Université USTHB – Bab-Ezzouar Bab-Ezzouar, 28 Septembre 2014

Faculté de l’Electronique et de l’Informatique, Département de l’Informatique Année universitaire 2014/2015

1ère année Master Informatique, Semestre 1 Semestre 1

Module : Conception et Complexité des Algorithmes

-------------------------------------------------------------------------------------------------------------------------------------------------------

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

**Rappels sur l’Algorithmique et la Programmation,**

**Introduction à la Complexité,**

**et Mesure du Temps d’Exécution d’un programme**

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

L’objet de ce TP est d’une part, de faire quelques rappels sur l’algorithmique, la programmation et la complexité ; et, d’autre part d’apprendre à mesurer le temps d’exécution d’un programme. On utilise le langage de programmation C.

**I. Partie I(Somme des N premiers nombres entiers naturels)**

Dans cette partie, on fait quelques rappels sur l’algorithmique, la programmation et la complexité avec un premier problème de calcul de la somme de nombres entiers. Ce problème est à résoudre en utilisant la technique de la programmation itérative.

1. Développer un algorithme itératif, noté Somme\_1, qui permet de calculer la somme, notée S, des n premiers nombres entiers naturels :

Le nombre entier naturel n est à lire en entrée (n>=1).

Utiliser les trois formes de la répétition :

1. pour … faire ;
2. tant que … faire ;
3. répéter … jusqu’à.

2. 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).

3. Calculer la complexité spatiale de chacun de ces algorithmes (là aussi, on a la même remarque que celle de la question 2°).

4. Développer le programme itératif correspondant, noté Psomme\_1, avec le langage C. Utiliser les 3 formes de la répétition.

Ind : Pour pouvoir utiliser les grandes valeurs de n, utiliser le type réel en simple ou double précision.

**II. 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.

1. Reprendre le programme précédent et inclure les instructions qui permettent de mesurer les temps d’exécution T pour l’échantillon des valeurs de n données dans le tableau ci-dessous (le nouveau programme est noté PSomme\_2).

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | … | 106 | 2\*106 | 107 | 2\*107 | 108 | 2\*108 | 109 | 2\*109 |
| T |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | 1010 | 2\*1010 | 1011 | 2\*1011 | 1012 | 2\*1012 | … | … | … |
| T |  |  |  |  |  |  |  |  |  |

2. Au lieu de mesurer séparément le temps d’exécution T pour chaque valeur de n, on automatise le processus des mesures. Pour cela, on utilise 2 tableaux : un tableau tab1 pour enregistrer (au préalable) les données en entrée de n, et un autre tableau tab2 pour les mesures du temps T.

Développer un programme de mesure du temps d’exécution du programme, noté PSomme\_3, qui a en entrée les données de l’échantillon ci-dessus (soit le tableau tab1) et en sortie les temps d’exécution (soit le tableau tab2).

3. Que remarquez-vous sur la variation (ie. la croissance) du temps d’exécution T par rapport à la variation de l’entrée n.

4. 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 la valeur de n.

5. Quelle est la validité de ce résultat ? (ie. La fonction T(n) est-elle valide pour toute valeur entière de n ?)

**Le Corrigé du TP n°1**

**I. Partie I(Somme des n premiers nombres entiers naturels)**

**1- Algorithme de la somme des n premiers nombres entiers naturels, noté Somme\_1**

**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 :

1. instruction de répétition pour …
2. instruction de répétition tant que …
3. 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. 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’algorithmecar 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 :

4- Pour calculer la complexité temporelle d'un algorithme, on doit :

1. calculer les fréquences d'exécutions de chacune des instructions de l'algorithme ;
2. 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).
3. 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 ;
4. 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), fi (i=1…L) la fréquence d’exécution de l’instruction Ii (i=1…L), et Fla 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 Ii Fréquence d'exécution fi**  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 fi sont explicitées dans ce qui suit :

**Note 1 :**L’instruction I5 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 I6 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 : . On a la même remarque pour l’instruction I7.

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 :

En conséquence, la fonction de la fréquence de l'algorithme Somme\_1 est :

1. En notation exacte :
2. En notation Asymptotique : instructions exécutées

Comme on a :

où :

On déduit que complexité temporelle, notée T(n) de l'algorithme Somme\_1 est :

1. En notation exacte :
2. En notation Asymptotique : 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 cette durée.

On a :

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

où :

Donc :

En conclusion, on peut énoncer ce résultat comme suit :

**Théorème :**

La fonction qui mesure le temps d’exécution d’un algorithme et la fonction 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.

**3. Calculer la complexité spatiale de chacun de ces algorithmes (là aussi, on a la même remarque que celle de la question 2°).**

**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.

**Remarques :**

1- La complexité spatiale est calculée en fonction des données en entrée de l'algorithme.

2- les principales classes de complexité sont :

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 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 = 8 instructions.

Comme 1 instruction occupe 1 mot mémoire (par hypothèse), alors α = 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 1 donnée occupe 1 mot mémoire (par hypothèse), alors β = 3 mots mémoires.

Soit S l'espace mémoire occupé par les instructions et les données de l'algorithme Somme\_1.

On a: S(n) = α + β = (8 + 3) mots mémoires =.11 mots mémoires.

En conséquence, la complexité spatiale S(n) de l'algorithme Somme\_1 est :

1. En notation exacte:
2. En notation Landau:

**4. Développer les programmes itératifs correspondants avec le langage C.**

**Solution :**

Le programme, noté PSomme\_1, est une traduction directe de l’algorithme. Il est présenté sur la figure 3 suivante :

|  |
| --- |
| //PSomme\_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 \* , -2.2 \* ] U [0] U [2.2 \* , 1.8 \* ]  //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.**

**II. Partie II (Mesure du temps d’exécution)**

**1.** Mesurer les temps d’exécution T de chacun des 3 programmes précédents pour les valeurs de n données dans le tableau ci-dessous :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | … | 106 | 2\*106 | 107 | 2\*107 | 108 | 2\*108 | 109 | 2\*109 |
| T |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | 1010 | 2\*1010 | 1011 | 2\*1011 | 1012 | 2\*1012 | … | … | … |
| 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>.

1. Définir 2 variables notées t1 et t2 de type clock\_t qui désigne le type temps comme suit :

clock\_t t1, t2 ;

1. Définir une variable réelle pour lui affecter le le temps d’exécution du programme :

double delta ;

1. 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

1. 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  //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 \* , -2.2 \* ] U [0] U [2.2 \* , 1.8 \* ]  //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 | … | 106 | 2\*106 | 107 | 2\*107 | 108 | 2\*108 | 109 | 2\*109 |
| T |  | 0 | 0 .05 | 0.16 | 0.28 | 1.43 | 2.85 | 14.34 | 28.45 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | 1010 | 2\*1010 | 1011 | 2\*1011 | 1012 | 2\*1012 | … | … | … |
| 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 :** 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=1011, le temps d’exécution augmente considérablement. Avec un ordinateur plus performant, on aurait des temps plus courts.

**2. Au lieu de mesurer séparément le temps d’exécution T pour chaque valeur de n, on automatise le processus des mesures. Pour cela, on utilise 2 tableaux : un tableau tab1 pour enregistrer (au préalable) les données en entrée de n, et un autre tableau tab2 pour les mesures du temps T.**

**Développer un programme de mesure du temps d’exécution du programme, noté PSomme\_3, qui a en entrée les données de l’échantillon ci-dessus (soit le tableau tab1) et en sortie les temps d’exécution (soit le tableau tab2).**

**Solution :**

Dans le programme précédent PSomme\_2 (figure 4), on ajoute les déclarations des tableaux tab1 et tab2.Un 3ème tableau, noté som, est nécessaire pour enregistrer les sommes pour chacune des données de l’échantillon. On ajoute aussi l’instruction de répétition qui permet de répéter l’exécution du programme pour les données de l’échantillon. Le programme est présenté sur la figure 6 ci-dessous :

|  |
| --- |
| //PSomme\_3  //Ce programme, noté PSomme\_3, calcule la somme des n premiers nombres entiers naturels  //avec n>=1 et mesure de façon automatique les temps d’exécution pour un échantillon de //données n enregistrées au préalable..  //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 \* , -2.2 \* ] U [0] U [2.2 \* , 1.8 \* ]  //Ce programme est itératif  #include <stdlib.h>  #include <stdio.h>  #include <time.h> //la bibliothèque des fonctions de gestion du temps  int main()  {int L, j; //T désigne la taille de l'échantillon  //Dans l'énoncé, on a L=14 valeurs de n à tester  double i, n, S;  double tab1[14]; //un tableau tab1 de 14 éléments réels en double précision  //pour enregistrer l'échantillon des données n en entrée  double som[14]; //un tableau som de 14 éléments réels en double précision  //pour enregistrer les sommes pour chacune des 12 données n  double tab2[14]; //un tableau tab2 de 14 éléments réels en double précision  //pour enregistrer les mesures du temps T  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  //-----------------------------------------------------------------------------  printf("\n\n\n Série de Travaux Pratiques n°1 (TP n°1) \n");  printf("\n\n\n Programme 2 incluant le programme automatique de mesure du ");  printf("temps d'exécution \n");  printf("\n\n\ Il utilise l'instruction de répétition tant que qui s'écrit ");  printf("\n en langage C: while (condition) {...} \n");  printf("\n\nEntrer une touche quelconque pour continuer: "); getchar();  //Partie 1: Lecture des données  //Enregistrement préalable des données de l’échantillon  tab1[0] = 1.0E+6; tab1[1] = 2.0E+6; //1 et 2 millions  tab1[2] = 1.0E+7; tab1[3] = 2.0E+7; //10 et 20 millions  tab1[4] = 1.0E+8; tab1[5] = 2.0E+8; //100 et 200 millions  tab1[6] = 1.0E+9; tab1[7] = 2.0E+9; //1 et 2 milliard  tab1[8] = 1.0E+10; tab1[9] = 2.0E+10; //10 et 20 milliard  tab1[10] = 1.0E+11; tab1[11] = 2.0E+11; //100 et 200 milliard  tab1[12] = 1.0E+12; tab1[13] = 2.0E+12; //1000 et 2000 milliard  printf("\nDonner le nombre de valeurs à tester: L = "); //cela permet de  //contrôler le nombre de valeurs n à tester  scanf("%d", &L); //le format "%d" permet de lire le type entier en  //simple précision sur 2 octets = 16 bits  //Partie 2: Traitement  j = 0;  while (j<L)  {  n = tab1[j]; //n prend la jème (j=0...13) valeur de l'échantillon/des entrées n  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 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.  delta=(double)(t2-t1)/CLOCKS\_PER\_SEC; //formule permettant de calculer la  //durée d'exécution du programme entre les points t1 et t2  tab2[j] = delta; //enregistrer dans tab2[j] la durée d'exécution  som[j] = S; //som[j] contient la j ième (j=0...13) somme des n  //premiers entiers  j = j + 1;  }//fin du 1er while  //Partie 3: Ecriture des résultats  j = 0;  while (j<L)  {  printf("\n La somme S = 1+2+...+%3.1e = %lf", tab1[j], som[j]);  //le format "%e" permet d'afficher en notation scientifique  //le format "3.1" permet d'afficher sur 3 positions dont le  //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", tab2[j]);  printf("\n");  j = j + 1;  }//fin du while  //-----------------------------------------------------------------------------  printf("\n\nEntrer une touche quelconque pour terminer le programme: ");  getchar(); getchar();  return(0);  }//fin du programme PSomme\_3 |

**Figure 6.Programme de la somme des n premiers nombres entiers naturels avec n>=1**

**(noté PSomme\_3) avec mesure automatique des temps d’exécution**

**pour un échantillon de données.**

**Remarque:**L’intérêt principal avec ce type de programme est qu’on n’aura pas à refaire l’exécution du programme pour chaque donnée. Dans la pratique, les données des échantillons sont plutôt enregistrées de façon permanente dans des fichiers.

**3. Que remarquez-vous sur la variation (ie. la croissance) du temps d’exécution T par rapport à la variation de l’entrée n.**

**Solution :**

On constate sur le tableau des mesures obtenues expérimentalement avec l’échantillon des données que pour chaque couple de données (106, 2\*106), (107, 2\*107), (108, 2\*108), (109, 2\*109), etc., la valeur de la donnée n est doublée. De même, la valeur mesurée du temps T est pratiquement doublée :(0, 0.05), (0.16, 0.28), (1.43, 2.85), (14.34, 28.45), etc.

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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | … | 106 | 2\*106 | 107 | 2\*107 | 108 | 2\*108 | 109 | 2\*109 |
| T |  | 0 | 0 .05 | 0.16 | 0.28 | 1.43 | 2.85 | 14.34 | 28.45 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | 1010 | 2\*1010 | 1011 | 2\*1011 | 1012 | 2\*1012 | … | … | … |
| T | 142.42 | 284.73 | 1483.6 | 2965.2 | … | … | ... | … | … |

**Figure 5. Tableau des mesures du temps d’exécution obtenues**

**avec le programme PSomme\_2.**

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 :

(1)

On constate aussi sur le tableau des mesures :

;

;

; etc.

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

(2)

**4. 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 la valeur de n.**

**Solution :**

On réécrit la relation (2) comme suit :

Pour

(3)

où : représente le temps nécessaire pour calculer la somme de 2 élément.

On note :.

On déduit que :

(4)

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

**Remarque :** 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 en fonction des valeurs de . A partir de la disposition géométrique des points de ce tracé graphique, on déduit la nature du graphequi approche au mieux le graphe de la fonction :une droite, une parabole, une hyperbole, une sinusoïde, etc., et, ensuite, on détermine l’expression de la fonction .

**5. Quelle est la validité de ce résultat ? (ie. La fonction T(n) est-elle valide pour toute valeur entière de n ?)**

**Solution :**

La fonction : est valide pour les valeurs de n qui sont dans l’intervalle des mesures effectuées, soit l’intervalle.

Mais en dehors de cet intervalle, on ne peut pas conclure. On peut, tout au plus, prédire que la fonction déduite par expérimentation reste valide (çad, vraie). Ce sont là les limites de la méthode expérimentale.

Dans le cas général (çàd, pour toute valeur entière de n), on doit utiliser une autre approche plus générale et plus rigoureuse. Elle (cette approche) constitue la discipline qu’on appelle : **″la complexité des algorithmes″**.

On rappelle qu’on a appliqué quelques techniques de cette approche dans la question 2 de la partie 1 (ci-dessus).

On rappelle ici la définition de la discipline de la complexité des algorithmes.

**Définition :** La complexité des algorithmes est une discipline de l’informatique qui étudie formellement, c'est-à-dire scientifiquement, les quantités des ressources, notamment en temps et en espace mémoire, nécessaires à l’exécution d’un programme par un ordinateur.