TP6: Tri rapide en place

Dans ce sujet, pour T un tableau et a, b des indices, la notation $T[a..b]$ indique le sous-tableau composé des cases $T[a], T[a+1], \ldots, T[b]$. Par convention, si $a > b$, $T[a..b]$ représente un tableau **vide**, constitué d'aucune case. 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 l'une des extrémités seulement.

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** :

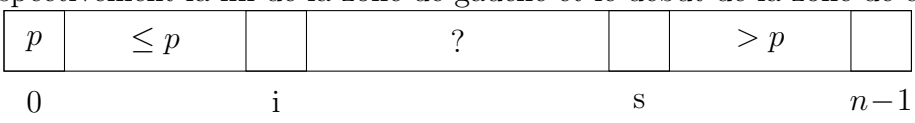
Algorithme 1 : `partition`(T, n)

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. *Réfléchissez bien à s'il faut inclure ou non i et s dans les zones qu'ils délimitent.*

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 une partie du tableau, comprise entre deux indices a et b, et pour qu'il renvoie l'indice du pivot après la partition :

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) : $j \in [\![a, b]\!]$ tel que l'ancienne valeur de $T[a]$ est maintenant à l'indice j, et $\forall x \in T[a..j[, x \leq T[j]$ 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 **en place** (c'est à dire sans allouer de mémoire additionnelle) :

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.

Q5. En déduire un algorithme `tri_rapide(T, n)` permettant de trier un tableau selon l'algorithme de tri rapide.

Notons que même si l'on dit que ce tri est en place, il ne va pas s'exécuter en utilisant un espace mémoire $\mathcal{O}(1)$. En effet, les appels récursifs font **grandir la pile d'appel**.

Q6. En fonction de n, quelle serait la taille maximale atteinte par la pile d'appel au cours du tri d'un tableau de taille n ?

Implémentation en C et chronométrage

Q7. 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.

Q8. 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()` qui permet de faire des mesures plus précises. 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. 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 forcer une division flottante, afin que le résultat puisse être un nombre à virgule.

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

Q9. 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).

Q10. Écrire une fonction analogue pour le tri quadratique choisi précédemment.

Q11. Mesurer le temps moyen d'exécution du tri rapide sur $n \in [100, 200, 300, \dots 4000]$ et stocker les valeurs mesurées dans un fichier.

Q12. Faire de même avec le tri quadratique.

☆ Compétences numériques ☆ : Comparaison des performances

Traçons les courbes des temps d'exécution obtenus aux questions précédentes, afin de comparer nos différents algorithmes. Le code python suivant permet de lire dans un fichier et d'en extraire une liste de nombres :

```
# ouverture en mode lecture
f = open(nom_fichier, "r")
# chaîne de caractère avec tout le contenu du fichier
contenu = f.read()
# sépare en sous-chaînes selon les espaces et les '\n'
nombres = contenu.split()
# transforme chaque sous-chaîne en flottant
nombres = list(map(float, nombres))
# fermeture du fichier
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 $[0.21, 1.37, 52.12]$.

Q13. Avec `matplotlib`, tracer les courbes des temps d'exécution du tri rapide, et du tri quadratique choisi.

Q14. Vérifier graphiquement que les complexités sont en $\mathcal{O}(n \log n)$ et $\mathcal{O}(n^2)$.

Pour aller plus loin : choix du pivot

Plutôt que de choisir le pivot aléatoirement, ou de toujours prendre le premier élément, on peut choisir un pivot qui a de bonnes chances de couper le tableau en deux parties équitables. Une méthode simple est de regarder trois éléments du tableau : le premier, le dernier, et celui du milieu. On prend alors la **médiane** des trois, ce qui réduit en pratique les chances de mauvais pivot.

Q15. Implémenter le choix de pivot des trois médianes, et comparer avec le tri rapide basique.

Pour aller encore plus loin : Tri hybride

Le tri par insertion en général plus lent que le tri rapide, mais nous avons vu en cours qu'il est **adaptatif** : il peut s'exécuter relativement vite sur des tableaux partiellement triés (et même en $\mathcal{O}(n)$ pour un tableau déjà trié).
Une amélioration du tri rapide consiste alors à arrêter les appels récursifs dès que l'on atteint des sous-tableaux d'une taille inférieure à un seuil K. On obtient alors un tableau qui n'est pas trié, mais qui est divisé en zones de tailles $\leq K$, de telle sorte que les zones sont triées entre elles. Par exemple pour $K = 3$, après avoir lancé le tri rapide partiel proposé ci-dessus sur le tableau suivant :

$$[6, 2, 3, 1, 8, 4, 9, 7, 2, 8, 3, 5]$$

on pourrait obtenir le tableau suivant :

$$[2, 1, 2, \quad 3, 4, 3, \quad 7, 6, 5, \quad 8, 8, 9]$$

Chaque bloc individuel de 3 éléments n'est pas trié, mais les blocs sont dans le bon ordre global : chaque élément du premier bloc est inférieur à chaque élément du deuxième bloc, et ainsi de suite.

Q16. A la main, appliquer un tri par insertion sur le tableau obtenu ci-dessus après le tri rapide partiel. Combien d'échanges ont été nécessaires ?

Q17. De manière général, en fonction de K et n, quel est le coût du tri par insertion sur un tableau divisé en blocs de taille K placés dans le bon ordre (mais où les éléments au sein de chaque bloc ne sont pas forcément triés) ?

Le tri hybride rapide-insertion consiste donc à appliquer le tri rapide partiel jusqu'à avoir partitionné le tableau en blocs d'au plus K éléments, puis à appliquer un tri par insertion sur le tableau obtenu.

Q18. Implémenter le tri hybride, et le tester avec $K = 100$, en le comparant au tri rapide pour des tableaux de tailles $10000, 20000, \ldots, 100000$.

Q19. Tester d'autres valeurs de K et les comparer.