COULIBALY Mariame
FERCHE Adelin-Flaviu
Grp 4

MINI PROJET 1

LA DESCRIPTION DES DOCUMENTS ENVOYES

- tme1_exo1p1.c
- tme1_exo1p2.c
- tme1_exo1p3.c

- tme1_exo2p1.c
  - void allouer_tableau(int **T, int n)
  - void desallouer_tableau(int *T)
  - void remplir_tableau(int *T, int n, int V)
  - void afficher_tableau(int *T, int n)
  - int Algo1(int *T, int n)
  - int Algo2(int *T, int n)
  - int main()

- tme1_exo2p2.c
  - void alloue_matrice(int ***M, int n)
  - void desalloue_matrice(int **M, int n)
  - void remplir_matrice(int **M, int n, int V)
  - void afficher_matrice(int **M, int n)
  - int premier_algo(int **M, int n)
  - int second_algo(int **M, int n, int V)
  - void Algo1(int **M1, int **M2, int **P, int n)
  - void Algo2(int **Msup, int **Minf, int **P, int n)
  - int main()

- 01_courbes_vitesse
  - Algo1.png
  - Algo2.png
  - donnee1.txt

- 02_courbes_vitesse.png
  - donnee2.txt

- 03_courbes_vitesse.png
  - donnee3.txt

EXERCICE 1

PARTIE 1 :

Q 1.1)
Le programme crée un tableau dynamique de taille 10 dont chaque case contient la valeur i allant de 0 à 9.
Lorsque l'on compile cela fonctionne mais lorsqu'on exécute le programme, on obtient une erreur de type "Segmentation fault".

Q 1.2)
Après l'itération dans laquelle i valait 0, on a que i vaut 4294967295 alors que normalement on devrait avoir que i = -1 pour sortir de la boucle
On essaye d'accéder à une case du tableau qui n'existe pas d'où l'erreur Segmentation Fault.

Q 1.3)
Pour résoudre cette erreur, on enlève le mot clé unsigned pour laisser que le int lors de la déclaration de i, pour que i puisse avoir la valoir -1.

(voir la modification apportée dans tme1_exo1p1.c)

PARTIE 2 :

Q 1.4)
Le programme créer une adresse, plus précisément, une élément de type Adresse dont on ajoute le numéro de rue, le nom de la rue et le code postale, et la renvoi.
La compilation marche et crée un exécutable mais lorsqu'on exécute cet exécutable on obtient une erreur de type "Segmentation fault".

Q 1.5)
-Lorsqu'on fait print new->rue dans le gdb, on obtient 0x0
-Lorsque l'on poursuit l'exécution, on obtient l'erreur de "Segmentation fault" qui survient à la ligne 15 lorsqu'on fait le strcpy.
-En effet, pour utiliser strcpy on doit allouer de la mémoire pour la destination, ici new->rue, car la taille de la source, ici r, peut ne pas avoir la même taille que new->rue. Pour y remedier, l'ordinateur va stocker chaque caractère de r dans des cases mémoires dont il a pas accès d'où le "Segmentation fault".
Donc pour éviter cette erreur, il faut allouer de la mémoire avec la commande "new->rue = (char*)malloc(sizeof(char)*strlen(r))", strlen permet de connaître la longueur de la chaîne de caractère r, qu'il faut placer avant "strcpy(new->rue, r)".

(voir la modification apportée dans tme1_exo1p2.c)

PARTIE 3 :

Q 1.6)
Le programme initialise un tableau de taille 100, puis ajoute des éléments dans le tableau et enfin l'affiche.
La compilation marche et l'exécution aussi, cela affiche:
t->position = 5

[5 18 99999 -452 4587]

Q 1.7)
Le problème provient de la mémoire. En effet, on alloue de la mémoire pour t→tab mais on ne libère pas cette mémoire allouée.

Q 1.8)
On constate qu'il y a un problème de mémoire. En effet, lorsqu'on lance la commande, on observe qu'on à une fuite mémoire, à la fin de l'exécution du programme, qui correspond à la perte de 400 bytes de mémoire.
Les 400 bytes proviennent de la mémoire alloué pour t->tab qui n'ont pas été libérer.

Q 1.9)
Pour cela, on ajoute un free(t->tab) après la boucle for de la fonction affichageTableau. En utilisant de nouveau valgrind on obtient «All heap blocks were freed -- no leaks are possible» ce qui prouve que l'on n'a plus de fuite mémoire.

(voir la modification apportée dans tme1_exo1p3.c)

EXERCICE 2

PARTIE 1: voir tme1_exo2p1.c

Q 2.1)

1) On a choisi la version avec **void alloue_tableau(int **T, int n)** car il prend en argument un pointeur qui pointe vers l'adresse d'un tableau et ainsi modifier le tableau sans que l'on fasse de return dans la fonction, soit avoir une opération élémentaire en moins.

2) On a créer une fonction qui prend en paramètre : **void desalloue_tableau(int *T)** car on a pas besoin de pointeur et on a seulement fait free(T).

3) On a créer une fonction de la forme **void remplir_tableau(int *T, int n, int V)** qui parcourt le tableau grâce à une boucle for et dont chaque case est rempli par rand()%V, une valeur entre 0 et V (exclus).
On a avant cela fait srand(time(NULL)) pour initialiser la fonction rand().

4) Initialisation de la fonction **void afficher_tableau(int *T, int n)** dont on parcours le tableau à l'aide d'une boucle en affichant à l'aide de printf les éléments du tableau.

Q 2.2)

1) On a créer une fonction **int Algo1(int *T, int n)** dont on initialise une variable s à 0 et on parcours le tableau avec 2 boucles prenant les éléments deux à deux et dont au la différence au carré qu'on additionne à s, ainsi s prend une nouvelle valeur.

2) Grâce à la formule donnée dans l'énoncé, on peut simplifier la première formule à $2n*s1-2*s2^2$, dont s1,la somme des carrées des éléments du tableau allant de i à n, et s2, le carré de la somme des éléments du tableau allant de i à n . Ainsi, on crée une fonction **int Algo2(int *T, int n)** dont on initialise s1 et s2 à 0 puis on parcours le tableau avec seulement 1 boucle et non plus 2 comme à l'Algo 1.

3) **voir 01_courbes_vitesse**

On observe que la courbe de l'Algo2 est en dessous de celle de l'Algo1 ce qui signifie que l'Algo2 met largement moins de temps à se finir que l'Algo1. On peut également voir sur les graphiques que les valeurs du temps ne sont pas au même ordre de grandeur, 10^-5 pour l'Algo2 et 10^-2 pour l'Algo1. On peut expliquer cette différence par leur pire cas : leur pire cas dépendent de la taille n, plus n est grand plus les programmes vont mettre du temps à se finir. En effet pour l'Algo 1, il faut $6n^2+2$ opérations élémentaires dans le pire des cas, donc une complexité temporelle de $O(n^2)$, alors que pour l'Algo2, il faut 6n+7 opérations élémentaires dans le pire des cas, donc une complexité temporelle de O(n). Donc l'Algo2 a une complexité temporelle inférieure à celle de l'Algo1 d'où le graphique obtenue.

PARTIE 2 : voir tme1_exo2p2.c

Q 2.4)

1) **void alloue_matrice(int ***M, int n)** avec en paramètre un pointeur sur une matrice pour éviter le return puis on alloue de la mémoire pour M selon n et enfin on parcourt la matrice à l'aide d'une boucle pour alloué de la mémoire pour chaque case de la matrice.

2) **void desalloue_matrice (int **M, int n)** dont d'abord on libère la mémoire des cases une par une à l'aide d'une boucle puis la mémoire de la matrice.

3) **void remplir_matrice (int **M, int n, int V)** dont on accède au case grâce à deux boucles puis on ajoute une valeur compris entre 0 et V (exclus) à l'aide de rand()%V

4) **void afficher_matrice(int **M, int n)** dont on affiche chaque case de la matrice en la parcourant à l'aide de deux boucles.

Q 2.5)

1) **int premier_algo(int **M, int n)**: on teste l'égalité de la valeur d'une case à toutes les autres valeurs des autres cases de la matrices, ainsi de suite, grâce à 4 boucles. Si on tombe sur une égalité on vérifie que ce n'est pas la même case, si ce n'est pas le cas on renvoi 0 qui correspond à False. Si le programme à fini de parcourir les 4 boucles alors on renvoi 1, c'est-à-dire True.

2) **int second_algo(int **M, int n, int V)**: on initialise et on alloue de la mémoire pour un tableau de taille V qui correspond à la borne maximale des valeurs contenues dans la matrice. Puis on parcourt la matrice à l'aide de 2 boucles et pour chaque valeur de la matrice, on regarde dans le tableau à l'indice de la valeur de la matrice s'il y a un 1, s'il n'y en a pas on le rajoute sinon on libère la mémoire alloué du tableau et on renvoi un 0 (False). Si on a fini de parcourir la matrice alors on libère également la mémoire avant de renvoyer un 1 (True).

3) **voir 02_courbes_vitesse.png et donnee2.txt**

On observe que la courbe de l'Algo2 est en dessous de celle de l'Algo1 même si par moment elle passe au-dessus. De plus, on remarque que la différence de temps entre les deux algorithmes est minime. On peut expliquer cette différence par leur pire cas : la matrice ne contient que des éléments de valeur différentes. Ainsi, pour l'Algo1, il faut $9n^4+1$ opérations élémentaires dans le pire des cas, donc une complexité temporelle de $O(n^4)$, alors que pour l'Algo2, il faut $4n^2+4$

opérations élémentaires dans le pire des cas, donc une complexité temporelle de $O(n^2)$. Donc l'Algo2 a une complexité temporelle inférieure à celle de l'Algo1 d'où le graphique obtenue.

Q 2.6)

1) **void Algo1(int **M1, int **M2, int**P, int n)** : on crée 2 boucles puis on initialise la case de P à 0 ensuite on fait une troisième boucle dont à chaque tour on fait la somme des produits des éléments de la ligne i de la matrice 1 avec les éléments de la colonne j de la matrice 2.

2) **void Algo2 (int **Msup, int **Minf, int **P, int n)** : on fait à peu près la même chose que dans Algo1 mais cette fois on a deux matrice triangulaires donc on crée une condition qui teste si les valeurs des cases de Msup et Minf sont différent de 0 avant d'effectuer le produit sinon on ne fait rien et on passe au tour suivant.

3) Le deuxième algorithme n'est pas de meilleure complexité car on a 3 boucles for donc une complexité temporelle de $O(n^3)$ comme le premier algorithme.

4) **voir 03_courbes_vitesse.c et donnee3.txt**

On observe que les courbes sont confondues au début mais qu'a partir d'une certaine taille n, ici à partir de n=300, la courbe de l'Algo1 passe au dessus de la courbe de l'Algo2, c'est-à-dire qu'à partir d'une valeur de n, l'Algo1 prend plus de temps à se finir que l'Algo2. On peut expliquer cette différence par leur pire cas : tout deux dépendent de n, plus n est grand plus les algorithmes vont mettre du temps à se finir comme le montre le graphique qui croit de façon exponentielle. Pour l'Algo1, il faut $7n^3$ opérations élémentaires dans le pire cas, donc une complexité temporelle de $O(n^3)$, pour l'Algo2, il faut $10n^3$ opérations élémentaires dans le pire de cas, donc une complexité temporelle de $O(n^3)$. Or l'Algo2 prend moins de temps à s'exécuter, ceci s'explique par le fait que l'Algo2 est un produit matriciel entre deux matrices triangulaires, une supérieure et une inférieure, par conséquent, lorsqu'une case d'une des deux matrices est égale à 0, on ne fait pas le produit et on passe à la case suivante alors que pour l'Algo1, il faut faire la somme des produits pour chaque case, d'où une meilleure rapidité de l'Algo2.