ZENG Fanxiang
FRANCOISE Raphaël

TME 1 – Structure de données

Le but de ce TME est de se familiariser dans un premier temps avec les outils de débogage par l'utilisation de Valgrind et de GDB. Puis dans un second temps, nous allons évaluer la complexité temporelle des différentes fonctions de manipulation des tableaux et des matrices (qui sont des structures de données). Pour cela nous allons utiliser Gnuplot pour générer des graphes afin de voir le temps mis par le CPU pour exécution de chaque fonction en fonction de la taille des tableaux et des matrices.

Organisation globale de nos fichiers :

- Un dossier « parti1 » correspondant à l'exercice 1 et comprenant les fichiers disponibles sur Moodle et leurs modifications (q2 et q3).
- Un dossier « parti2 » correspondant à l'exercice 2 :
  - « tab1.c » est le ficher de la première partie. Nous avons fait un header « tab1.h » afin de pouvoir avoir un fichier qui contient le main « testtab.c ».
  - « matrice.c » est le fichier de la seconde partie. De même nous avons fait un header « matrice.h » afin de pouvoir avoir un fichier « testMatrice.c » et « testProduitMatrice.c ».
  - Enfin nous avons mis en place un Makefile afin de simplifier la compilation.

Exercice 1 :

Partie 1

1.1) Le programme est censé remplir le tableau en commençant par (longueur du tableau -1) jusqu'à 0. Il n'y a pas d'erreur à la compilation. Cependant nous rencontrons une *segmenation fault* à l'exécution du programme.

1.2) Après l'itération où i vaut 0, i vaut 294967295. Nous nous attendions cependant à ce que i soit égal à -1. On essaie donc d'accéder à tab[294967295]. Cette *segmentation fault* s'explique par le fait que nous avons essayé d'accéder à une case qui n'existe pas dans le tableau.

1.3) Pour résoudre cette erreur il suffit d'enlever le *unsigned* devant la variable i, pour qu'elle puisse devenir négative et que l'on puisse donc sortir de la boucle.

Partie 2

1.4) Le programme est censé créer une adresse et afficher les différentes composantes de la structure fraichement créer (nom, rue, code_postal) à l'aide d'un *printf*. Lors de la compilation il n'y a pas de soucis. Cependant nous avons une *segmenation fault* à l'exécution du programme.

1.5) *print new->rue* nous donne une adresse correspondant à 0x0. Nous observons ligne 16 une *segmentation fault*. Cette erreur survient au lancement de la fonction *strcpy*.
La chaine *new->rue* n'a pas été allouée avant d'être appelée dans *strcpy.* Or cela doit être fait. Pour résoudre le problème, nous pouvons donc allouer la chaine new->rue de manière dynamique avec un *malloc(sizeof(char)*n))*, *n* pouvant être choisi de façon arbitraire (ex : 200 pour être large).

Partie 3

1.6) Le programme est censé créer une structure de type *Tableau** et initialiser le tableau de cette structure à 100 cases (de manière dynamique). Nous ajoutons ensuite des valeurs aux 5 premières cases. Puis nous affichons les n premières cases (de tab[0] à tab[(t->position)-1]) du tableau contenu dans cette structure.
La compilation et l'exécution ont l'air de bien se passer. L'affichage attendu est celui obtenu.

1.7) Il y a un problème de fuite de mémoire.

1.8) En exécutant le programme avec Valgrind nous remarquons une fuite de mémoire de 400 bytes. Cette fuite provient donc de *t->tab* car un *int* correspond à 4 bytes et ici la taille de *t->tab* est de 100 donc 400 bytes.

1.9) Pour résoudre ce problème il faut libérer *t->tab* avec la commande *free(t->tab)* et ensuite libérer *t* avec *free(t).*

Exercice 2 :

Partie 1

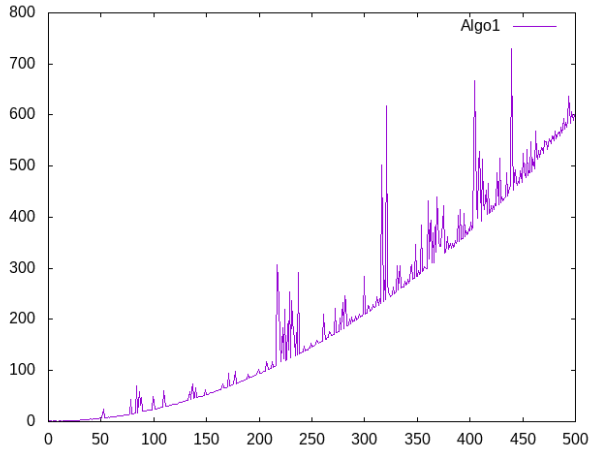
2.1) Nous avons choisi le passage par référence car cela permet de modifier directement le tableau dans le *main* sans passer par une

copie. Passer par une copie du tableau est beaucoup plus couteux et donc moins intéressant.

2.2) Dans la 2ème partie de la question il faut développer et réduire la somme que nous avons eu précédemment pour obtenir quelque chose de la forme (xi * yi) et ensuite utiliser la formule donnée dans l'énoncé. Après avoir fait cela nous obtenons quelque chose du type : 2 * ((n * S(tab[i])) – (S(tab[i]))^2). Avec S qui correspond à la somme de i=1 à n, n étant la taille du tableau.

2.3) Nous pouvons voir que le temps mis par le CPU est inferieur pour la fonction de complexité O(n) que pour la fonction de complexité O(n^2). Le temps passe en général de 3-4s à 1-2s avec la nouvelle version.

temps_ cpu

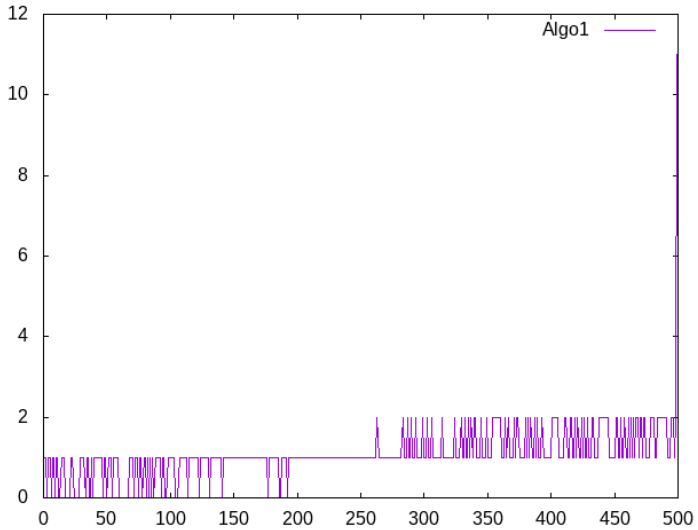


nb_tour

Titre : courbe sumcarre_1

Avec cette courbe nous voyons que le temps mis par le CPU pour exécuter la fonction devient de plus en plus grand avec l'augmentation de la taille du tableau

temps_ cpu



Titre : courbe sumcarre_2

Avec la courbe de sumcarre_2 on voit que le temps mis par le CPU pour exécuter cette fonction est beaucoup plus faible que la première fonction

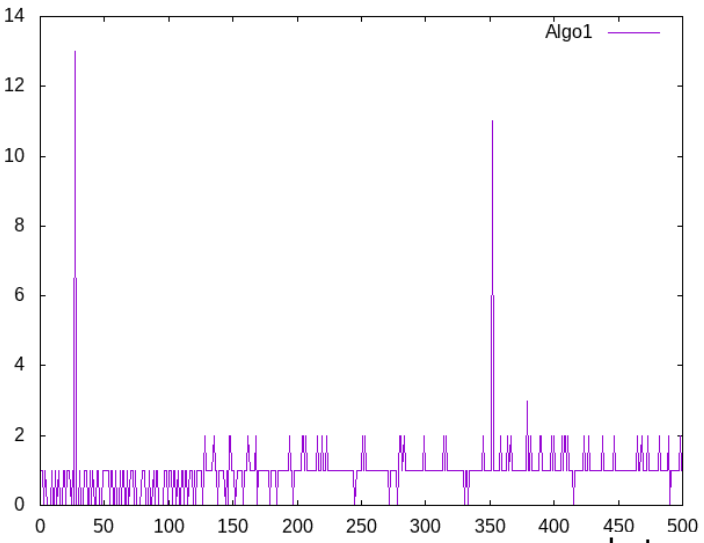
nb_tour

Partie 2

2.4) Voir code.

2.5.1) Voir code commentaire
2.5.2) Voir code, et commentaire
2.5.3) On ne voit pas une grande différence en terme d'allure de courbe car la fonction s'arrête lorsqu'un élément différent est trouvé. Mais la deuxième fonction possède certainement une meilleur complexité temporelle avec deux tour de boucle en moins

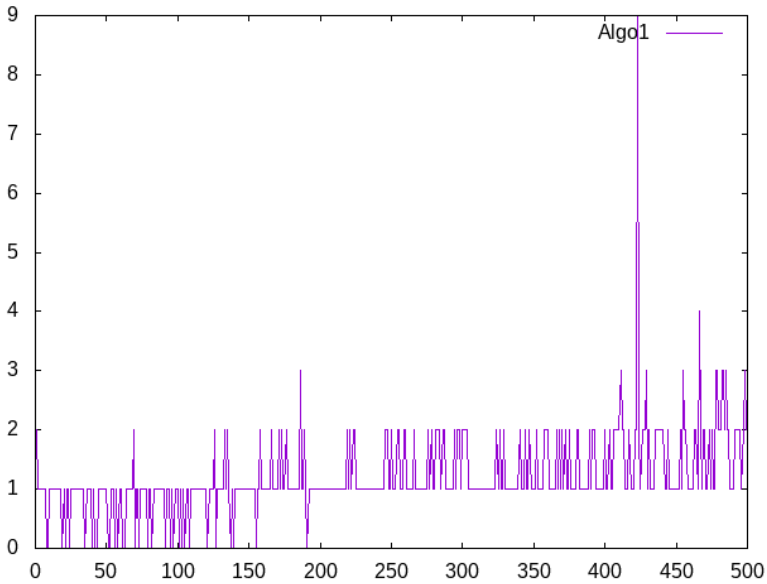
temps_cpu



nb_tour

Titre : courbe all_diff_1

temps_cpu



Titre : courbe all_diff_2

nb_tour

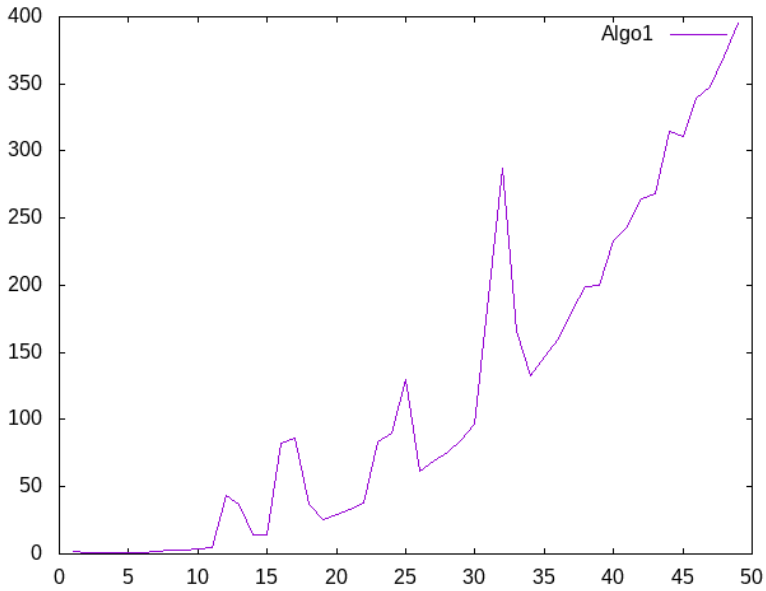
2.6.1) Voir code.

2.6.2) Voir code.

2.6.3) Le premier algorithme est de complexité $O(n^3)$ et le second de complexité $O(n^2)$, car la boucle for(k = 0 ; k < tmp ; k++) n'est pas de complexité O(n) mais de O(1). Cela s'explique par le fait que les conditions au-dessus de la boucle, tmp soit une constante donc de complexité O(1) et non de O(n).

2.6.4) En observant ces deux graphes on voit que le temps mis par le CPU pour la courbe produit_matrice2 est bien inférieur à la courbe produit_matrice1, ce qui montre bien une meilleure complexité de notre deuxième fonction

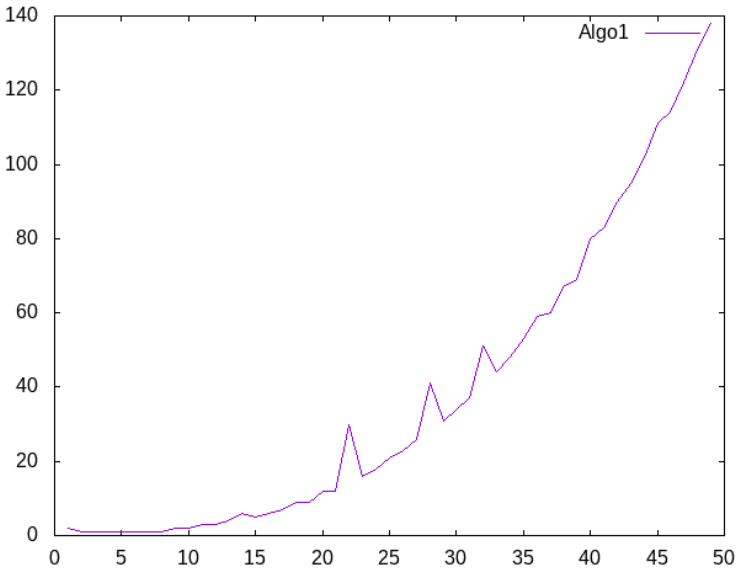
temps_cpu



Titre : courbe produit_matrice1

nb_tour

temps_cpu



Titre : courbe produit_matrice2

nb_tour