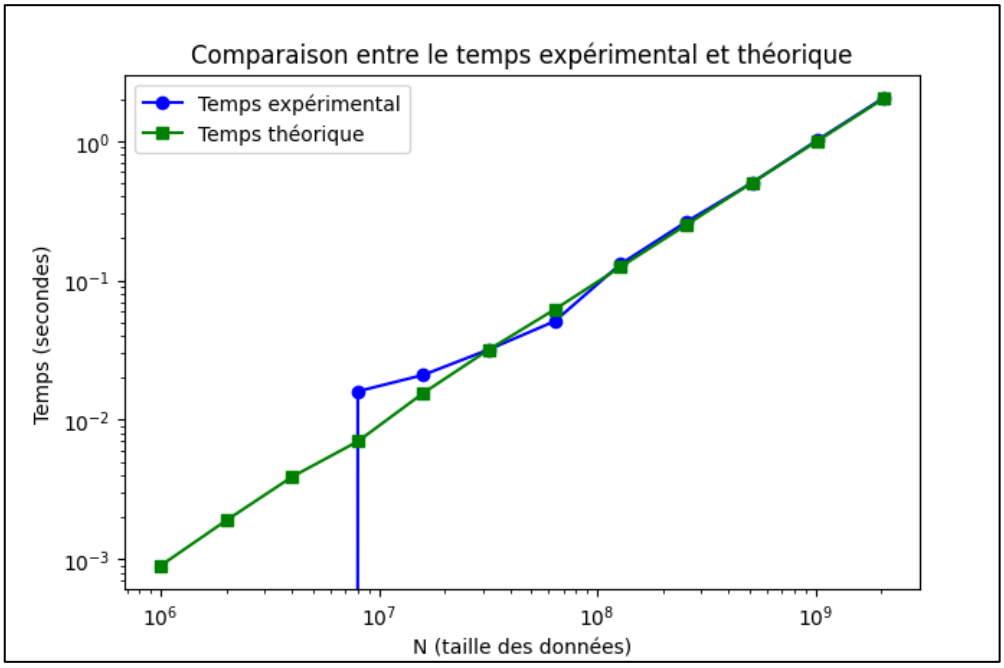
$$\boldsymbol{T(n) = f(n) * \Delta t}$$

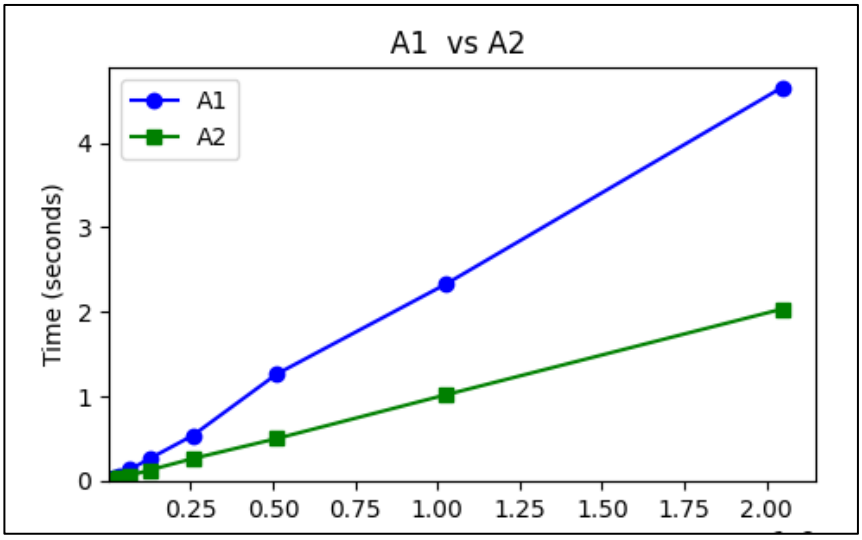
$$\boldsymbol{\Delta t = \frac{T(n)}{2n+6}} = \frac{0.032}{2 * 32000011 + 6} = \boldsymbol{4.9 * 10^{-10}}$$

n	1000003	2000003	4000037	8000009	16000057	32000011	64000031
T(s)	0.000	0.000	0.000	0.016	0.021	0.032	0.051
T theori	0.0009	0.0019	0.0039	0.007	0.0156	0.032	0.062

n	128000003	256000001	512000009	1024000009	2048000011
T	0.131	0.263	0.501	1.017	2.034
T theori	0.125	0.25	0.5017	1.003	2.007

Les prédictions théoriques et les mesures expérimentales sont très proches.





La deuxième solution A2 prend environ la moitié du temps de la première solution A1, donc la deuxième solution est plus efficace

Algorithm 3 (A3)

Écrivez l'algorithme correspondant

1. n, cmpt, i integer;
2. Begin
3. cmp ← 0;
4. i ← 1;
5. print(‘entrez un nombre pour tester s’il est premier ’)
6. lire(n);
7. Pour i de 1 a $\sqrt{n}$ faire
8. if (n%i == 0) then
9. cmp ← cmp + 1;
10. end if
11. i ← i + 1;
12. fait
13. if (cmp == 1) then
14. print(‘Oui, n est un nombre premier.’);
15. else
16. Print("Non, n n'est pas un nombre premier.")
17. end if
18. End

Calculate the best and worst case theoretical complexity of this algorithm in Big Oh

Meilleur cas:

Le meilleur cas se produit lorsque n est égal à 1:

Si n=1, la condition $n <= 1$ est vraie, donc la boucle ne s'exécute pas.

Complexité asymptotique dans le meilleur cas : O(1)

Pire cas:

Se produit pour des grandes valeurs de n, où la boucle s'exécute complètement jusqu'à i=n/2.

```
1.  cmp ← 0; →1
2.   print('entrez un nombre pour tester s'il est premier ') →1
3.   lire(n); →1
4.   Pour i de 1 a √n faire →√n
5.   if ( n%i == 0 ) then  →2
6.   cmp ← cmp + 1;  →2
7.   end if
8.  fait
9.   if ( cmp == 1 ) then →2
10.  print('Oui, n est un nombre premier.' ); →1
11.  else
12. Print("Non, n n'est pas un nombre premier.")
13.  end if
```

f(n)=1+1+1+$\frac{n}{2}$ (4)+3=4$\sqrt{n}$+6=2$\sqrt{n}$+6

alors, la complexité dans le pire cas est O($\sqrt{n}$)

n	1000003	2000003	4000037	8000009	16000057	32000011	64000031
T(s)	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

n	128000003	256000001	512000009	1024000009	2048000011
T	0.0000	0.0000	0.0000	0.0000	0.0000

Algorithm 4 (A4)

Écrivez l'algorithme correspondant

n, cmpt, i integer;

1. Begin
2. cmp ← 0;
3. i ← 1;
4. print('entrez un nombre pour tester s'il est premier ')
5. lire(n);
6. If (n%2 =0) and (n !=2) then
7. print('Non, n n'est pas un nombre premier');
8. else
9. Pour i de 1 a $\sqrt{n}$ faire
10. if (n%i == 0) then
11. cmp ← cmp + 1;
12. end if
13. i ← i + 1;
14. fait
15. if (cmp == 1) then
16. print('Oui, n est un nombre premier.');
17. else
18. Print("Non, n n'est pas un nombre premier.")
19. end if
20. enif
21. End

Calculate the best and worst case theoretical complexity of this algorithm in Big Oh

Meilleur cas:

Le meilleur cas se produit lorsque n est égal à 1:

Si n=1, la condition $n <= 1$ est vraie, donc la boucle ne s'exécute pas.

Complexité asymptotique dans le meilleur cas : O(1)

Pire cas:

Se produit pour des grandes valeurs de n, où la boucle s'exécute complètement jusqu'à i=n/2.

14. cmp ← 0; →1
1. print('entrez un nombre pour tester s'il est premier ') →1
2. lire(n); →1

3. If (n%2 =0) and (n !=2) then → 2
4. print('Non, n n'est pas un nombre premier'); →1
5. else
6. end if
7. Pour i de 1 a $\sqrt{n}$ faire →$\sqrt{n}$
8. if (n%i == 0) then →2
9. cmp ← cmp + 1; →2
10. end if
11. fait
12. if (cmp == 1) then →2
13. print('Oui, n est un nombre premier.'); →1
14. else
15. Print("Non, n n'est pas un nombre premier.")
16. end if

f(n)=1+1+1+1+2+$\frac{n}{2}$ (4)+3=4$\sqrt{n}$+6=2$\sqrt{n}$+8

alors, la complexité dans le pire cas est O($\sqrt{n}$)

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

$$\Delta \boldsymbol{t} = \frac{\boldsymbol{T(n)}}{\boldsymbol{2\sqrt{n} + 8}} = \frac{0.032}{2 * \sqrt{\ } + 8}$$

n	1000003	2000003	4000037	8000009	16000057	32000011	64000031
T(s)	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
T theori							

n	128000003	256000001	512000009	1024000009	2048000011
T	0.0000	0.0000	0.0000	0.0000	0.0000
T theori					

L'ordre de complexité des Algorithmes 3 **et** 4 **est de** $\sqrt{N}$ **ce qui est bien plus rapide que les algorithmes vérifiant tous les diviseurs jusqu'à N, mais cela peut entraîner des temps d'exécution trop petits pour la machine donc 0 est affiché**

RESULTAT GLOBAL ET COMPARISON ENTRE LES ALGORITHMES :

Algorithme	A1	A2	A3	A4
complexité	O(n)	O(n)	$O(\sqrt{n})$	$O(\sqrt{n})$

- L'algorithme 2 est une petite amélioration de l'algorithme 1, l'ordre de complexité est le même O(N), les deux algorithmes évoluent linéairement mais l'algorithme 2 fait la moitié d'itération. On voit bien dans le tableau que le temps d'exécution de l'algorithme 2 est approximativement ½ du temps de l'algorithme 1
- L'algorithme 3 représente une amélioration meilleure, car la complexité théorique est réduite à racine carré de N.
- Le temps d'exécution de l'algorithme 3 est négligeable par rapport aux algorithmes 1 et 2, o(N) >>>> o ($\sqrt{N}$)
- L'algorithme 4 est encore une bonne amélioration car les nombres premiers sont des nombres impairs donc il n'est pas nécessaire de tester la divisibilité au nombre pairs Selon le tableau, le temps d'exécution est encore plus petit que l'algorithme 3

LAB 3

Exercice 1: Multiplication de matrices

1- Écrivez l'algorithme qui permet de calculer le produit de 2 matrices A et B :

```
Debut
for i=1 to n do
   for j=1 to p do
          C[i][j] =0
   end if
end if
for i = 1 to n do
    for j = 1 to p do
        for k = 1 to m do
          C[i][j] = C[i][j] + A[i][k] × B[k][j]
        end for
   end for
end for
for i=1 to n do
   for j=1 to p do
          printf(C[i][j] );
   end if
endif
fin
```

Explication

L'algorithme prend en entrée deux matrices A de dimensions N×M et B de dimensions M×P.

On initialise une matrice C de dimensions N×P à des valeurs nulles.

Le calcul du produit matriciel est réalisé en parcourant les lignes de A, les colonnes de B, et en effectuant une somme pondérée par les éléments correspondants.

2- Calculer la complexité temporelle théorique de ce programme en termes de n, m et p

```
1- Debut
2- for i=1 to n do  → n
3-    for j=1 to p do  → p
4-      C[i][j] =0  →1
5-    end if
6- end if
7- for i = 1 to n do →n
8-    for j = 1 to p do →p
9-      for k = 1 to m do →m
10-         C[i][j] = C[i][j] + A[i][k] × B[k][j]  →3
11-      end for
12-     end for
13-  end for
14-  for i=1 to n do → n
15-      for j=1 to p do →p
16-         printf(C[i][j] );→1
17-      end for
18-  endfor
19-  fin
```

$f(n) = (n \times p) + (n \times p \times m \times 3) + (n \times p) = 2(n \times p) + 3(n \times p \times m)$

D'où $f(n) \in O(n \times p \times m)$

<u>Dans le cas où n=m=p (cas des matrices carrées), donner la nouvelle formulation de la complexité.</u>

Si n=m=p alors $f(n) = 2n^2 + 3n^3$

D'où la complexité de O(n^3)

3- Calculez l'espace mémoire nécessaire pour exécuter ce problème.

L'espace mémoire est principalement occupé par les trois matrices A, B et C. Chaque matrice nécessite n^2unités de mémoire, soit un total de $3n^2$unités de mémoire.

Les variables i,j et k sont des entiers et nécessitent un espace mémoire constant =3

Et on a 10 instructions

Donc la complexité spéciale=$3n^2$+3+10=$3n^2$+13

4- Écrivez le programme C correspondant et mesurez les temps d'exécution T du produit de deux matrices carrées (nxn) pour un échantillon de données de la variable n et représentez les résultats sous forme de tableau

Remarque : Tous les matrices crées sont de type dynamique en utilisant des double pointeur int **Matrice

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

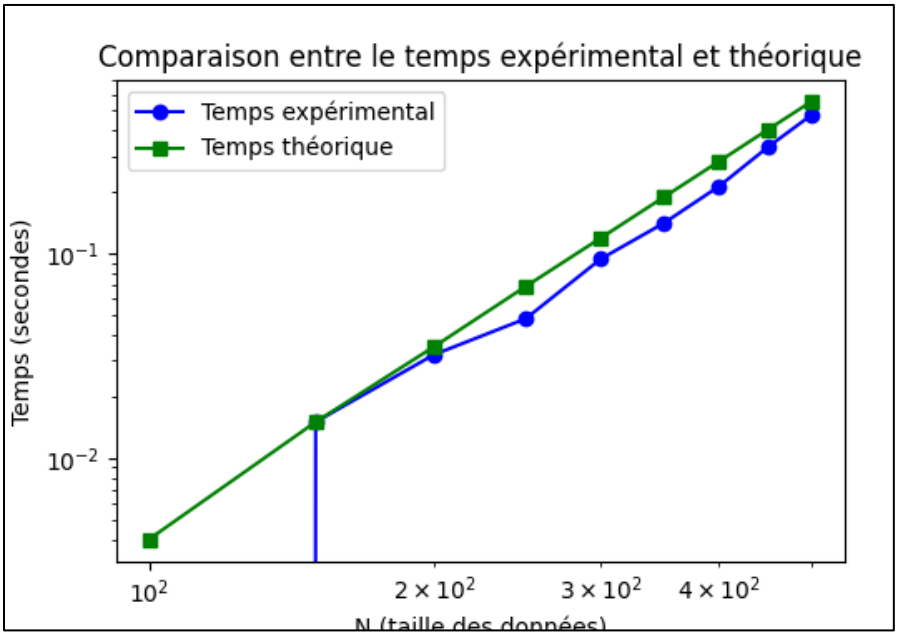
n	100	150	200	250	300	350	400	450	500
T exp	0.000	0.015	0.032	0.048	0.094	0.141	0.213	0.330	0.473
T theo	0.004	0.015	0.035	0.069	0.119	0.189	0.282	0.402	0.551

$$f(n) \times \Delta t = T(n)$$

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

$$\Delta t = \frac{T(n)}{2n^2+3n^3} = \frac{0.015}{2(150)^2+3(150)^3} = 1.47 * 10^{-9}$$

5- Représenter par un graphique les variations du temps T en fonction des valeurs de n.



5- Comparer la complexité théorique et la complexité expérimentale. Y a-t-il une concordanceentre le modèle théorique et les mesures expérimentales ?

Les deux mesures ($Texp\ et\ Ttheo$) augmentent de manière cubique, comme prévu par le modèle théorique O(n^3).
Cela confirme que la croissance expérimentale suit bien la complexité théorique.

Exercice 2 : Recherche d'une sous-matrice

1- En supposant que les éléments de A et B ne sont pas triés, écrire une fonction subMat1 qui recherche B dans A. Évaluer sa complexité temporelle théorique.

```
int SousMat1( int N, int M, int N_prime, int M_prime , int **A, int **B) {
int i, j, x, y;

if( N_prime > N || M_prime  > M){
printf("La matrice B est plus grande que A \n"); return 0;
}
for (i = 0; i <= N - N_prime; i++) {
   for (j = 0; j <= M - B; j++) {
      int trouve = 1; // si on trouve une valeure different on met trouve a 0 et on sort de boucle
    for (x = 0; x < N_prime; x++) {
       for (y = 0; y < M_prime; y++) {
        if (A[i + x][j + y] != B[x][y]) {
        trouve = 0; // Submatrix doesn't match at this position
     break;
    }
    }
   if (!trouve) {
break;
}}
if (trouve) {
printf("Soumatrice trouvee a ligne =    %d , colonne =  %d\n", i,j);
return 1;
}}}
printf("Soumatrice n'existe pas \n"); return 0;
}
```

Explication

L'algorithme recherche l'existence de B(N_prime,M_prime) dans A(N,M)
On commence d'abord par le test sur la taille des deux matrices
Comme on voit il y'a 4 boucles imbriqué
Les deux premières boucles extérieures sont pour parcourir les cases de la matrice A

On utilise les indice i et j pour le parcours de lignes et colonne respectivement

On arrête le parcours a N-N_prime et M-M_prime car sinon la sou matrice recherché va dépasser

l'extrémité de la matrice A

Évaluer sa complexité temporelle théorique.

```
int SousMat1( int N, int M, int N_prime, int M_prime , int
**A, int **B) {
int i, j, x, y;

if( N_prime > N || M_prime  > M){
printf("La matrice B est plus grande que A \n"); return 0;
}
for (i = 0; i <= N - N_prime; i++) {
   for (j = 0; j <= M - B; j++) {
      int trouve = 1; // si on trouve une valeure different
on met trouve a 0 et on sort de boucle
    for (x = 0; x < N_prime; x++) {
       for (y = 0; y < M_prime; y++) {
        if (A [i + x][j + y] != B [x][y]) {
        trouve = 0; // Submatrix doesn't match at this
position
     break;
    }
S
   if (!trouve) {
break;
}}
if (trouve) {
printf("Soumatrice trouvee a ligne =    %d , colonne =
     %d\n", i,j);
return 1;
}}}
printf("Soumatrice n'existe pas \n"); return 0;
}
```

1ere boucle: $N - N\_prime$ iteration
2eme boucle: $M - M\_prime$ iterations
3eme boucle: N_prime iterations
4eme boucle: M_prime iterations

Nombre itérations totale : $(N - N\_prime)(M - M\_prime)(N\_prime)(M\_prime)$

Complexité temporelle :

$$O((n - n_{prime}) * (m - m_{prime}) * n_{prime} * m_{prime})$$

2- En supposant que chacune des lignes de A et B est triée par ordre croissant (voir figure), écrivez une fonction subMat2 non naïve de complexité minimale pour trouver B dans A
Évaluez sa complexité temporelle théorique.

```
int SousMat2(int N, int M, int N_prime, int M_prime, int **A, int **B) {
if (N_prime > N || M_prime > M) {
printf("La matrice B est plus grande que A \n"); return 0;
}
int i, j, x, y;
for (i = 0; i <= N -N_prime; i++) {
for (j = 0; j <= M - M_prime; j++) {
int trouve = 1;
for (x = 0; x < N_prime; x++) {
// tester seulement la premiere case de chaque ligne
// si vrai , on parcours la matrice B , sinon on passe
if (A[i + x][j] != B[x][0]) { trouve = 0;
break;
}
//  parcourir la matrice B si la premiere case est trouvé
for (y = 1; y < B; y++) {
if (A[i + x][j + y] != B[x][y]) { trouve = 0;
break;
}}
if (!trouve) {
break;
}}
if (trouve) {
printf("Soumatrice trouvee a ligne =    %d , colonne =  %d\n", i, j);
return 1;
}}}
printf("Soumatrice n'existe pas \n"); return 0;
```

Explication :

L'amélioration par rapport au premier algorithme est que :

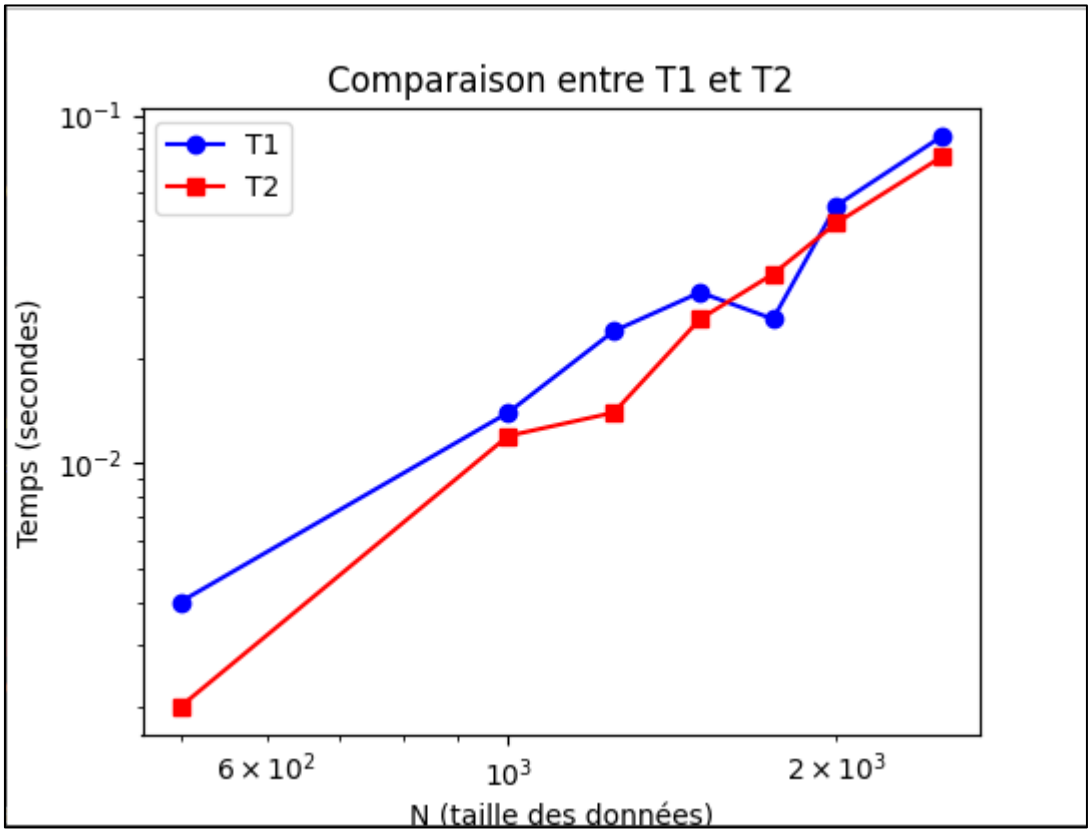
Dans l'algorithme 1 : On va tester pour chaque case matrice B si elle existe dans matrice A
Dans l'algorithme 2 : On ne teste que la première valeur de chaque ligne de la matrice B dans la matrice A
Si on trouve, alors on rentrer dans la dernière boucle pour parcourir toute la matrice B sinon on pace a la ligne suivante

Complexité temporelle :

$$O((n - n_{prime}) * (m - m_{prime}) * n_{prime} * m_{prime})$$

3- Mesurer les temps d'exécution en faisant varier n, m puis n', m' et représenter les résultats sous forme d'un tableau pour les fonctions subMat1 et subMat2.

Taille de la matrice	500	1000	1250	1500	1750	2000	2500
T1	0.004	0.014	0.024	0.031	0.026	0.055	0.087
T2	0.002	0.012	0.014	0.026	0.035	0.049	0.076



Le graph montre bien que l'algorithme 2(T2) est une optimisation de l'algorithme 1(T1), en diminuant le nombre d'itération que l'algorithme fait dans la recherche des éléments de la petite matrice

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);
}
```

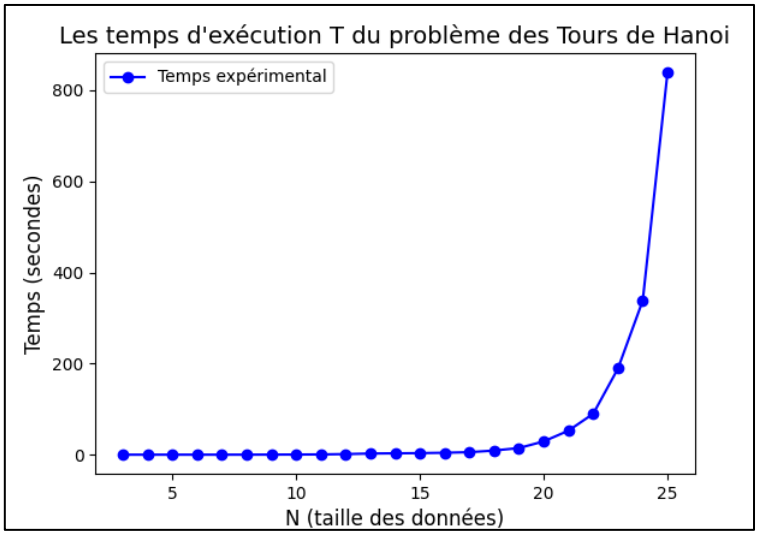
4) Mesurer les temps d'exécution T (en secondes) pour un échantillon de données de variable n et représenter les résultats sous forme de tableau :

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

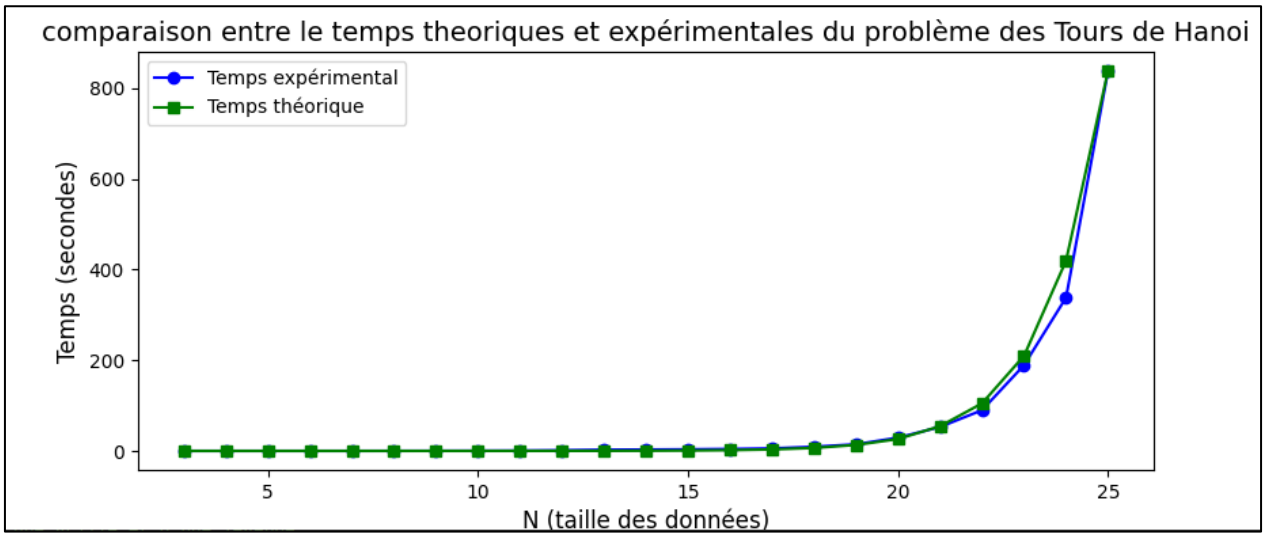
12	13	14	15	16	17	18	19	20	21
1.312	2.546	3.015	3.505	4.252	5.520	9.116	14.582	29.135	52.756
0.10	0.20	0.41	0.81	1.63	3.27	6.55	13.10	26.21	54.42

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.

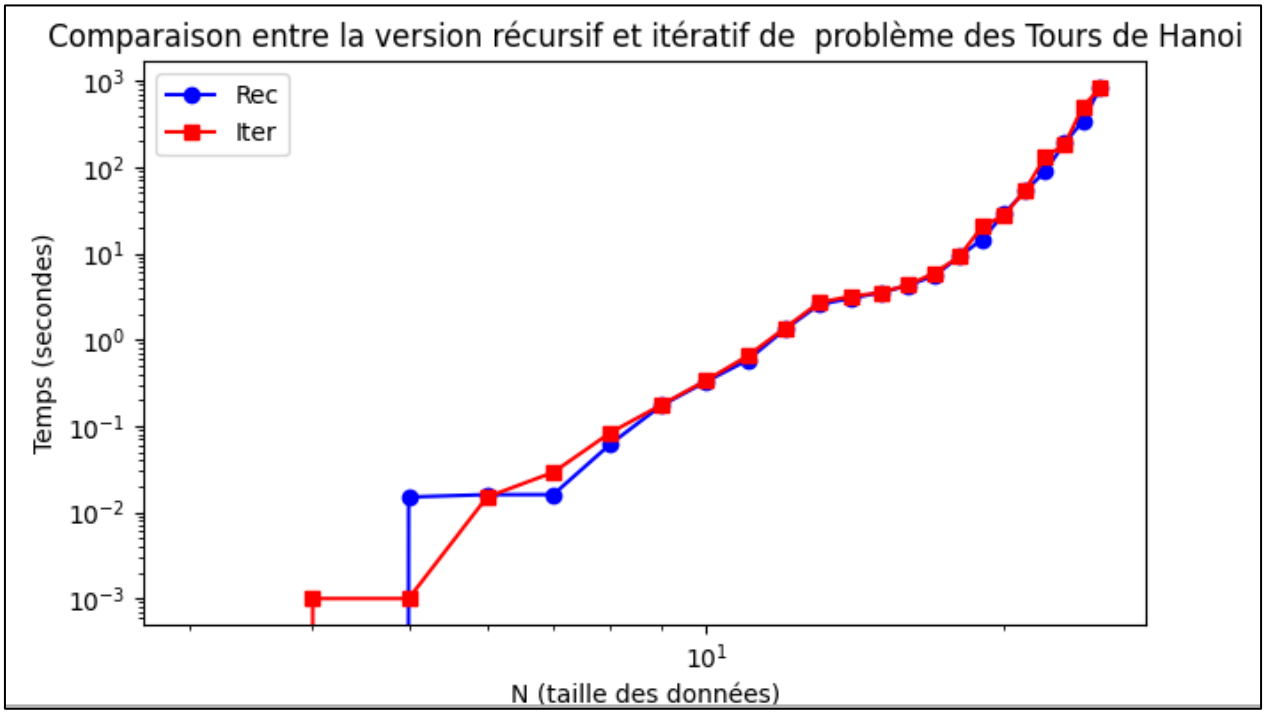
```
void moveDisk(int d, int ar) {
    printf("Deplacer le disque de %d a %d\n", d, ar);
}
void hanoiIterative(int n) {
    int total = pow(2, n) - 1;
    int tours[3];
 for (int i = 0; i < 3; i++) {
        tours[i] = i + 1;
    }
 if (n % 2 == 0) {
        int temp = tours[1];
        tours[1] = tours[2];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

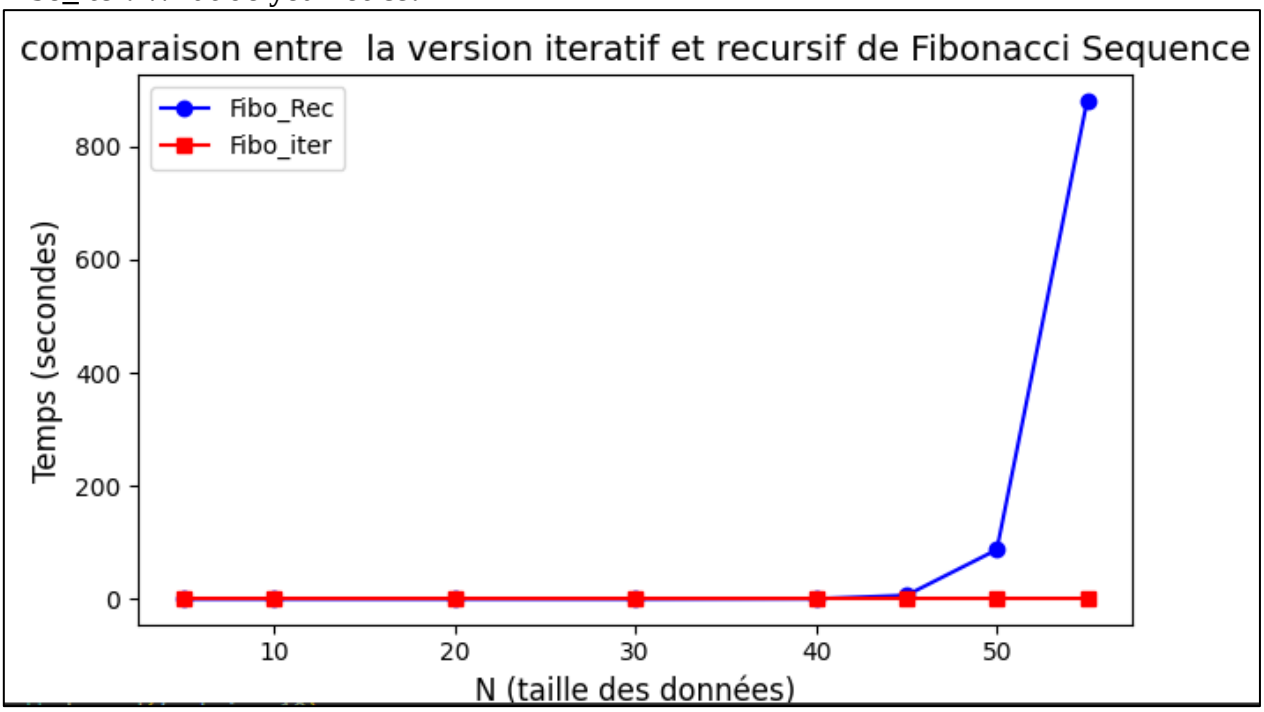
Fonction Fibo_Rec

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.