Mini Projet 1 : Outils de débogue, tableaux et complexité

Ce mini projet est à réaliser pendant les deux premières séances de TME. Il va vous permettre de découvrir deux techniques qui seront utiles pour l'ensemble des TME du semestre (et notamment pour le projet final). Plus précisément, le premier exercice traite de techniques de débogue, tandis que deuxième exercice est l'occasion d'étudier une façon d'évaluer la vitesse d'un programme et de comparer les vitesses de plusieurs programmes. Les outils fournis pour ces deux exercices ne sont pas obligatoires et pourront être remplacés par des outils équivalents. Les fichiers nécessaires à la réalisation de ce TME peuvent être récupérés sur moodle.

Exercice 1 – Utilisation d'outils de débogue

La bonne compilation d'un programme écrit en C n'assure malheureusement pas son bon fonctionnement. Il existe souvent des erreurs qui ne sont révélées qu'au moment de l'exécution et qu'il faut trouver. Parmi ces erreurs, la plus courante est l'"erreur de segmentation" qui survient souvent lors de la manipulation de tableaux, ou de pointeurs. L'exercice suivant vous présente l'utilisation d'un outil de débogue qui vous permettra la plupart du temps de localiser vos erreurs.

Partie 1 : *problème avec l'indice de boucle.*
Récupérez le fichier C nommé `tme1_exo1p1.c` :

```c
#include<stdio.h>
#include<stdlib.h>

const static int len = 10;

int main(void) {
    int *tab;
    unsigned int i;

    tab = (int*)malloc(len*sizeof(int));

    for (i=len-1; i>=0; i--) {
        tab[i] = i;
    }

    free(tab);
    return 0;
}
```

Q 1.1 À la lecture du code, qu'est censé faire le programme d'après vous ? Compilez le avec la commande suivante `gcc -o tme1_exo1p1 tme1_exo1p1.c` puis lancer le. Que se passe-t-il ?

Q 1.2 Pour trouver l'erreur dans le programme, nous allons utiliser gdb, un outil de débogue très pratique. Recompilez votre programme avec la commande suivante : `gcc -ggdb -o tme1_exo1p1 tme1_exo1p1.c`. L'option `-ggdb` permet d'inclure dans l'exécutable des informations supplémentaires qui facilite le débogue. Voici ci-dessous un résumé des commandes importantes sous gdb :

- `quit` (q) : quitter `gdb`
- `run` (r) : lancer l'exécution
- `break, watch` (b,w) : introduire un point d'arrêt ou bien "surveiller une variable"
- `continue` (c) : relance l'exécution jusqu'au prochain point d'arrêt rencontré.
- `print` (p) : afficher la valeur d'une variable.
- `clear, delete` (cl,d) : effacer un point d'arrêt. Pour `clear`, il faut indiquer un numéro de ligne ou nom de fonction, tandis que pour `delete`, il faut indiquer le numéro du breakpoint (`delete` tout court permet d'effacer tous les points d'arrêt).

Lancez l'outil gdb sur l'exécutable de votre programme dans un terminal : `gdb tme1_exo1p1`. Pour trouver l'erreur du programme, vous allez l'exécuter pas-à-pas et "donner la trace" de la variable de boucle `i`, c'est-à-dire examiner l'évolution de sa valeur dans le temps. Pour cela, commencer par taper la commande `break 13` pour définir un point d'arrêt au niveau de l'instruction `tab[i] = i;` qui se trouve à la ligne 13 du fichier. Lancez ensuite l'exécution à l'aide de l'instruction `run`. Le programme va alors s'exécuter jusqu'au point d'arrêt que vous venez de définir. Lorsque l'exécution du programme est interrompue, on peut examiner l'état de la mémoire à ce moment là, par exemple en affichant la valeur d'une variable à l'aide de la commande `print`. Pour tracer la variable `i`, exécutez la commande `print i`. Vous pouvez maintenant continuer l'exécution du programme :

- Soit avec la commande `continue`, qui permet de continuer l'exécution jusqu'à tomber sur le prochain point d'arrêt (ici on a un seul point d'arrêt, donc on va retomber sur le même).
- Soit avec la commande `step`, qui permet d'avancer pas à pas dans l'exécution, afin de bien contrôler l'évolution du programme.

Faites itérer la boucle jusqu'au bout, en suivant l'évolution de la valeur de `i` (à l'aide de l'instruction `print i`). Après l'itération dans laquelle `i` valait 0, que vaut la valeur de `i` ? Que devrait valoir `i` pour sortir de la boucle ? À quelle case du tableau essaie-t-on d'accéder ? Déduisez-en le sens de l'"erreur de segmentation".

Q 1.3 Sachant que le mot clé `unsigned` force un type C standard (`char, short, int`) à ne jamais être négatif (c'est-à-dire "non signé"), que proposez-vous pour résoudre cette erreur ?

Partie 2 : *problème d'allocation mémoire.*
Récupérez le fichier C nommé `tme1_exo1p2.c` :

```c
#include<stdio.h>
#include<stdlib.h>
#include <string.h>

typedef struct adresse {
    int numero;
    char* rue;
    int code_postal;
} Adresse;

Adresse* creer_adresse(int n, char* r, int c) {
    Adresse* new = (Adresse*) malloc(sizeof(Adresse));

    new->numero = n;
    strcpy(new->rue, r);
    new->code_postal = c;

    return new;
```

```
19 }
20
21 int main(void) {
22     Adresse* maison = creer_adresse (12, "manoeuvre", 15670);
23
24     printf("Adresse courante : %d rue %s %d France\n", maison->numero, maison->rue,
           maison->code_postal);
25
26     return 0;
27 }
```

Q 1.4 À la lecture du code, qu'est censé faire le programme d'après vous ? Compilez le programme et lancez le. Que se passe-t-il ?

Q 1.5 Recompilez le programme en activant l'option de débogue `-ggdb` et lancez l'outil `gdb` sur l'exécutable résultant. Créez un point d'arrêt au niveau de la ligne 15 du fichier avec l'instruction `break 15`. Faites de même avec la ligne 16 avec l'instruction `break 16`.

- Lancez l'exécution du programme avec la commande `run`. Le programme va s'arrêter au premier point d'arrêt (ligne 15). Afficher la valeur de `new->rue` avec l'instruction `print new->rue`. Que voyez-vous ?
- Continuer l'exécution pour arriver au deuxième point d'arrêt (ligne 16). Que constatez-vous ? À quel moment survient l'erreur ?
- Expliquez la cause de l'erreur et proposer une correction.

Partie 3 : *fuite mémoire.*

Récupérez le fichier C nommé `tme1_exo1p3.c` :

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  typedef struct tableau{
5     int* tab;
6     int maxTaille;
7     int position;
8  }Tableau;
9
10 void ajouterElement(int a,Tableau *t){
11    t->tab[t->position]=a;
12    t->position++;
13 }
14
15 Tableau* initTableau(int maxTaille){
16    Tableau* t = (Tableau*)malloc(sizeof(Tableau));
17    t->position=0;
18    t->maxTaille=maxTaille;
19    t->tab=(int*)malloc(sizeof(int)*maxTaille);
20    return t;
21 }
22
23 void affichageTableau(Tableau* t){
24    printf("t->position = %d\n",t->position);
25    printf("[ ");
26    for (int i=0;i<(t->position);i++){
27       printf("%d ",t->tab[i]);
28    }
```

```
29      printf("]\n");
30  }
31
32  int main(){
33      Tableau* t;
34      t = initTableau(100);
35      ajouterElement(5,t);
36      ajouterElement(18,t);
37      ajouterElement(99999,t);
38      ajouterElement(-452,t);
39      ajouterElement(4587,t);
40      affichageTableau(t);
41      free(t);
42  }
```

Q 1.6 À la lecture du code, qu'est censé faire le programme d'après vous ? Compilez le programme et lancez le. Que se passe-t-il ?

Q 1.7 Quels sont les problèmes de ce programme ?

Q 1.8 Nous allons utiliser l'outil `valgrind` pour repérer les fuites mémoires éventuelles. Lancer la commande `valgrind --leak-check=yes ./tme1_exo1p3`. Que constatez-vous ? À quoi correspond les 400 bytes relevés par l'outil ?

Q 1.9 Proposez une correction du programme et vérifiez qu'il n'y a plus de fuite mémoire (en utilisant l'outil `valgrind`).

Exercice 2 – Algorithmique et tableaux

Dans cet exercice, il s'agit d'étudier la complexité temporelle de différents algorithmes permettant de résoudre un même problème. On s'intéressera à la complexité temps-pire cas, en donnant des bornes avec la notation Landau O.

Partie 1 : Tableaux à une dimension.

Dans cette partie, on s'intéresse à des algorithmes travaillant sur des tableaux à une dimension.

Q 2.1 Commencez par créer les fonctions suivantes :

1. une fonction `alloue_tableau` permettant d'allouer la mémoire utilisée par un tableau T d'entiers de taille n. Noter qu'il existe deux versions possibles pour cette fonction :

   `alloue_tableau(int n);` et `alloue_tableau(int **T, int n);`.

   Dans la première version, le tableau est retourné par la fonction, tandis que dans la seconde version, on utilise le passage par référence. Justifier votre choix.
2. une fonction `desalloue_tableau` permettant de désallouer la mémoire utilisée par un tableau d'entiers de taille n.
3. une fonction `remplir_tableau` permettant de remplir le tableau avec des valeurs entières aléatoirement générées entre 0 et V (non-compris).
4. une fonction `afficher_tableau` permettant d'afficher les valeurs du tableau.

Q 2.2 On souhaite écrire un algorithme efficace permettant de calculer la somme des carrés des différences entre les éléments du tableau pris deux à deux :

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\Big(T(i)-T(j)\Big)^2$$

1. Écrivez un premier algorithme de complexité $O(n^2)$.
2. Écrivez un second algorithme de meilleure complexité, en vous aidant du fait que :

$$\sum_{i=1}^{n}\sum_{j=1}^{n}x_i x_j = \Big(\sum_{i=1}^{n}x_i\Big)^2$$

Q 2.3 L'objectif est maintenant de comparer les temps de calcul des deux algorithmes en faisant varier la taille n du tableau. Pour mesurer le temps mis par le CPU pour effectuer une fonction `fct(int n)`, on peut utiliser le code suivant où `temps_cpu` contient le temps CPU utilisé en secondes.

```
#include <time.h>
...
clock_t temps_initial;
clock_t temps_final;
double temps_cpu;
...
temps_initial = clock(); //Nombre de ''ticks'' consomm'es par le programme jusqu'ici
fct(n);
temps_final = clock (); //Nombre de ''ticks'' consomm'es par le programme jusqu'ici

//On convertit les ''ticks'' consomm'es par fct en secondes :
temps_cpu =  ((double)(temps_final - temps_initial))/CLOCKS_PER_SEC;

printf("%d %f ",n,temps_cpu);
```

Pour analyser les résultats obtenus, on veut visualiser par des courbes les séries de temps obtenues. Pour cela, on peut utiliser un logiciel extérieur lisant un fichier texte créé par le programme. Par exemple, le logiciel gnuplot qui est utilisable en ligne sous linux permet de faire des courbes en lisant un fichier. Pour notre problème, le fichier d'entrée de gnuplot, nommé "sortie_vitesse.txt", est simplement la donnée en lignes de `n` suivi des mesures de temps. On peut lancer gnuplot puis taper interactivement les commandes pour lire le fichier "sortie_vitesse.txt", ou bien utiliser une redirection de la forme `gnuplopt -p < commande.txt` avec un fichier `commande.txt` du type :

```
plot "sortie_vitesse.txt" using 1:2 title 'Algo1' with lines
replot "sortie_vitesse.txt" using 1:3 title 'Algo2' with lines
set term postscript portrait
set output "01_courbes_vitesse.ps"
set size 0.7, 0.7
```

L'option `-p` permet de garder la fenêtre du graphique ouverte. Ces lignes de commande créent sur le disque le fichier postscript contenant deux courbes sur un même graphique. Justifiez les courbes obtenues avec votre programme en fonction de la complexité pire-cas attendue.

Partie 2 : Tableaux à deux dimensions (matrices)

Dans cette partie, on souhaite écrire des algorithmes efficaces pour tester si une matrice contient uniquement des valeurs différentes, puis écrire des programmes pour réaliser le produit de deux matrices.

Q 2.4 Commencez par créer les fonctions suivantes :

1. une fonction `alloue_matrice` permettant d'allouer la mémoire utilisée par une matrice entière carrée de taille n x n.
2. une fonction `desalloue_matrice` permettant de désallouer la mémoire utilisée par une matrice carrée.
3. une fonction `remplir_matrice` permettant de remplir une matrice carrée avec des valeurs entières aléatoirement générées entre 0 et V (non-compris).
4. une fonction `afficher_matrice` permettant d'afficher les valeurs d'une matrice carrée.

Q 2.5 On souhaite écrire un programme permettant de vérifier efficacement que tous les éléments de la matrice ont des valeurs différentes.

1. Écrivez un premier algorithme de complexité $O(n^4)$.
2. Écrivez un second algorithme de meilleure complexité dans le cas où la borne maximale V sur les valeurs contenues dans la matrice est connue. Pour obtenir une meilleure complexité, utilisez un tableau de taille V alloué en mémoire.
3. Comparez les temps de calcul des deux algorithmes et représentez les courbes obtenues en fonction du nombre n d'éléments de la matrice. Analysez les résultats en justifiant les courbes obtenues en fonction de la complexité pire-cas attendue.

Q 2.6 Dans cette question, on souhaite comparer deux algorithmes permettant d'effectuer le produit de deux matrices.

1. Écrivez un premier algorithme de complexité $O(n^3)$.
2. Écrivez un second algorithme avec moins d'instructions, en tenant compte du fait que le produit concerne une matrice triangulaire supérieure et une matrice triangulaire inférieure.

   Rappel : une matrice m est triangulaire supérieure si $m[i][j] = 0$ pour tout $i > j$, et triangulaire inférieure si $m[i][j] = 0$ pour tout $i < j$.

   N.B. : Vous pouvez utiliser un `struct` pour créer le type `MatTriangulaire` (comme en TD), mais ce n'est pas obligatoire.
3. Ce dernier algorithme est-il de meilleur complexité ? Justifiez votre réponse.
4. Comparez les temps de calcul des deux algorithmes et représentez les courbes obtenues en fonction du nombre n d'éléments de la matrice. Justifiez les courbes obtenues en fonction de la complexité pire-cas attendue.