Série de Travaux Pratiques n° 1 (TP n° 1)

Mesure du temps d'exécution d'un programme

L'objet de ce TP est d'apprendre à mesurer le temps d'exécution d'un programme. On utilise les fonctions de gestion du temps du langage de programmation C qui sont incluses dans la bibliothèque time.h. On prend comme exemple d'étude le problème élémentaire du calcul de la somme des n premiers nombres entiers naturels.

Partie I : Développement de l'algorithme et du programme du problème de la somme des n premiers nombres entiers naturels

Dans cette partie, on développe un algorithme et le programme associé pour le problème du calcul de la somme des n premiers nombres entiers naturels.

1- Développer un algorithme itératif qui permet de calculer la somme, notée S, des n premiers nombres entiers naturels (n est à lire en entrée et n>=1) :

$$S=\sum_{i=1}^{n} i=1+2+3+\ldots+n$$

Utiliser les trois formes de la répétition :

a) pour … faire ;
b) tant que … faire ;
c) répéter … jusqu'à.

2- Développer les programmes itératifs correspondants avec le langage C.
Ind : Pour tester les grandes valeurs de n, on peut utiliser :

- le type entier double précision noté long int et %Ld ;
- le type réel en simple précision noté float et %f ;
- le type réel en double précision noté double et %lf.

3- Développer un algorithme récursif pour ce problème.
4- Développer le programme récursif correspondant.

Partie II : Mesure du temps d'exécution

Dans cette partie, on apprend à mesurer le temps d'exécution d'un programme en utilisant les fonctions de gestion du temps du langage de programmation C qui sont incluses dans la bibliothèque time.h.

5- Mesurer les temps d'exécution T de chacun des programmes précédents pour l'échantillon suivant des données de la variable en entrée n :

N	…	10^6	$2*10^6$	10^7	2*107	10^8	$2*10^8$	10^9	$2*10^9$

T									

N	10^{10}	$2*10^{10}$	10^{11}	$2*10^{11}$	10^{12}	$2*10^{12}$	…	…	…
T									

6- Représenter avec un graphe les variations du temps mesuré T en fonction de la variable n.
7- Déduire le temps d'exécution moyen des instructions contenues dans les programmes.
8- Rédiger un rapport décrivant le travail réalisé.

Corrigé du TP n°1

Partie I : Développement de l'algorithme et du programme du problème de la somme des n premiers nombres entiers naturels

1- Développer un algorithme itératif qui permet de calculer la somme, notée S, des n premiers nombres entiers naturels (n est à lire en entrée et n>=1) :

$$S=\sum_{i=1}^{n} i=1+2+3+\ldots+n$$

Utiliser les trois formes de la répétition :

a) pour … faire ;
b) tant que … faire ;
c) répéter … jusqu'à.

Solution :

L'algorithme de la somme des n premiers nombres entiers naturels avec n>=1 est un algorithme élémentaire. On utilise une variable, notée S, qui cumule le résultat des additions, 2 à 2, et, successives, de tous les entiers de 1 à n. on utilise pour cela une variable intermédiaire i qui prend, au fur et à mesure, les différentes valeurs : 1, 2, …, n. Au préalable, on doit initialiser S à 0.

La répétition des opérations d'addition peut s'effectuer à l'aide des 3 instructions de contrôle de la répétition disponibles dans tous les langages algorithmiques et la plupart des langages de programmation :

a- instruction de répétition pour …
b- instruction de répétition tant que …
c- instruction de répétition répéter …

L'algorithme que l'on développe, noté Somme_1, utilise la 2ème forme de la répétition : tant que … Il est présenté sur la figure 1 ci-dessous. Les 2 autres formes peuvent en être déduites aisément.

```
Algorithme Somme_1 ;          //Algorithme de la somme  des N premiers nombres entiers
                              //naturels avec n>=1
                              //Il utilise l'instruction de répétition : tant que …
Var
  n, S : réel ;
  i : réel;                   //i est déclaré en réel car on a de grandes valeurs pour n

Debut
  //Partie 1: Lecture des données
  ecrire("\nDonner la valeur n = ");
  lire(n);

  //Partie 2: Traitement
  S = 0;
  i = 1;
  tant que (i<=n) faire
  {
      S = S + i;
      i = i + 1;
  }//fin de tant que

 //Partie 3: Ecriture des résultats
  ecrire("La somme S = 1+2+...+ ", n, " =", S);
Fin. //fin de l'algorithme Somme_1
```

Figure 1. Algorithme de la somme des n premiers nombres entiers naturels avec n>=1 (noté Somme_1) et écrit en langage algorithmique.

2- Développer les programmes itératifs correspondants avec le langage C.

Ind : Pour tester les grandes valeurs de n, on peut utiliser :

- le type entier double précision noté long int et %Ld ;
- le type réel en simple précision noté float et %f ;
- le type réel en double précision noté double et %lf.

Solution :

Le programme, noté PSomme_1, est une traduction directe de l'algorithme. Il est présenté sur la figure 3 ci-dessous.

```
//PSomme_1 (Programme Somme_1)
//Ce programme, noté PSomme_1, calcule la somme  des n premiers nombres entiers naturels
//avec n>=1.
//Il utilise l'instruction de répétition : while (condition) {…}
//Le nombre n est donné en entrée. Il est déclaré de type double qui désigne le type réel en
//double précision sur 8 octets = 64 bits. L'intervalle des valeurs est:
//        [-1.8 * 10^308, -2.2 * 10^-308] U [0] U [2.2 * 10^-308, 1.8 * 10^308]

//Ce programme est itératif

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

int main()                      //programme principal
{
  double n, i, S ;

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

  //Partie 2: Traitement des données
  S = 0 ;
  i = 1 ;
  while (i<=n)
  {
       S = S + i ;
       i = i + 1 ;
  }//fin de while

  //Partie 3: Sortie des résultats
  printf("\n       La somme S = 1+2+...+%3.1e  = %lf", n, S);
                          //le format "%e" permet d'afficher en notation scientifique. Le format
                          //"3.1" permet d'afficher sur 3 positions dont la virgule et 1 position
                          //après la virgule (et il reste 1 position pour la mantisse)
  printf("\n\nEntrer une touche quelconque pour terminer le programme: ");  getchar();
  return(0);
}//fin du programme PSomme_1
```

Figure 3. Programme de la somme des n premiers nombres entiers naturels avec n>=1, noté PSomme_1, et écrit en langage C.

3- Développer un algorithme récursif pour ce problème.

Solution :

A traiter par les étudiants.

4- Développer le programme récursif correspondant.

Solution :

A traiter par les étudiants.

Partie II : Mesure du temps d'exécution

5- Mesurer les temps d'exécution T de chacun des programmes précédents pour l'échantillon suivant des données de la variable en entrée n :

n	…	10^6	$2*10^6$	10^7	2*107	10^8	$2*10^8$	10^9	$2*10^9$
T									

N	10^{10}	$2*10^{10}$	10^{11}	$2*10^{11}$	10^{12}	$2*10^{12}$	…	…	…
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 time.h. 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 type temps comme suit :
   clock_t t1, t2 ;
3- Définir une variable réelle pour lui affecter le temps d'exécution du programme :
   double delta ;
4- 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
5- Calculer le temps d'exécution du code inclus entre les points de mesure t1 et t2 avec la formule suivante :
   delta = (double) (t2-t1)/CLOCKS_PER_SEC ;

Remarque :

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, noté PSomme_2, incluant les instructions de la mesure du temps d'exécution du programme est présenté sur la figure 4 ci-dessous. Il reprend le programme précédent (figure 3) et on lui a ajouté les instructions 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

```
//PSomme_2 (Programme Somme_2)
//Ce programme, noté PSomme_3, calcule la somme  des n premiers nombres entiers naturels
//avec n>=1 ; et mesure le temps d'exécution.
//Il utilise l'instruction de répétition : while (condition) {…}
//Le nombre n est donné en entrée. Il est déclaré de type double qui désigne le type réel en
//double précision sur 8 octets = 64 bits. L'intervalle des valeurs est:
//        [-1.8 * 10^308, -2.2 * 10^-308] U [0] U [2.2 * 10^-308, 1.8 * 10^308]
//Ce programme est itératif

#include <stdlib.h>             //La bibliothèque des fonctions générales.
#include <stdio.h>              //La bibliothèque des fonctions des entrées/sorties.
#include <time.h>               //L a bibliothèque des fonctions de gestion du temps
int main()                      //programme principal
{
  double n, i, S ;
  clock_t  t1, t2;              //clock_t désigne le type temps
  double delta;  //delta désigne la durée d'exécution du programme entre les points t1 et t2

  //Partie 1: Entrée des données
  printf("Donner un nombre: n= ") ;
  scanf("%lf", &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.
  S = 0 ;
  i = 1 ;
  while (i<=n)
  {
      S = S + i ;
      i = i + 1 ;
  }//fin de 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.
  delta=(double)(t2-t1)/CLOCKS_PER_SEC;    //formule permettant de calculer la durée
                                //d'exécution du programme entre les points t1 et t2
  //Partie 3: Sortie des résultats
  printf("\n       La somme S = 1+2+...+%3.1e  = %lf", n, S);
                        //le format "%e" permet d'afficher en notation scientifique. Le format
                        //"3.1" permet d'afficher sur 3 positions dont la virgule et 1 position
                        //après la virgule (et il reste 1 position pour la mantisse)
  printf("\n       Le temps d'exécution est delta = %10.3lf secondes", delta);
```

```
   printf("\n\nEntrer une touche quelconque pour terminer le programme: ");  getchar();
   return(0);
}//fin du programme PSomme_2
```

Figure 4. Programme de la somme des n premiers nombres entiers naturels avec n>=1, noté PSomme_2, avec mesure du temps d'exécution.

Les mesures obtenues avec le programme PSomme_2 sont présentées sur le tableau suivant :

n	…	10^6	$2*10^6$	10^7	2*107	10^8	$2*10^8$	10^9	$2*10^9$
T		0	0.05	0.16	0.28	1.43	2.85	14.34	28.45

n	10^{10}	$2*10^{10}$	10^{11}	$2*10^{11}$	10^{12}	$2*10^{12}$	…	…	…
T	142.42	284.73	1483.6	…	…	…	...	…	…

Figure 5. Tableau des mesures du temps d'exécution obtenues avec le programme PSomme_2.

Remarque 1 : Bien entendu, avec un autre micro ordinateur, ayant d'autres caractéristiques, on aura des mesures du temps différentes.

Remarque 2 : Il faut noter que dans le programme de la mesure du temps, le code à mesurer n'inclut pas les instructions d'entrée/sortie.

Remarque 3 : A partir de la valeur de n = 10^{11}, et, avec ce mico-ordinateur (précisé ci-dessus), le temps d'exécution augmente considérablement. On a : T = 1483,6 s (soit, 1483,6/60 = 24,7 m). On a arrêté les mesures à ce niveau.

6- Représenter avec un graphe les variations du temps mesuré T en fonction de la variable n.

Solution :

A traiter par les étudiants.

7- Déduire le temps d'exécution moyen des instructions contenues dans les programmes.

Solution :

A traiter par les étudiants.

8- Rédiger un rapport décrivant le travail réalisé.

Solution :

A traiter par les étudiants.

Questions supplémentaires :

9- Calculer la complexité temporelle de chacun de ces algorithmes (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 exécuter 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 notation asymptotique appelée aussi notation Landau. Cette deuxième forme est la forme couramment utilisée.

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- Le calcul de la complexité temporelle peut se ramener au calcul du nombre d'instructions exécutées par cet algorithme qu'on appelle aussi fréquence d'exécution des instructions de l'algorithme car la fonction T(n) qui mesure le temps d'exécution d'un algorithme et la fonction F(n) qui mesure le nombre d'instructions exécutées par cet algorithme appartiennent à la même classe de complexité (voir la remarque ci-dessous).
3- les principales classes de complexité sont :

$$O(n), O(n^2), O(n^3), O(\log(n)), O(nlog(n)), O(a^n)$$

4- 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 somme de la fréquence de la condition de l'instruction conditionnelle et le maximum des fréquences entre ses 2 branches dans le pire cas ou le minimum des fréquences dans le meilleur cas (car une seule branche est exécutée).
c) Dans le cas d'une instruction de répétition (pour, tant que ou répéter), on doit calculer la somme de la fréquence de la condition de l'instruction de la répétition et des fréquences des instructions contenues dans le corps (ou le bloc) de l'instruction de la répétition ;
d) Calculer la somme des fréquences des instructions de l'algorithme.

5- Généralement, le calcul de la complexité temporelle exacte n'est pas possible car cela dépend des données en entrée de l'algorithme, et on se restreint alors au calcul des complexités au pire cas, au moyen cas et au meilleur cas. La complexité au pire cas est celle qui est la plus utilisée.
6- 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()).

7- Les fonctions prédéfinies comme sin, cos, log, modulo, etc. sont supposées se comporter comme des opérations exécutables par le processeur de manière indivisible (cette hypothèse ne change pas la complexité de l'algorithme).
8- 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 de l'algorithme).
9- 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.

Soit L le nombre d'instructions de l'algorithme Somme_1 (ci-dessus), f_i (i=1…L) la fréquence d'exécution de l'instruction I_i (i=1…L), et F la somme de ces fréquences.
On a: L = 8 instructions.

Pour déterminer F, on doit calculer les fréquences d'exécutions de chacune des L=8 instructions de l'algorithme Somme_1 comme il est montré sur le tableau suivant (figure 2.) :

N°	Instruction I_i		Fréquence d'exécution f_i	
1	ecrire("\nDonner la valeur n = ");	→	1	
2	lire(n);	→	1	
3	S = 0 ;	→	1	
4	i = 1 ;	→	1	
5	tant que (i<=n) faire	→	$(n+1)^1$	
	{			
6	S = S + 1 ;	→	2n	//(2 opérations : + et =)^2
7	i = i + 1 ;	→	2n	//(2 opérations : + et =)^2
	}//fin tant que			
8	ecrire ("La somme est S= ", S) ;	→	1	

Figure 2. Tableau des fréquences d'exécutions des instructions de l'algorithme Somme_1

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

Note 1 : L'instruction I_5 est exécutée (n+1) fois car la condition (i<=n) est testée (n+1) fois : elle est vraie n fois et fausse 1 fois.

Note 2: L'instruction I_6 est composée de 2 opérations exécutables de façon indivisible : l'opération d'addition (+) et l'opération d'affectation (=). La fréquence d'exécution est alors comptée 2 fois : $f_7 = 2*2$. On a la même remarque pour l'instruction I_7.

Dans cet algorithme, la complexité est exacte. Il n'y pas de meilleur cas ou de pire cas.

La somme des fréquences d'exécution des instructions est :

$$F(n) = \sum_{i=1}^{L=8} f_i = f_1 + f_2 + f_3 + f_4 + f_5 + f_6 + f_7 + f_8$$
$$\text{¿ } 1+1+1+1+(N+1)+2N+2N+1$$
$$\text{¿ } 5n+6 \text{ instructions exécutées}$$

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

En conséquence, la fonction de la fréquence $F(n)$ de l'algorithme Somme_1 est :

1- **En notation exacte :** $F(n) = 5n+6$ *instructions exécutées*
2- **En notation Asymptotique :** $F(n) = O(n)$ **instructions exécutées**

Comme on a : $T(n) = F(n) * \Delta t$ *unité de temps*
où : Δt est la durée d'exécution, supposée constante, de toutes les instructions (règle n°8).

On déduit que la complexité temporelle, notée T(n) de l'algorithme Somme_1 est :

1- **En notation exacte :** $T(n) = (5n+6) * \Delta t$ *unité de temps*
2- **En notation Asymptotique :** $T(n) = O(n)$ **unités de temps**

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

On a supposé l'hypothèse (règle n°8 ci-dessus) : toutes les instructions prennent la même durée de temps d'exécution. Soit Δt cette durée.

On a : $F(n)=(5n+6)\,instructions\,exécutées$

Il s'ensuit que le temps d'exécution est :

$$T(n)=(F(n)*\Delta t)\,unités\,de\,temps$$
$$¿\,((5n+6)*\Delta t)\,unités\,de\,temps$$
$$¿\,(5n*\Delta t+6*\Delta t)\,unités\,de\,temps$$
$$¿\,((5*\Delta t)*n+(6*\Delta t))\,unités\,de\,temps$$
$$¿\,(an+b)\,unités\,de\,temps$$

où : $a=5*\Delta t, b=6*\Delta t$
et Δt est la durée d^' exécution, suposée constante, de toutes les instructions.

Donc :

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

En conclusion, on peut énoncer ce résultat comme suit :

Théorème :
La fonction $T(n)$ qui mesure le temps d'exécution d'un algorithme et la fonction $F(n)$ qui mesure le nombre d'instructions exécutées par cet algorithme sont égales en notation asymptotique. On dit aussi qu'elles appartiennent à la même classe de complexité.

10. Calculer la complexité spatiale de chacun de ces algorithmes (là aussi, on a la même remarque que celle de la question 9).

Solution :
Rappel :
La complexité spatiale d'un algorithme est la mesure exacte (rarement utilisée) ou en notation asymptotique (couramment utilisée) de l'espace mémoire occupé par les instructions et les données lors de son exécution.

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

1- La complexité spatiale est calculée en fonction des données en entrée de l'algorithme.
2- les principales classes de complexité sont :
$O(n), O(n^2), O(n^3), O(\log(n)), O(nlog(n)), O(a^n)$
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 une donnée de type scalaire (entier, réel, logique ou caractère) occupe aussi $1\,mot\,mémoire$.

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: $L=8\,instructions$.
Comme 1 instruction occupe 1 mot mémoire (par hypothèse), alors $\alpha=8\,mots\,mémoires$.

Soit D le nombre de données de l'algorithme Somme_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 une donnée occupe un mot mémoire (par hypothèse), alors $\beta=3\,mots\,mémoires$.

Soit S l'espace mémoire occupé par les instructions et les données de l'algorithme Somme_1.
On a: $S(n)=\alpha+\beta=(8+3)\,mots\,mémoires=.11\,mots\,mémoires$.

$$S(n)\,est\,une\,fonction\,constante\,en\,n\,du\,type: S(n)=b \Rightarrow S(n)=O(1)$$

En conséquence, la complexité spatiale $S(n)$ de l'algorithme Somme_1 est :

1-	**En notation exacte:**	$S(n)=11\,mots\,mémoires$
2-	**En notation Landau:**	$S(n)=O(1)\,mots\,mémoires$

Xxx
Le résuktat trouvé du yemps d'exécution d'un programme T(n) et F(n)

Le détailler car c'est un résultat fondamental en complexité des algorithmes