Université des Sciences et de la Technologie
Houari Boumediene

Conception et Complexité des Algorithme

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

2 novembre 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>=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 2 jusqu'à n.

```
Algorithme_nombre_premier1

VAR
i,N : entier;
prem : booleen;

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 calcule de la complexité exact 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 ou le nombre est égale à 2 donc la boucle n'est exécutée aucune fois, tel que :
   f1(n) = 1(=) + 1(=) + 4(<= , - , == , et)
   f1(n) = 6 (Opérations) ⇒ f1(n) = O(1)

2. Calcule de la complexité au pire cas :
   Il s'agit du cas ou 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(<=,-,==,et)(n-1) + [2(mod,==)+2(+,=)](n-2)
   f2(n) = 8n - 10 (Opérations) ⇒ 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).

Il s'agit du nombre de case mémoire ou octets utilisés par le programme.
Pour calculer la complexité spatiale de cet algorithme nous allons considérer les tailles mémoires des types en langage C ; tel que nous aurons : int/entier : 2 octets

1. En notation exacte :
   Nous avons dans cet algorithme que trois (3) variables de type "entier"
   Ce qui nous fait : 3(2 octets) = 6 octets

2. En notation asymptotique :
   Le nombre de case mémoire est constant, donc on obtient :
   s(n) = O(1)

3 Développement de programme correspondant avec le langage C.

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

return 0;
}
```

4.2 Résultats de l'exécution du programme :

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

On remarque que tous les nombres données sont premiers !

5 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" .

5.1 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]-1 && prem == 1){

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

    fin = clock();

    if(prem == 1)
    {
        printf("Le nombre saisie : %ld est premier! \n",tab[j]);
    }
    else{
        printf("Le nombre saisie : %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 saisie : 1000003 est premier!
temps d'exécution = 0.009000
Le nombre saisie : 2000003 est premier!
temps d'exécution = 0.009000
Le nombre saisie : 4000037 est premier!
temps d'exécution = 0.013000
Le nombre saisie : 8000009 est premier!
temps d'exécution = 0.026000
```

```
Le nombre saisie : 16000057 est premier!
temps d'exécution = 0.051000
Le nombre saisie : 32000011 est premier!
temps d'exécution = 0.102000
Le nombre saisie : 64000031 est premier!
temps d'exécution = 0.205000
Le nombre saisie : 128000003 est premier!
temps d'exécution = 0.411000
Le nombre saisie : 256000001 est premier!
temps d'exécution = 0.820000
Le nombre saisie : 512000009 est premier!
temps d'exécution = 1.642000
Le nombre saisie : 1024000009 est premier!
temps d'exécution = 3.289000
Le nombre saisie : 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.

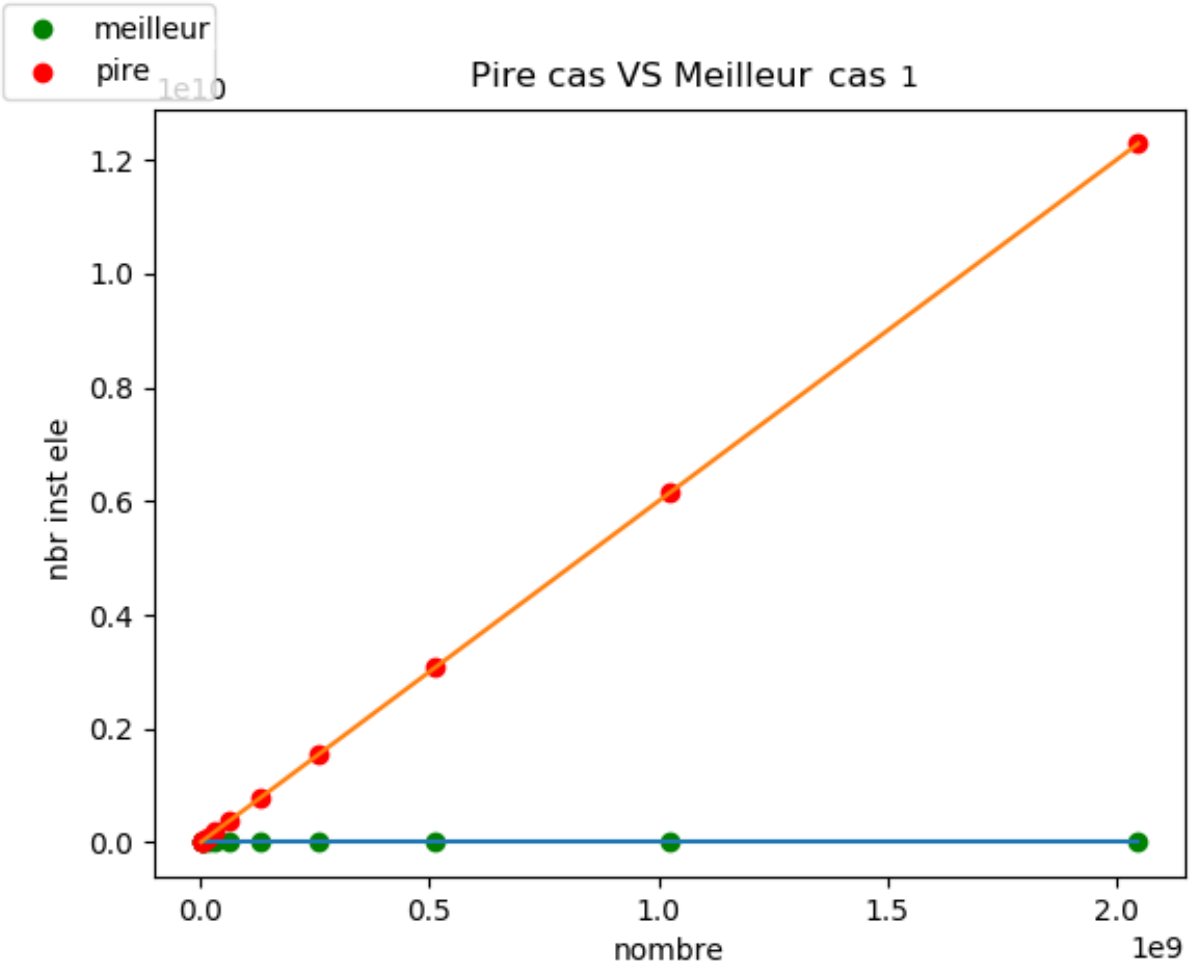
```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

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;

}
```

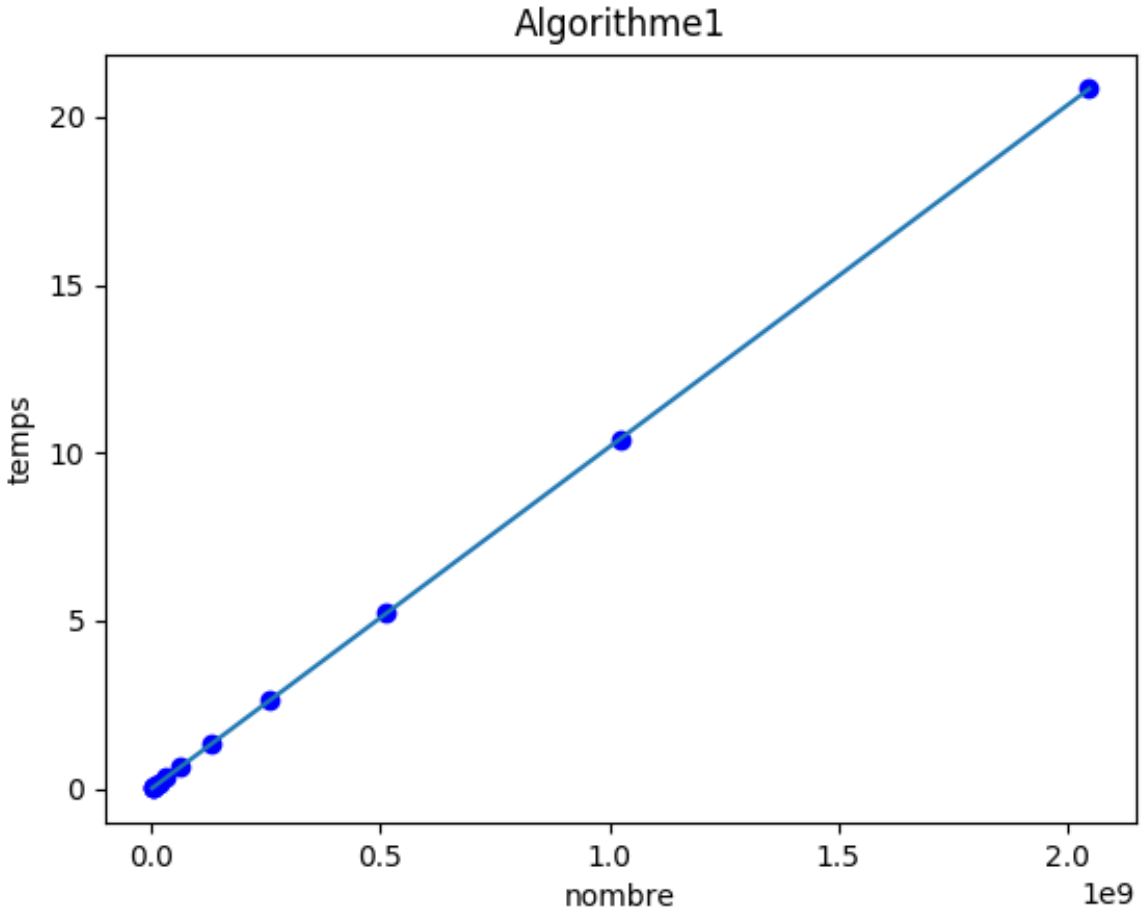
7 Représentation par un graphe, Gf1(n) et Gf2(n), les variations de la fonction de la complexité temporelle correspondant au meilleur cas f1(n) et au pire cas f2(n) en fonction de n respectivement ; et par un autre graphe, noté GT(n), les variations du temps d'exécution T(n) en fonction de n.

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.


meilleur
pire
1e10
Pire cas VS Meilleur cas 1
1.2
1.0
0.8
0.6
0.4
0.2
0.0
nbr inst ele
0.0
0.5
1.0
1.5
2.0
nombre
1e9

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 mesures de temps d'exécution avec le pire et meilleur cas :

1. Remarque :
   En comparant le graphe de la variation du temps d'exécution de l'échantillon GT avec les graphes du meilleur f1 et pire f2 cas, nous constatons que GT tant à ressembler au graphe f1 ; ils ont tous deux une forme polynomial linéaire.

2. Signification :
   Les données de l'échantillon correspondent au pire cas, effectivement se sont tous des nombres premiers.

8.2 Remarque et 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 ⇒ T1 = 0.013000
N2 = 8000009 ≈ 2 ∗ $N1$ ⇒ T2 = 0.026000 ≈ 2 ∗ $T1$

Aussi
N1 = 128000003 ⇒ T1 = 0.411000
N2 = 256000001 ≈ 2 ∗ $N1$ ⇒ T2 = 0.820000 ≈ 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.

Dans le cas des données de l'échantillon la complexité théorique et expérimentale sont du même ordre de grandeur que la complexité théorique du pire cas,
Mais en générale la complexité expérimentale d'un échantillon quelconque est compris entre la complexité au pire cas et au meilleur cas.

Complexité Meilleur cas ≤ complexité expérimentale ≤ complexité Pire cas
f2(n) ≤ GT(n) ≤ f1(n)

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 2 jusqu'à $n/2$.

```
Algorithme_nombre_premier2

VAR
i,N : entier;
prem : booleen;

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 calcule de la complexité exact 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 ou le nombre est égale à 2 donc la boucle n'est exécutée aucune fois, tel que :
   f1(n) = 1(=) + 1(=) + 4(<= , div , == , et)
   f1(n) = 6 (Opérations) $\Rightarrow$ f1(n) = O(1)

2. Calcule de la complexité au pire cas :
   Il s'agit du cas ou le nombre est premier c'est à dire le contenu de la boucle est exécuté $\lfloor \frac{n}{2}$ - 2 + 1 fois = $\lfloor \frac{n}{2}$ - 1 fois. (selon la règle : fin-debut+1)
   ce qui donne :
   f2(n) = 1(=) + 1(=) + 4(<=,div,==,et)($\lfloor \frac{n}{2}$) + [2(mod,==)+2(+,=)]($\lfloor \frac{n}{2}$-1)
   f2(n) = 8$\lfloor \frac{n}{2}$ - 2 (Opérations) $\Rightarrow$ f2(n) = O($\lfloor \frac{n}{2}$) = 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).

Il s'agit du nombre de case mémoire ou octets utilisés par le programme.
Pour calculer la complexité spatiale de cet algorithme nous allons considérer les tailles mémoires

des types en langage C ; tel que nous aurons : int/entier : 2 octets

1. En notation exacte :
   Nous avons dans cet algorithme que trois (3) variables de type "entier"
   Ce qui nous fait : $3(2 \text{ octets}) = 6 \text{ octets}$

2. En notation asymptotique :
   Le nombre de case mémoire est constant, donc on obtient :
   $s(n) = O(1)$

11 Développement de programme correspondant avec le langage C.

```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 saisie : %ld est premier! \n",tab[j]);
    }
    else{
        printf("Le nombre saisie : %ld n'est pas premier! \n",tab[j]);
    }
}
return 0;
}
```

12.2 Résultats de l'exécution du programme :

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

```
Le nombre saisie : 1024000009 est premier!
Le nombre saisie : 2048000011 est premier!
```

On remarque que tous les nombres 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 saisie : %ld est premier! \n",tab[j]);
    }
    else{
        printf("Le nombre saisie : %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 saisie : 1000003 est premier!
temps d'exécution = 0.004000
Le nombre saisie : 2000003 est premier!
temps d'exécution = 0.007000
Le nombre saisie : 4000037 est premier!
temps d'exécution = 0.009000
Le nombre saisie : 8000009 est premier!
temps d'exécution = 0.016000
Le nombre saisie : 16000057 est premier!
temps d'exécution = 0.026000
Le nombre saisie : 32000011 est premier!
temps d'exécution = 0.051000
Le nombre saisie : 64000031 est premier!
temps d'exécution = 0.102000
Le nombre saisie : 128000003 est premier!
temps d'exécution = 0.205000
Le nombre saisie : 256000001 est premier!
temps d'exécution = 0.415000
Le nombre saisie : 512000009 est premier!
temps d'exécution = 0.821000
Le nombre saisie : 1024000009 est premier!
temps d'exécution = 1.647000
Le nombre saisie : 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

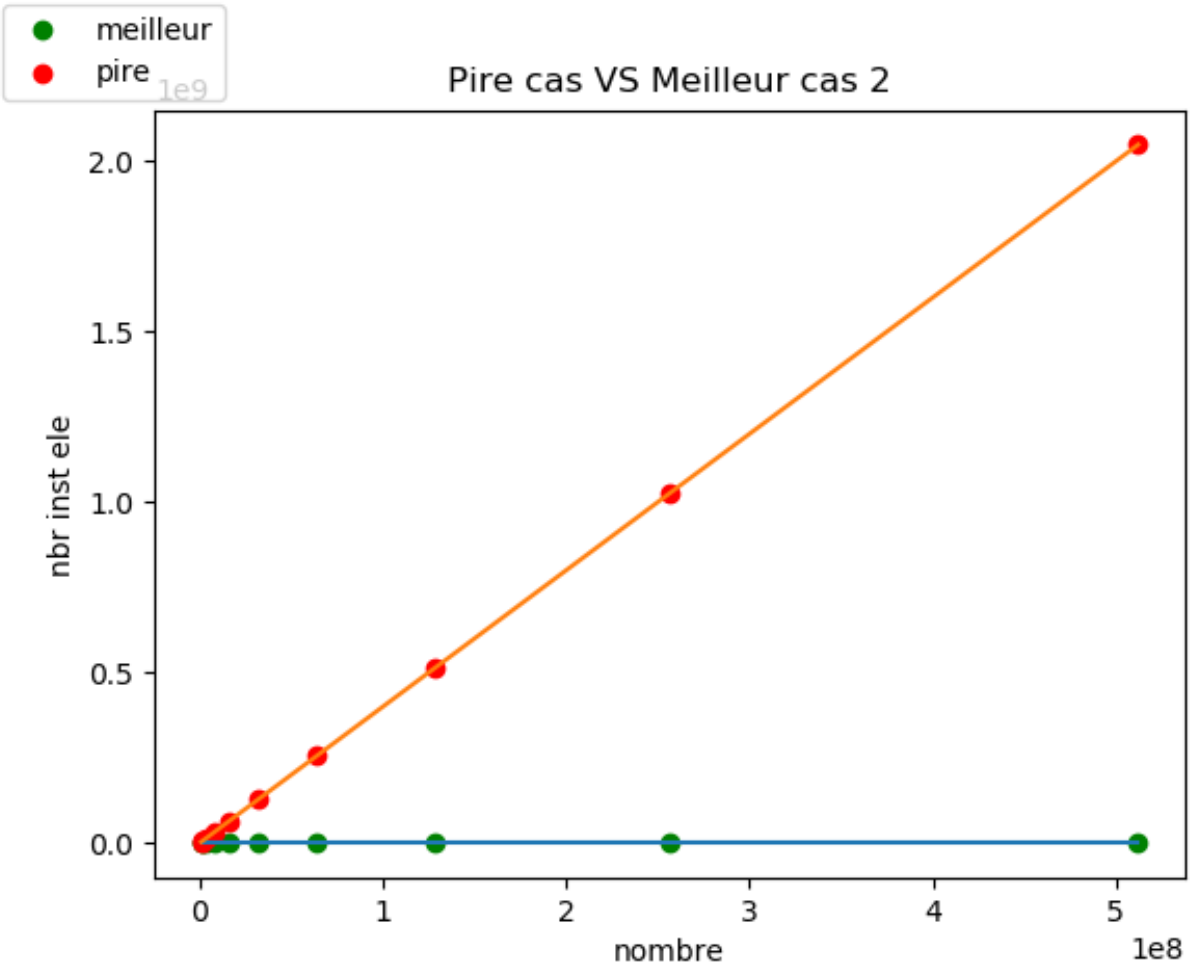
14 Développement du programme de mesure du temps d'exécution du programme qui a en entrée les données de l'échantillon dans tab1 et en sortie les temps d'exécution dans tab2.

```
#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;
}
```

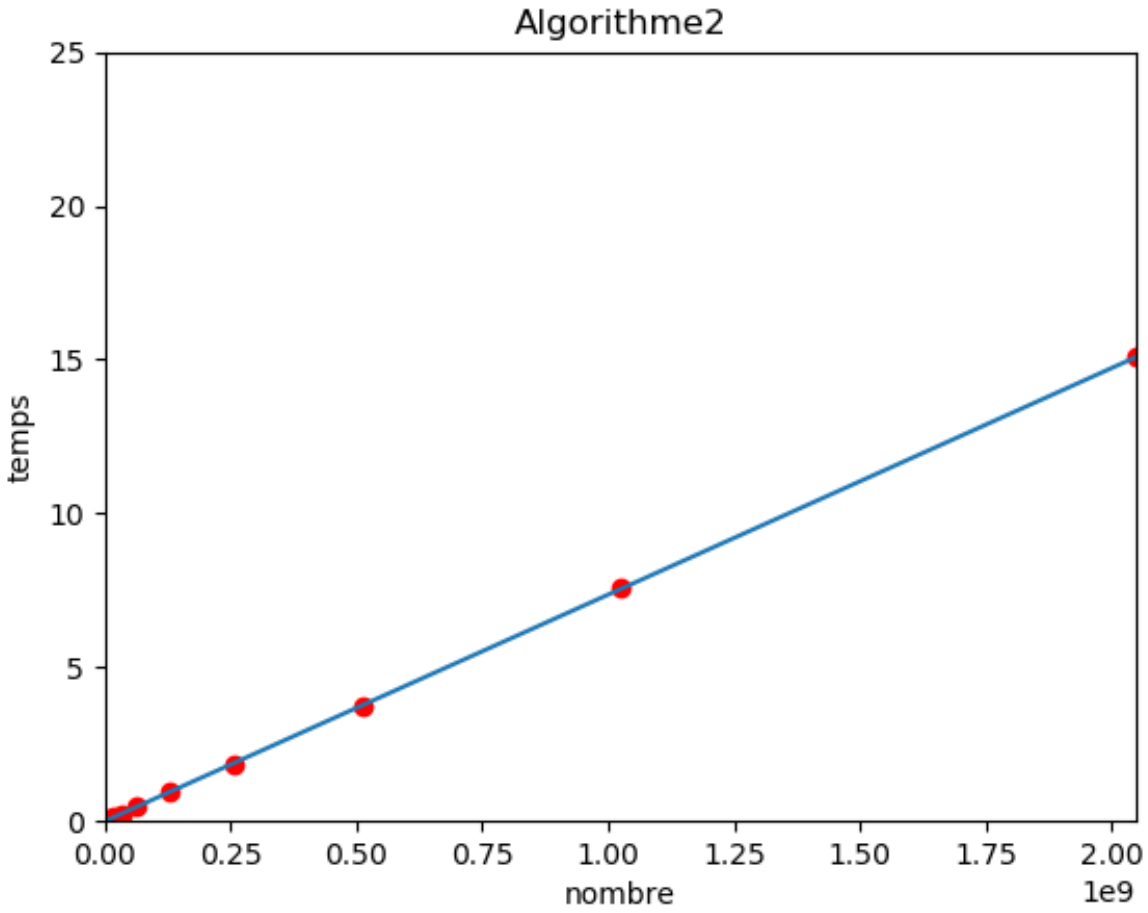
15 Représentation par un graphe, Gf1(n) et Gf2(n), les variations de la fonction de la complexité temporelle correspondant au meilleur cas f1(n) et au pire cas f2(n) en fonction de n respectivement ; et par un autre graphe, noté GT(n), les variations du temps d'exécution T(n) en fonction de n.

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 mesures de temps d'exécution avec le pire et meilleur cas :

1. Remarque :
   En comparant le graphe de la variation du temps d'exécution de l'échantillon GT avec les graphes du meilleur f1 et pire f2 cas, nous constatons que GT tant à ressembler au graphe f1 ; ils ont tous deux une forme polynomial linéaire.

2. Signification :
   Les données de l'échantillon correspondent au pire cas, effectivement se sont tous des nombres premiers.

16.2 Remarque et 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,

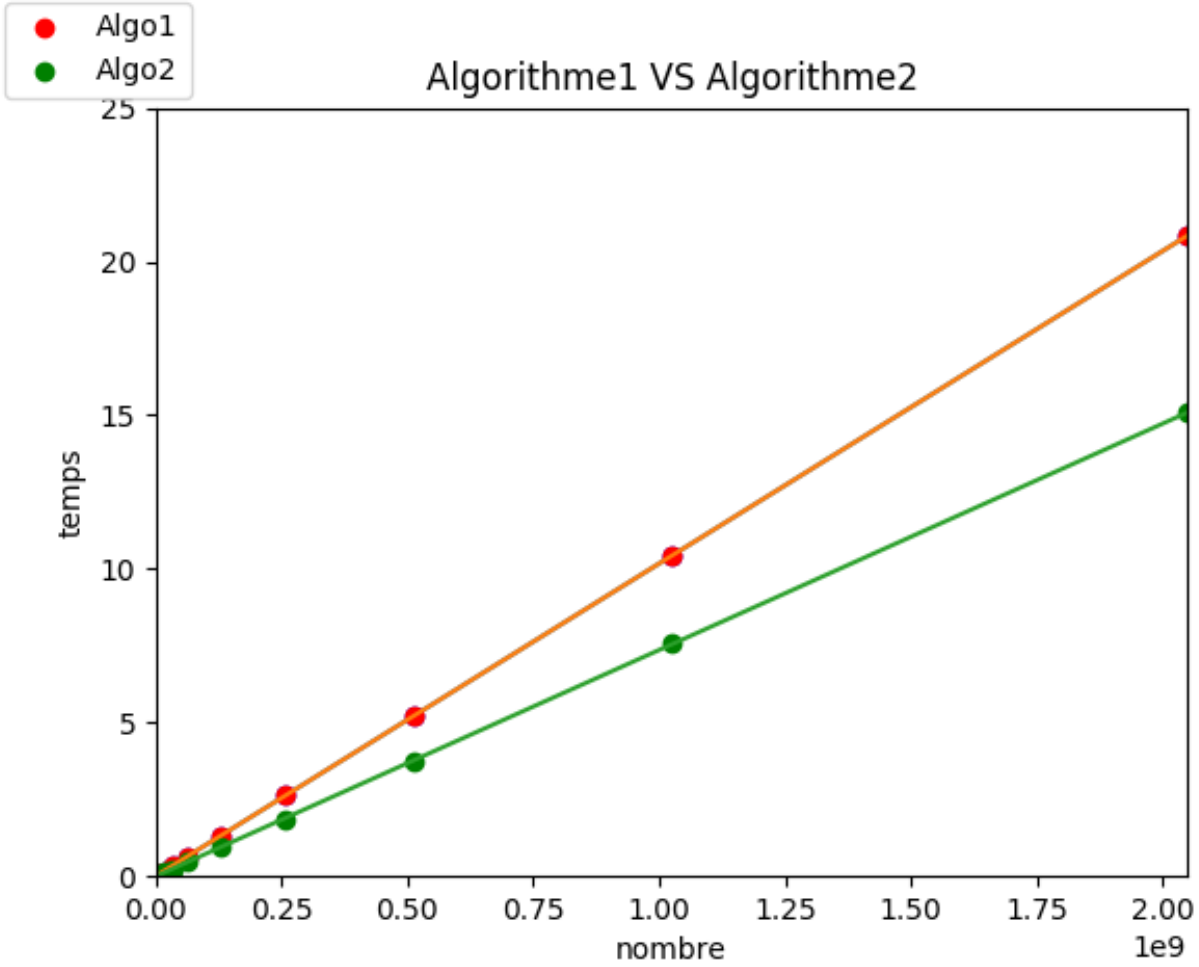
16.3 Comparaison de la complexité théorique et expérimentale.

Dans le cas des données de l'échantillon la complexité théorique et expérimentale sont du même ordre de grandeur que la complexité théorique du pire cas,
Mais en générale la complexité expérimentale d'un échantillon quelconque est compris entre la complexité au pire cas et au meilleur cas.

Complexité Meilleur cas $\leq$ complexité expérimentale $\leq$ complexité Pire cas
f2(n) $\leq$ GT(n) $\leq$ f1(n)

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'algorithme2 ($\frac{n}{2}$) est plus rapide (temps d'exécution) que l'algorithme1 (n).

Donc en terme de rapidité nous choisissons l'Algorithme 2. (vu qu'en terme d'espace mémoire ils sont équivalent)

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>=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 sqrt qui donne la racine carré de i, i variant de 2 jusqu'à $\sqrt{(n)}$.

```
Algorithme_nombre_premier3

VAR
i,N : entier;
prem : booleen;

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 f1(n), et au pire cas, notée f2(n).

Le calcule de la complexité exact 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 ou le nombre est égale à 2 donc la boucle n'est exécutée aucune fois, tel que :
   f1(n) = 1(=) + 1(=) + 4(<= , sqrt , == , et)
   f1(n) = 6 (Opérations) $\Rightarrow$ f1(n) = O(1)

2. Calcule de la complexité au pire cas :
   Il s'agit du cas ou le nombre est premier c'est à dire le contenu de la boucle est exécuté $\lfloor\sqrt{n}$ - 2 + 1 fois = $\lfloor\sqrt{n}$ - 1 fois. (selon la règle : fin-debut+1)
   ce qui donne :

f2(n) = 1(=) + 1(=) + 4(<=,sqrt,==,et)($\lfloor\sqrt{n}$) + [2(mod,==)+2(+,=)]($\lfloor\sqrt{n}$-1)
f2(n) = 8$\lfloor\sqrt{n}$ - 2 (Opérations) $\Rightarrow$ f2(n) = O($\lfloor\sqrt{n}$)= 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).

Il s'agit du nombre de case mémoire ou octets utilisés par le programme.
Pour calculer la complexité spatiale de cet algorithme nous allons considérer les tailles mémoires des types en langage C ; tel que nous aurons : int/entier : 2 octets

1. En notation exacte :
   Nous avons dans cet algorithme que trois (3) variables de type "entier"
   Ce qui nous fait : 3(2 octets) = 6 octets

2. En notation asymptotique :
   Le nombre de case mémoire est constant, donc on obtient :
   s(n) = O(1)

20 Développement de programme correspondant avec le langage C.

```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 saisie : %ld est premier! \n",tab[j]);
    }
    else{
        printf("Le nombre saisie : %ld n'est pas premier! \n",tab[j]);
    }
}
return 0;
}
```

21.2 Résultats de l'exécution du programme :

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

On remarque que tous les nombres 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 calcule 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 saisie : %ld est premier! \n",tab[j]);
    }
    else{
        printf("Le nombre saisie : %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 saisie : 1000003 est premier!
temps d'exécution = 0.000000
Le nombre saisie : 2000003 est premier!
temps d'exécution = 0.000000
Le nombre saisie : 4000037 est premier!
temps d'exécution = 0.000000
Le nombre saisie : 8000009 est premier!
temps d'exécution = 0.000000
Le nombre saisie : 16000057 est premier!
temps d'exécution = 0.000000
Le nombre saisie : 32000011 est premier!
temps d'exécution = 0.001000
Le nombre saisie : 64000031 est premier!
temps d'exécution = 0.001000
Le nombre saisie : 128000003 est premier!
temps d'exécution = 0.001000
Le nombre saisie : 256000001 est premier!
temps d'exécution = 0.002000
Le nombre saisie : 512000009 est premier!
temps d'exécution = 0.002000
Le nombre saisie : 1024000009 est premier!
temps d'exécution = 0.002000
Le nombre saisie : 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.

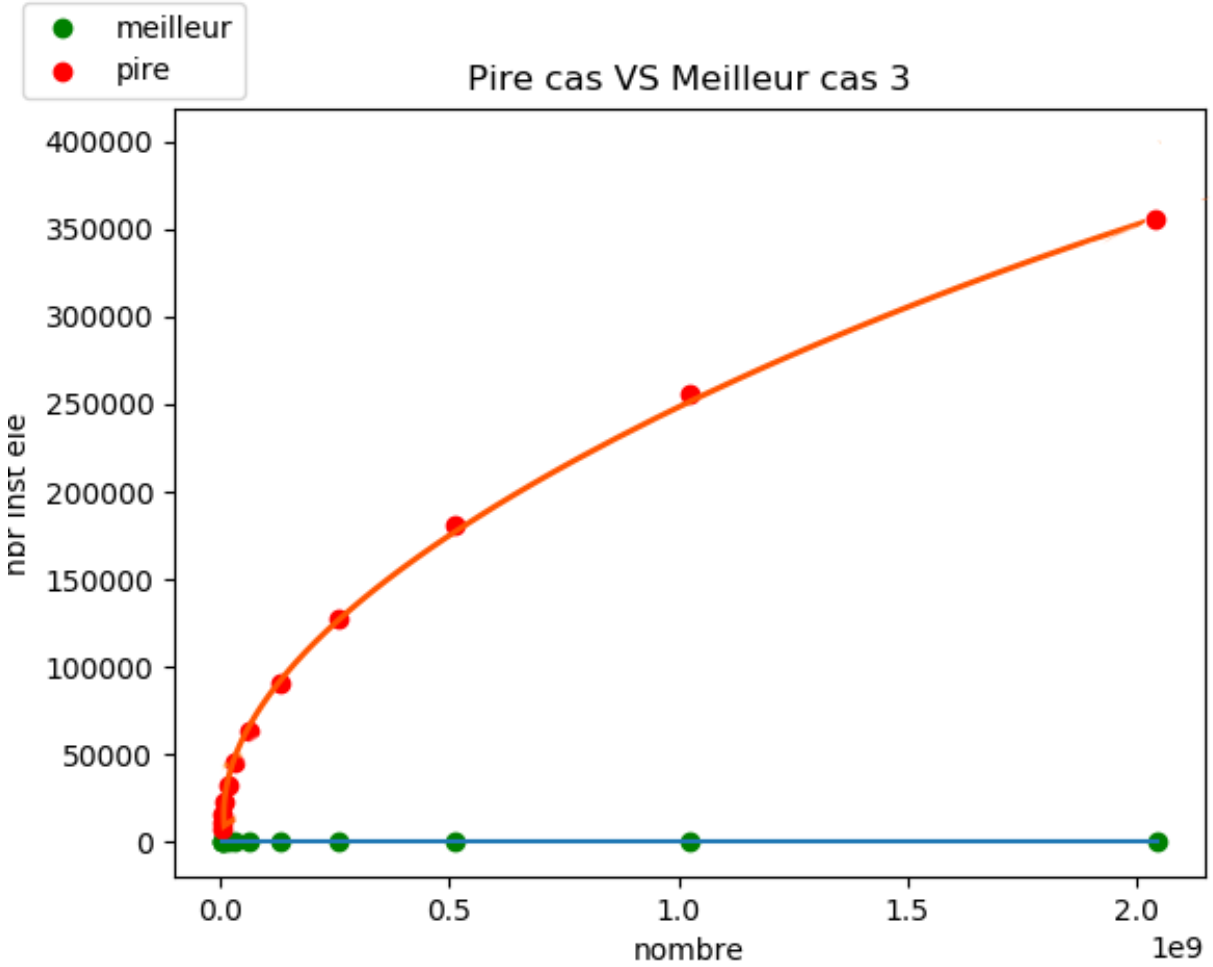
```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

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;
}
```

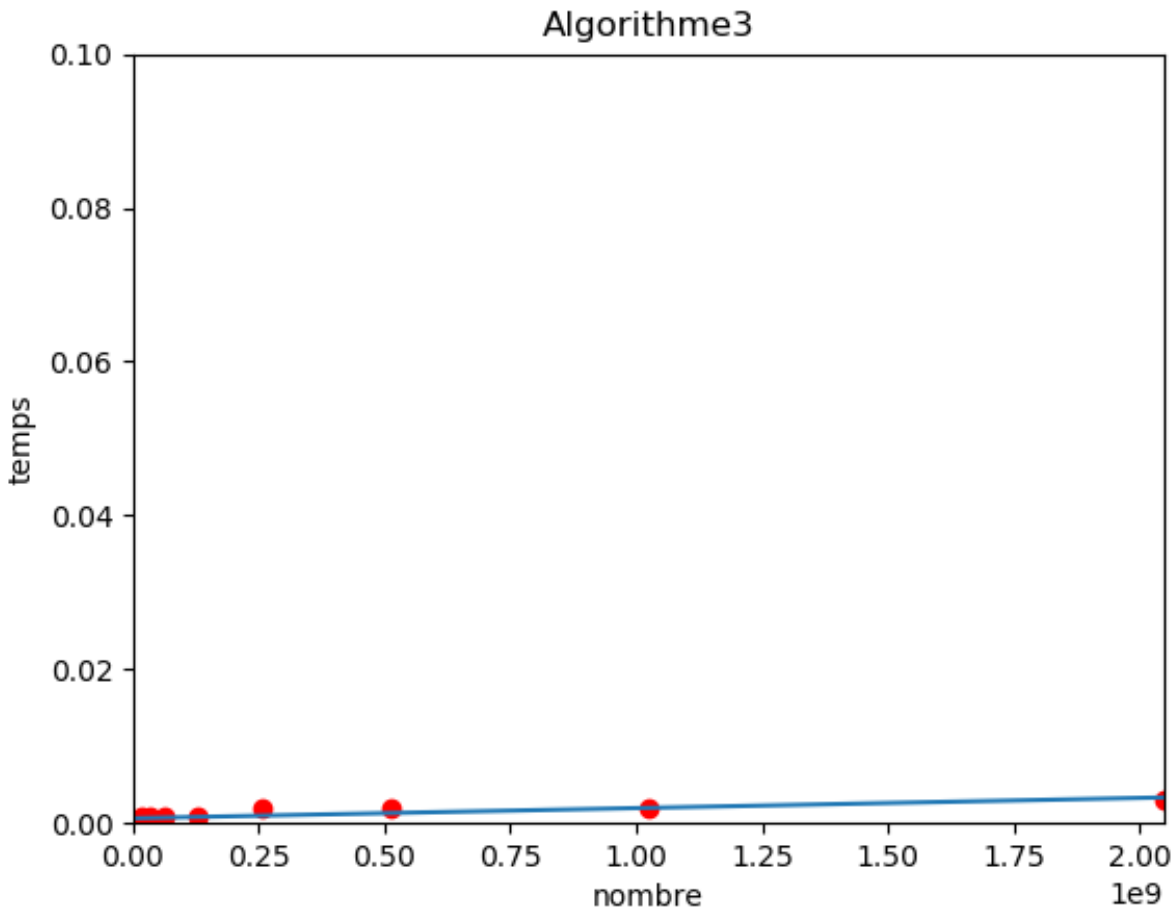
24 Représentation par un graphe, Gf1(n) et Gf2(n), les variations de la fonction de la complexité temporelle correspondant au meilleur cas f1(n) et au pire cas f2(n) en fonction de n respectivement ; et par un autre graphe, noté GT(n), les variations du temps d'exécution T(n) en fonction de n.

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 mesures de temps d'exécution avec le pire et meilleur cas :

1. Remarque :
   En comparant le graphe de la variation du temps d'exécution de l'échantillon GT avec les graphes du meilleur f1 et pire f2 cas, nous constatons que GT tant à ressembler au graphe f1 ; ils ont tous deux une forme racinaire.

2. Signification :
   Les données de l'échantillon correspondent au pire cas, effectivement se sont tous des nombres premiers.

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

On remarque que les temps d'exécution sont augmentés de 0.001000 lorsque N est multipliée par 8.

Exemples :
N1 = 32000011 ⇒ T1 = 0.001000
N2 = 256000001 ≈ 8 ∗ $N1$ ⇒ T2 = 0.002000 ≈ $T1 + 0.00100$

Aussi
N1 = 256000001 ⇒ T1 = 0.002000
N2 = 2048000011 ≈ 8 ∗ $N1$ ⇒ T2 = 0.003000 ≈ $T1 + 0.001000$

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 :

$$T(x * N) = T(N) + y \text{ pour tous } x \in [1000003 - 2048000011] \text{ et } y < 1$$

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

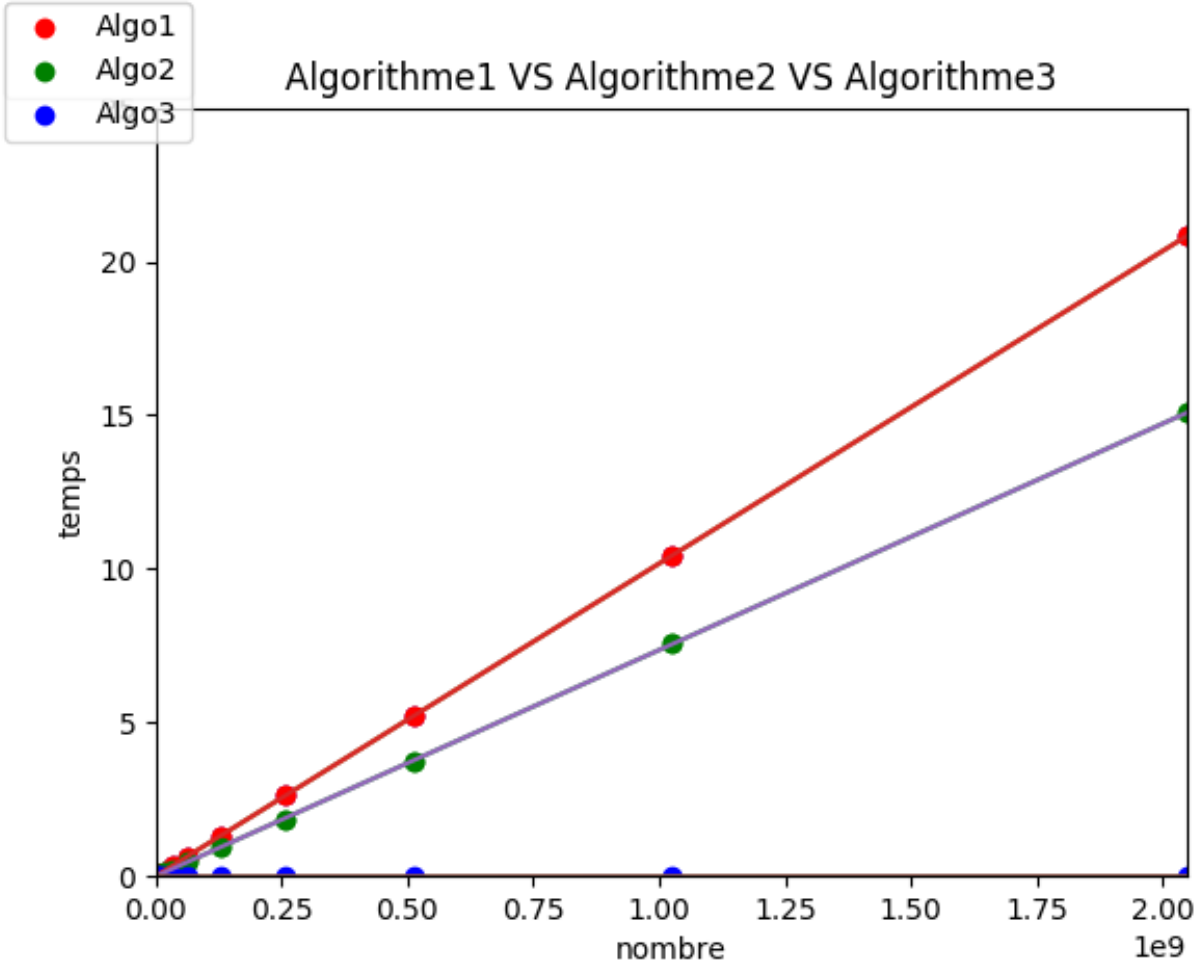
25.3 Comparaison de la complexité théorique et expérimentale.

Dans le cas des données de l'échantillon la complexité théorique et expérimentale sont du même ordre de grandeur que la complexité théorique du pire cas,
Mais en générale la complexité expérimentale d'un échantillon quelconque est compris entre la complexité au pire cas et au meilleur cas.

Complexité Meilleur cas ≤ complexité expérimentale ≤ complexité Pire cas
f2(n) ≤ GT(n) ≤ f1(n)

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. (vu qu'en terme d'espace mémoire ils sont équivalent)