GUERY Alexandre
PIASENTIN Béryl

Compte-rendu du TP5

QuickSort :

Q1) La fonction renvoie 1 si le tableau passé en argument est trié par ordre croissant, 0 si ce n’est pas le cas. Pour cette fonction, afin de pouvoir tester des tableaux triés et les cas limites, on a determinisé les tests : N doit donc être supérieur ou égal à 10.

Q2) La fonction chapeau (qsortu) appelle la fonction récursive (qsortuRec) sur tout le tableau, donc avec d = 0 et f = Tlen – 1. Le cas terminal de la fonction récursive est le tri d’un seul élément : d = f donc f-d = 0. Dans les autres cas, on applique la partition puis on appelle à nouveau notre fonction de tri sur la partie du tableau que l’on trie avant le pivot, et sur cette même partie après le pivot.
La fonction testQsortu affiche un tableau aléatoire de taille N non trié, puis une fois passé dans la fonction qsort, et termine par afficher s’il est trié à l’aide de la fonction sorted.
Pour évaluer la complexité, on a lancé la fonction evalComplexite (qui crée un tableau aléatoire de taille N, le trie, et vérifie le trie à l’aide de la fonction sorted) avec les valeurs de N proposées. Voilà les résultats obtenus pour une exécution : (N : temps)
1 000 000 : 0.184s
10 000 000 : 2.137s
20 000 000: 4.285s
40 000 000: 9.035s
80 000 000: 18.524s
160 000 000: 38.964s
Pour une entrée de taille 2*n, le temps est d'environ 2*Temps(n). La complexité est donc linéaire.

Q3) Si les éléments sont triés par ordre décroissant, à chaque appel de la fonction récursive l’élément pivot (le dernier du tableau, stocké dans la variable x) sera placé au début du bout de tableau à trier, ce qui engendrera un décalage de tous les éléments : la complexité sera linéaire pour chaque appel, donc quadratique.
Voici les valeurs obtenues en fonction de la valeur de N :
25 000 : 1.057s
50 000 : 4.271s
100 000 : 16.711s
200 000 : 51.805s
Le temps semble être quadruplé lorsque la taille de l'entrée est doublée : la complexité semble bien être quadratique si le tableau est décroissant.

Une solution est de prendre un pivot aléatoire (sans oublier de commencer par placer le dernier élément à la place du pivot). En faisant cela, on obtient les résultats donnés par la evaluationComplexiteQsortuDecroissantAlea :
250 000 : 0.035s
500 000 : 0.058s
1 000 000 : 0.111s
2 000 000 : 0.244s
Le temps semble doubler lorsque la taille de l'entrée est doublée : la complexité est donc devenue linéaire, lorsqu’on prend un pivot aléatoire.

Q4) La fonction qsortuStd a le même effet que la fonction qsortu avec le prototype de qsort : on a changé les T[a] = T[b] en memcpy(T+a*size,T+b*size,size) car T est de type void.
Pour les tableaux aléatoires, voici les temps obtenus en fonction de N, pour le tri par qsortuStd :
1 000 000 : 0.360s
10 000 000 : 3.937s
20 000 000: 8.265s
40 000 000: 16.933s
80 000 000: 35.451s
160 000 000: 1min12.566s
La complexité est linéaire, mais les valeurs sont deux fois plus élevées qu'avec la fonction qsortu.

On peut faire les mêmes constatations dans le cas décroissant, comme le montrent les valeurs :
250 000 : 0.075s
500 000 : 0.110s
1 000 000 : 0.264s
2 000 000 : 0.518s

Voilà les résultats que nous donne qsort, de la bibliothèque standard (obtenus à l'aide des fonctions evaluationComplexiteQsort et evaluationComplexiteDecroissantQsort) :
Pour les tableaux remplis aléatoirement :
1 000 000 : 0.240s
10 000 000 : 2.616s
20 000 000: 5.507s
40 000 000: 11.462s
80 000 000: 23.807s
160 000 000: 49.670s
Pour les tableaux décroissants :
250 000 : 0.024s
500 000 : 0.035s
1 000 000 : 0.072s
2 000 000 : 0.130s
Dans les deux cas, la complexité est linéaire.

Récapitulatif des résultats :

Tableau rempli aléatoirement

	qsortu	qsortuAlea	qsortuStd	qsort
1 000 000	0.184s	0.200s	0.360s	0.240s
10 000 000	2.137s	2.236s	3.937s	2.616s
20 000 000	4.285s	4.718s	8.265s	5.507s
40 000 000	9.035s	9.685s	16.933s	11.462s
80 000 000	18.524s	20.183s	35.451s	23.807s
160 000 000	38.964s	41.683s	1min12.566s	49.670s

Tableau décroissant

	qsortuAlea	qsortuStd	qsort
250 000	0.035s	0.075s	0.024s
500 000	0.058s	0.110s	0.035s
1 000 000	0.111s	0.264s	0.072s
2 000 000	0.244s	0.518s	0.130s

On n’a pas placé qsortu dans ce tableau, car on n’a pas pris les mêmes N, les temps étaient trop longs (Si on fait 10*N, le temps est multiplié par 100).

Dans le cas d’un tableau quelconque, c’est qsortu qui est la plus rapide. Si le tableau est décroissant, c’est qsort qui l’est. Dans tous les cas, la complexité est linéaire.

Le fait que qsortuStd soit plus lente que les autres peut s’expliquer en partie par les appels à la fonction compare, et les memcpy.

RadixSort :

Q5) testDenomsort permet d’afficher un tableau avant et après qu’il soit trié par denomsort_r. Le tri consiste en un tri suivant les deux bits de poids faibles (et uniquement ceux-là).
Pour trier un tableau de 60 millions de nombres de 8 bits, la fonction Denomsort met environ 1.940 seconde (on l’a appelée sur les 8 bits à la position 0, donc les nombres entiers)
Lorsqu’on essaye d’exécuter le tri de 60 millions de nombres de 8 bits via qsortuStd sur 32 bits, on obtient une erreur de mémoire : le serveur turing.e.ujf-grenoble.fr n’a pas assez de mémoire. On ne peut donc pas comparer. Ce n’est pas le nombre d’éléments qui posent problèmes, mais le fait que bien trop de nombres soient identiques.
17 secondes 129 ont été nécessaires pour trier un tableau de 60 000 000 de nombres de 24 bits par qsortuStd, contre 27 secondes 396 dans le même cas pour qsortuStd : il est plus efficace de trier les nombres selon leur nombre de bits, à l’aide de radixsort ! (Le modulo étant plus grand, on n’a plus de problèmes de mémoire).
Q6)

Test de la fonction radixsortu avec un tableau de 240 000 000 nombres de 24bits:

	2^2	2^3	2^4	2^6	2^8	2^12	2^24
24 bits	33.587s	24.216s	19.726s	16.836s	14.072s	12.907s	problème

Le problème pour 2^24 vient d’un manque de mémoire, de la même manière que pour qsortuStd.

Test de la fonction radixsortu avec un tableau de 240 000 000 nombres de 32bits:

	2^2	2^4	2^8	2^16	2^32
32 bits	33.544s	18.877s	13.699s	10.554s	4.761s

On a choisi les bases suivantes car elles divisent 32 pour faire une nombre entier de fois denomsort.

Q7) Pour cette question, on choisit la base 2^12 car c’est la plus rapide d’après la question précédente, pour des nombres de 24 bits.

	1000	10 000	100 000	1 000 000	50 000 000	500 000 000
qsortu	0.003s	0.008s	0.043s	0.366s	22.219s	4m27.564s
radixsortu	0.006s	0.007s	0.013s	0.053s	2.704s	23.510s

Qsortu est quadratique, on le revoit bien ici. Radixsortu est linéaire. Les différences de temps sont donc normales.
On remarque que pour 1000 valeurs, qsortu est plus rapide : les complexités dont on parle sont asymptotique, donc c’est normal.