

UNIVERSITÉ DES SCIENCES ET DE LA TECHNOLOGIE  
HOUARI BOUMEDIENE



CONCEPTION ET COMPLEXITÉ DES ALGORITHMES

---

**Rapport de Travaux Pratiques N°2**  
**Algorithmes de Complexités temporelles**  
**linéaire  $O(n)$  et racine carrée  $O(\sqrt{n})$ .**

---

*Binôme*

MOHAMMEDI HAROUNE  
HOUACINE NAILA AZIZA

*Professeur*

Pr. AMANI FERHAT

30 octobre 2017

# 1 Partie I : Algorithme 1 du test de la primalité.

## 1 Développement de l'algorithme qui permet de déterminer si un nombre entier naturel $n$ est premier ( $n \geq 2$ ).

Afin de vérifier si un nombre entier naturel " $n$ " est premier ou pas nous allons tester s'il est divisible par un autre nombre entier naturel appartenant à l'intervalle  $[2 - n]$ . Pour cela nous allons utiliser la fonction modulo qui donne le reste de la division de  $n$  par  $i$ ,  $i$  variant de 1 jusqu'à  $n$ .

```
Algorithme_nombre_premier1

VAR
i,N : entier;
prem : boolean;

DEBUT

    écrire("donner la valeur de N = ");
    lire(N);

    i=2;
    prem = vrai;

    tant que( i <= N-1 ET prem == vrai){

        si( N mod i == 0)
            alors
                prem = faux;
            sinon
                i = i + 1;
        }

    si(prem == 1)
        alors
            écrire("Le nombre saisi :",N,"est premier!");
        sinon
            écrire("Le nombre saisi :",N,"n'est pas premier! ");
    FIN.
```

## 2 Complexité :

### 2.1 Calcule des complexités temporelles en notation exacte et/ou en notation asymptotique de Landau $O$ (Grand $O$ ) de cet algorithme au meilleur cas, notée $f1(n)$ , et au pire cas, notée $f2(n)$ .

Le calcul de la complexité exacte de cet algorithme n'est pas possible car nous ne pouvons pas déterminer une forme générale représentant les nombres premiers.

1. Calcule de la complexité au meilleur cas :

Il s'agit du cas où le nombre est égal à 2 donc la boucle n'est exécutée aucune fois, tel que :

$f1(n) = 1(=) + 1(=) + 4(<=, -, ==, \text{et})$

$f1(n) = 6 \text{ (Opérations)} \Rightarrow f1(n) = O(1)$

2. Calcule de la complexité au pire cas :

Il s'agit du cas où le nombre est premier c'est à dire le contenu de la boucle est exécuté  $(n-1)-2 + 1$  fois =  $n-2$  fois. (selon la règle : fin-debut+1)

ce qui donne :

$f2(n) = 1(=) + 1(=) + 4(<=, -, ==, \text{et})(n-1) + [2(\text{mod}, ==) + 2(+, =)](n-2)$

$f2(n) = 8n - 10 \text{ (Opérations)} \Rightarrow f2(n) = O(n)$

## 2.2 Calculer la complexité spatiale en notation exacte et/ou en notation asymptotique de Landau $O$ (Grand $O$ ) de cet algorithme notée $s(n)$ .

## 3 Développement de programme correspondant avec le langage C.

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

int main()
{
    int i, N, prem;

    printf("donner la valeur de N = ");
    scanf("%d", &N);

    i=2;
    prem = 1;

    while( i <= N-1 && prem == 1){

        if( N%i == 0)
            prem = 0;
        else
            i = i + 1;
    }

    if(prem == 1)
    {
        printf("Le nombre saisi : %d est premier! \n", N);
    }
    else{
        printf("Le nombre saisi : %d n'est pas premier! \n", N);
    }
}
```

```
return 0;
}
```

#### 4 Vérification par programme si les nombres n donnés dans le tableau de l'énoncé (1.000.003, 2.000.003, ...) sont premiers.

Pour répondre à cette question, notre programme doit contenir une tableau `tab[ ]` des valeurs à tester, ainsi le programme de teste de primalité devra être exécuté pour chaque élément de ce tableau.

##### 4.1 Programme C à exécuter :

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

int main()
{
    long int i,j,prem;

    long int tab[]={1000003, 2000003, 4000037, 8000009, 16000057,
        ↪ 32000011, 64000031, 128000003, 256000001, 512000009,
        ↪ 1024000009, 2048000011};

    for(j=0 ; j<12 ; j++)
    {
        i=2;
        prem = 1;

        while( i <= tab[j]-1 && prem == 1){

            if( tab[j]%i == 0)
                prem = 0;
            else
                i = i + 1;
        }

        if(prem == 1)
        {
            printf("Le nombre saisié : %ld est premier! \n",tab[j]);
        }
        else{
            printf("Le nombre saisié : %ld n'est pas premier! \n",tab[j]);
        }
    }
}
```

```
return 0;
}
```

## 4.2 Résultats de l'exécution du programme :

```
Le nombre saisié : 1000003 est premier!
Le nombre saisié : 2000003 est premier!
Le nombre saisié : 4000037 est premier!
Le nombre saisié : 8000009 est premier!
Le nombre saisié : 16000057 est premier!
Le nombre saisié : 32000011 est premier!
Le nombre saisié : 64000031 est premier!
Le nombre saisié : 128000003 est premier!
Le nombre saisié : 256000001 est premier!
Le nombre saisié : 512000009 est premier!
Le nombre saisié : 1024000009 est premier!
Le nombre saisié : 2048000011 est premier!
```

On remarque que tous les nombres donnés sont premiers!

## 5 Mesure des temps d'exécution $T$ pour les nombres $n$ donnés.

Pour mesurer le temps d'exécution du programme nous utilisons les fonctions de gestion du temps qui sont fournies dans la bibliothèque "time.h" .

### 5.1 Programme C correspondant au calcul du temps d'exécution pour chaque valeur du tableau :

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

int main()
{
    long int i,j,prem;
    clock_t deb,fin;
    double total;

    long int tab[]={1000003, 2000003, 4000037, 8000009, 16000057,
        ↪ 32000011, 64000031, 128000003, 256000001, 512000009,
        ↪ 1024000009, 2048000011};

    for(j=0 ; j<12 ; j++)
    {
        deb = clock();

        i=2;
        prem = 1;
```

```

while( i <= tab[j]-1 && prem == 1){

    if( tab[j]%i == 0)
        prem = 0;
    else
        i = i + 1;
}

fin = clock();

if(prem == 1)
{
    printf("Le nombre saisié : %ld est premier! \n",tab[j]);
}
else{
    printf("Le nombre saisié : %ld n'est pas premier! \n",tab[j]);
}

total = (double) (fin - deb)/CLOCKS_PER_SEC;
printf("temps d'exécution = %lf \n",total);
}
return 0;
}

```

## 5.2 Résultat de l'exécution du programme :

```

Le nombre saisié : 1000003 est premier!
temps d'exécution = 0.009000
Le nombre saisié : 2000003 est premier!
temps d'exécution = 0.009000
Le nombre saisié : 4000037 est premier!
temps d'exécution = 0.013000
Le nombre saisié : 8000009 est premier!
temps d'exécution = 0.026000
Le nombre saisié : 16000057 est premier!
temps d'exécution = 0.051000
Le nombre saisié : 32000011 est premier!
temps d'exécution = 0.102000
Le nombre saisié : 64000031 est premier!
temps d'exécution = 0.205000
Le nombre saisié : 128000003 est premier!
temps d'exécution = 0.411000
Le nombre saisié : 256000001 est premier!
temps d'exécution = 0.820000
Le nombre saisié : 512000009 est premier!
temps d'exécution = 1.642000
Le nombre saisié : 1024000009 est premier!
temps d'exécution = 3.289000

```

```
Le nombre saisi : 2048000011 est premier!  
temps d'exécution = 6.576000
```

### 5.3 Remplissage du tableau :

Valeur N :	1000003	2000003	4000037	8000009	16000057	32000011
Temps d'exe :	0.009000	0.009000	0.013000	0.026000	0.051000	0.102000
Valeur N :	64000031	128000003	256000001	512000009	1024000009	2048000011
Temps d'exe :	0.205000	0.411000	0.820000	1.642000	3.289000	6.576000

## 6 Développement du programme de mesure du temps d'exécution du programme qui a en entrée les données de l'échantillon dans tab1 et en sortie les temps d'exécution dans tab2.

```
#include <stdlib.h>  
#include <stdio.h>  
#include <time.h>  
  
int main()  
{  
    long int i,j,prem;  
    clock_t deb,fin;  
    double total;  
  
    long int tab1[]={1000003, 2000003, 4000037, 8000009,  
        ↪ 16000057, 32000011, 64000031,  
    128000003, 256000001, 512000009, 1024000009, 2048000011};  
  
    double tab2[12];  
  
    for(j=0 ; j<12 ; j++)  
    {  
        deb = clock();  
  
        i=2;  
        prem = 1;  
  
        while( i <= tab1[j]-1 && prem == 1){  
  
            if( tab1[j]%i == 0)  
                prem = 0;  
            else  
                i = i + 1;  
        }  
    }
```

```
    fin = clock();

    total = (double) (fin - deb)/CLOCKS_PER_SEC;

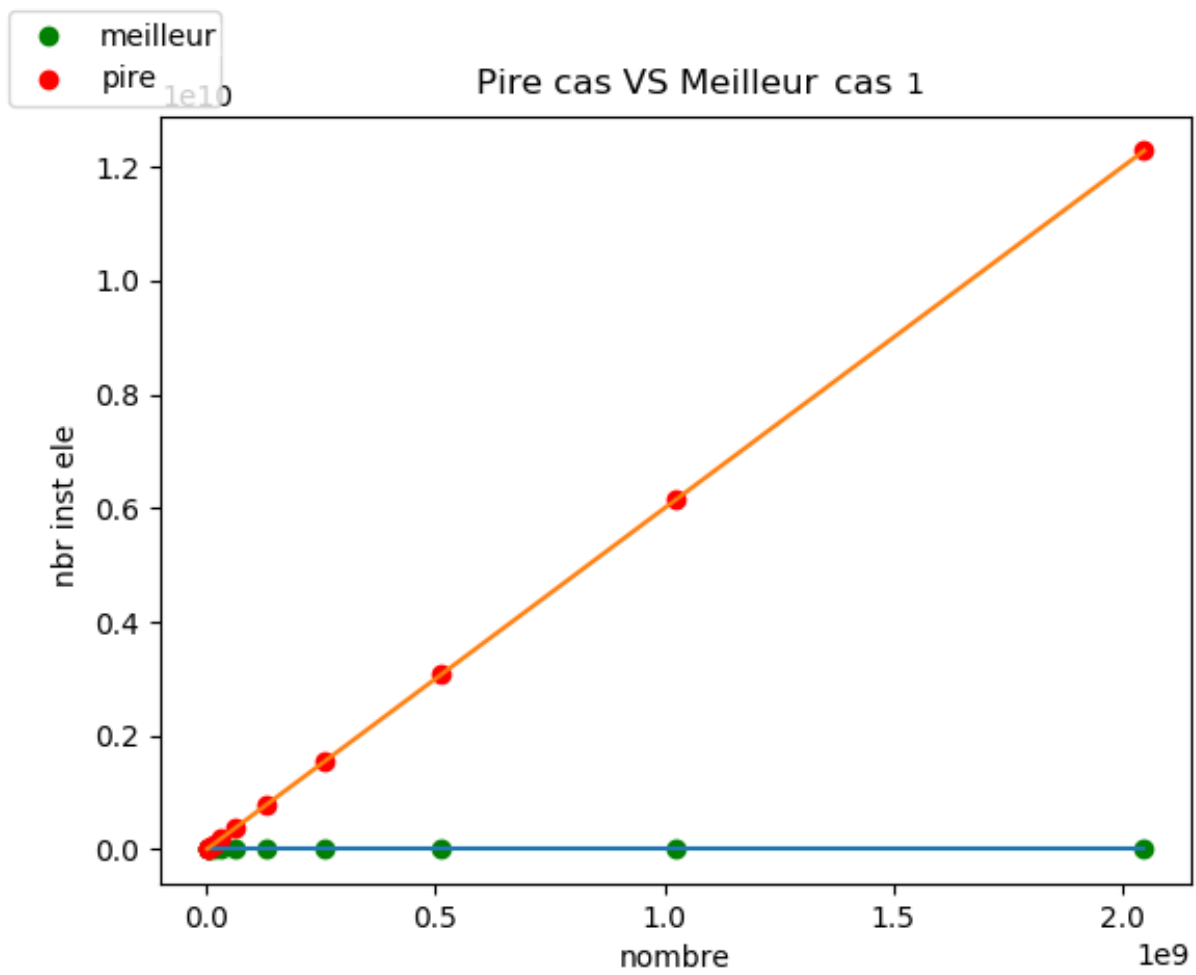
    tab2[j]= total ;
    printf("%lf, ", tab2[j]);
}
return 0;
}
```

**7 Représentation par un graphe,  $Gf1(n)$  et  $Gf2(n)$ , les variations de la fonction de la complexité temporelle correspondant au meilleur cas  $f1(n)$  et au pire cas  $f2(n)$  en fonction de  $n$  respectivement ; et par un autre graphe, noté  $GT(n)$ , les variations du temps d'exécution  $T(n)$  en fonction de  $n$ .**

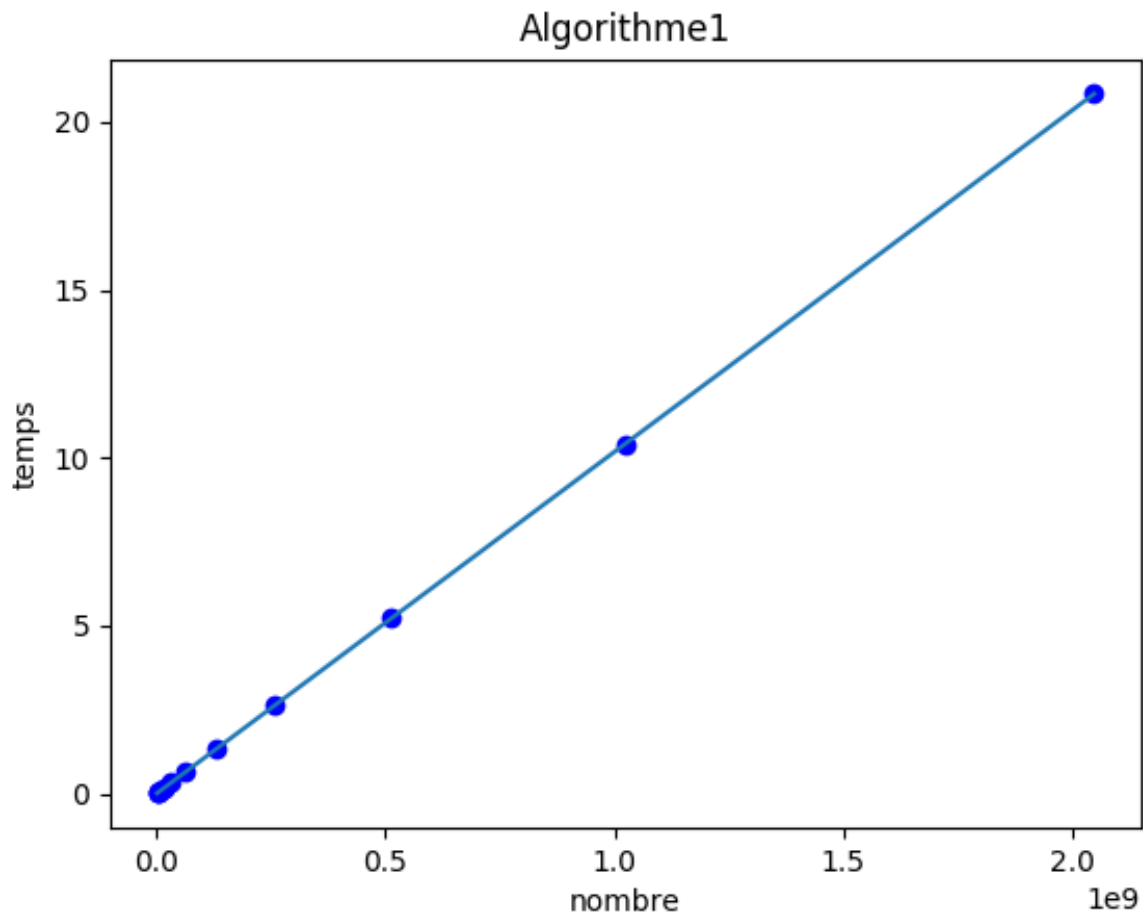
**7.1 Représentation des deux graphes  $Gf1$  et  $Gf2$  du meilleur et pire cas respectivement :**

Sachant que pour le meilleur cas c'est une constante et pour le pire cas le graphe aura une forme linéaire vu sa fonction.





## 7.2 Représentation du graphe GT de la variation du temps d'exécution selon les données de teste



## 8 Interprétation des résultats.

### 8.1 Comparaison des mesure de temps d'exécution avec le pire et meilleur cas :

- 1.
2. Remarque :

### 8.2 Remarque est déduction d'une fonction $T(n)$ reliant $n$ au temps d'exécution.

On remarque que les temps d'exécution sont approximativement doublés lorsque  $N$  est doublé.

Exemples :

$$N1 = 4000037 \Rightarrow T1 = 0.013000$$

$$N2 = 8000009 \approx 2 * N1 \Rightarrow T2 = 0.026000 \approx 2 * T1$$

Aussi

$$N1 = 128000003 \Rightarrow T1 = 0.411000$$

$$N2 = 256000001 \approx 2 * N1 \Rightarrow T2 = 0.820000 \approx 2 * T1$$

On peut constater la linéarité du graphe.

On en déduit que le temps d'exécution est proportionnel à N, ce que l'on peut représenter par la formule suivante :

$$T(x * N) = x * T(N) \text{ pour tous } x * N \in [1000003 - 2048000011]$$

(x étant la tangente d'un point sur le graphe).

Nous ne pouvant pas généraliser car les testes que nous avons fait n'englobent pas toutes les valeurs possibles,

### 8.3 Comparaison de la complexité théorique et expérimentale.

1. Complexité théorique :

2. Complexité expérimentale :

Nous l'avons déjà calculé dans le tableau 5.3

3. Nous constatons que :

## 2 Partie II : Algorithme 2 du test de la primalité.

### 9 Développement de l'algorithme qui permet de déterminer si un nombre entier naturel n est premier (n ≥ 2).

Afin de vérifier si un nombre entier naturel "n" est premier ou pas nous allons tester s'il est divisible par un autre nombre entier naturel appartenant à l'intervalle  $[2 - n/2]$ . Pour cela nous allons utiliser la fonction modulo qui donne le reste de la division de n par i, i variant de 1 jusqu'à n/2.

```
Algorithme_nombre_premier2

VAR
i,N : entier;
prem : boolean;

DEBUT

    écrire("donner la valeur de N = ");
    lire(N);

    i = 2;
    prem = vrai;
```

```

    tant que( i <= (N div 2) ET prem == vrai){

        si( N mod i == 0)
            alors
                prem = faux;
            sinon
                i = i + 1;
    }

    si(prem == 1)
        alors
            écrire("Le nombre saisi :",N,"est premier!");
        sinon
            écrire("Le nombre saisi :",N,"n'est pas premier! ");
FIN.

```

## 10 Complexité :

### 10.1 Calcule des complexités temporelles en notation exacte et/ou en notation asymptotique de Landau O (Grand O) de cet algorithme au meilleur cas, notée $f1(n)$ , et au pire cas, notée $f2(n)$ .

Le calcul de la complexité exacte de cet algorithme n'est pas possible car nous ne pouvons pas déterminer une forme générale représentant les nombres premiers.

1. Calcul de la complexité au meilleur cas :

Il s'agit du cas où le nombre est égal à 2 donc la boucle n'est exécutée aucune fois, tel que :

$f1(n) = 1(=) + 1(=) + 4(<=, \text{div}, ==, \text{et})$

$f1(n) = 6 \text{ (Opérations)} \Rightarrow f1(n) = O(1)$

2. Calcul de la complexité au pire cas :

Il s'agit du cas où le nombre est premier c'est à dire le contenu de la boucle est exécuté  $\lfloor \frac{n}{2} - 2 + 1 \text{ fois} = \lfloor \frac{n}{2} - 1 \text{ fois. (selon la règle : fin-debut} + 1)$

ce qui donne :

$f2(n) = 1(=) + 1(=) + 4(<=, \text{div}, ==, \text{et})(\lfloor \frac{n}{2} \rfloor) + [2(\text{mod}, ==) + 2(+, =)](\lfloor \frac{n}{2} \rfloor - 1)$

$f2(n) = 8\lfloor \frac{n}{2} \rfloor - 2 \text{ (Opérations)} \Rightarrow f2(n) = O(\lfloor \frac{n}{2} \rfloor) = O(n)$

### 10.2 Calculer la complexité spatiale en notation exacte et/ou en notation asymptotique de Landau O (Grand O) de cet algorithme notée $s(n)$ .

## 11 Développement de programme correspondant avec le langage C.

```

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

int main()
{

```

```

int i,N,prem;

printf("donner la valeur de N = ");
scanf("%d",&N);

i=2;
prem = 1;

while( i <= N/2 && prem == 1){

    if( N%i == 0)
        prem = 0;
    else
        i = i + 1;
}

if(prem == 1)
{
    printf("Le nombre saisi : %d est premier! \n",N);
}
else{
    printf("Le nombre saisi : %d n'est pas premier! \n",N);
}

return 0;
}

```

## 12 Vérification par programme si les nombres n donnés dans le tableau de l'énoncé (1.000.003, 2.000.003, ...) sont premiers.

Pour répondre à cette question, notre programme doit contenir une tableau `tab[]` des valeurs à tester, ainsi le programme de teste de primalité devra être exécuté pour chaque élément du tableau.

### 12.1 Programme C à exécuter :

```

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

int main()
{
    long int i,j,prem;

    long int tab[]={1000003, 2000003, 4000037, 8000009, 16000057,
        ↪ 32000011, 64000031, 128000003, 256000001, 512000009,
        ↪ 1024000009, 2048000011};

    for(j=0 ; j<12 ; j++)

```

```

{
    i=2;
    prem = 1;

    while( i <= tab[j]/2 && prem == 1){

        if( tab[j]%i == 0)
            prem = 0;
        else
            i = i + 1;
    }

    if(prem == 1)
    {
        printf("Le nombre saisié : %ld est premier! \n",tab[j]);
    }
    else{
        printf("Le nombre saisié : %ld n'est pas premier! \n",tab[j]);
    }
}

return 0;
}

```

## 12.2 Résultats de l'exécution du programme :

```

Le nombre saisié : 1000003 est premier!
Le nombre saisié : 2000003 est premier!
Le nombre saisié : 4000037 est premier!
Le nombre saisié : 8000009 est premier!
Le nombre saisié : 16000057 est premier!
Le nombre saisié : 32000011 est premier!
Le nombre saisié : 64000031 est premier!
Le nombre saisié : 128000003 est premier!
Le nombre saisié : 256000001 est premier!
Le nombre saisié : 512000009 est premier!
Le nombre saisié : 1024000009 est premier!
Le nombre saisié : 2048000011 est premier!

```

On remarque que tous les nombre données sont premiers!

## 13 Mesure des temps d'exécution T pour les nombres n données.

Pour mesurer le temps d'exécution du programme nous utilisons les fonctions de gestion du temps qui sont fournies dans la bibliothèque "time.h" .

### 13.1 Programme C correspondant au calcule du temps d'exécution pour chaque valeur du tableau :

```

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

int main()
{
    long int i,j,prem;
    clock_t deb,fin;
    double total;

    long int tab[]={1000003, 2000003, 4000037, 8000009, 16000057,
        ↪ 32000011, 64000031, 128000003, 256000001, 512000009,
        ↪ 1024000009, 2048000011};

    for(j=0 ; j<12 ; j++)
    {
        deb = clock();

        i=2;
        prem = 1;

        while( i <= tab[j]/2 && prem == 1){

            if( tab[j]%i == 0)
                prem = 0;
            else
                i = i + 1;
        }

        fin = clock();

        if(prem == 1)
        {
            printf("Le nombre saisié : %ld est premier! \n",tab[j]);
        }
        else{
            printf("Le nombre saisié : %ld n'est pas premier! \n",tab[j]);
        }

        total = (double) (fin - deb)/CLOCKS_PER_SEC;
        printf("temps d'exécution = %lf \n",total);
    }
    return 0;
}

```

### 13.2 Résultat de l'exécution du programme :

```
Le nombre saisié : 1000003 est premier!
```

```

temps d'exécution = 0.004000
Le nombre saisi : 2000003 est premier!
temps d'exécution = 0.007000
Le nombre saisi : 4000037 est premier!
temps d'exécution = 0.009000
Le nombre saisi : 8000009 est premier!
temps d'exécution = 0.016000
Le nombre saisi : 16000057 est premier!
temps d'exécution = 0.026000
Le nombre saisi : 32000011 est premier!
temps d'exécution = 0.051000
Le nombre saisi : 64000031 est premier!
temps d'exécution = 0.102000
Le nombre saisi : 128000003 est premier!
temps d'exécution = 0.205000
Le nombre saisi : 256000001 est premier!
temps d'exécution = 0.415000
Le nombre saisi : 512000009 est premier!
temps d'exécution = 0.821000
Le nombre saisi : 1024000009 est premier!
temps d'exécution = 1.647000
Le nombre saisi : 2048000011 est premier!
temps d'exécution = 3.284000

```

### 13.3 Remplissage du tableau :

Valeur N :	1000003	2000003	4000037	8000009	16000057	32000011
Temps d'exe :	0.004000	0.007000	0.009000	0.016000	0.026000	0.051000
Valeur N :	64000031	128000003	256000001	512000009	1024000009	2048000011
Temps d'exe :	0.102000	0.205000	0.415000	0.821000	1.647000	3.284000

## 14 Développement du programme de mesure du temps d'exécution du programme qui a en entrée les données de l'échantillon dans tab1 et en sortie les temps d'exécution dans tab2.

```

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

int main()
{
    long int i,j,prem;
    clock_t deb,fin;
    double total;

```



```

long int tab1[]={1000003, 2000003, 4000037, 8000009,
    ↪ 16000057, 32000011, 64000031,
128000003, 256000001, 512000009, 1024000009, 2048000011};

double tab2[12];

for(j=0 ; j<12 ; j++)
{
    deb = clock();

    i=2;
    prem = 1;

    while( i <= tab1[j]/2 && prem == 1){

        if( tab1[j]%i == 0)
            prem = 0;
        else
            i = i + 1;
    }
    fin = clock();

    total = (double) (fin - deb)/CLOCKS_PER_SEC;

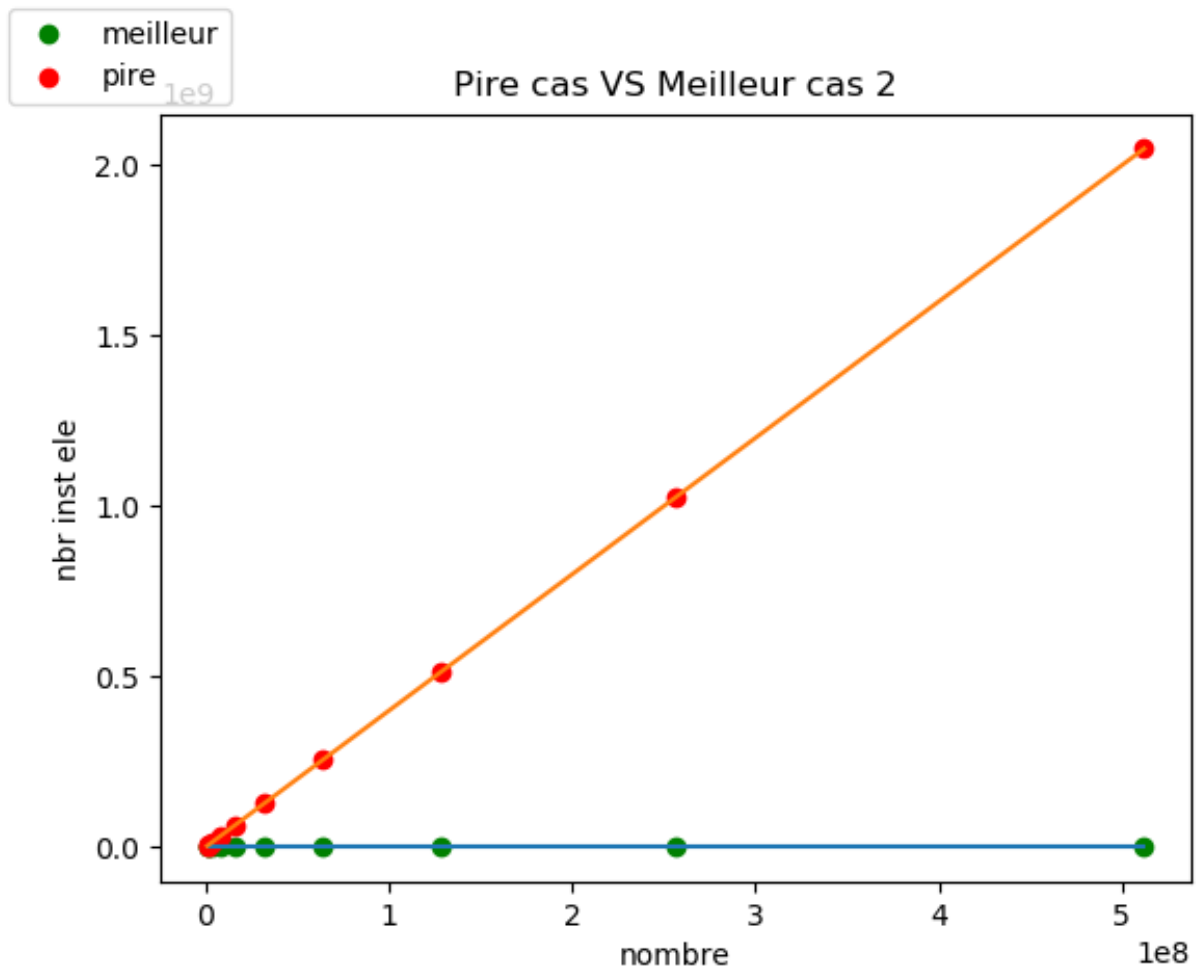
    tab2[j]= total ;
    printf("%lf, ", tab2[j]);
}
return 0;
}

```

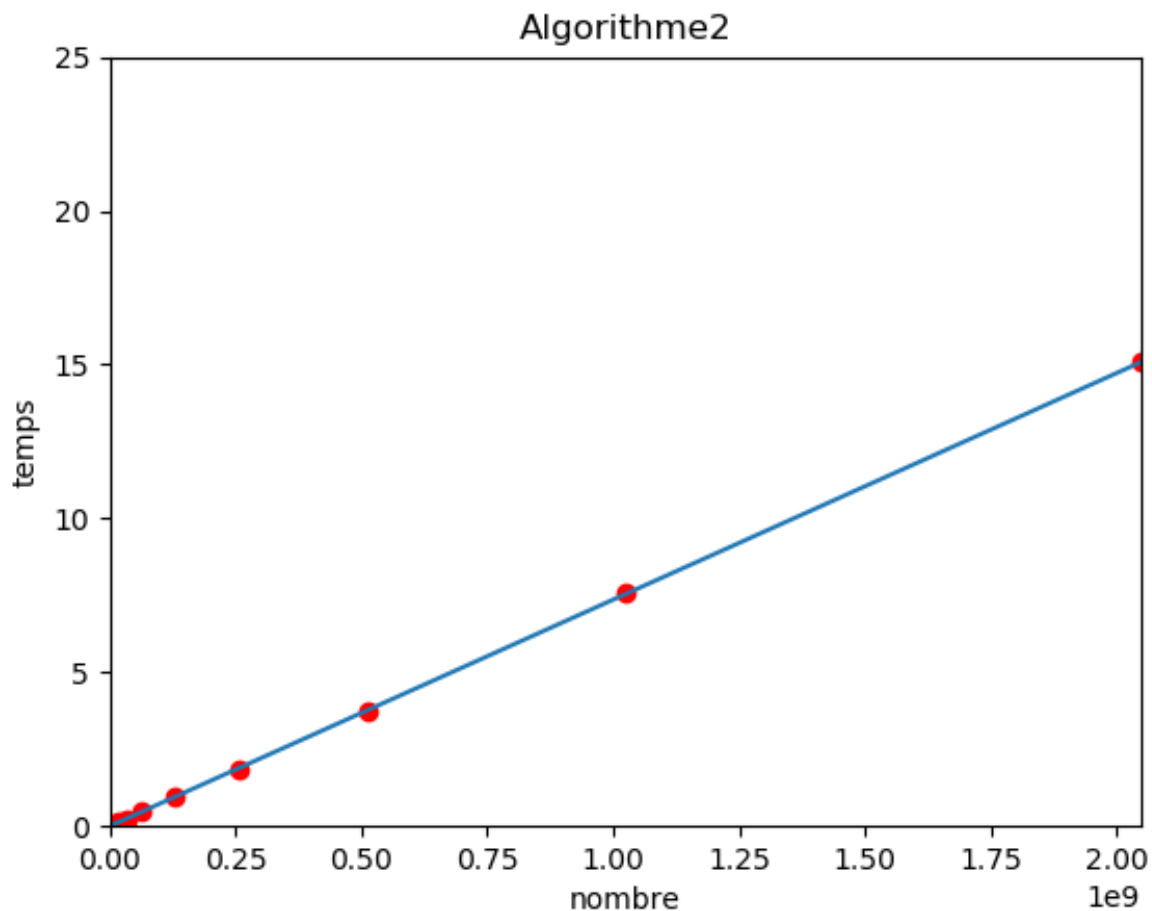
**15 Représentation par un graphe,  $Gf1(n)$  et  $Gf2(n)$ , les variations de la fonction de la complexité temporelle correspondant au meilleur cas  $f1(n)$  et au pire cas  $f2(n)$  en fonction de  $n$  respectivement; et par un autre graphe, noté  $GT(n)$ , les variations du temps d'exécution  $T(n)$  en fonction de  $n$ .**

**15.1 Représentation des deux graphes  $Gf1$  et  $Gf2$  du meilleur et pire cas respectivement :**

Sachant que pour le meilleur cas c'est une constante et pour le pire cas le graphe aura une forme linéaire vu sa fonction.



## 15.2 Représentation du graphe GT de la variation du temps d'exécution selon les données de teste



## 16 Interprétation des résultats.

### 16.1 Comparaison des mesure de temps d'exécution avec le pire et meilleur cas :

1. Le graphe :

2. Remarque :

On constate que le graphe obtenu à partir de la mesure des temps d'exécution de l'échantillon tant à ressembler au graphe du pire cas.

Donc le graphe GT correspond au graphe du pire cas car tous les nombres de l'échantillon sont Premiers.

### 16.2 Remarque est déduction d'une fonction $T(n)$ reliant $n$ au temps d'exécution.

On remarque que les temps d'exécution sont approximativement doublés lorsque  $N$  est doublé.

Exemples :

$$N1 = 4000037 \Rightarrow T1 = 0.009000$$

$$N2 = 8000009 \approx 2 * N1 \Rightarrow T2 = 0.016000 \approx 2 * T1$$

Aussi

$$N1 = 128000003 \Rightarrow T1 = 0.205000$$

$$N2 = 256000001 \approx 2 * N1 \Rightarrow T2 = 0.415000 \approx 2 * T1$$

On peut constater la linéarité du graphe.

On en déduit que le temps d'exécution est proportionnel à N, ce que l'on peut représenter par la formule suivante :

$$T(x * N) = x * T(N) \text{ pour tous } x * N \in [1000003 - 2048000011]$$

(x étant la tangente d'un point sur le graphe).

Nous ne pouvant pas généraliser car les testes que nous avons fait n'englobent pas toutes les valeurs possibles,

### 16.3 Comparaison de la complexité théorique et expérimentale.

1. Complexité théorique :

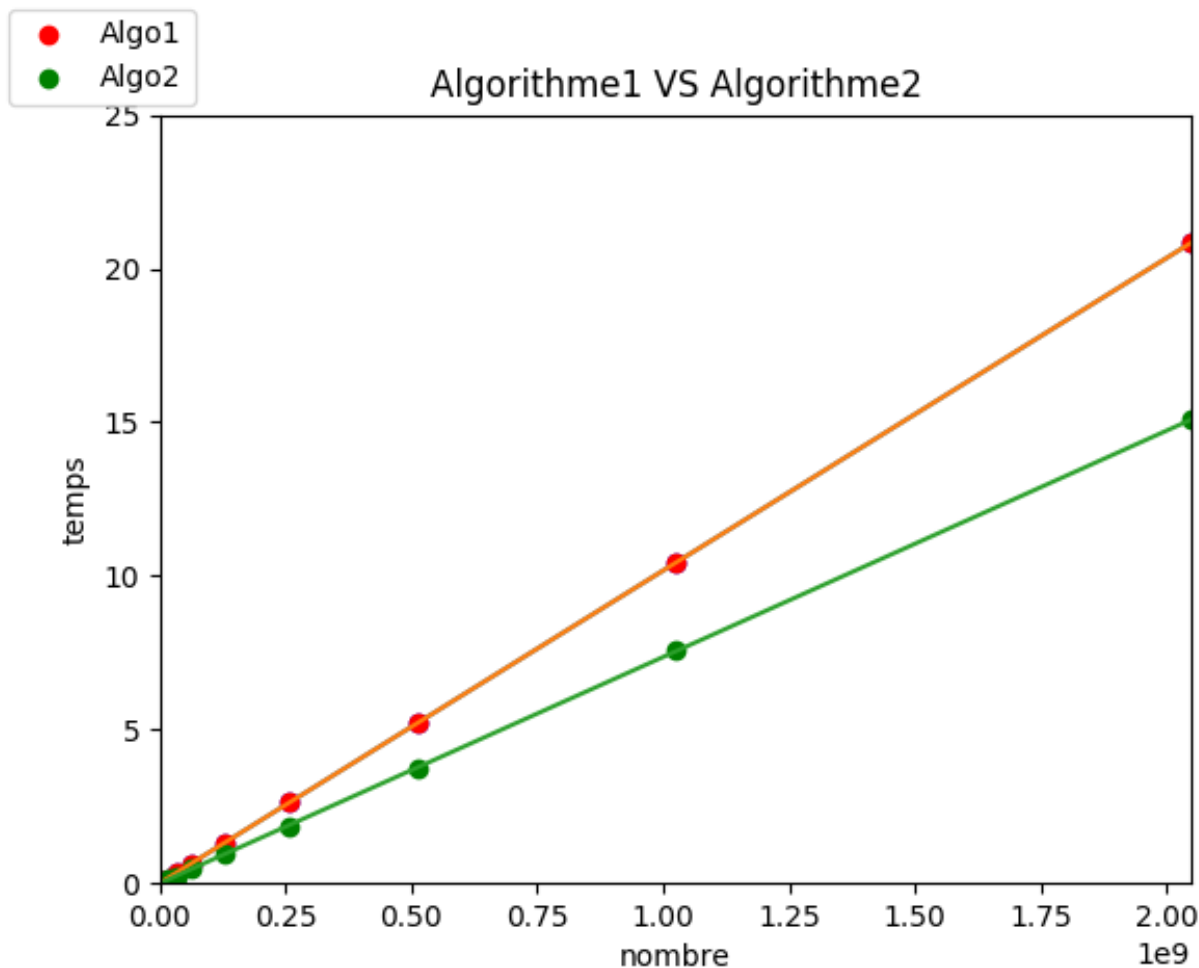
2. Complexité expérimentale :

Nous l'avons déjà calculé dans le tableau 5.3

3. Nous constatons que :

## 17 Comparaison des deux algorithmes précédents.

### 17.1 Représentation des deux graphes :



### 17.2 Choix de l'algorithme le plus performant :

Il est clairement visible à travers la représentation des deux graphes ci-dessus que l'algorithme 2 ( $\frac{n}{2}$ ) est plus rapide (temps d'exécution) que l'algorithme 1 ( $n$ ).

Donc en terme de rapidité nous choisissons l'Algorithme 2.

## 3 Partie III : Algorithme 3 du test de la primalité.

### 18 Développement de l'algorithme qui permet de déterminer si un nombre entier naturel $n$ est premier ( $n \geq 2$ ).

Afin de vérifier si un nombre entier naturel " $n$ " est premier ou pas il nous allons tester s'il est divisible par un autre nombre entier naturel appartenant à l'intervalle  $[2 - \sqrt{(n)}]$ . Pour cela nous allons utiliser la fonction modulo qui donne le reste de la division de  $n$  par  $i$ ,  $i$  variant de 1 jusqu'à  $n$ .

```

Algorithme_nombre_premier3

VAR
i,N : entier;
prem : boolean;

DEBUT

    écrire("donner la valeur de N = ");
    lire(N);

    i=2;
    prem = vrai;

    tant que( i <= sqrt(N) ET prem == vrai){

        si( N mod i == 0)
            alors
                prem = faux;
            sinon
                i = i + 1;
    }

    si(prem == 1)
        alors
            écrire("Le nombre saisi :",N,"est premier!");
        sinon
            écrire("Le nombre saisi :",N,"n'est pas premier! ");
FIN.

```

## 19 Complexité :

### 19.1 Calcule des complexités temporelles en notation exacte et/ou en notation asymptotique de Landau O (Grand O) de cet algorithme au meilleur cas, notée $f_1(n)$ , et au pire cas, notée $f_2(n)$ .

Le calcul de la complexité exacte de cet algorithme n'est pas possible car nous ne pouvons pas déterminer une forme générale représentant les nombres premiers.

1. Calcul de la complexité au meilleur cas :  
 Il s'agit du cas où le nombre est égal à 2 donc la boucle n'est exécutée aucune fois, tel que :  
 $f_1(n) = 1(=) + 1(=) + 4(<=, \text{sqrt}, ==, \text{et})$   
 $f_1(n) = 6 \text{ (Opérations)} \Rightarrow f_1(n) = O(1)$
2. Calcul de la complexité au pire cas :  
 Il s'agit du cas où le nombre est premier c'est à dire le contenu de la boucle est exécuté  $\lfloor \sqrt{n} - 2 + 1 \text{ fois} = \lfloor \sqrt{n} - 1 \text{ fois. (selon la règle : fin-debut} + 1)$   
 ce qui donne :

$$f_2(n) = 1(=) + 1(=) + 4(<=, \text{sqrt}, ==, \text{et})(\lfloor \sqrt{n} \rfloor) + [2(\text{mod}, ==) + 2(+, =)](\lfloor \sqrt{n} - 1 \rfloor)$$

$$f_2(n) = 8\lfloor \sqrt{n} \rfloor - 2 \text{ (Opérations)} \Rightarrow f_2(n) = O(\lfloor \sqrt{n} \rfloor) = O(\sqrt{n})$$

**19.2** Calculer la complexité spatiale en notation exacte et/ou en notation asymptotique de Landau  $O$  (Grand  $O$ ) de cet algorithme notée  $s(n)$ .

**20** Développement de programme correspondant avec le langage C.

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

int main()
{

    int i,N,prem;

    printf("donner la valeur de N = ");
    scanf("%d",&N);

    i=2;
    prem = 1;

    while( i <= sqrt(N) && prem == 1){

        if( N%i == 0)
            prem = 0;
        else
            i = i + 1;
    }

    if(prem == 1)
    {
        printf("Le nombre saisi : %d est premier! \n",N);
    }
    else{
        printf("Le nombre saisi : %d n'est pas premier! \n",N);
    }

    return 0;
}
```

## 21 Vérification par programme si les nombres n donnés dans le tableau de l'énoncé (1.000.003, 2.000.003, ...) sont premiers.

Pour répondre à cette question, notre programme doit contenir une tableau `tab[]` des valeurs à tester, ainsi le programme de teste de primalité devra être exécuté pour chaque élément du tableau.

### 21.1 Programme C à exécuter :

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

int main()
{
    long int i,j,prem;

    long int tab[]={1000003, 2000003, 4000037, 8000009, 16000057,
        ↪ 32000011, 64000031, 128000003, 256000001, 512000009,
        ↪ 1024000009, 2048000011};

    for(j=0 ; j<12 ; j++)
    {
        i=2;
        prem = 1;

        while( i <= sqrt(tab[j]) && prem == 1){

            if( tab[j]%i == 0)
                prem = 0;
            else
                i = i + 1;
        }

        if(prem == 1)
        {
            printf("Le nombre saisié : %ld est premier! \n",tab[j]);
        }
        else{
            printf("Le nombre saisié : %ld n'est pas premier! \n",tab[j]);
        }
    }

    return 0;
}
```

### 21.2 Résultats de l'exécution du programme :

```
Le nombre saisié : 1000003 est premier!
Le nombre saisié : 2000003 est premier!
```



```
Le nombre saisi : 4000037 est premier!  
Le nombre saisi : 8000009 est premier!  
Le nombre saisi : 16000057 est premier!  
Le nombre saisi : 32000011 est premier!  
Le nombre saisi : 64000031 est premier!  
Le nombre saisi : 128000003 est premier!  
Le nombre saisi : 256000001 est premier!  
Le nombre saisi : 512000009 est premier!  
Le nombre saisi : 1024000009 est premier!  
Le nombre saisi : 2048000011 est premier!
```

On remarque que tous les nombre données sont premiers!

## 22 Mesure des temps d'exécution T pour les nombres n données.

Pour mesurer le temps d'exécution du programme nous utilisons les fonctions de gestion du temps qui sont fournies dans la bibliothèque "time.h" .

### 22.1 Programme C correspondant au calcul du temps d'exécution pour chaque valeur du tableau :

```
#include <stdlib.h>  
#include <stdio.h>  
#include <time.h>  
#include <math.h>  
  
int main()  
{  
    long int i,j,prem;  
    clock_t deb,fin;  
    double total;  
  
    long int tab[]={1000003, 2000003, 4000037, 8000009, 16000057,  
        ↪ 32000011, 64000031, 128000003, 256000001, 512000009,  
        ↪ 1024000009, 2048000011};  
  
    for(j=0 ; j<12 ; j++)  
    {  
        deb = clock();  
        i=2;  
        prem = 1;  
  
        while( i < tab[j] && prem == 1){  
  
            if( tab[j]%i == 0)  
                prem = 0;  
            else  
                i = i + 1;  
        }  
    }
```

```

    fin = clock();

    if(prem == 1)
    {
        printf("Le nombre saisié : %ld est premier! \n",tab[j]);
    }
    else{
        printf("Le nombre saisié : %ld n'est pas premier! \n",tab[j]);
    }

    total = (double) (fin - deb)/CLOCKS_PER_SEC;
    printf("temps d'exécution = %lf \n",total);
}
return 0;
}

```

## 22.2 Résultat de l'exécution du programme :

```

Le nombre saisié : 1000003 est premier!
temps d'exécution = 0.000000
Le nombre saisié : 2000003 est premier!
temps d'exécution = 0.000000
Le nombre saisié : 4000037 est premier!
temps d'exécution = 0.000000
Le nombre saisié : 8000009 est premier!
temps d'exécution = 0.000000
Le nombre saisié : 16000057 est premier!
temps d'exécution = 0.000000
Le nombre saisié : 32000011 est premier!
temps d'exécution = 0.001000
Le nombre saisié : 64000031 est premier!
temps d'exécution = 0.001000
Le nombre saisié : 128000003 est premier!
temps d'exécution = 0.001000
Le nombre saisié : 256000001 est premier!
temps d'exécution = 0.002000
Le nombre saisié : 512000009 est premier!
temps d'exécution = 0.002000
Le nombre saisié : 1024000009 est premier!
temps d'exécution = 0.002000
Le nombre saisié : 2048000011 est premier!
temps d'exécution = 0.003000

```

## 22.3 Remplissage du tableau :

Valeur N :	1000003	2000003	4000037	8000009	16000057	32000011
Temps d'exe :	0.000000	0.000000	0.000000	0.000000	0.000000	0.001000
Valeur N :	64000031	128000003	256000001	512000009	1024000009	2048000011
Temps d'exe :	0.001000	0.001000	0.002000	0.002000	0.002000	0.003000

## 23 Développement du programme de mesure du temps d'exécution du programme qui a en entrée les données de l'échantillon dans tab1 et en sortie les temps d'exécution dans tab2.

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

int main()
{
    long int i,j,prem;
    clock_t deb,fin;
    double total;

    long int tab1[]={1000003, 2000003, 4000037, 8000009,
        ↪ 16000057, 32000011, 64000031,
        128000003, 256000001, 512000009, 1024000009, 2048000011};

    double tab2[12];

    for(j=0 ; j<12 ; j++)
    {
        deb = clock();
        i=2;
        prem = 1;

        while( i < tab1[j] && prem == 1){

            if( tab1[j]%i == 0)
                prem = 0;
            else
                i = i + 1;
        }
        fin = clock();

        total = (double) (fin - deb)/CLOCKS_PER_SEC;

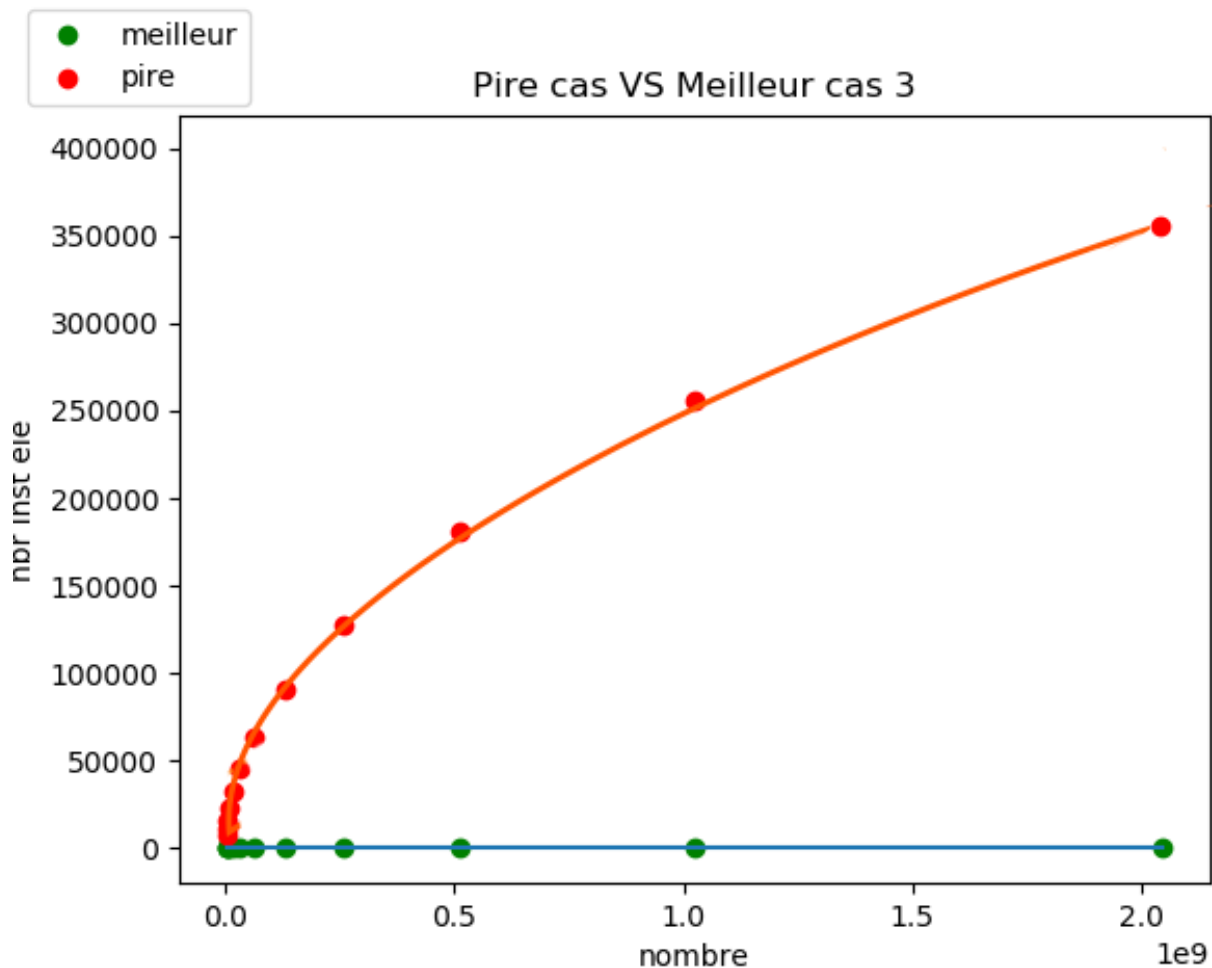
        tab2[j]= total ;
        printf("%lf, ", tab2[j]);
    }
    return 0;
}
```

}

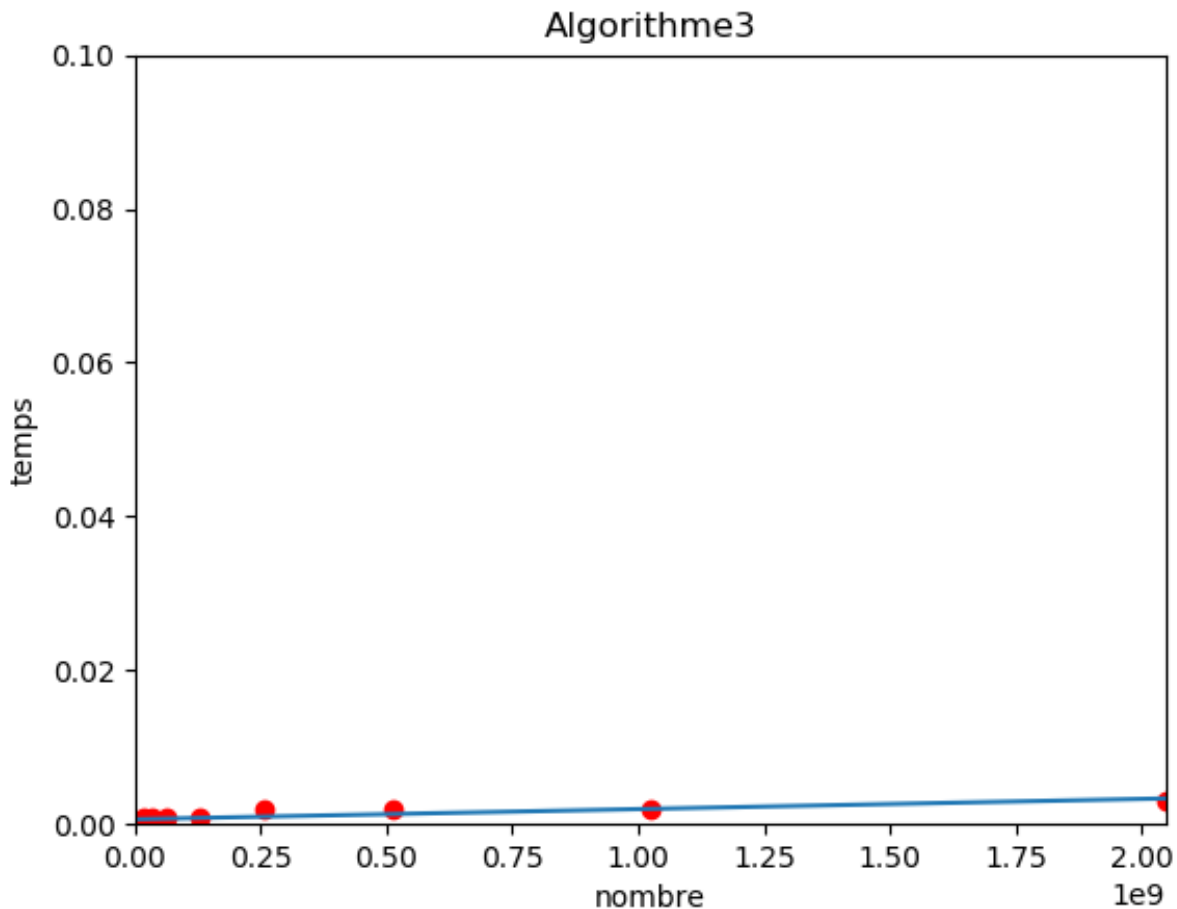
24 Représentation par un graphe,  $Gf1(n)$  et  $Gf2(n)$ , les variations de la fonction de la complexité temporelle correspondant au meilleur cas  $f1(n)$  et au pire cas  $f2(n)$  en fonction de  $n$  respectivement ; et par un autre graphe, noté  $GT(n)$ , les variations du temps d'exécution  $T(n)$  en fonction de  $n$ .

24.1 Représentation des deux graphes  $Gf1$  et  $Gf2$  du meilleur et pire cas respectivement :

Sachant que pour le meilleur cas c'est une constante et pour le pire cas le graphe aura une forme linéaire vu sa fonction.



## 24.2 Représentation du graphe GT de la variation du temps d'exécution selon les données de teste



## 25 Interprétation des résultats.

### 25.1 Comparaison des mesure de temps d'exécution avec le pire et meilleur cas :

1. Le graphe :
2. Remarque :  
On constate que le graphe obtenu à partir de la mesure des temps d'exécution de l'échantillon tant à ressembler au graphe du pire cas.  
Donc le graphe GT correspond au graphe du pire cas car tous les nombres de l'échantillon sont Premiers.

### 25.2 Remarque est déduction d'une fonction $T(n)$ reliant $n$ au temps d'exécution.

### 25.3 Comparaison de la complexité théorique et expérimentale.

1. Complexité théorique :

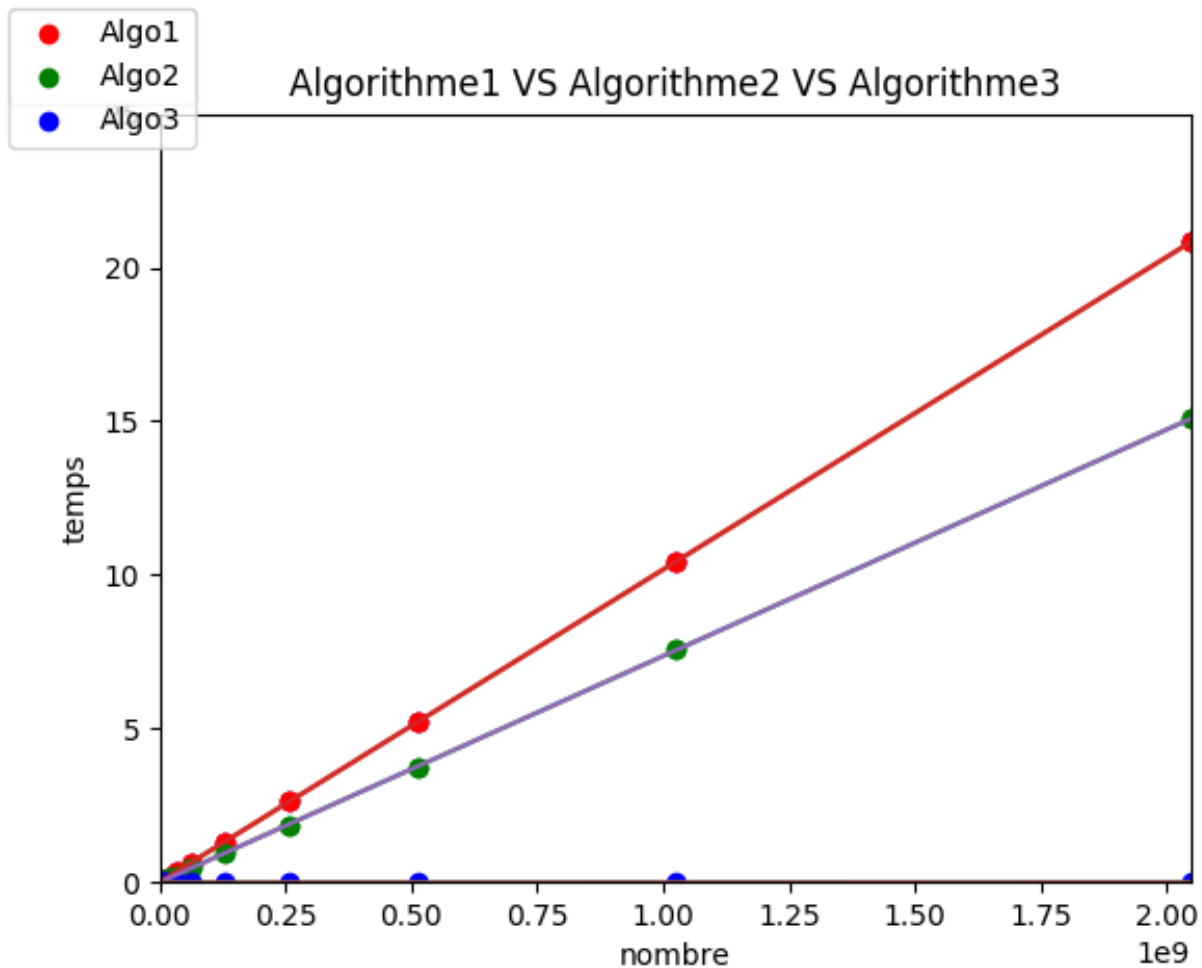
## 2. Complexité expérimentale :

Nous l'avons déjà calculé dans le tableau 5.3

## 3. Nous constatons que :

# 26 Comparaison des trois algorithmes précédents.

## 26.1 Représentation des trois graphes :



## 26.2 Choix de l'algorithme le plus performant :

Il est clairement visible à travers la représentation des trois graphes que l'algorithme 3 ( $\sqrt{n}$ ) est plus rapide que d'algorithme 2 ( $\frac{n}{2}$ ) qui est lui même plus rapide que l'algorithme 1 ( $n$ ).

Donc en terme de rapidité nous choisissons l'Algorithme 3.