جامعة هواري بومدين
للعلوم والتكنولوجيا
USTHB

RAPPORT PROJET 1

TP 1 Algorithmique et complexité

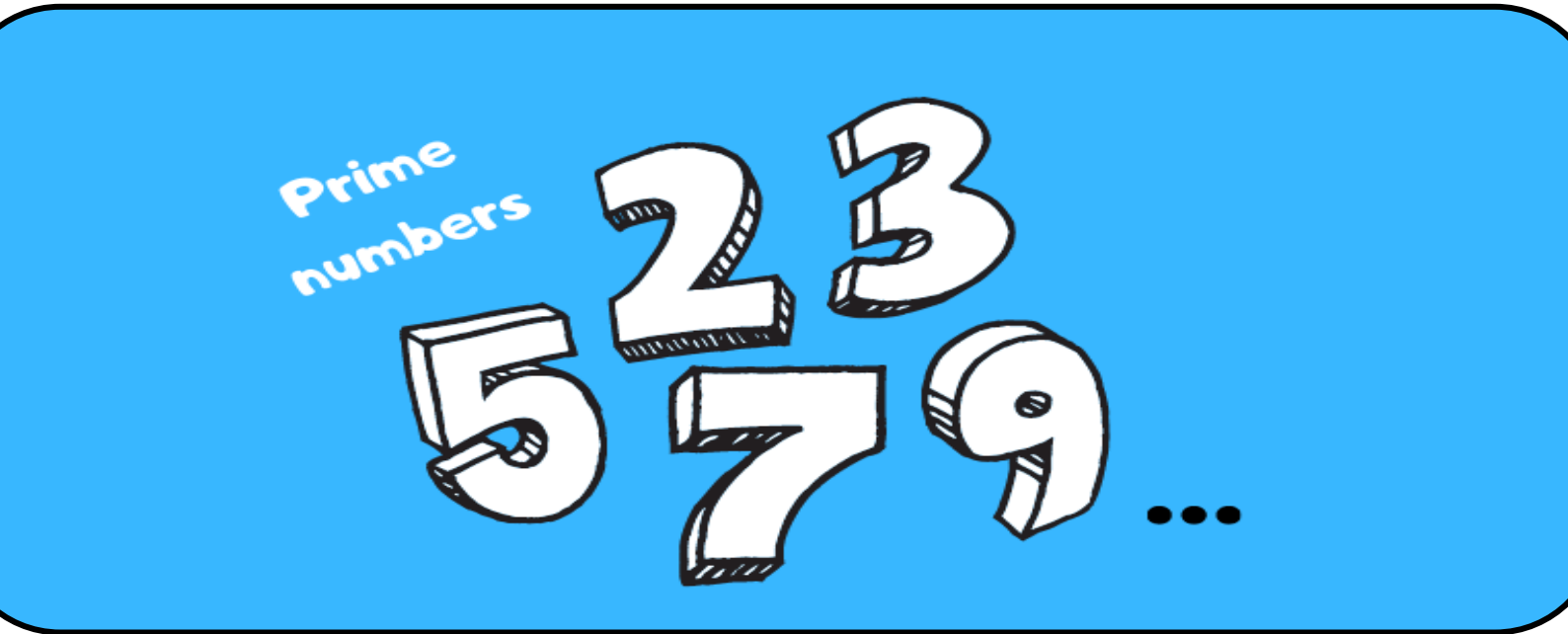
Test de primalité

Réalisé Par :

- Mahmoud Bacha Rabah Sami 202031050494
- Saadallah Abderahim 202031047723

GROUP 3

M1 RSD 2023/2024



INTRODUCTION :

DEFINITION D'UN NOMBRE PREMIER :

Un nombre premier est un entier positif qui n'a pas d'autres diviseurs positifs que lui-même et 1.

Par exemple : 2, 3, 5 .. 890 …

Dans ce mini projet nous allons étudier 4 algorithmes qui permettent de vérifier si un nombre est premier ou non, et puis comparer ces algorithmes en termes de complexité théorique et de temps de calcul.

ENVIRONEMENT DE TRAVAIL :

Les algorithmes seront tous exécuter sur le même échantillon de nombres premiers et aussi sur les mêmes machines pour avoir une comparaison équitable

- Langage : python 3.9
- CPU: i5 8365U 3.9 GHZ max frequency
- Ram: 8 GB
- OS: windows 11
- Logical: VScode Jupiter notebook

Echantillon de donnee N :

1000003 , 2000003 , 4000037 , 8000009 , 16000057 , 32000011 , 64000031 , 128000003 , 256000001 , 512000009 , 1024000009 , 2048000011

ALGORITHM 1 :

PRINCIPE :

Parcourir dans une boucle de 2 jusqu'à N-1, et si N est divisible par i alors on arrête.
pire cas est si N est premier et dans ce cas le nombre d'itération est N-2

Q1-2 : IMPLEMENTATION ET ETUDE DE COMPLEXITE :

```
def algo1( N ):                          //appel 1
    for i in range(2,N):                     //initialization 1
      if N % i == 0:                     //(N-2) (test for 1 + test if 1 + incrementation 1 +affect 1 )
        return False                         //test de sortie 1
    return True
```

Donc la complexité est : f(N) = 4N -5

La complexité théorique de cet algorithme est linéaire O(N)

Q3 : CACUL DE TEMPS D'EXECUTION :

N	temps(s) algo 1
1000003	0.093751669
2000003	0.173330307
4000037	0.324709892
8000009	0.70889473
16000057	1.861647367
32000011	2.754846096
64000031	5.381466866
128000003	10.97707963
256000001	22.63799381
512000009	43.22730517
1024000009	87.10564756
2048000011	335.1974342

Q3_C :

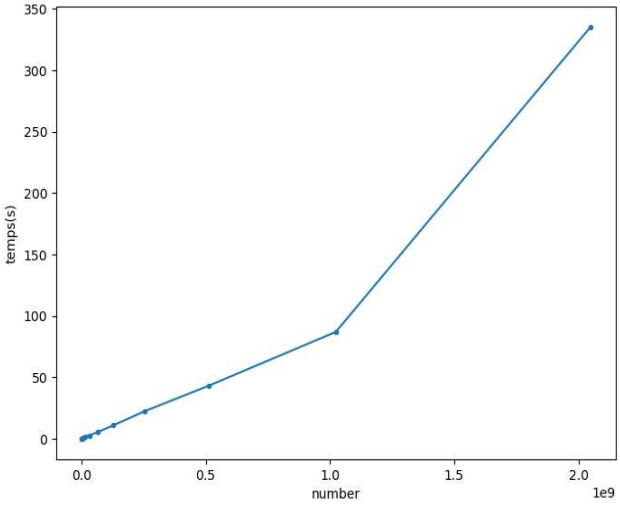
On compare les données, on voit que que le temps évolue de la même façon que N évolue, N se double de valeur, et le temps d'exécution est aussi en train de se doubler

Donc le temps évolue linéairement

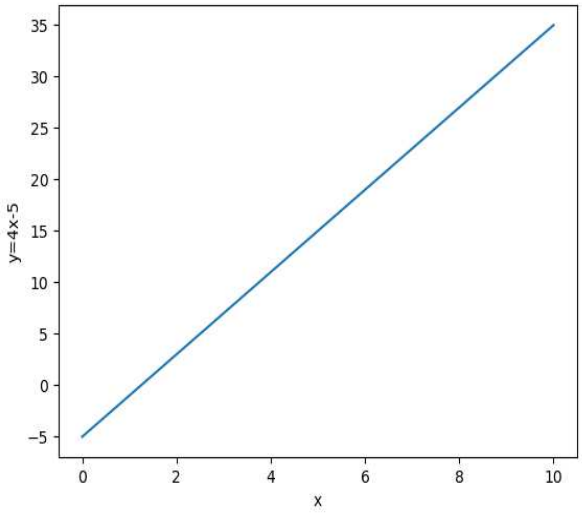
Q3_D :

La complexité théorique est linaire o (N) et les résultats expérimentaux est compatible avec

Q3_E : REPRESENTATION GRAPHIQUE :



Graph T(N) Algo1



Graph O(N) Algo1

ALGORITHM 2 :

PRINCIPE :

Mathématiquement, les diviseurs d'un nombre sont tous inferieur ou égal a sa moitié $\frac{N}{2}$, donc on peut arrêter l'itération a $\frac{N}{2}$.

Nombre d'itération devient : $\frac{N-2}{2}$

Q1-2 : IMPLEMENTATION ET ETUDE DE COMPLEXITE :

```
def algo2( N ):                                  //appel 1
    end=int(N/2)                                 // affect 1 + operation 1
    for i in range(2,end):                       //initialization 1
      if N % i == 0:                             //((N-2)/2) (test for 1 + test if 1 + incrementation 1 +affect 1 )
        return False                             //test de sortie 1
    return True
```

donc la complexité est : f(N) = 2N -3

La complexité théorique de cet algorithme est linéaire O(N)

Q3 : CACUL DE TEMPS D'EXECUTION :

number	temps(s) algo 2
1000003	0.040854454
2000003	0.083082914
4000037	0.186151505
8000009	0.353016376
16000057	0.75756526
32000011	1.582297564
64000031	3.467319727
128000003	7.325021267
256000001	11.4237752
512000009	29.13390851
1024000009	50.48400879
2048000011	143.7383599

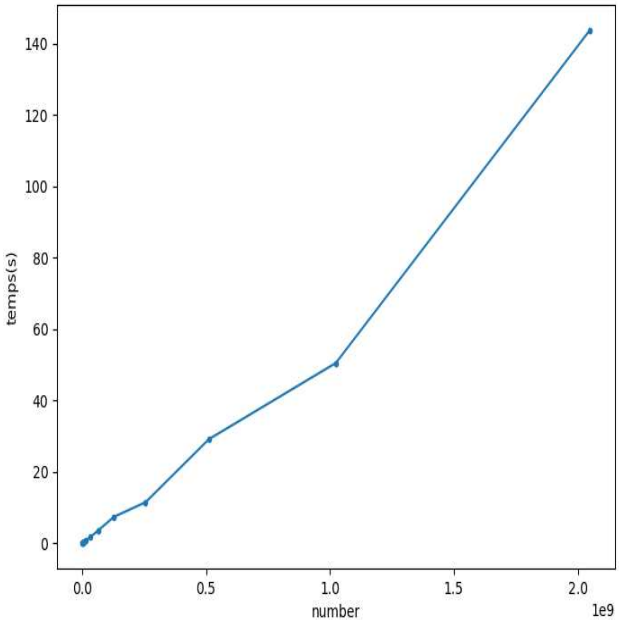
Q3_C :

On compare les données, on voit que le temps est en train d'évoluer linéairement.
les valeurs sont petit par rapport a l'algorithme 1 car on un moins d'itérations

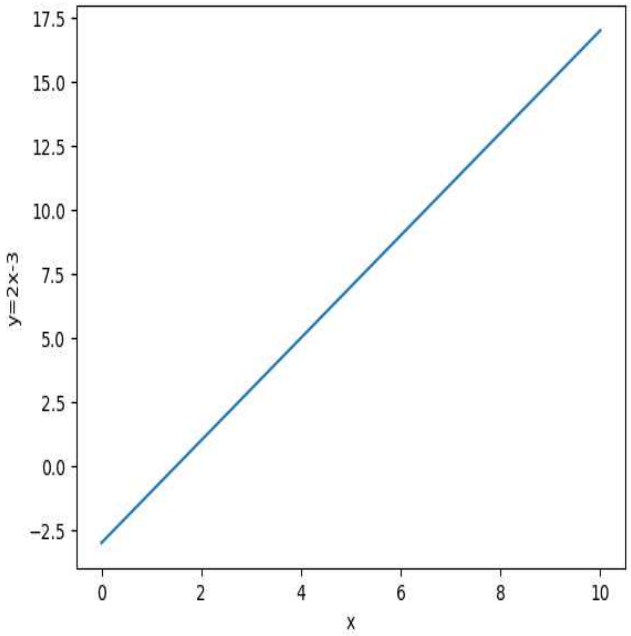
Q3_D :

La complexité théorique est linaire o (N) et les résultats expérimentaux est compatible avec

Q3_E : REPRESENTATION GRAPHIQUE :



Graph T(N) Algo2



Graph O(N) Algo 2

ALGORITHM 3 :

PRINCIPE :

En utilisant la propriété mathématique qui dit que la moitié des diviseurs d'un nombre entier N est

$\leq \sqrt{N}$, on peut limiter le nombre d'itération a $\sqrt{N} - 2$

Q1-2 : IMPLEMENTATION ET ETUDE DE COMPLEXITE :

```
def algo3( N ):                                  //appel 1
    end=int(math.sqrt(N))                        // affect 1 + operation 1
    for i in range(2,end):                       //initialization 1
      if N % i == 0:                             //( √N-2) (test for 1 + test if 1 + incrementation 1 +affect 1 )
        return False                             //test de sortie 1
    return True
```

donc la complexité est : f(N) =$4\sqrt{N} - 3$

La complexité théorique de cet algorithme est linéaire O($\sqrt{N}$)

Q3 : CACUL DE TEMPS D'EXECUTION :

number	temps(s) algo 3
1000003	0
2000003	0.001114607
4000037	0
8000009	0
16000057	0.001015902
32000011	0
64000031	0.001216173
128000003	0.002327919
256000001	0.002730846
512000009	0.001645565
1024000009	0.002563238
2048000011	0.005600452

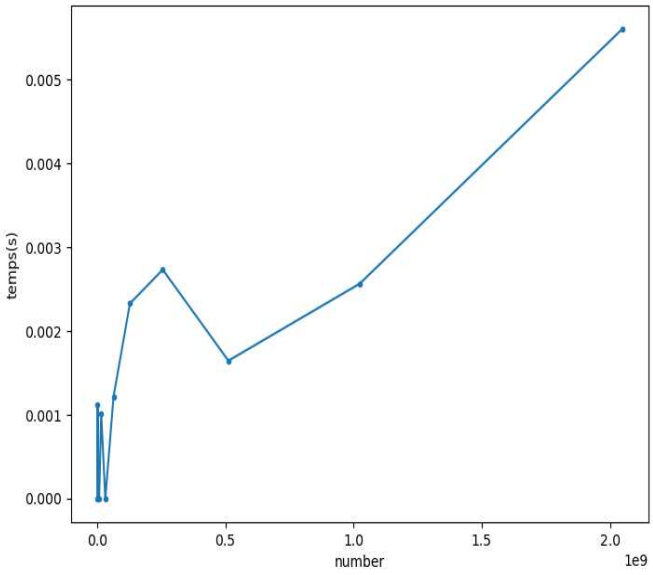
Q3_C :

L'algorithme n'évolue pas d'une manière linaire, l'ordre de complexité est de $\sqrt{N}$, pour les petites valeurs de N, le temps était trop petit pour la machine donc 0 est affiché

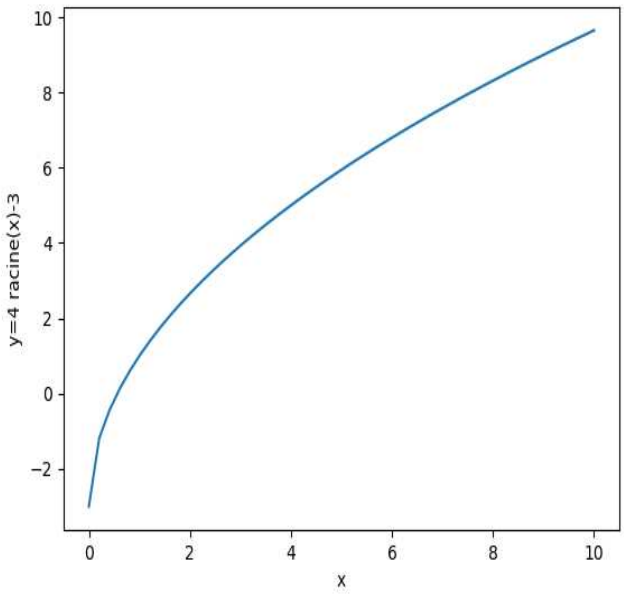
Q3_D :

La complexité théorique est linaire o($\sqrt{N}$) et les résultats expérimentaux est compatible avec

Q3_E : REPRESENTATION GRAPHIQUE :



Graph T(N) Algo3



Graph O(N) Algo3

ALGORITHM 4 :

PRINCIPE :

On fait une itération sur les nombres impairs seulement
Un test est nécessaire pour voir si le nombre est pair.
Si N est pair alors on exécute que le test, sinon s'il est impair, on passe à la boucle
Nombre d'itération $\frac{\sqrt{N}-3}{2}$

L'ordre de complexité reste toujours O ($\sqrt{N}$)

Q1-2 : IMPLEMENTATION ET ETUDE DE COMPLEXITE :

```
def estpremier_algo4( x ):                          // appel 1
        if x % 2==0  and x!=2:                      // Test 3
            return False
        else:
        end=int(math.sqrt(x))+1                     // operation 1 + affect 1
        for i in range(3,end,2):                    // initialization 1
            if x % i == 0:          //(√N −3 / 2) (test for 1 + test if 1 + incrementation 1 +affect 1 )
                return False                        // test de sortie 1
        return True
```

donc la complexité est : f(N) =$2\sqrt{N}+2$

La complexité théorique de cet algorithme est linéaire O($\sqrt{N}$)

Q3 : CACUL DE TEMPS D'EXECUTION :

number	temps(s) algo 4
1000003	0
2000003	0
4000037	0
8000009	0
16000057	0.001050711
32000011	0.00034833
64000031	0
128000003	0.000637293
256000001	0
512000009	0.00205493
1024000009	0.001031876
2048000011	0.003072262

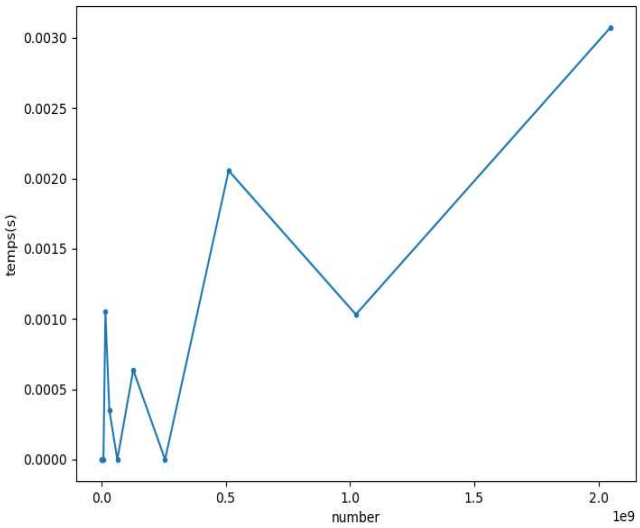
Q3_C :

L'algorithme n'évolue pas d'une manière linaire, l'ordre de complexité est de $\sqrt{N}$, pour les petites valeurs de N, le temps était trop petit pour la machine donc 0 est affiché
l'algorithme est une amélioration d'algorithme 3

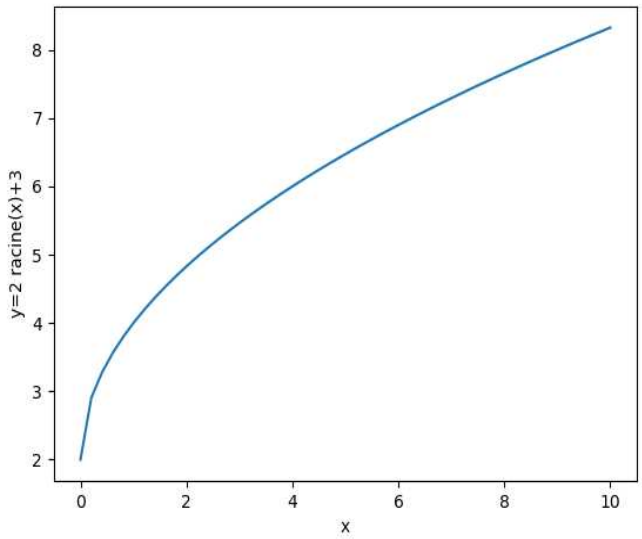
Q3_D :

La complexité théorique est linaire o($\sqrt{N}$) et les résultats expérimentaux est compatible avec

Q3_E : REPRESENTATION GRAPHIQUE :



Graph T(N) Algo 4



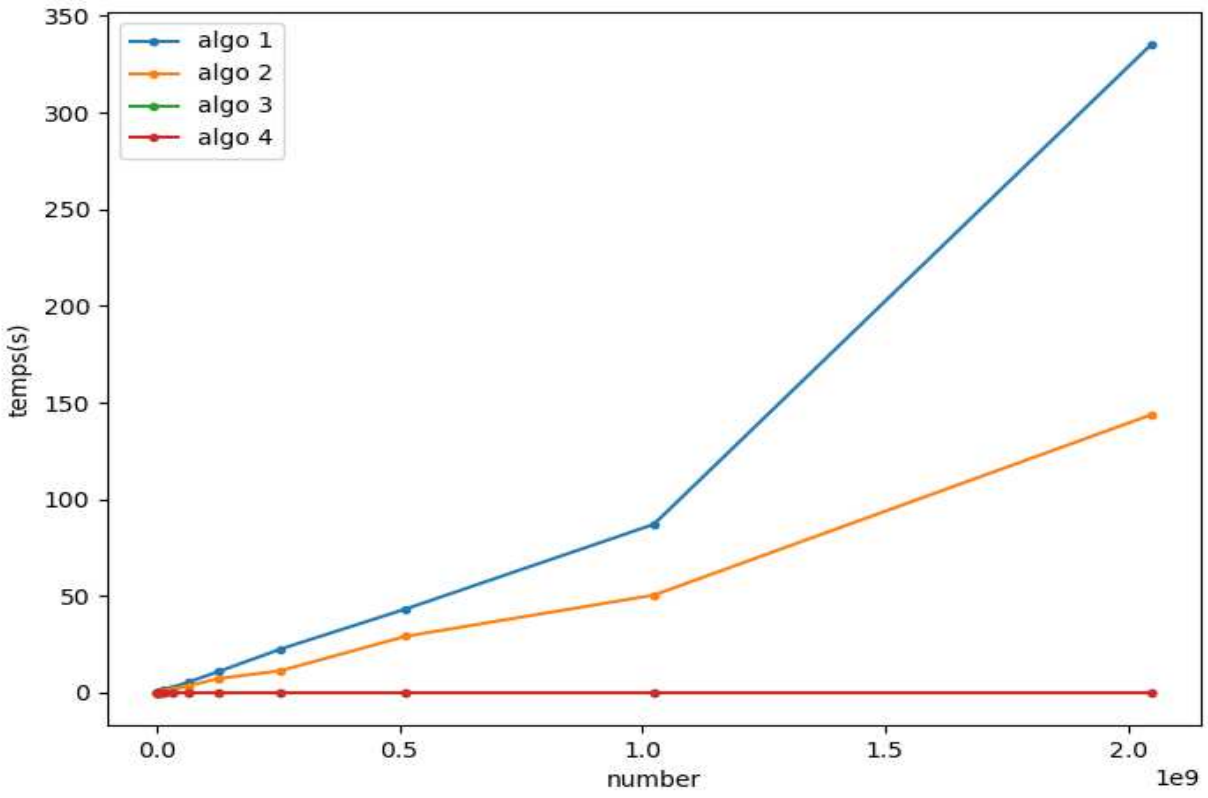
Graph O(N) Algo 4

RESULTAT GLOBAL ET COMPARISON ENTRE LES ALGORITHMES :

Algorithme	1	2	3	4
Nombre d'itérations	$N-2$	$\frac{N-2}{2}$	$\sqrt{N}-2$	$\frac{\sqrt{N}-3}{2}$
Complexité	O(N)	O(N)	O($\sqrt{N}$)	O($\sqrt{N}$)

N	temps(s) algo 1	temps(s) algo 2	temps(s) algo 3	temps(s) algo 4
1000003	0.093751669	0.040854454	0	0
2000003	0.173330307	0.083082914	0.001114607	0
4000037	0.324709892	0.186151505	0	0
8000009	0.70889473	0.353016376	0	0

16000057	1.861647367	0.75756526	0.001015902	0.001050711
32000011	2.754846096	1.582297564	0	0.00034833
64000031	5.381466866	3.467319727	0.001216173	0
128000003	10.97707963	7.325021267	0.002327919	0.000637293
256000001	22.63799381	11.4237752	0.002730846	0
512000009	43.22730517	29.13390851	0.001645565	0.00205493
1024000009	87.10564756	50.48400879	0.002563238	0.001031876
2048000011	335.1974342	143.7383599	0.005600452	0.003072262



Graph T(N) comparaison des 4 algorithmes

- L'algorithme 2 est une petite amélioration de l'algorithme 1, l'ordre de complexité est le même O(N), les deux algorithmes évolue 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

- Dans les algorithmes 3 et 4 , pour certaine valeurs la machine a donné 0 comme temps d'exécution car le temps réels était très petit

- Les algorithmes ayant la même complexité théorique sont proche en termes de temps d'exécution expérimental

CONCLUSION :

Le test de primalité peut se faire en plusieurs approches, mais le choix de l'algorithme optimal joue un rôle important en termes de rapidité de temps d'exécution.
On a pu améliorer l'algorithme de test de primalité par réduire le nombre d'itérations, mais le plus optimal était d'opter pour un algorithme ayant une complexité théorique plus petite.
Un algorithme O ($\sqrt{N}$) est très puissant par rapport à un algorithme O(N)

END