Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique
Département Intelligence Artificielle et
Sciences des données

Spécialité: Système Informatique Intelligent

Module: Complexité des algorithmes

Rapport du TP 1 : Mesure du temps d'exécution d'un programme

Année Universitaire: 2022/2023

Ce rapport a été réalisé par:

"TEAM 28"

NOM:	PRÉNOM:	MATRICULE:
BEHLOUL	HALA LYNA	191931068962
HENDEL	LYNA MARIA	191931068959
NAAR	SARAH	181831087978
NADIR	SOMIA	181833020729

TABLE DES MATIÈRES

I. Introduction

En informatique, il y a toujours plus d'une façon de résoudre un problème en générant différents algorithmes . Par conséquent, il est fortement nécessaire d'utiliser une méthode pour mesurer la performance de l'exécution de ces derniers afin de juger laquelle est le plus optimale

1. Description de l'objectif de l'algorithme

L'objectif de ce TP est de mesurer le temps d'exécution d'un programme en C. En utilisant les fonctions de gestion du temps (<time.h>) de ce langage de programmation. L'exemple choisi dans ce tp est le calcul de primalité dans 6 différents algorithmes.

2. Qu'est ce qu'un nombre premier ?

On dit qu'un nombre entier naturel est premier s'il possède exactement deux diviseurs positifs : 1 et lui-même.

3. Qu'est ce qu'une complexité d'un algorithme ?

Le calcul de la complexité d'un algorithme consiste à évaluer la quantité des ressources utilisées par l'algorithme en question (ou programme informatique qui en est l'implémentation) en termes de temps d'exécution et de ressources matérielles allouées. sachant que L'objectif premier d'un calcul de complexité algorithmique est de **pouvoir comparer l'efficacité d'algorithmes résolvant le même problème qui dans ce cas représente** le calcul de primalité. Il est important de ne pas confondre performances et complexité. Lorsqu'on mesure les performances d'un algorithme on cherche qu'elle est l'implémentation la plus efficace, car par
Par exemple, une implémentation en java sera moins rapide qu'une implémentation en langage natif tel que le C.

- ***Environnement de travail***

Système d'exploitation : Windows 10 .

On a opté pour l'implémentation des algorithmes : Langage C, dev-C++ (5.11).

Compilateur : gcc 4.9.2 ,

- ***Environnement matériel***

  Les expérimentations pour chaque algorithme ont été effectuées sous la machine ayant les caractéristiques suivantes

Ordinateur portable Dell Intel'R) cORE(™) i7-7500U CPU @ 2.70GHz , 2901MHz 2 coeurs , 8Go RAM

- ***Fonctionnement des l'algorithme:***

Un problème représente un ensemble de données sur lesquelles on veut appliquer un traitement pour aboutir à un résultat voulu, dans ce cas on souhaite écrire un algorithme qui demande à l'utilisateur de saisir un entier supérieur à 1 et qui ensuite détermine si ce dernier est premier ou pas.

Partie 1

- ***les algorithmes et leurs complexité :***

1- La première technique d'approche utilisée consiste en une boucle dans laquelle on va tester si le nombre n donné est divisible par 2,3 .. jusqu'à n-1, le pire scénario d'exécution en l'occurrence celui qui prend le plus de temps est le cas ou notre nombre n'est pas premier, la boucle va donc s'exécuter avec un nombre maximum d'itération et donc n-2. La complexité de cet algorithme est O(n).

Algorithme A1 :
début premier = vrai ; i = 2 ; tant que (i <= n-1) et premier faire si (n mod i = 0) alors premier = faux sinon i = i+1 ; fin.

- **La complexité de A1 : O(n).**

2- Le deuxième algorithme consiste à optimiser l'algorithme précédent (A1) , on sait que si n est divisible par 2, il l'est aussi par n/2, pareil pour 3 etc, plus globalement quand un nombre est divisible par n, il est divisible par n/i et c'est ce que démontre cet algorithme. Le cas le moins avantageux est là ou n est premier donc il exécute n/2-1 itération, sa complexité est pareil que son prédécesseur et donc O(n).

Algorithme A2 :
début premier = vrai ; i = 2 ; tant que (i <= [n/2]) et premier faire si (n mod i = 0) alors premier = faux sinon i = i+1 ; fin.

- **La complexité de A2 : O(n).**

3- Pour le troisième algorithme on prend n qui est divisible par x, il reste divisible par n√x, nous essayons donc d'optimiser le deuxième algorithme en appliquant le même principe et en ajoutant la racine, dans ce cas le nombre maximum d'itération sera égale à [√n]-1 obtenant ainsi une complexité de O(√n) ce qui le rend plus optimal que les deux algorithmes précédent

Algorithme 3
début premier = vrai ; i = 2 ; tant que i <= √n et premier faire si (n mod i = 0) alors premier = faux sinon i = i+1 ; fin.

- **La complexité de A3 : est en O(√ n).**

4- Dans ce 4eme algorithme nous considérons le cas où n est impair, donc on ne teste la divisibilité que par des nombres impairs, le pire cas de figure est la ou notre nombre est effectivement premier, le nombre d'itérations de la boucle sera égal à [n/2]-2, avec une complexité égale aux deux premiers algorithmes O(n).

algorithme A4 :
début premier = vrai ; si (n <> 2) et (n mod 2 = 0) alors premier = faux sinon si (n <> 2) alors début i=3; tant que (i <= n-2) et premier faire si (n mod i = 0) alors premier = faux sinon i = i+2 ; fin.

- **La complexité de A4 : O(n).**

5- Quant au cinquième algorithme , il présente une hybridation entre le A2 et A4 , sachant que si un nombre est divisible par 2 il sera divisible aussi par n/2 , en prenant en considération que n est impair on le divisera seulement sur les chiffres impairs entre 3 et n/2 , le nombre d'itérations sera n/4 dans les pires cas

Algorithme A5 :
début premier = vrai ; si (n <> 2) et (n mod 2 = 0) alors premier = faux sinon si (n <> 2) alors début i=3 ; tant que (i <= [n/2])) et premier faire si (n mod i = 0) alors premier = faux sinon i = i+2 ; fin .

- **La complexité de A5 : O(n).**

6- Et pour le dernier algorithme c'est une combinaison entre le A3et le A4 on divise le nombre n que sur les chiffres impairs entre 3 et √n , le nombre max d'itérations de la boucle est égale à (√n /2)-1 sa complexité est donc en O (√n)

Algorithme 6 :
début premier = vrai ; si (n <> 2) et (n mod 2 = 0) alors premier = faux sinon si (n <> 2) alors début i=3 ; tant que (i <= Vn) et premier faire si (n mod i = 0) alors premier = faux sinon i = i+2 ; fin, fin.

- **La complexité de A6 : O(√ n).**

Partie 2

1 - Le temps d'exécution des six algorithmes sur des nombres premiers (de huit chiffres):

	Algorithme					
Nombre	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**
163063	0.002000	0.000000	0.000000	0.001000	0.000000	0.000000
331423	0.005000	0.001000	0.000000	0.001000	0.001000	0.000000
89385071	0.768000	0.373000	0.000000	0.377000	0.196000	0.000000
92130761	0.798000	0.378000	0.000000	0.385000	0.197000	0.000000
171746501	1.560000	0.719000	0.000000	0.711000	0.364000	0.000000
574070509	4.710000	2.294000	0.000000	2.559000	1.0406000	0.000000
738111317	6.171000	2.957000	0.000000	3.333000	1.6220000	0.000000
4021238159	18.266	16.436000	0.001000	8.884000	8.746000	0.000000
5701254253	18.174000	18.174000	0.001	8.684	9.103	0.000000
716984015687	19.583	19.319	0.009	10.214	10.742	0.006

Table 1 : tests sur des nombre premiers de longueur variante entre 6 et 12

Les résultats de la table 1 montrent que pour un nombre premiers relativement petit (de longueur 6) les temps d'exécution pour les six algorithmes sont très petit, et qu'il existe une relation de corrélation directe entre la longueur du nombre premier et de son temps d'exécution, plus la longueur augmente, plus le temps d'exécution est plus important et cela se reflète de manière très claire notamment pour les 3 derniers nombres (lignes du tableau).

Les algorithmes A3 et A6 sont les plus rapides, suivis respectivement par A5, A4, A2 et pour finir A1.

2 - Tableau et graphe représentants les temps d'exécution des six algorithmes en nanoseconde selon la variation du nombre n donnée :

	Algorithme					
Nombre	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**
13775449	0.078000	0.062000	0.000000	0.046000	0.028000	0.000000
27502207	0.184000	0.078000	0.000000	0.094000	0.048000	0.000000
31942387	0.194000	0.093000	0.000000	0.110000	0.060000	0.000000
44438983	0.323000	0.151000	0.000000	0.144000	0.093000	0.000000
51404893	0.315000	0.182000	0.000000	0.167000	0.084000	0.000000
66826651	0.455000	0.219000	0.000000	0.259000	0.114000	0.000000

Table 2 : Mesure de temps d'exécution à des nombre de longueur 8

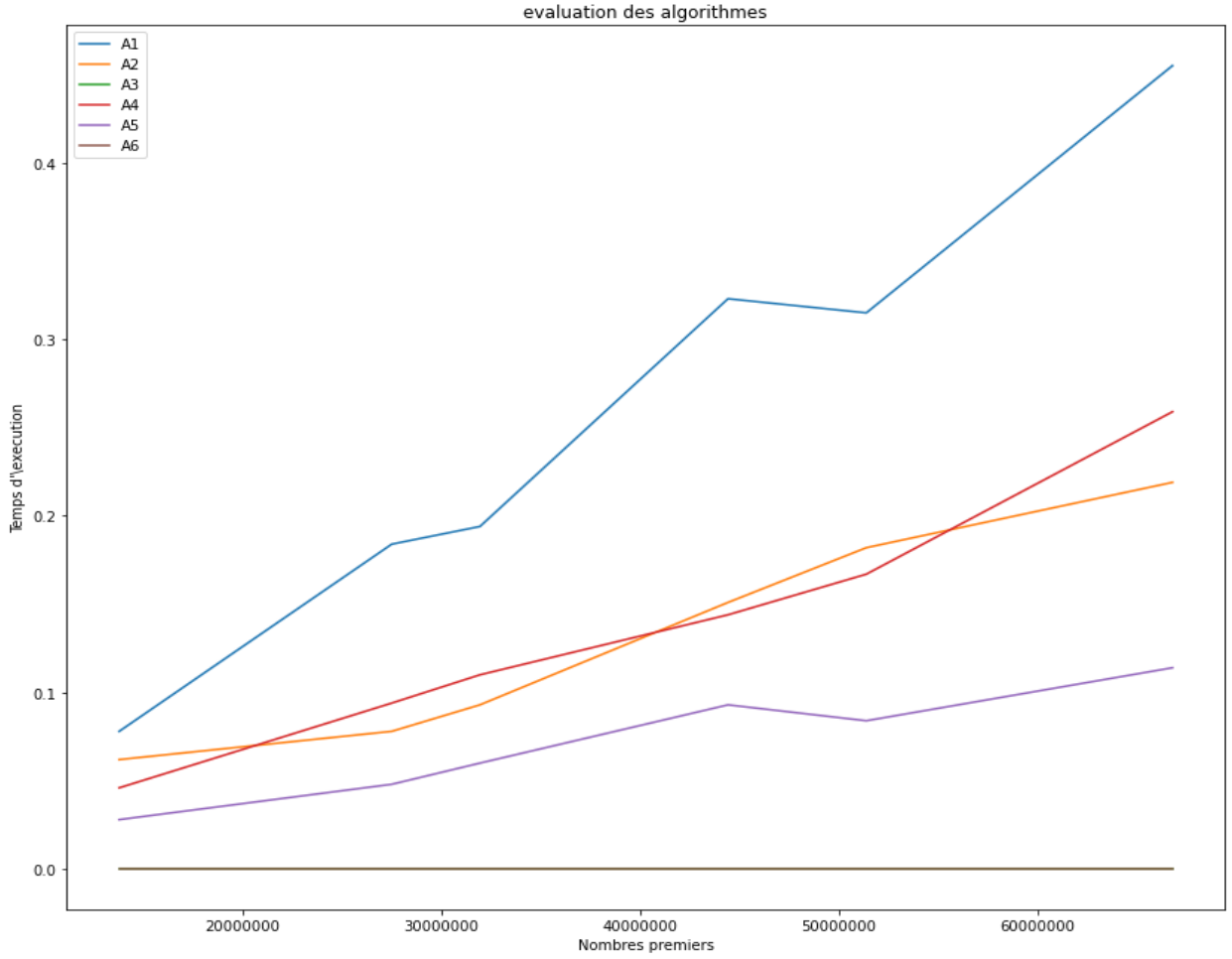

Figure 1: Graphique du temps d'exécution

Le tableau et le graphe ci-dessus représentent le temps d'exécution de six nombres premiers 3 chacun d'une longueur de 8 chiffres, on constate qu'il y'a une relation de corrélation directe entre la taille du nombre premier et le temps d'exécution, plus la taille du nombre augmente, plus on a d'itérations , plus le temps d'exécution est plus important, cela ne s'applique par

contre pas sur les temps d'exécution des algorithmes (A3) et (A6) cela est dû à la structure du code et du nombre d'itérations et donc la complexité des algorithmes, ce qui rend le temps d'exécution beaucoup moins important. On en conclut qu'un algorithme ayant une complexité égale à O(√n) sera beaucoup plus optimal et beaucoup plus rapide qu'un algorithme ayant une complexité de O(√n).

3-Comparaison des moyennes entre les six algorithmes:

Après avoir exécuté 50 fois les six algorithmes précédents sur les nombres indiqués ci dessous et en calculant leurs moyennes voici les résultats obtenus:

	Algorithme					
Nombre	A1	A2	A3	A4	A5	A6
636539	0.183000	0.080000	0.000000	0.09	0.046	0.00000
1000609	0.01535	0.00725	0.0002	0.00665	0.0033	0.00000
13775449	0.113	0.064	0.000000	0.0531	0.028	0.00000
567891089	5.781200	2.915300	0.001800	2.900100	1.474550	0.001050
6750001331	21.09	20.4	0.0012	13.26	13.74	0.001

Table 3: Moyenne de temps d'exécution 50 fois

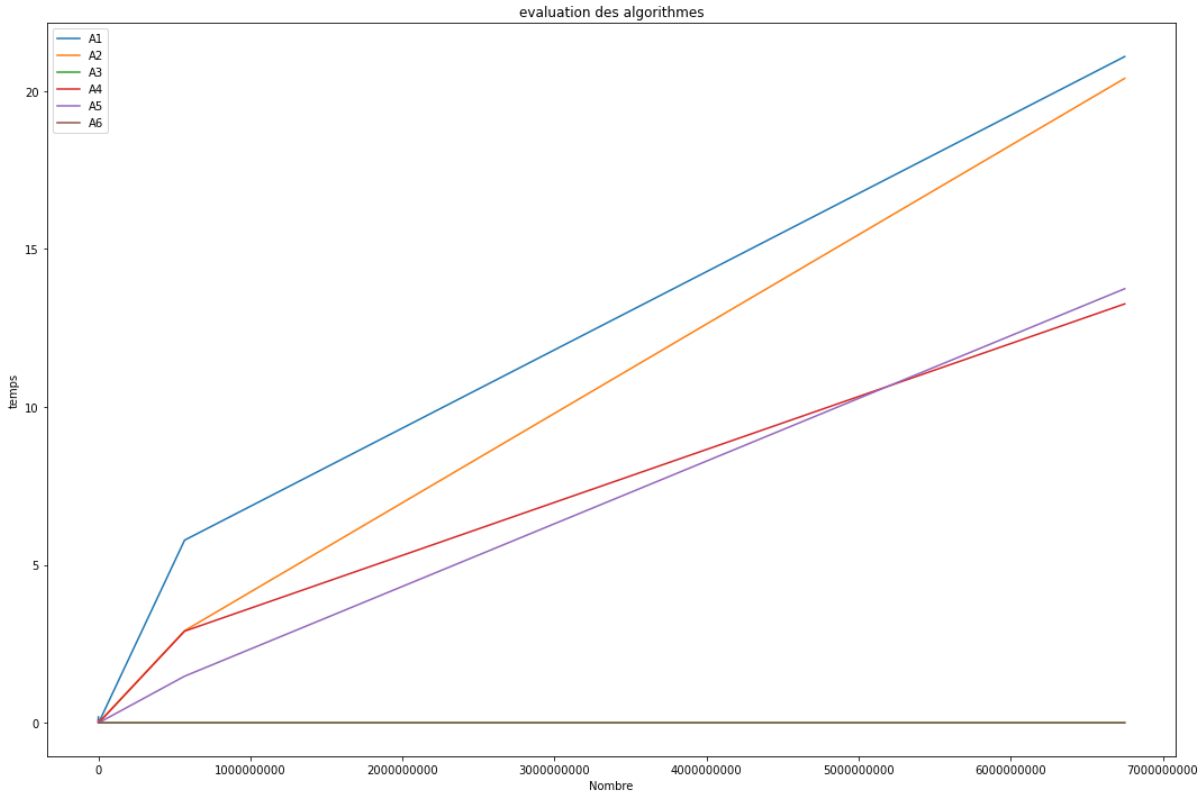

Figure 2: Graphique du temps d'exécution 50 fois

analyse :

On remarque que la représentation graphique des moyennes d'exécution de chaque nombre 50 fois est un outil de compréhension , en premier lieu nous remarquons que si on exécute le même nombre avec le même algorithme ,on n'obtient pas le même temps d'exécution .

En comparant les tables 3 & 1 et les figures 1 & 2 on remarque que les valeurs de la figure 2 sont plus cohérentes et précises que celles de la figure 1 et la table 1 malgré la petite différence ,cette dernière ne peut être négligée, cela peut être expliqué et argumenté par l'existence de certains facteurs externe qui peuvent impacter et causer un changement du temps d'exécution durant l'exécution de notre algorithme,ces changement font qu'il n'est pas possible de comparer des algorithmes lorsqu'on a pas la même combinaison matérielles (processeur,bus de la carte mère, vitesse et capacité de la RAM), car cette dernière influence sur le temps d'exécution.,

Dans un autre cas on remarque de manière plus flagrante grâce à la figure 2 que l'algorithme A1 est le plus lent et le moins optimale des autres algorithmes, plus le nombre n donné augmente plus la différence entre A1 et les autres algo est plus visible (relation de corrélation directe), par ailleurs nous remarquons de la table que A2 et A4 sont presque similaire en ce qui concerne leurs temps d'exécution vu leurs complexité similaire qui est de **O(n) .**

Pour finir il est important de constater que les algorithmes A3 et A6 ont le même ordre de complexité (O(√n) avec un temps d'exécution de 0.00000 la majorité du temps ce qui fait d'eux les algorithmes les plus optimaux par rapport aux autres.

4-Conclusion :

Au cours de ce tp nous avons calculé le temps d'exécution des différents algorithmes chacun ayant sa propre complexité en utilisant des nombres premiers de longueurs différentes afin de pouvoir illustrer le changement dans le temps d'exécution et cela que dépend essentiellement de la structure de l'algorithme et donc en l'occurrence sa complexité, la longueur du nombre premier et bien évidemment l'état du cpu, cela confirme les notions acquises en cours et en td qui démontrent qu'une complexité réduite donne un temps d'exécution meilleur.

Répartition des tâches:

- BEHLOUL Hala Lyna : Réalisation des algorithmes A1 et A3 et calcul de leurs complexités, réalisation des tables et rédaction du rapport

- HENDEL Lyna Maria: Réalisation de l'algorithme A5 et calcul de sa complexité, calcul des moyennes et rédaction du rapport

- NAAR Sarah: Réalisation de l'algorithme A6 et calcul de sa complexité, exécutions et évaluations et calcul des moyennes

- NADIR Somia: Réalisation des algorithmes A2 et A4 et calcul de leurs complexités, réalisation des graphes (Les graphes ont été réalisés avec PYTHON)

Annexe (code source)

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

bool testPremA1 (  unsigned long long n ){
   int i,j;
   bool premier;
   clock_t t1,t2;
   double delta;
   premier=true;
float somme , moy;
somme =0;
for (j=0;j<=50;j++){

   t1 = clock();
   i=2;
   while(i<n && premier){
      if(n % i == 0){
         premier=false;
      }else {
      i++;}
   }
   t2 = clock();
   delta = (double) (t2-t1)/CLOCKS_PER_SEC ;

  printf("Le temps d'exécution est : %lf\n",delta);
      somme=somme+delta;
}
moy = somme /50;
printf("La moyenne est :%f\n",moy);
  return premier;


}
bool testPremA2 (  unsigned long long n){
```

```
   int i,j;
   double delta;
   bool premier;
   clock_t t1,t2;

float somme , moy;
somme =0;
for (j=0;j<=50;j++){
   t1 = clock();
   premier=true;
   i=2;
   while(i<=n/2 && premier==true){
      if(n % i == 0){
         premier=false;
      }
      else {
      i++;}
   }
   t2 = clock();
   delta = (double) (t2-t1)/CLOCKS_PER_SEC ;

   printf("Le temps d'exécution est : %lf\n",delta);
       somme=somme+delta;
}
moy = somme /50;
printf("La moyenne est :%f\n",moy);
   return premier;

}

bool testPremA3 (  unsigned long long n ){
   int i,j;
   bool premier;
   clock_t t1,t2;
   double delta;

float somme , moy;
somme =0;
```

```
for (j=0;j<=50;j++){

   t1 = clock();
   premier=true;
   i=2;
   while(i<=sqrt(n) && premier==true){
      if(n % i == 0){
         premier=false;
      }else {
      i++;}
   }
   t2 = clock();
   delta = (double) (t2-t1)/CLOCKS_PER_SEC ;
   printf("Le temps d'exécution est : %lf\n",delta);
       somme=somme+delta;
}
moy = somme /50;
printf("La moyenne est :%f\n",moy);
   return premier;



}

bool testPremA4 (  unsigned long long n){
   int i,j;
   double delta;
   bool premier;
   clock_t t1,t2;
   premier=true;

float somme , moy;
somme =0;
for (j=0;j<=50;j++){
   t1 = clock();


   if(n!=2 && n%2==0){
```

```
        premier=false;
    }
    else{ if(n!=2) {
    i=3;
    while(i<=n-2 && premier){
        if( n % i ==0){
            premier=false;
        }
        else{
        i=i+2;}
    }}}
    t2 = clock();
    delta = (double) (t2-t1)/CLOCKS_PER_SEC ;
    printf("Le temps d'exécution est : %lf\n",delta);
     somme=somme+delta;
}
moy = somme /50;
printf("La moyenne est :%f\n",moy);
    return premier;

}
bool testPremA5 (unsigned long long n )
{
 int j,i;
 bool premier;
 clock_t t1, t2 ;
 double delta ;
float somme , moy;
somme =0;
for (j=0;j<=50;j++){
  t1 = clock();
  premier=true;

    if(n!=2 && n%2==0){
        premier=false;
    }
   else { if(n!=2) {
       i=3;
```

```
      while(i<=n/2 && premier){
       if(n%i==0){
          premier=false;
       }
       else {
          i=i+2;
       }
      }
   }}

  t2 = clock();
  delta = (double) (t2-t1)/CLOCKS_PER_SEC ;
  printf("Le temps d'exécution est : %lf\n",delta);
 somme=somme+delta;
}
moy = somme /50;
printf("La moyenne est :%f\n",moy);
  return premier;

}
bool testPremA6 ( unsigned long long n )
{   int i,j ;
float somme , moy;
somme =0;
    bool premier;
   clock_t t1, t2 ;
   double delta ;
for (j=0;j<=30;j++){
          t1 = clock();
  premier=true;
   if (n != 2 && n % 2 == 0)
   { premier = false ;
          }
  else
  {
    if (n!=2){
          i =3;
          while (i<= sqrt(n) &&  premier == true){
```

```
                if(n %i == 0 ){
                        premier =false ;
                        }
                        else {
                        premier = true ;
                        }
                        i =i+2;
                 }
          }
  }
  t2 = clock();
  delta = (double) (t2-t1)/CLOCKS_PER_SEC ;
  printf("Le temps d'exécution est : %lf\n",delta);
  somme=somme+delta;
}
moy = somme /50;printf("La moyenne est :%f\n",moy);

  return premier;

}



int main()
{
   unsigned long long n;
   int i;

   bool premier;


   printf("Donner un nombre :\n");
   scanf("%llu",&n);
   premier=testPremA1(n);
   premier=testPremA2(n);
   premier=testPremA3(n);
   premier=testPremA4(n);
   premier=testPremA5(n);
```

```
   premier=testPremA6(n);


   if(premier)

  {
     printf("\n%llu est un nombre premier \n",n);
  }
  else
  {
     printf("\n%llu n'est pas un nombre premier \n",n);
  }

   return 0;
}
```