Série de Travaux Pratiques n° 2 (TP n°2)

Algorithmes de Complexités temporelles linéaire $O(n)$ et racine carrée $O(\sqrt{n})$

Note : Un minimum de connaissances en programmation est indispensable pour le suivi des séances des travaux pratiques.

L'objet de ce TP est une étude expérimentale de 3 algorithmes du problème du test de la primalité d'un nombre entier naturel. Les 2 premiers algorithmes ont une complexité linéaire $O(n)$ et le 3ème algorithme a une complexité en racine carrée $O(\sqrt{n})$. On utilise le langage de programmation C.

Rappel : Un nombre entier naturel n est premier s'il n'a que 2 diviseurs : le nombre 1 et le nombre n lui-même.

Partie I (Algorithme 1 du test de la primalité)

1- Développer un algorithme qui permet de déterminer si un nombre entier naturel n est premier $(n \geq 2)$.

Ind : Utiliser la fonction *modulo* qui donne le reste de la division de n *par* i, i *variant de* 1 *jusqu'à* n.

2.1- Calculer les 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 $f2(n)$.

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- Développer le programme correspondant avec le langage C.

4- Vérifier par programme si les nombres n donnés dans le tableau ci-dessous $(1.000.003, 2.000.003, \ldots)$ sont premiers. On doit remarquer que comme les nombres n varient dans l'intervalle $1\,million\,à\,environ\,2\,milliards$, le nombre de tests (le reste de la division de n *par* i) varie aussi dans le même intervalle.

5- Mesurer les temps d'exécution T pour ces nombres n et compléter le tableau ci-dessous.

Ind : Pour mesurer le temps d'exécution d'un programme avec le langage C, on utilise les fonctions de gestion du temps qui sont fournies dans la bibliothèque *time.h* (inclure l'instruction : ¿*include*<*time.h*>¿).

n	1.000.003	2.000.003	4.000.037	8.000.009	16.000.057	32.000.011	64.000.031
T							

n	128.000.003	256.000.001	512.000.009	1024.000.009	2048.000.011
T					

6- Développer un programme de mesure du temps d'exécution du programme qui a en entrée les données de l'échantillon ci-dessus et en sortie les temps d'exécution. Les données et les mesures du temps sont à enregistrer dans des tableaux notés respectivement $Tab1$ *et* $Tab2$.
7- Représenter par un graphe, noté $Gf1(n)$, les variations de la fonction de la complexité temporelle au meilleur cas $f1(n)$ en fonction de n ; et par un autre graphe, noté $GT(n)$, les variations du temps d'exécution $T(n)$ en fonction de n. Utiliser pour cela un logiciel graphique tel que excel.
8- Interprétation des résultats :
8.a- Les mesures du temps obtenues correspondent-elles au meilleur cas ou au pire cas ?
8.b- Que remarque-t-on sur les données de l'échantillon et sur les mesures obtenues ? Peut-on déduire, même de façon approximative, une fonction $T(n)$ reliant T *et* n ; c'est-à-dire une fonction $T(n)$ permettant de déterminer directement la valeur de T à partir de$n$.
Ind: comparer chaque nombre n avec le suivant ; et chaque mesure du temps avec la suivante.
8.c- Comparer entre la complexités théorique et la complexité expérimentale (çàd., les mesures expérimentales). Les prédictions théoriques sont-elles compatibles avec les mesures expérimentales ?

Partie II (Algorithme 2 du test de la primalité)
On cherche à améliorer l'algorithme précédent.
On sait que tout diviseur i du nombre n vérifie la relation : $i \leq n/2, avec\ i \neq n$.
1- Développer un 2ème algorithme en tenant compte de cette propriété et refaire les questions 2 à 8 de la partie 1.
2- Comparer les 2 algorithmes (représenter pour cela dans une même figure les graphes des 2 algorithmes). Lequel des 2 algorithmes est meilleur (ou plus performant) ?

Partie III (Algorithme 3 du test de la primalité)
On cherche à améliorer encore l'algorithme du test de la primalité.
Il existe une propriété mathématique sur les nombres entiers :
Propriété : les diviseurs d'un nombre entier n sont pour la moitié $\leq \sqrt{n}$ et pour l'autre moitié $>\sqrt{n}$.
1- Développer un 3ème algorithme en tenant compte de cette propriété et refaire toutes les questions 2 à 8 de la partie 1.
2- Comparer les 3 algorithmes (représenter pour cela dans une même figure les graphes des 3 algorithmes). Lequel des 3 algorithmes est meilleur (ou plus performant) ?

Le Corrigé du TP n°2

I. Partie I (Algorithme 1 du test de la primalité)

1- Développer un algorithme qui permet de déterminer si un nombre entier naturel n est premier $(n \geq 2)$.

Ind : Utiliser la fonction *modulo* qui donne le reste de la division de *n par i , i variant de* 1 *jusqu' à n*.

Solution :

Il existe plusieurs algorithmes du test de la primalité. Celui qu'on présente ici est basé sur le comptage du nombre de diviseurs du nombre n à tester. On utilise pour cela l'opérateur *modulo , noté "mod"*, qui donne le reste de la division entière. On parcourt tous les diviseurs potentiels $i$du nombre n qui varient de 1 *jusqu' à* n. A chaque rencontre d'un diviseur i, on incrémente un compteur, noté nb, qui compte le nombre de diviseurs. Si ce dernier est égal à 2 (càd, il n'y a que 2 *diviseurs* : 1 *et* n) alors le nombre n est premier ; sinon il n'est pas premier. L'algorithme est présenté sur la figure 1 ci-dessous.

```
Algorithme Primalité_1 ;        //Algorithme 1 du test de la primalité d'un nombre entier
                                //naturel n
Var     n, i, nb  :         entier ;

Debut
        //Partie 1: Entrée des données
 1      ecrire('Donner un nombre n : ') ;
 2      lire(n) ;

        //Partie 2: Traitement des données
 3      nb := 0 ;           //:= désigne l'instruction d'affectation
 4      i := 1 ;
 5      tant que (i<=n) faire
        debut
 6              si (n mod i = 0)
 7                 alors nb := nb + 1 ;
                fin si ;
 8              i := i + 1 ;        //tester le diviseur éventuel suivant
        fin tant que ;

        //Partie 3: Sortie des résultats
 9      si (nb = 2)
10         alors  ecrire (n, "est premier") ;
11         sinon ecrire (n, "n'est pas premier") ;
       fin si ;
Fin. //fin de l'algorithme Primalité_1
```

(Les instructions sont numérotées de 1 à 11)

Commentaire :

Dans cet algorithme, on peut diminuer le nombre de parcours de la boucle tant que (instruction n° 5). En effet, il ne sert à rien de parcourir tous les diviseurs potentiels i *de* 1 *à* n puisque le plus grand diviseur de n est au plus égal à $n/2$ (sauf pour $i=n$). On peut aussi arrêter le parcours de la boucle si on rencontre un diviseur autre que 1 *et* n car, dans ce cas, n n'est pas un nombre premier. De plus, on peut éviter de tester les nombres pairs puisqu'ils ne sont pas premiers.

En fait, on peut développer plusieurs algorithmes pour le test de la primalité ; mais ici on prend cet algorithme tel qu'il est proposé car ce qui nous importe le plus est le calcul de sa complexité. L'objectif est avant tout pédagogique. On propose justement 2 autres algorithmes dans les parties 2 et 3 de ce TP.

2.1- Calculer les complexités temporelles de cet algorithme au pire cas, notée f1(n), et au meilleur cas, notée f2(n). (cette question est supposée être traitée en séances de TD, sinon la laisser pour les séances ultérieures).

Solution :

Rappel : La complexité temporelle d'un algorithme désigne la mesure du temps nécessaire pour l'exécution de cet algorithme. Elle est donnée sous la forme d'une fonction dont les variables sont les données en entrée de l'algorithme. Cette fonction peut être une expression mathématique exacte mais cette forme est rarement utilisée. Elle peut être aussi une expression mathématique en ordre de grandeur. Cette deuxième forme est la forme couramment utilisée et elle est donnée en notation asymptotique appelée aussi notation Landau.

Comme le temps d'exécution d'un algorithme est proportionnel au nombre d'instruction exécutées par cet algorithme (appelé aussi somme des fréquences d'exécution des instructions), on se ramène dans la pratique au calcul de ce dernier nombre.

On montre, dans ce qui suit, **le calcul pratique de la complexité temporelle d'un algorithme itératif**. On utilise les règles suivantes :

1- La complexité temporelle est calculée en fonction des données en entrée de l'algorithme.
2- Par ordre de grandeur, il faut comprendre la classe de complexité O(n), O(n^2), O(n^3), O($\sqrt{n}$), O(log(n)), O(nlog(n)), O(a^n), etc.
3- Pour calculer la complexité temporelle d'un algorithme, on doit :
   a) calculer les fréquences d'exécutions de chacune des instructions de l'algorithme.
   b) Dans le cas d'une instruction conditionnelle complète ayant les 2 branches alors et sinon, on doit calculer la fréquence de la condition de l'instruction conditionnelle et le maximum des fréquences entre ses 2 branches dans le pire cas et le minimum des fréquences dans le meilleur cas (car une seule branche est exécutée). On calcule ensuite la somme de ces 2 fréquences.
   c) Dans le cas d'une instruction de répétition (pour, tant que ou répéter), on doit calculer la fréquence de la condition de l'instruction de la répétition et les fréquences des instructions contenues dans le corps de l'instruction de la répétition. On calcule ensuite la somme de ces 2 fréquences.
   d) Calculer la somme des fréquences de toutes les instructions de l'algorithme.
4- Généralement, le calcul de la complexité temporelle exacte n'est pas possible, et on se restreint alors au calcul des complexités au pire cas, au moyen cas et au meilleur cas.
5- Les instructions exécutables correspondent aux opérations exécutables par le processeur de manière indivisible :
   - les opérations arithmétiques (+, -, *, /) ;
   - les opérations de condition (==, !=, <, <=, >, >=) ;
   - les opérations d'entrée/sortie (lire(), ecrire()).
6- Les fonctions prédéfinies comme sin, cos, log, modulo, etc. sont supposées se comporter comme des opérations exécutables par le processeur de manière indivisible (on admet cette hypothèse car elle ne change pas la complexité temporelle de l'algorithme).
7- On suppose que toutes les instructions prennent la même durée de temps d'exécution, ce qui permet de faire l'addition de leurs fréquences d'exécutions (là aussi, cette hypothèse ne change pas la complexité temporelle de l'algorithme).
8- Le meilleur cas d'un algorithme correspond au nombre minimal du nombre d'instructions exécutées par cet algorithme alors que le pire cas correspond au nombre maximal du nombre d'instructions exécutées.

Remarque 1 :
La règle n°8 (ci-dessus) s'applique pour cet algorithme comme suit : le pire cas de cet algorithme correspond aux nombres n qui ne sont pas premiers et le meilleur cas correspond aux nombres N qui sont premiers. Cela peut paraître étrange (ie., non habituel) car, d'habitude, dans les autres algorithmes du test de la primalité, c'est la détermination des nombres premiers qui correspond au pire cas. En fait, cette caractéristique est incluse dans cet algorithme dans un but pédagogique d'illustration des notions du pire cas et du meilleur cas.

Soit L le nombre d'instructions de l'algorithme Primalité_1 (ci-dessus) et soit F la somme des fréquences d'exécutions de toutes les instructions de l'algorithme.

On a: L = 11 instructions.

Pour calculer F, on utilise les fréquences d'exécutions f_i de chacune des L=11 instructions de l'algorithme Primalité_1 comme il est montré sur le tableau suivant :

N°	Instruction I_i		Fréquence d'exécution f_i	
1	ecrire('Donner un nombre n : ') ;	→	1	
2	Lire(n) ;	→	1	
3	nb := 0 ;	→	1	
4	i := 1 ;	→	1	
5	tant que (i<=n) faire	→	(n+1)	
6	si (n mod i = 0)	→	2n	//(2 opérations : mod et =)[1]
7	alors nb := nb + 1 ;	→	2*2	au meilleur cas[2] //(2 opérations : + et :=)[1]
			2*(n/2 + 1)	au pire cas[3] //(2 opérations : + et :=)[1]
8	i := i + 1 ;	→	2n	//(2 opérations : + et :=)[1]
9	si (nb = 2) ;	→	1	
10	alors ecrire (N, "est premier") ;	→	1	au meilleur cas[4]
			∄	car le pire cas correspond au sinon de la condition[4]
11	sinon ecrire (N, "n'est pas premier") ;	→	∄	car le meilleur cas correspond au alors de la condition[4]
			1	au pire cas[4]

Figure 2. Tableau des fréquences d'exécutions des instructions de l'algorithme Primalité_1

Remarque 2 : Les notes en exposant dans les fréquences f_i sont explicitées dans ce qui suit :

- Note 1 : L'instruction I_6 est composée de 2 opérations exécutables de façon indivisible : l'opération reste de la division entière (mod) et l'opération du test d'égalité (=). La fréquence d'exécution est alors comptée 2 fois : $f_6 = 2*n$. On a des remarques similaires pour les instructions I_7 et I_8.

- Note 2 : Un nombre n premier possède par définition 2 diviseurs : 1 et le nombre n lui-même. Cela correspond au meilleur cas de l'algorithme parce que l'instruction I_7 est exécutée un nombre minimum de fois, soit 2 fois (c'est le nombre de fois que la condition de l'instruction I_6 est vraie). La fréquence de l'instruction est alors : $f_7 = 2*2$.

- Note 3 : Comme les diviseurs d'un nombre n sont par définition (<=n/2), et en supposant qu'ils le sont tous, et sachant que n est diviseur de lui-même, alors le nombre de diviseurs est au maximum (n/2 + 1). Cela correspond au pire cas de l'algorithme parce que l'instruction I_7 est exécutée un nombre maximum de fois, soit (n/2 + 1) (c'est le nombre de fois que la condition de l'instruction I_6 serait vraie). La fréquence de l'instruction est alors : $f_7 = 2* (n/2 + 1)$.

- Note 4 : L'instruction I_{10} est exécutée 1 fois car nb=2, et cela correspond au meilleur cas vu que n est premier. Le pire cas (correspondant à nb≠2 et donc N n'est pas premier) ne peut pas exister et il est pris en compte par la branche sinon (instruction I_{11}). L'instruction I_{11} se comporte comme le contraire de l'instruction I_{10}.

Du fait qu'il existe dans l'algorithme un meilleur cas et un pire cas, la somme des fréquences d'exécutions des instructions de l'algorithme est scindée en 2 cas : la somme les fréquences au pire cas notée $f1(n)$ et la somme les fréquences au meilleur cas notée $f2(n)$.

1- Au pire cas :

$f1(n)=\sum_{i=1}^{L=11}(f_i)+f_{11}f_{10}$ *n' existe pas car on est au pire cas*

¿ $1+1+1+1+(n+1)+2n+2\left(\frac{n}{2}+1\right)+2n+1+1$

¿ $\ldots=6n+9$

$f1(n)$ *est une fonction linéaire en n du type :* $f1(n)=an+b \Rightarrow f1(n)=O(n)$

En conséquence, la complexité temporelle de l'algorithme Primalité_1 est au pire cas :

1- En notation exacte : $f1(n)=6n+9$ *instructions exécutées*
2- En notation asymptotique : $f1(n)=O(n)$ *instructions exécutées*

2- Au meilleur cas :

$f2(n)=\sum_{i=1}^{L=11}(f_i)+f_{10}f_{11}$ *n' existe pas car on est au meilleur cas*

¿ $1+1+1+1+(n+1)+2n+2(2)+2n+1+1$

¿ $\ldots=5n+11$

$f2(n)$ *est une fonction linéaire en n du type :* $f2(n)=an+b \Rightarrow f2(n)=O(n)$

En conséquence, la complexité temporelle de l'algorithme Primalité_1 est au meilleur cas :

1- En notation exacte : $f2(n)=5n+11$
2- En notation asymptotique : $f2(n)=O(n)$

Remarque 3 :
La complexité temporelle qu'on vient de calculer est relative au nombre d'instructions exécutées par l'algorithme. Elle ne diffère de la complexité temporelle relative au temps d'exécution de l'algorithme que par un facteur multiplicatif. Donc, en notation asymptotique, ces 2 complexités sont égales. On peut montrer ce résultat comme suit :

On a supposé l'hypothèse (règle n°7 ci-dessus) : toutes les instructions prennent la même durée de temps d'exécution. Soit Δt cette durée.

On a au meilleur cas : $f2(n)=(5n+11)$ *instructions exécutées*

Il s'ensuit que le temps d'exécution au meilleur cas est :

$T2(n)=(f2(n)*\Delta t)$ *unités de temps*
$T2(n)=((5n+11)*\Delta t)$ *unités de temps*
¿ $(5\Delta t\, n+11\Delta t)$ *unités de temps*
¿ $(5\Delta t\, n+11\Delta t)$ *unités de temps*
¿ $((5\Delta t)n+(11\Delta t))$ *unités de temps* ¿ $(an+b)$ *unités de temps*

où : $a=5\Delta t, b=11\ \Delta t$
et Δt est une valeur fixe pour un ordinteur donné

Donc :

$$T2(n) \text{ est une fonction linéaire en } n \text{ du type} : T2(n)=an+b \Rightarrow T2(n)=O(n)$$

Le même raisonnement peut être fait dans le pire cas.

En conclusion, on peut énoncer ce résultat comme suit :

Théorème :
La complexité temporelle relative au nombre d'instructions exécutées par l'algorithme et la complexité temporelle relative au temps d'exécution de l'algorithme sont égales en notation asymptotique.

2.2- Calculer la complexité spatiale de cet algorithme notée s(n). (là aussi, on a la même remarque que celle de la question 2.1).

Solution :
Rappel : La complexité spatiale d'un algorithme désigne la mesure de l'espace mémoire nécessaire pour enregistrer en mémoire centrale les instructions et les données de cet algorithme lors de son exécution. Elle est donnée sous la forme d'une fonction dont les variables sont les données en entrée de l'algorithme. Cette fonction peut être une expression mathématique exacte mais cette forme est rarement utilisée. Elle peut être aussi une expression mathématique en ordre de grandeur. Cette deuxième forme est la forme couramment utilisée et elle est donnée en notation asymptotique appelée aussi notation Landau.

Les données peuvent être de type statique, c'est-à-dire les données dont l'allocation de la mémoire est effectuée **avant l'exécution du programme**, ou de type dynamique, c'est-à-dire les données dont l'allocation de la mémoire est effectuée **en cours d'exécution du programme**. On retrouve ce deuxième type de données avec les structures de données dynamiques (dont l'allocation de la mémoire s'effectue avec l'opérateur new dan le langage Pascal et avec l'opérateur malloc dans le langage C). On le retrouve aussi avec les programmes récursifs où chaque appel d'une fonction récursive génère une allocation de la mémoire pour les données déclarées dans cette fonction. On doit noter que les instructions de la fonction récursive ne sont allouées qu'une seule fois.

On montre, dans ce qui suit, **le calcul pratique de la complexité spatiale d'un algorithme itératif ou récursif**. On utilise les règles suivantes :

1- La complexité spatiale est calculée en fonction des données en entrée de l'algorithme.
2- Par ordre de grandeur, il faut comprendre la classe de complexité O(N), $O(N^2)$, $O(N^3)$, $O(\sqrt{n})$, O(log(N)), O(Nlog(N)), $O(a^N)$, etc.
3- Généralement, le calcul de la complexité spatiale des algorithmes itératifs ne pose pas de problèmes.
4- Avec les algorithmes récursifs, on doit tenir compte de chaque appel récursif d'une fonction qui génère de façon dynamique, et donc de façon transparente au programmeur, l'allocation d'un espace mémoire pour sauvegarder les paramètres d'appels de la fonction et des variables locales de la fonction. Dans la pratique, on calcule le nombre d'appels récursifs et on le multiplie par la taille de l'espace mémoire occupé par les paramètres d'appels et les variables locales de la fonction.

5- L'unité de mesure de la complexité spatiale est le mot mémoire qui est un nombre multiple d'octets, mais généralement en puissance de 2 (1, 2, 4 ou 8 octets). On suppose qu'une instruction occupe 1 mot mémoire et qu'une donnée de type scalaire (entier, réel, logique ou caractère) occupe aussi un mot mémoire.

Soit L le nombre d'instructions de l'algorithme Somme_1 (ci-dessus) et soit α l'espace mémoire occupé par ces instructions.

On a: $L=11\,instructions$.
Comme 1 instruction occupe 1 mot mémoire (par hypothèse),
alors : $\alpha=11\,mots\,mémoires$.

Soit D le nombre de données de l'algorithme Primalité_1 (ci-dessus) et soit β l'espace mémoire occupé par ces données.

On a: 3 variables scalaires N, S et i de type entier et/ou réel. Donc: $D=3\,données$.
Comme 1 donnée occupe 1 mot mémoire (par hypothèse), alors $\beta=3\,mots\,mémoires$.

Soit s l'espace mémoire occupé par les instructions et les données de l'algorithme Primalité_1.

On a : $s(n)=\alpha+\beta=(11+3)\,mots\,m\acute{e}\,moires=14\,mots\,m\acute{e}\,moires$.

$s(n)\,est\,une\,fonction\,constante\,en\,N\,du\,type: s(n)=a\Rightarrow s(n)=O(1)$

En conséquence, la complexité spatiale $s(n)$ de l'algorithme Primalité_1est :

1- En notation exacte : $s(n)=14\,mots\,m\acute{e}\,moires$
2- En notation asymptotique : $s(n)=O(1)\,mots\,m\acute{e}\,moires$

3- Développer le programme correspondant avec le langage C.

Solution :
Le programme est une traduction directe de l'algorithme. Il est présenté sur la figure 3.

```
//Primalité_1
//Série TP n°3, Partie 1 (programme 1 du test de la primalité)
//archive: c:\bc45\bin\s3_1 (s1_ex1_1 (série 3, partie 1))

//Ce programme vérifie (teste) si un nombre n est premier.
//Le nombre n est donné en entrée.
//Ce programme est itératif.
//Ce programme fait une itération de 1 à n;
//Ce programme ne génère pas (ne cherche pas) des nombres premiers mais on peut générer

#include <stdlib.h>  //La bibliothèque des fonctions générales.
#include <stdio.h>   //La bibliothèque des fonctions des entrées/sorties.

int main()                  //programme principal
{ long int n, i, reste ;    //long int désigne le type entier de taille double, soit 4 octets (32 bits),
                            //on peut vérifier cela avec l'instruction sizeof().
  long int nb ;             //La variable nb compte le nombre de diviseurs de n.

  //Partie 1: Entrée des données
  printf("Donner un nombre: n= ") ;
  scanf("%ld", &n) ;

  //Partie 2: Traitement des données
  nb =0 ;
  i =1 ;
  while (i<=n)
  { If (n % i ==0)   //% désigne la fonction modulo et == désigne l'opération du test d'égalité.
     //then        (On rappelle que le langage C utilise le then implicitement sans le déclarer)
     {
       nb =nb + 1 ;       //On compte les diviseurs de n
      }//fin du then
         i=i + 1 ;        // tester le diviseur éventuel suivant
  }//fin du while

  //Partie 3: Sortie des résultats
  if (nb == 2)
    printf("\n     %ld est un nombre premier\n", n) ;
    else p printf("          %ld n'est pas un nombre premier\n", n) ;

  return(0);
}//fin du programme
```

Figure 3. Programme du test de la primalité (appelé Primalité_1) en C

4- Vérifier par programme si les nombres n donnés dans le tableau ci-dessous (1.000.003, 2.000.003, …) sont premiers. On doit remarquer que comme les nombres n varient dans l'intervalle 1 million à environ 2 milliards, le nombre de tests (le reste de la division de n par i) varie aussi dans le même intervalle.

Solution :

On vérifie aisément avec le programme précédent que tous les nombres de l'échantillon sont premiers.

5- Mesurer les temps d'exécution T pour ces nombres n et compléter le tableau ci-dessous.

Ind : Pour mesurer le temps d'exécution d'un programme avec le langage C, on utilise les fonctions de gestion du temps qui sont fournies dans la bibliothèque time.h (inclure l'instruction : #include <time.h>).

n	1.000.003	2.000.003	4.000.037	8.000.009	16.000.057	32.000.011	64.000.031
T							

n	128.000.003	256.000.001	512.000.009	1.024.000.009	2.048.000.011
T					

Solution :

La mesure du temps avec le langage C fait appel aux fonctions de la gestion du temps qui sont fournies dans la bibliothèque time.h. On suit les étapes suivantes :

1- Utiliser la bibliothèque des fonctions de gestion du temps ;
   Pour cela, inclure dans le programme la directive : #include <time.h>.
2- Définir 2 variables notées t1 et t2 de type clock_t qui désigne le temps comme suit :
   clock_t t1, t2 ;
3- Mesurer le temps avec la fonction clock() comme suit :
   t1 = clock() ; //au début du code à mesurer
   t2 = clock() ; //à la fin du code à mesurer
4- Calculer le temps d'exécution du code inclus entre les points de mesure t1 et t2 avec la formule suivante :
   float delta = (float) (t2-t1)/CLOCKS_PER_SEC ;

Rem : Le temps d'exécution delta est donné en secondes. Le paramètre CLOCKS_PER_SEC est inclus dans la bibliothèque des fonctions de gestion du temps time.h.

Le programme incluant les instructions de la mesure du temps d'exécution du programme est présenté sur la figure 4.

```
//Primalité_1
//… inchangé
#include <time.h>    //La bibliothèque des fonctions de gestion du temps.

int main()                  //programme principal
{ long int N, i, reste ;    //long int désigne le type entier de taille double, soit 4 octets (32 bits),
                            //on peut vérifier cela avec l'instruction sizeof().
  long int nb ;             //La variable nb compte le nombre de diviseurs de N.
  clock_t  t1, t2;          //clock_t désigne le type temps.

  //Partie 1: Entrée des données
  printf("Donner un nombre: N= ") ;
  scanf("%ld", &N) ;

  //Partie 2: Traitement des données
 t1 = clock() ;       //La variable t1 reçoit la valeur du temps fournie par la fonction clock().
                      //C'est le début de la mesure du temps.
  nb =0 ;
  i =1 ;
  while (i<=N)
  { if (N % i ==0)  //% désigne la fonction modulo et == désigne l'opération du test d'égalité.
      //then        (On rappelle que le langage C utilise le then implicitement sans le déclarer)
      {
        nb =nb + 1 ;        //On compte les diviseurs de N.
       }//fin du then
          i=i + 1 ;         // tester le diviseur éventuel suivant
  }//fin du while
  t2 = clock() ;       //La variable t2 reçoit la valeur du temps fournie par la fonction clock().
                       //C'est la fin de la mesure du temps.

  float delta=(float)(t2-t1) / CLOCKS_PER_SEC ;
                    //Cette formule permet de calculer le temps d'exécution du code inclu
                    //entre les 2 points t1 et t2. (float) effectue une conversion du type clock_t()
                    //vers le type float.

  //Partie 3: Sortie des résultats
  if (nb == 2)
    printf("\n     %ld est un nombre premier\n", N) ;
    else printf("  %ld n'est pas un nombre premier\n", N) ;

  printf("\n       Le temps d'exécution est delta = %f secondes", delta)) ;
  return(0) ;
}//fin du programme
```

Figure 4. Programme du test de la primalité (appelé Primalité_1) avec les fonctions de la mesure du temps d'exécution

Les mesures sont effectuées avec un micro ordinateur PC portable ayant les caractéristiques suivantes :

- Pentium Dual Core T2390, fréquence 1.86 GHz
- RAM 2Go

Les mesures obtenues avec le programme Primalité_1 de la figure 5 sont :

N	1.000.003	2.000.003	4.000.037	8.000.009	16.000.057	32.000.011	64.000.031
T(s)	0.22	0.44	0.93	1.86	3.79	7.47	14.94

N	128.000.003	256.000.001	512.000.009	1024.000.009	2048.000.011
T(s)	29.94	59.91	119.79	239.52	479.17

Figure 5. Tableau des mesures du temps d'exécution obtenues avec le programme Primalité_1.

6- Développer un programme de mesure du temps d'exécution du programme qui a en entrée les données de l'échantillon ci-dessus et en sortie les temps d'exécution. Les données et les mesures du temps sont à enregistrer dans des tableaux notés respectivement Tab1 et Tab2.

Solution :

Dans le programme précédent, on ajoute les déclarations des tableaux Tab1 et Tab2 et on ajoute aussi l'instruction de répétition qui permet de répéter l'exécution du programme pour les 12 données n de l'échantillon. Le programme est présenté sur la figure 6.

```
//Primalité_1
//… inchangé

#include <stdlib.h>  //La bibliothèque des fonctions générales.
#include <stdio.h>   //La bibliothèque des fonctions des entrées/sorties.
#include <time.h>    //La bibliothèque des fonctions de gestion du temps.

int main()                  //programme principal
{ long int n, i, j, reste ;  //long int désigne le type entier de taille double, soit 4 octets (32 bits),
                             //on peut vérifier cela avec l'instruction sizeof().
  long int nb ;              //La variable nb compte le nombre de diviseurs de n.
  clock_t  t1, t2;           //clock_t désigne le type temps.
  int Tab1[12] ;             //pour les 12 données n de l'échantillon
  float   Tab2[12] ;         //pour les 12 mesures du temps d'exécution du programme

  //Partie 1: Entrée des données
  for (j=0 ; j<12 ; i++)
  { printf("Donner le %ld ème nombre %ld : n= ", j) ;
    scanf("%ld", &Tab1[j]) ;
  }

  //Partie 2: Traitement des données


  j = 0 ;
 while (j< 12)
 {
```

```
    t1 = clock() ;       //La variable t1 reçoit la valeur du temps fournie par la fonction clock().
                         //C'est le début de la mesure du temps.
    nb =0 ;
    i =1 ;
    while (i<= Tab1[j])
    { if (Tab1[j] % i ==0)          //% désigne la fonction modulo
                                    //== désigne l'opération du test d'égalité.
      //then        (On rappelle que le langage C utilise le then implicitement sans le déclarer)
       {
          nb =nb + 1 ;      //On compte les diviseurs de Tab1[j].
       }//fin du then
          i=i + 1 ;          // tester le diviseur éventuel suivant
    }//fin du 2ème while
    t2 = clock() ;       //La variable t2 reçoit la valeur du temps fournie par la fonction clock().
                         //C'est la fin de la mesure du temps.

    float delta=(float)(t2-t1) / CLOCKS_PER_SEC ;
          //Cette formule permet de calculer le temps d'exécution du code inclu
          //entre les 2 points t1 et t2. (float) effectue une conversion du type clock_t()
          //vers le type float.
    Tab2[j] = delta;
    j = j + 1;          //passer au nombre suivant de l'échantillon
  }//fin du 1er while

   //Partie 3: Sortie des résultats
  for (j=0 ; j<12 ; i++)
  {    printf("\n  %ld est un nombre premier\n", Tab1[j] ) ;
       printf("\n  Le temps d'exécution est delta = %f secondes", Tab2[j] ) ;
  }

   return(0) ;
}//fin du programme
```

Figure 6. Programme de mesure du temps d'exécution du test de la primalité (appelé Primalité_1) pour un échantillon de données

Rem : Ce programme ne diffère du programme précédent que dans la répétition des mesures. Ainsi, on n'aura pas à refaire l'exécution du programme pour chaque donnée. De plus, les données et les mesures obtenues sont enregistrées ce qui permet d'éviter de les ressaisir à chaque fois.

7- Représenter par un graphe, noté $G_{f2}(n)$, les variations de la fonction de la complexité au meilleur cas f2(n) en fonction de n ; et par un autre graphe, noté $G_T(n)$, les variations du temps d'exécution T(n) en fonction de n. Utiliser pour cela un logiciel graphique tel que excel.

Solution :

Pour représenter le graphe expérimental $G_T(n)$, on reporte les mesures obtenues avec le programme de mesure du temps d'exécution du programme Primalité_1 (question 5° ci-dessus) sur un graphe. Les nombres n sont représentés sur l'axe des abscisses et les temps T sont représentés sur l'axe des ordonnées. On obtient alors un graphe $G_T(n)$ constitué par un ensemble de points. Il est représenté sur la figure 7 (le temps T est noté T1).

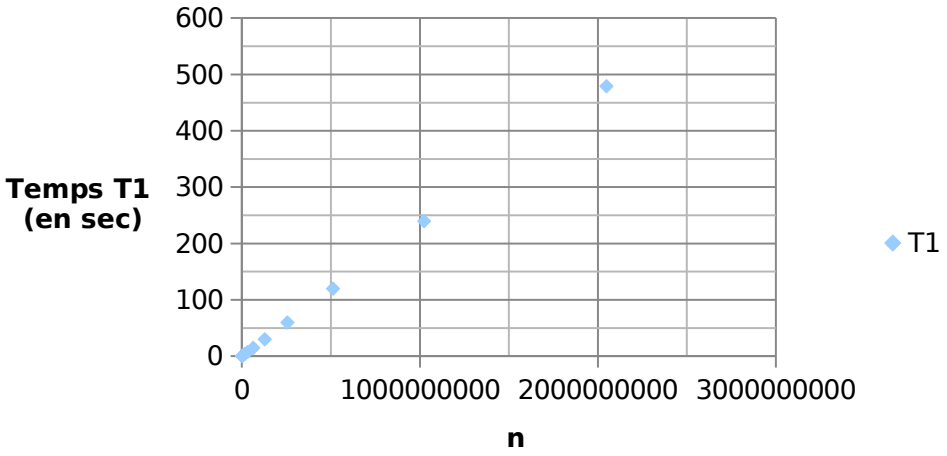

Figure 7. Graphe expérimental des mesures du temps sous la forme d'un ensemble de points

A partir de la disposition géométrique de ces points, on peut parfois (mais pas toujours) déduire la nature du graphe qui approche au mieux le graphe $G_T(n)$. Est-ce une droite, une parabole, une hyperbole, une sinusoïde, etc. ? Cela permet alors de connaître expérimentalement le comportement ou l'évolution du temps d'exécution du programme en fonction des données en entrées n. Pour l'échantillon considéré, la droite semble être le graphe qui représente le mieux l'ensemble des points $G_T(n)$. Le nouveau graphe $G_T(n)$ approché par une droite est représenté sur la figure 8.

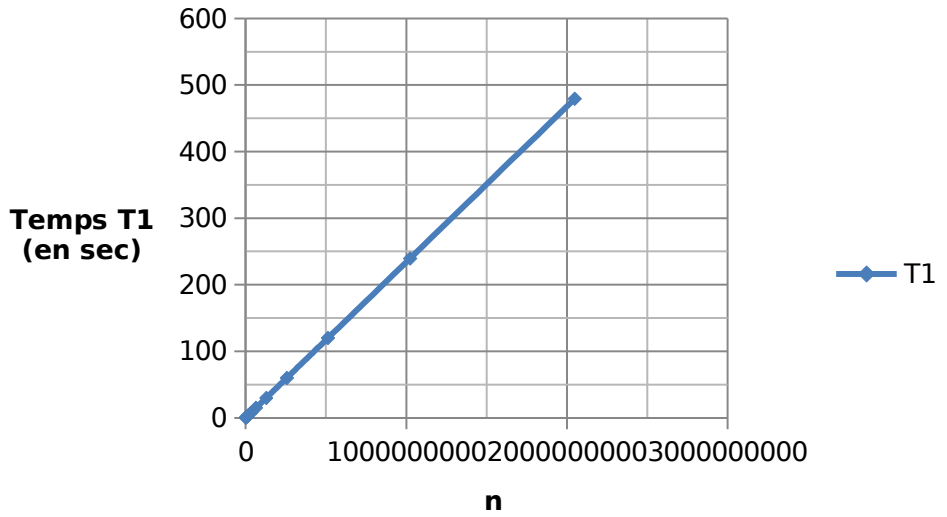

Figure 8. Graphe expérimental des mesures du temps approché par une droite

Cette droite peut ne pas passer par tous les points mesurés, mais cela n'est pas gênant car les mesures expérimentales sont obtenues dans des conditions qui ne pas parfaites. En fait, l'exécution d'un programme est faite en concurrence avec d'autres programmes du système

d'exploitation. Ceci peut générer des interruptions et donc des pertes de temps lors des commutations (ou des changements) de contexte. Les mesures expérimentales peuvent alors inclure des temps additifs qui altèrent (ou modifient) les temps réels d'exécution. Mais en général, l'effet de ces fluctuations sur les temps des mesures doit diminuer avec l'augmentation de la taille des données.

Pour représenter le graphe théorique $G_{f2}(N)$ de la fonction de la complexité au meilleur cas : $f2(n)=5n+11$, on procède comme suit :

1- Détermination de Δt à partir des mesures obtenues ;
2- Calculer le temps $T2(n)$ défini par : $T2(n)=f2(n)*\Delta t$

Où : Δt la durée d'exécution d'une instruction quelconque. (voir ci-dessus la remarque 3 de la question 2° pour plus de détails)

On détermine Δt à partir du tableau des mesures de la question 5 comme suit :

Pour n = 2048.000.011, le nombre d'instructions exécutées par l'algorithme est donné par :

$$f2(n)=5n+11=5*2048.000.011+11=10.240.000.066 \text{ instructions.}$$

On rappelle que ce nombre est premier et donc il correspond à l'exécution au meilleur cas de l'algorithme.
Comme $T(n=2048.000.011)=479,17\,s$,
on a alors : $\Delta t=479,17/10.240.000.066=46,79*10^{-9}\,s$.
(car : $10.240.000.066\,instr\;479,17\,s$

$$1\,instr\;\Delta t=479,17/10.240.000.066=46,79*10^{-9}\,s$$

La valeur Δt ainsi calculée est une approximation du temps d'exécution moyen des instructions du programme. On dit bien **"moyen"** car toutes les instructions n'ont pas la même durée d'exécution. Néanmoins, c'est une très bonne approximation et qui est meilleure que celle que l'on peut prendre à partir du manuel des instructions du microprocesseur équipant l'ordinateur utilisé. En effet, le programme n'utilise pas toutes les instructions de l'ensemble des instructions du microprocesseur.

Pour chacun des nombres n, on calcule le temps $T2(n)$:

$$T2(n)=f2(n)*\Delta t=(5N+11)*\Delta t=(5N+11)*46,79*10^{-9}$$

On regroupe ces données dans le tableau suivant (figure 9) :

n	1.000.003	2.000.003	4.000.037	8.000.009	16.000.057	32.000.011	64.000.031
T(s)	0.22	0.44	0.93	1.86	3.79	7.47	14.94

n	128.000.003	256.000.001	512.000.009	1024.000.009	2048.000.011
T(s)	29.94	59.91	119.79	239.52	479.17

Figure 9. Tableau des mesures du temps d'exécution théorique.

On les reporte sur le graphe théorique $G_{f2}(N)$ qui est représenté sur la figure 10 (le temps est noté T2).

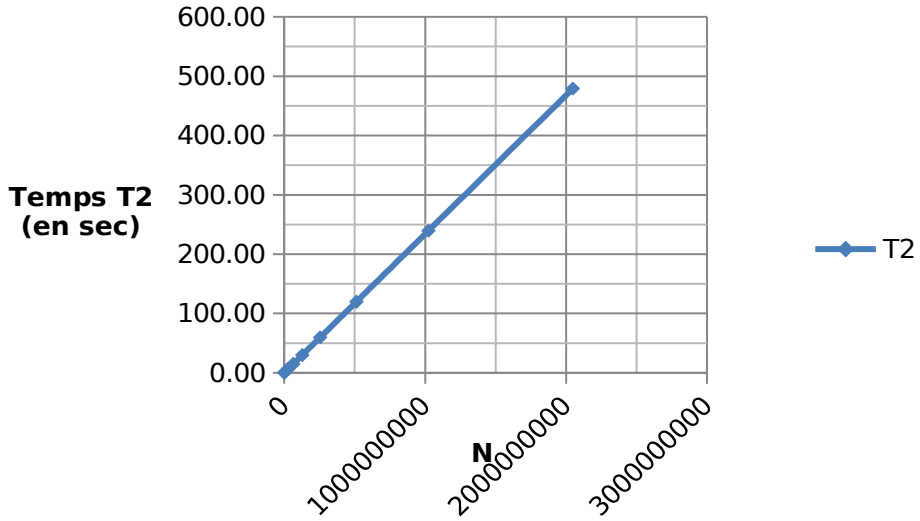

Figure 10. Graphe théorique des mesures du temps approché par une droite.

8. Interprétation des résultats.

8.a- Les mesures du temps obtenues correspondent-elles au meilleur cas ou au pire cas ?

Solution :

Les mesures obtenues expérimentalement correspondent au meilleur cas car chacun des nombres de l'échantillon est un nombre premier et on a vu ci-dessus que la détermination d'un nombre premier par l'algorithme proposé Primalité_1 correspond au meilleur cas et la détermination d'un nombre non premier correspond au pire cas (voir question 2° ci-dessus).

8.b- Que remarque-t-on sur les données de l'échantillon et sur les mesures obtenues ? Peut-on déduire, même de façon approximative, une fonction T(n) reliant T et n ; c'est-à-dire une fonction T(n) permettant de déterminer directement la valeur de T à partir de n.

Ind: comparer chaque nombre n avec le suivant ; et chaque mesure du temps avec la suivante.

Solution :

On constate sur le tableau des mesures obtenues expérimentalement avec l'échantillon des données que pour chaque couple de données (1.000.003, 2.000.003), (4.000.037, 8.000.009), (16.000.057, 32.000.016), etc., la valeur de la donnée n est approximativement doublée. De même, la valeur mesurée du temps T est approximativement doublée : (0.22, 0.44), (0.93, 1.86), (3.79, 7.47), etc.

Les mesures obtenues sont (voir la figure 5 ci-dessous qu'on représente ici une 2ème fois) :

N	1.000.003	2.000.003	4.000.037	8.000.009	16.000.057	32.000.011	64.000.031
T(s)	0.22	0.44	0.93	1.86	3.79	7.47	14.94

N	128.000.003	256.000.001	512.000.009	1024.000.009	2048.000.011
T(s)	29.94	59.91	119.79	239.52	479.17

Figure 5. Tableau des mesures du temps d'exécution obtenues avec le programme Primalité_1.

On peut donc déduire à partir de ces mesures que le temps d'exécution est doublé quand la valeur de n est doublée, ce qu'on représente formellement par :

$$T(2*n)=2*T(n)\,\forall n\in[10^6,2*10^{11}] \qquad (1)$$

On constate aussi sur le tableau des mesures :

$T(4000037)=0.93\approx 4*T(1000003)=4*0.22=0.88;$
$T(8000009)=1.86\approx 8*T(1000003)=8*0.22=1.76;$
$T(32000011)=7.47\approx 16*T(2000003)=16*0.44=7.04;$
$T(64000031)=14.94\approx 4*T(16000057)=4*3.79=14.16;$ etc.

On déduit alors que le temps d'exécution est proportionnel à n, ce qu'on représente formellement par :

$$T(k*n)=k*T(n)\,\forall(k*n)\in[1000003,2048000011],n,k\in N \qquad (2)$$

On réécrit la relation (2) comme suit :

$$T(k*n)=k*T(n)\,\forall(k*n)\in[10^6,2*10^{11}],n,k\in N$$
$$\Leftrightarrow T(k*n)=n*T(k)\,\forall(k*n)\in[10^6,2*10^{11}],n,k\in N$$

Pour $k=1000003, T(1000003*n)=n*T(1000003)$

$$\forall(1000003*n)\in[1000003,2048000011]$$
$$\Leftrightarrow T(n)=n*T(2)\,\forall(2*n)\in[10^6,2*10^{11}] \qquad (3)$$

où : $T(2)$ représente le temps nécessaire pour calculer la somme de 2 élément.
On note : $\gamma=T(2)$.

On conclut :

$$T(n)=n*T(2)=n*\gamma\,\forall(2*n)\in[10^6,2*10^{11}],\gamma\in R^{+¿¿}$$
$$\Leftrightarrow T(n)=\gamma*n\,\forall(2*n)\in[10^6,2*10^{11}],\gamma\in R^{+¿¿} \qquad (4)$$

La relation (4) permet de déterminer directement la valeur de $T(n)$ à partir de la valeur de n.

Remarque : La relation (4) traduit le graphe d'une droite. On a déjà établit ce résultat dans la question 7° ci-dessus.

En général, il est préférable d'utiliser une autre méthode largement répandue dans les sciences expérimentales. Elle consiste en la représentation graphique des phénomènes (càd, des fonctions représentant ces phénomènes) à étudier. Dans notre cas, on établit le tracé graphique des variations du temps d'exécution $T(n)$ en fonction des valeurs de n. A partir de la

disposition géométrique des points de ce tracé graphique, on déduit la nature du graphe qui approche au mieux le graphe de la fonction $T(n)$: une droite, une parabole, une hyperbole, une sinusoïde, etc., et, ensuite, on détermine l'expression de la fonction $T(n)$.

8.c- Comparer entre la complexités théorique et la complexité expérimentale (çàd., les mesures expérimentales). Les prédictions théoriques sont-elles compatibles avec les mesures expérimentales ?

Solution :

On trace sur une même figure les 2 graphes expérimental $G_T(N)$ et théorique $G_f(N)$ (figure 9).

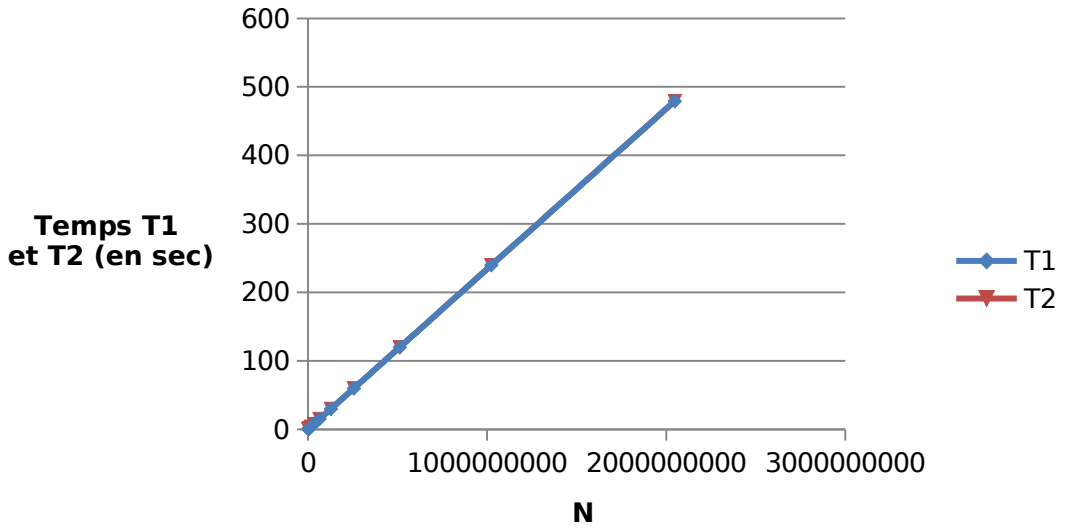

Figure 9. Graphes expérimental et théorique des mesures du temps approchés par 2 droites

(En bleu: graphe expérimental, en rouge: graphe théorique)

On remarque qu'il y a superposition des 2 graphes. Ceci était prévisible au vu des différences très faibles entre les valeurs expérimentales et les valeurs théoriques.

Au vu du tracé des 2 graphes $G_T(N)$ et $G_f(N)$, il existe une compatibilité entre la complexité expérimentale et théorique.

Partie 2

1- Algorithme 2 du test de la primalité

Dans ce 2ème algorithme du test de la primalité, on reprend la même structure de l'algorithme 1 pour pouvoir faire une comparaison cohérente avec l'algorithme 1. On parcourt les diviseurs potentiels i du nombre N de 1 jusqu'à N/2. On change N par N/2 dans l'instruction I_5.

Le reste de l'analyse est similaire à la partie 1.

Partie 3

1- Algorithme 3 du test de la primalité

Dans ce 3ème algorithme du test de la primalité, on reprend la même structure de l'algorithme 1 pour pouvoir faire une comparaison cohérente avec l'algorithme 1. On parcourt les diviseurs potentiels i du nombre N de 1 jusqu'à $\leq N^{1/2}$ ($=\sqrt{N}$). On change N par $\sqrt{N}$N/2 dans l'instruction I_5.

Le reste de l'analyse est similaire à la partie 1.