

## 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 :
  - o « tab1.c » est le fichier de la première partie. Nous avons fait un header « tab1.h » afin de pouvoir avoir un fichier qui contient le main « testtab.c ».
  - o « 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 ».
  - o 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 *segmentation 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 fraîchement créée (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 *segmentation 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 chaîne *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 chaîne *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 :

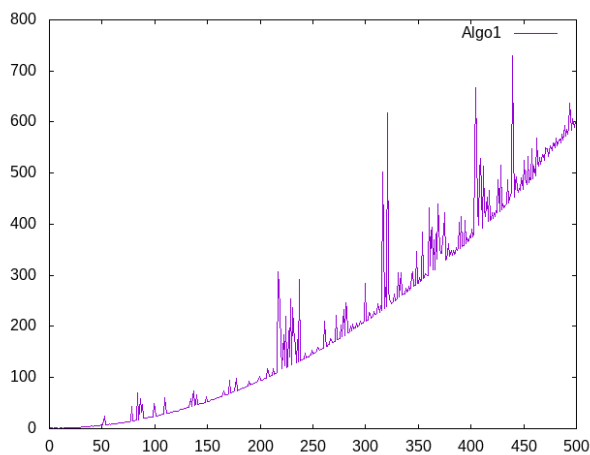
### Partie 1

- 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 2<sup>ème</sup> partie de la question il faut développer et réduire la somme que nous avons eu précédemment pour obtenir quelque chose de la forme  $(x_i * y_i)$  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

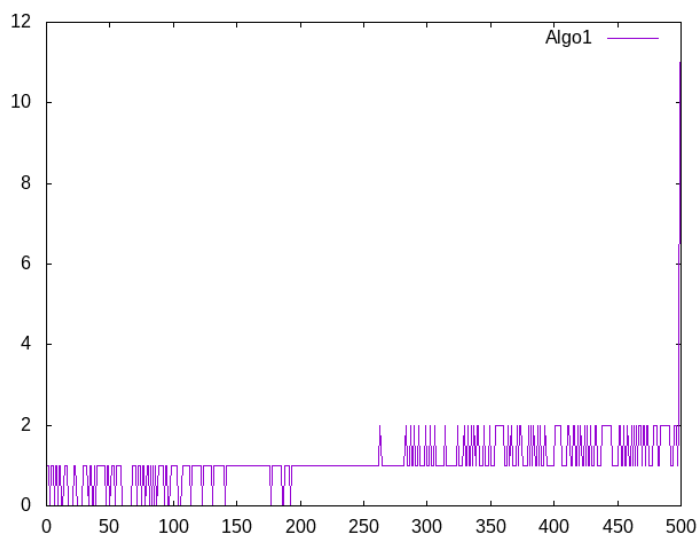


**Titre : courbe sumcarre\_1**

**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**

nb\_tour

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.

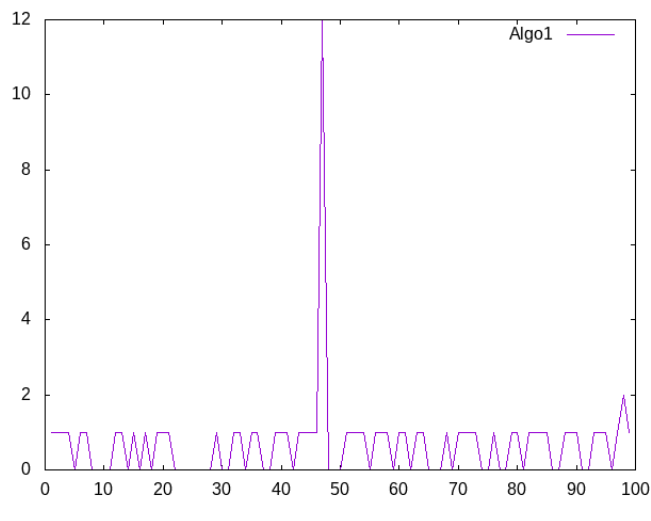
2.5.1) Voir code commentaire ; une fonction de complexité  $O(n^4)$

2.5.2) Voir code, et commentaire ; une fonction de  $O(n^2)$  (sans compter memset)

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. De plus, lors des piques d'incertitude, la première fonction a une pique beaucoup plus haute que la 2eme, ce qui prouve également une meilleur complexité de la 2eme fonction

**Titre : courbe all\_diff\_1**

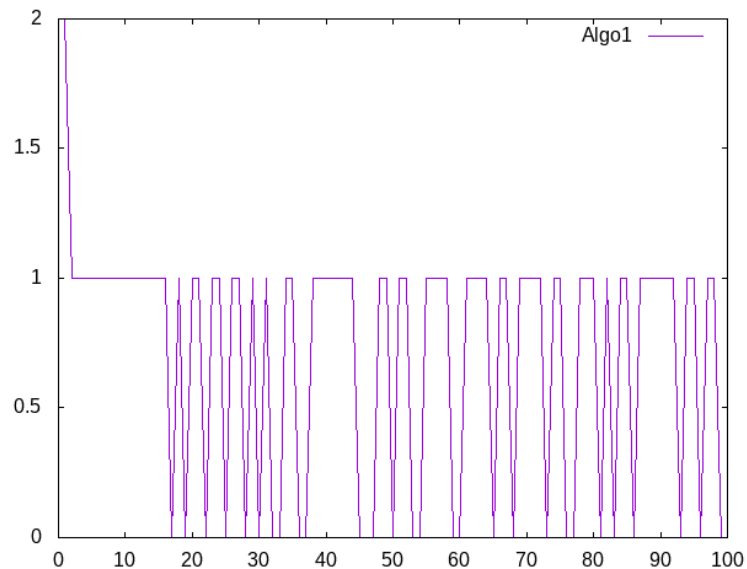
temps\_  
cpu



nb\_tour

temps\_  
cpu

**Titre : courbe all\_diff\_2**



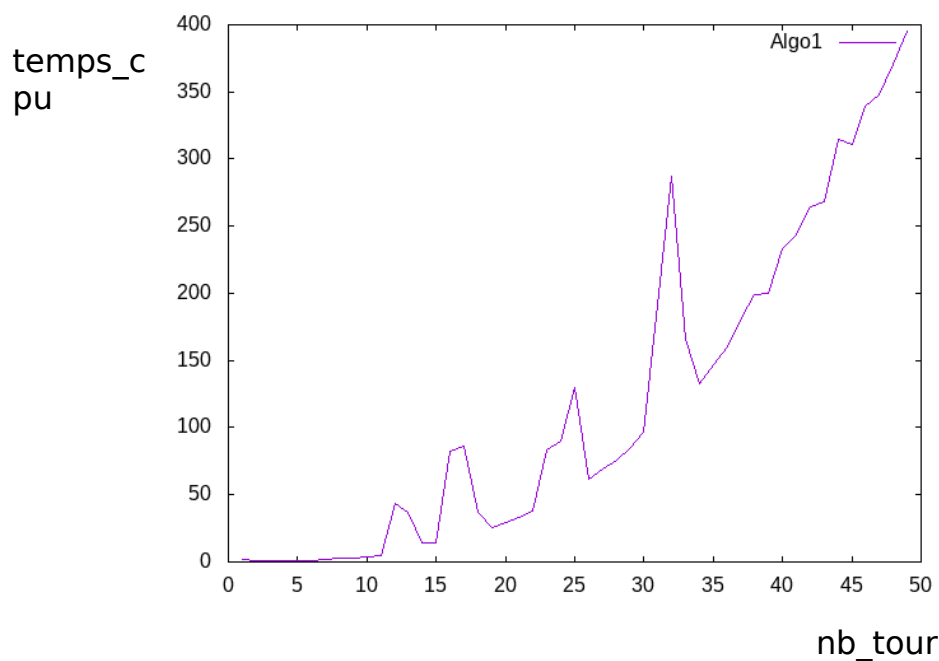
nb\_tour

2.6.1) Voir code.

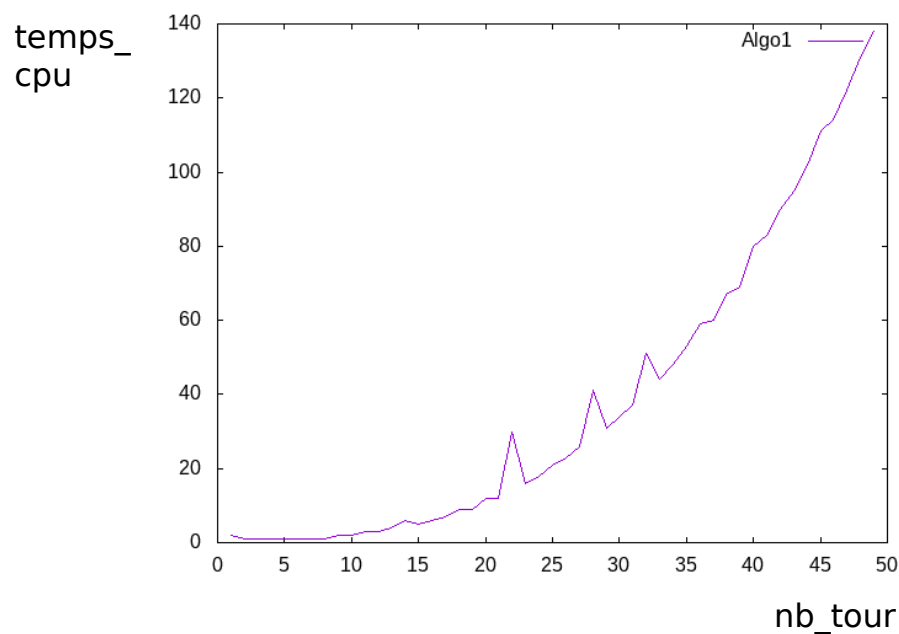
2.6.2) Voir code.

2.6.3) Non ce dernier algorithme n'a pas une meilleur complexité, car ce deuxième algorithme est borné par  $O(n^3)$ , et notre première algorithme est aussi borné par  $O(n^3)$ . Or notre deuxième algorithme avec les conditions, la troisième tour de boucle est en de  $O(tmp)$  avec  $tmp \leq n$ , donc notre 2ème fonction va être théoriquement plus rapide mais reste toujours de complexité  $O(n^3)$ .

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 que même si les deux fonction sont tous borné par  $O(n^3)$  la deuxième reste toujours plus efficace.



**Titre : courbe  
produit\_matrice1**



**Titre : courbe  
produit\_matrice2**