TP6 : Tri rapide en place

MP2I Lycée Pierre de Fermat

Dans ce sujet, pour T un tableau et a, b des indices, la notation $T[a..b]$ indique le sous-tableau composé des cases $T[a], T[a+1], \ldots, T[b]$. De même, la notation $T]a..b[$ indique le sous-tableau composé des cases $T[a+1], \ldots, T[b-1]$. On pourra écrire $T[a..b[$ et $T]a..b]$ pour exclure seulement l'une des extrémités.

Étude théorique

L'algorithme de tri rapide vu en cours **renvoie une copie triée** de son entrée, et n'est donc pas un algorithme en place : il nécessite de réserver de l'espace additionnel. On se propose d'implémenter une version du tri rapide qui travaille directement dans le tableau d'entrée. Le cœur de cet algorithme est la procédure de **partition** :

Algorithme 1 : `partition`(T, n)

Entrée(s) : T tableau de taille n

Sortie(s) : T est modifié, l'ancienne valeur de $T[0]$ est maintenant à un indice j tel que $\forall x \in T[0..j[, x \leq T[j]$ et $\forall x \in T]j..n[, x > T[j]$

La procédure de partition permet donc de couper un tableau en deux selon le pivot $T[0]$: les éléments inférieurs ou égaux à gauche, les éléments strictement supérieurs à droite. Tout cela est fait en place, sans réserver de mémoire additionnelle.

Par exemple, si $T = [5, 2, 7, 8, 1, 5, 9, 8, 6, 2]$, après avoir appelé **Partition**(T), on a séparé T en utilisant $T[0] = 5$ comme pivot. Après l'exécution de l'algorithme, T peut donc contenir $[2, 1, 5, 2, \mathbf{5}, 7, 8, 9, 8, 6]$.
L'ordre au sein de chaque partie est arbitraire et dépend de l'implémentation de l'algorithme. Dans l'exemple précédent, l'ordre choisi est celui d'apparition dans le tableau T avant la partition.

Implémentons l'algorithme de partition. Le principe est de maintenir deux zones dans le tableau : une qui contient les éléments $\leq T[0]$ et qui grandit depuis le bord gauche du tableau, et une qui contient les éléments $> T[0]$ et qui grandit depuis le bord droit.
On pourra donc utiliser deux variables i, s, avec $i = 1, s = n - 1$ initialement, qui marquent respectivement la fin de la zone de gauche et le début de la zone de droite. Une fois que l'on a rangé correctement toutes les cases, on déplace $T[0]$ au bon endroit.

Q1. Traduire le rôle des variables i et s en un invariant.

Q2. Proposer un algorithme en pseudo-code pour la partition, en suivant les indications précédentes et en respectant l'invariant trouvé à la question précédente. Complexité attendue : $\mathcal{O}(n)$.

Q3. Adapter l'algorithme pour qu'il puisse partitionner seulement une partie du tableau comprise entre deux indices a et b :

Algorithme 2 : `partition_entre`(T, n, a, b)

Entrée(s) : T tableau de taille n, $a, b \in [\![0, n-1]\!]$ avec $a \leq b$
Sortie(s) : T est modifié, l'ancienne valeur de $T[a]$ est maintenant à un indice $j \in [\![a, b]\!]$ tel que $\forall x \in T[a..j[, x \leq T[j]$ et $\forall x \in T]j..b], x > T[j]$

Q4. Écrire un algorithme récursif `tri_rapide_entre` permettant de trier une partie de T :

Algorithme 3 : `tri_rapide_entre`(T, n, a, b)

Entrée(s) : T tableau de taille n, $a, b \in [\![0, n-1]\!]$ avec $a \leq b$
Sortie(s) : T est modifié de sorte que $T[a..b]$ est trié.

En déduire un algorithme `tri_rapide(T, n)` permettant de trier un tableau selon l'algorithme de tri rapide.

Implémentation en C

Q5. Implémenter un tri par sélection ou par insertion, et le tester en utilisant une fonction `bool est_trie (int* T, int n)` déterminant si un tableau est trié dans l'ordre croissant.

Q6. Implémenter le tri rapide en place, et le tester.

Nous allons comparer les performances du tri rapide avec celles du tri quadratique choisi. Pour cela, nous devons mesurer leur temps d'exécution, mais la fonction `time()` ne permet de faire des mesures qu'à la seconde près.

La librairie `<time.h>` contient la fonction `clock_t clock()`. Cette fonction renvoie le temps écoulé depuis le lancement du programme, dans une unité arbitraire appelé les *clocks*.[1] Un **clock** vaut environ un millionième de seconde, ce qui permet d'avoir des mesures plus précises. Le nombre exact de **clocks** par seconde est accessible avec la macro `CLOCKS_PER_SEC`, ce qui permet de mesurer des durées comme suit :

```
1 clock_t debut = clock();
2 /* Code à chronométrer */
3 clock_t fin = clock();
4 float duree = (float) (fin - debut) / CLOCKS_PER_SEC;
```

Le `(float)` sert à éviter une division entière, qui empêcherait d'avoir des durées inférieures à la seconde.

Q7. Écrire une fonction `float test_tri_rapide (int n)` qui génère 20 tableaux aléatoires de taille n, les trie avec le tri rapide, chronomètre le tout, et renvoie le temps moyen écoulé par tableau, en secondes (on négligera le temps pris par la génération des tableaux).

Q8. Écrire une fonction analogue pour le tri quadratique choisi précédemment.

Q9. Mesurer le temps moyen d'exécution du tri rapide sur $n \in [100, 200, 300, \ldots 4000]$ et stocker les valeurs mesurées dans un fichier.

Q10. Faire de même avec le tri quadratique.

1. Si vous avez des notions d'architecture des ordinateurs : attention, un clock ne correspond pas à un tic d'horloge du CPU !

En python, tracer les courbes afin de comparer les deux algorithmes. Le code python suivant permet de lire dans un fichier et d'en extraire une liste de nombres :

```
1 f = open(nom_fichier, "r")
2 contenu = f.read() # chaîne de caractère avec tout le contenu du fichier
3 nombres = contenu.split() # sépare en sous-chaînes selon les espaces et les '\n'
4 nombres = list(map(float, nombres)) # transforme chaque sous-chaîne en flottant
5 f.close()
```

Par exemple, si le fichier contient :

```
0.21
1.37
52.12
```

alors après avoir lancé le code python ci-dessus, `nombres` sera la liste $[0.21, 1.37, 52.12]$.

Q11. Avec `matplotlib`, tracer les courbes des temps d'exécution des deux tris.

Q12. Quelles courbes peut-on tracer pour voir graphiquement les complexités en $\mathcal{O}(n \log n)$ et $\mathcal{O}(n^2)$?