COCAULT Éléa - CARBONEAU Danaël

IN006 - Structures de Données

Compte Rendu

TME 1-2

Ce mini projet a pour but d'explorer les principales erreurs présentes en C, les moyens d'y remédiern ainsi que de dresser une première approche de la complexité algorithmique sur des exemples de structures simples (tableaux à une et deux dimensions).

Pour cela, nous avons choisi de découper notre code, et notamment celui du second exercice, en créeant deux fichiers `fonctions_tableau.c` et `fonctions_matrices.c` accompagnés de leurs headers afin de les réutiliser pour les fichiers des différentes questions.

Nous avons également fait un makefile facilitant la compilation de tous nos fichiers. Pour le second exercice, nous avons également conservé pour chaque question de complexité le fichier sur lequel sont inscrites les vitesses de calcul.

Exercice 1

Partie 1

Question 1.1

Le programme alloue la mémoire dynamiquement à un tableau de `len = 10` entiers `tab` et le remplit en partant du dernier indice avec la valeur dudit indice. Après cela, il libère la mémoire allouée à `tab`. Aucune erreur n'apparaît à la compilation, mais une erreur apparaît à l'exécution `Erreur de segmentation (core dumped)`.

Question 1.2

Après l'étape où `i` vaut 0, `i` prend la valeur 4294967295, qui est toujours positive, ce qui ne fait alors pas respecter la condition d'arrêt de boucle. `i` devrait valoir -1. Lors de l'exécution de la ligne 13 de ce tour de boucle, on cherche à accéder à l'indice 4294967295 de `tab`, ce qui est hors de la mémoire allouée à `tab` par notre programme. C'est là l'origine de l'erreur de segmentation (on essaie d'accéder à de la mémoire à laquelle on n'a pas accès).

Question 1.3

Puisque nous voulons que `i` vaille -1 à la fin de l'exécution de notre boucle for, il ne peut pas être "unsigned" lors de sa déclaration (cela le force à être toujours positif). Il faudrait alors retirer le unsigned de sa déclaration.

Partie 2

Question 1.4

La fonction `creer_adresse` prend en paramètre un entier `n`, une chaîne de caratères passée par adresse `r` et un entier `c`. Elle alloue la mémoire à une adresse `new`, initialise chacun des champs de la structure avec les variables passées en paramètres (en faisant un `strcpy` pour `r`) et retourne le pointeur vers cette adresse (`new`).

Le main créé une adresse à l'aide de la fonction `creer_adresse` puis en affiche les données.

Aucune erreur n'apparaît à la compilation, mais il y a une erreur de segmentation à l'exécution.

Question 1.5

L'affichage de `new->rue` à la ligne 15 nous donne `0x0`. Lorsqu'on veut continuer, gdb nous signale une erreur de segmentation, le programme s'arrête là. La cause en est que le champ `rue` de la structure `Adresse` n'a aucune mémoire allouée alors même que c'est un pointeur. Lors de l'appel à `strcpy`, l'on tente de copier les caractères de la chaine de caractères donnée ("manoeuvre") dans le champ `rue`. Cette opération nécessite d'accéder en mémoire au champ pour y mettre de nouvelles valeurs. De fait ici, on essaie d'accéder à une zone mémoire non allouée, et on a une erreur de segmentation.
Une solution serait de remplacer la ligne 15 par : `new->rue = strdup(r);` . Cette fonction permet d'allouer au préalable la mémoire nécessaire à stocker la chaîne de caractères donnée en argument, avant d'effectuer la recopie.

Partie 3

Question 1.6

Le programme créé une structure `tableau`, trois fonctions permettent d'y ajouter un élément (`ajouterElement`), de l'initialiser avec une valeur `maxTaille` (`initTableau`) et d'afficher le tableau (`afficherTableau`).

Le main initialise un `Tableau` (structure). Dans son champ `tab`, il ajoute à la suite 5 éléments avant de l'afficher. Enfin, il libère la structure.
Tout va bien à la compilation et à l'exécution.

Question 1.7

Il y a un problème de fuite mémoire : la mémoire allouée à la structure n'est pas complètement libérée. En effet, lorsqu'on fait `free (t)`, on libère le pointeur sur la structure, mais pas le tableau (`tab`) alloué lors de l'initialisation.

Question 1.8

`Valgrind` repère effectivement une fuite mémoire : `LEAK SUMMARY: ==4802== definitely lost: 400 bytes in 1 blocks`.
Comme expliqué plus tôt, es 400 bytes correspondent au tableau de taille 100 alloué dynamiquement à l'appel de `initTableau` à la `ligne 16`.

Question 1.9

On a proposé dans le code de créer une fonction `libererTableau` qui s'occupe de libérer toute la mémoire allouée dynamiquement, à laquelle on fait ensuite appel[1]

[1] On aurait aussi pu simplement rajouter `free(t->tab); free(t);` directement dans le main.

Exercice 2

Partie 1

Question 2.1

1. Le passage par référence permet de choisir la valeur de retour comme on le souhaite (pour tester nos fonctions par exemple).

2. Voir `fonctions_tableau.c` pour le code des fonctions.

note : Nous avons mis à part les fonctions sur les tableaux de la partie 1 dans le ce fichier et créé un header `fonctions_tableau.h` *afin de pouvoir les réutiliser dans la suite de l'exercice, pour nos tableaux à deux dimensions.*

Question 2.2

1. Le code est à retrouver dans `tme1_exo2p1.c`.
   `somme_v1` est bien de complexité en $O(n^2)$ car à chaque itération de la première boucle, on fait n itérations de la seconde.

2. On veut, pour réduire la complexité, ne faire qu'une seule boucle dans `somme_v2`, pour cela, il va falloir éliminer de notre formule une des deux sommes :

$$\sum_{i=1}^{n}\sum_{j=1}^{n}(x_i - x_j)^2 = \sum_{i=1}^{n}\sum_{j=1}^{n} x_i^2 - 2 \times x_i x_j + x_j^2 \tag{1}$$

$$= n \times \sum_{i=1}^{n} x_i^2 + n \times \sum_{j=1}^{n} x_j^2 + \sum_{i=1}^{n}\sum_{j=1}^{n} -2 \times x_i x_j \tag{2}$$

Puisque i et j varient sur le même intervalle, les deux sommes sont égales et on peut factoriser.

$$= 2 \times n \times \sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n}\sum_{j=1}^{n} -2 \times x_i x_j \tag{3}$$

$$= 2 \times n \times \sum_{i=1}^{n} x_i^2 - 2\sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j \tag{4}$$

On utilise la formule donnée dans l'énoncé pour supprimer la seconde somme.

$$= 2 \times n \times \sum_{i=1}^{n} x_i^2 - 2(\sum_{i=1}^{n} x_i)^2 \tag{5}$$

Cette nouvelle formulation du problème (5) nous permet désormais d'implémenter `somme_v2` avec une seule boucle (voir `fonctions_tableau.c`). Puisqu'il n'y a qu'une seule boucle, `somme_v2 est en` $O(n)$, ce qui est bien d'une complexité inférieure à `somme_v1`

Question 2.3

(Pour le main, voir code : `tme1_exo2p1.c`)

La différence entre les temps de calculs est flagrante. Pour peu d'opérations (une centaine réalisées de 1 en 1), l'algorithme 1 en complexité $O(n^2)$ augmente rapidement en temps de calcul alors que le second en $O(n)$ semble évoluer quasiment constamment et se maintenir à des valeurs bien plus faibles (Figure 1).

L'évolution exacte du temps de calcul devient plus clair lorsqu'on augmente le nombre d'itérations à représenter. Sur le second graphique (Figure 2), on obtient une demi-parabole pour le premier algorithme. Cependant, la droite du second, décrivant supposément une augmentation linéaire du temps de calcul, reste invisible sur notre graphique car devenue négligeable par rapport au premier.

On peut donc conclure sur l'immense avantage du second algorithme en matière de temps de calcul.

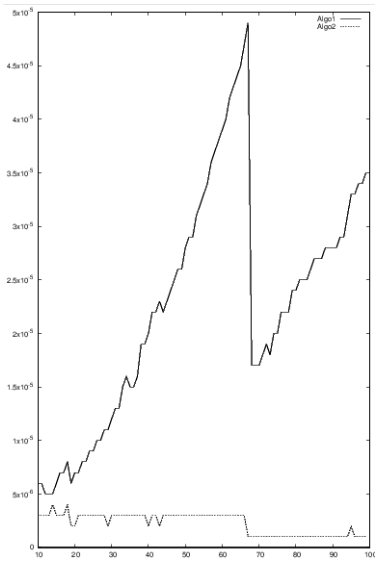

Figure 1: évolution du temps de calcul de `somme_v1` (ligne continue) et de `somme_v2`(pointillés) en fonction de la taille du tableau

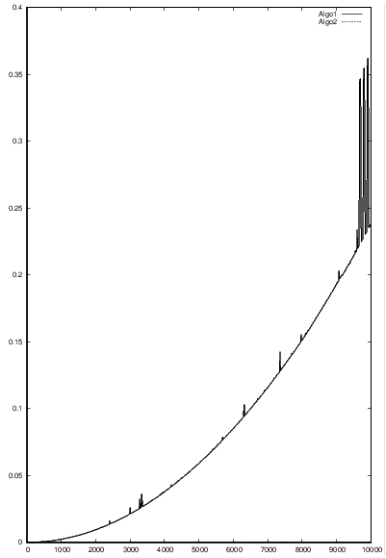

Figure 2: évolution du temps de calcul de `somme_v1` (ligne continue) et de `somme_v2`(pointillés) en fonction de la taille du tableau allant jusqu'à 10 000

Partie 2

Question 2.4

Note : comme pour la partie 1 de l'exercice, nous avons choisi de mettre nos fonctions élémentaires sur les matrices dans un fichier `fonctions_matrice.c` *afin de pouvoir les réutiliser dans la suite du programme. Les raisons du passage par adresse pour la fonction* `alloue_matrice` *s'explique de la même manière que pour la première partie de l'exercice.*

Question 2.5

1. On a fait un deux boucles pour accéder à chaque élément de la matrice, puis on fait à nouveau deux parcours afin de comparer tous les autres éléments de la matrice avec notre courant (en prenant soin de ne pas tenir compte du cas où on le compare avec lui-même, rôle assuré par l'entier `meme_element`.

2. Puisqu'on sait que nos éléments ne peuvent dépasser V, il suffit alors de créer un tableau de taille V qui contiendra donc dans ses indices toutes les valeurs que peuvent prendre nos

indices. On initialise toutes ses cases à 0 (ce qui signifie qu'a priori aucune n'apparaît), puis on modifie, au fur et à mesure, ce tableau, en mettant à 1 la valeur des cases dont l'indice a été rencontré dans la matrice, un simple test sur cette valeur suffit alors à savoir si la valeur est déjà présente dans la matrice. Il ne faut alors que deux boucles pour parcourir la matrice, ainsi qu'une pour remplir `tab_comparaison`. Notre algorithme est donc de complexité $O(n^2)$ puisque v est une constante définie, indépendante de la taille de la matrice, et de ses valeurs (et qui devient négligeable asymptotiquement[2].

3. La complexité de notre second algorithme dépendant également de la valeur de V, mais la fixer risquerait également de fausser notre analyse de la complexité : si la valeur de V est plus petite que n^2, alors il y aura forcément au moins deux valeurs identiques et selon leur position dans la matrice, une sortie anticipée viendrait "fausser" notre appréhension de la complexité temps-pire-cas. Il a donc été décidé d'indexer la valeur de V sur n^2, de sorte qu'il y ait 5 fois plus de valeurs possibles que d'éléments dans la matrice, ce qui permet une plus grande diversité dans nos tests et d'approcher du "pire cas" (il aurait été mieux d'augmenter ce facteur mais la mémoire de l'ordinateur nous limite pour de grandes valeurs).

   En observant ces graphiques, on voit que le premier algorithme est beaucoup plus "instable" (Figure 3) en terme de nombre d'opérations réalisées, ce qui vient des sorties anticipées, l'aléatoire, notamment dans le placement des éléments rend le nombre d'opérations très instable.

   Au contraire, le graphique représentant l'évolution du temps de calcul nécessaire au second algorithme (Figure 4) en fonction de la taille du tableau est assez régulier quant au temps de calcul, car les sorties anticipées ne dépendent alors pas de la position dans la matrice du second élément.

   On peut donc en conclure que le second algorithme est plus efficace, ce qui est en adéquation avec la comparaison des complexités asymptotiques.

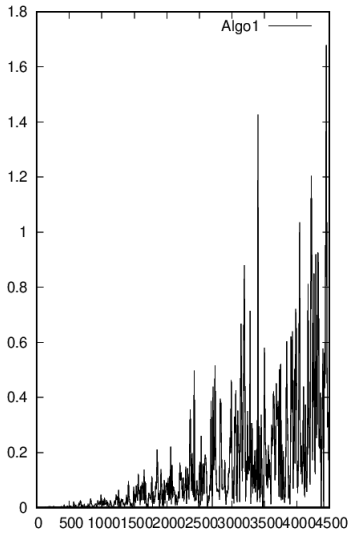

Figure 3: Graphe représentant l'évolution de la complexité temps-pire-cas de la fonction `mat_diff_v1`

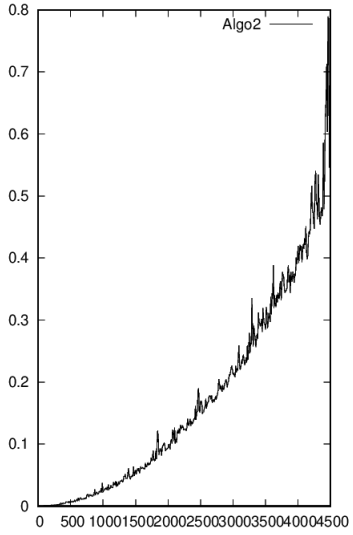

Figure 4: Graphe représentant l'évolution de la complexité temps-pire-cas de la fonction `mat_diff_v2`

[2]pour de très grandes valeurs de n

Question 2.6

1. (voir code : tme1_exo2p2_q6.c)
   On fait une fonction avec trois boucles imbriquées parcourant la matrice résultat ainsi que les deux matrices dont on fait le produit, chacune de ces boucles faisant dans le pire cas n itération, la complexité de `pdt_mat_v1` est donc bien en $O(n^3)$

2. Supposant maintenant que les deux matrices sont triangle respectivement triangle supérieur pour `mat1` et triangle inférieur pour `mat2`. On peut alors réduire notre troisième boucle car l'indice k ne sera plus nécessaire pour certaines valeurs.

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=j}^{n-1} 1 = \frac{1}{2}n^2(n+1) \tag{6}$$

   Cette complexité est également en $O(n^3)$, mais en temps de calcul concret, on en réalise moins avec notre second algorithme car on parcourt moins d'éléments de nos matrices (uniquement les triangles) lors du calcul des coefficients.

3. L'algorithme 1 et l'algorithme 2 sont tous deux de même complexité temps pire cas et ont donc un comportement asymptotique similaire, en $O(n^3)$, leur complexité temps pire cas est donc, théoriquement, identique.

4. Cependant, on remarque effectivement (voir Figure 5) une évolution plus lente du temps de calcul pour le premier algorithme. On note de plus que cette différence reste importante même pour des matrices dont la taille (1000x1000) commence à être conséquente : on remarque presque 1,5 seconde de différence entre nos algorithmes.

   Le second algorithme, s'il demeure malgré tout en complexité apparente identique au premier, reste en réalité dans la pratique considérablement moins coûteux en terme de calculs (mais nécessite une précondition très restrictive sur nos matrices).

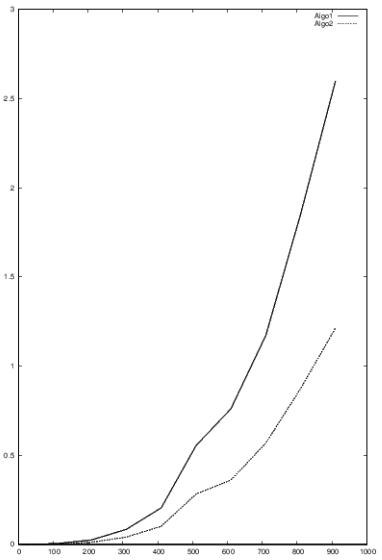

Figure 5: Graphe représentant l'évolution de la complexité temps-pire-cas de nos deux fonctions (`pdt_mat_v1` en plein et `pdt_mat_v2` en pointillé) en fonction de la taille de la matrice carrée