TP 18 : Tri rapide, sélection rapide et plus courte distance entre deux points du plan

Dans ce TP, on va implémenter le tri rapide et le comparer aux autres algorithmes de tri déjà implémentés. On va aussi comparer le schéma de Lomuto implémenté en cours avec un nouveau schéma de partitionnement : le schéma de l'algorithme de tri rapide original, appelé schéma de Hoare, dont l'intérêt est de diminuer le nombre de permutations effectuées d'un facteur constant. On va aussi étudier un exemple d'algorithme Diviser-pour-Régner dans un contexte où il est difficile de voir où il sort, en OCaml. Dans ce TP, les ★ correspondent à des exercices difficiles / trop longs pour être traités en TP, qui sont là pour vous proposer des pistes pour aller plus loin.

Exercice 1 – Tri rapide par le schéma de partitionnement de Hoare

On va écrire rapidement le tri rapide en C, en utilisant un schéma de partitionnement différent du schéma de Lomuto vu en cours. On note T le tableau où l'on va partitionner la tranche $T[deb : fin]$. Cette fois-ci, on place le pivot au centre de la tranche à partitionner (à l'indice deb). On prend toujours deux indices `i = deb + 1` et `j = fin`, et on va cette fois-ci maintenir l'invariant suivant en effectuant des permutations du tableau :

- la tranche $T[deb + 1 : i]$ contient des éléments inférieurs ou égal au pivot,
- la tranche $T[j : fin]$ contient des éléments strictement supérieurs au pivot,
- la tranche restante $T[i : j]$ n'a pas été touchée.

On s'arrête quand $i = j$ et on finit en mettant le pivot à la bonne place.

▶ **Question 1** Implémenter ce schéma de partitionnement en une fonction `int partitionner_hoare(int* t, int deb, int fin, int pivot)`, renvoyant la position finale du pivot initialement en deb.

▶ **Question 2** En déduire une fonction `void tri_rapide_aux(int* t, int deb, int fin)`, puis une fonction `void tri_rapide(int* t, int taille)` qui implémente le tri rapide.

Pour tester cette fonction et son temps d'exécution, on va écrire une fonction pour générer un tableau contenant des valeurs aléatoires.

Exercice 2 – Fonctions de test et de mesure du temps d'exécution

Pour tester notre fonction de tri rapide, on va utiliser plusieurs fonctions utiles dans des bibliothèques standard. Dans l'ordre :

C
```c
clock_t clock(); // dans time.h
// Renvoie le "nombre de cycles processeurs" écoulés depuis le début de
↪ l'exécution de votre programme. La fréquence du processeur est
↪ accessible avec la macro CLOCKS_PER_SEC.
```

Ainsi, on peut calculer deux valeurs de temps instantanées en évaluant `clock()` avant et après l'exécution d'une fonction, et avec `diff` leur différence on obtient leur temps d'exécution en secondes en calculant `(diff * 1.0 / CLOCKS_PER_SEC)` [a]. Pour générer des nombres aléatoires, on peut utiliser :

```c
void srand( unsigned int seed );
// Initialise le générateur de nombres aléatoires rand. Une façon correcte de l'utiliser est d'exécuter srand(time(NULL)) au début de la fonction main.
int rand();
// Renvoie un entier aléatoire entre 0 et RAND_MAX.
```

▶ **Question 1** Écrire une fonction `int* tableau_aleatoire(int n)` qui renvoie le pointeur vers un tableau de taille `n` contenant des valeurs générées aléatoirement.

▶ **Question 2** Reprendre ou réimplémenter un algorithme de tri naïf (sélection, insertion, bulle...) et l'algorithme de tri fusion.

▶ **Question 3** Écrire une fonction `void test_temps_execution(int* t, int n)` qui effectue les opérations suivantes :

1. copie le tableau `t` dans suffisamment d'autres tableaux de taille `n`,
2. lance chacun des tris sur une copie différente du tableau initial,
3. affiche son temps d'exécution dans un format lisible (par exemple, exprimé en secondes).

▶ **Question 4** En déduire une fonction `void test_temps_execution_aleatoire(int n)` qui teste les temps d'exécution des différents tris sur un tableau de taille `n` généré aléatoirement.

▶ **Question 5** Écrire la même fonction, sauf que le tableau initial est trié dans l'ordre croissant : `void test_temps_execution_croissant(int n)`, puis décroissant : `void test_temps_execution_decroissant(int n)`.

▶ **Question 6** Écrire la fonction `int main(int argc, char* argv[])` ; de telle sorte que l'exécutable compilé prennent un argument entier `n` en ligne de commande et affiche les trois tests de temps d'exécution pour cet entier.

▶ **Question 7** Interpréter le résultat obtenu pour $n = \{10000, 20000\}$.

▶ **Question 8** Implémenter la variante du tri rapide où le pivot est choisi aléatoirement entre `deb` inclus et `fin` exclus.

a. qui va calculer le temps d'exécution en seconde en forçant des opérations flottantes.

Exercice 3 – Comparaison du nombre d'opérations selon le schéma de partitionnement

Les deux opérations les plus importantes dans un algorithme de tri est l'opération de comparaison entre éléments et les accès dans le tableau. Dans cet exercice, on va comparer les deux schémas de partitionnement en fonction du nombre de permutations effectuées.

★ **Question 1** Adapter les codes de `partitionner_hoare`, `tri_rapide_aux` et `tri_rapide` pour que le dernier renvoie le nombre de permutations effectuées pour trier lors de l'exécution du tri rapide.

★ **Question 2** Implémenter le schéma de Lomuto dans une fonction `partitionner_lomuto`, et comparer le nombre de permutations engendrées par ce schéma avec celui du schéma de Hoare sur un tableau généré aléatoirement.

★ **Question 3** Faire de même pour le nombre de comparaisons.

★ **Question 4** Comparer les temps d'exécution des deux schémas de partitionnement sur un tableau généré aléatoirement ou non, et comparer le nombre de comparaisons.

Exercice 4 – Complexité en moyenne du tri rapide

▶ **Question 1** Justifier que deux éléments d'un tableau sont comparés au plus une fois dans le tri rapide (peu importe la façon de choisir les pivots).

▶ **Question 2** En déduire une borne supérieure au nombre de comparaisons effectuées par le tri rapide.

Contrairement à la définition que l'on a donné de la complexité en moyenne dans le Chapitre 7, ici l'aléatoire ne se situe pas dans la distribution de probabilités de l'entrée, mais plutôt dans la distribution de probabilité des choix de pivots. On suppose que notre fonction de pivot est la suivante :

OCaml
```
let pivot t deb fin = deb + Random.int fin
(* renvoie un entier aléatoire entre deb et fin - 1 compris *)
```

Dans la suite, on supposera que les éléments du tableau à trier sont distinctes [a]. On va aussi se restreindre à compter le nombre de comparaisons.
Notons $x_0 < \cdots < x_{n-1}$ les éléments du tableau dans l'ordre croissant et notons $X_{i,j}$ la variable aléatoire égale à 1 si x_i est comparé avec x_j, 0 sinon [b]. On ne va pas s'intéresser au nombre de comparaisons effectué pendant un appel récursif particulier, mais plutôt au nombre total de comparaisons effectuées par l'algorithme.

▶ **Question 3** Exprimer X le nombre de comparaisons totales en fonction des $X_{i,j}$.

Cela nous permet d'exprimer simplement $\mathbb{E}[X]$ en fonction des $\mathbb{P}(\{x_i \text{ est comparé avec } x_j\})$. Manque plus qu'à les évaluer elles : pour cela, encore quelques notations. Pour $i < j$, on note $E_{i,j} = \{x_i, \dots, x_j\}$.

▶ **Question 4** Justifier que x_i et x_j sont comparés au cours de l'exécution si et seulement si le premier pivot choisi dans l'ensemble $E_{i,j}$ est x_i ou x_j.

On peut donc montrer que la probabilité que x_i et x_j soient comparés est $\frac{2}{j-i+1}$.

▶ **Question 5** Conclure.

On pourrait montrer le même résultat avec une autre version du tri rapide randomisé, dans lequel un prétraitement effectue une permutation aléatoire sur le tableau avant d'enchaîner avec le tri rapide déterministe. En pratique, il est plus simple et plus rapide de choisir aléatoirement les pivots.

▶ **Question 6** Implémenter l'algorithme de tri rapide randomisé en adaptant le code précédent en OCaml, et comparer les temps d'exécutions avec l'algorithme de tri rapide déterministe dans deux cas : pour un tableau presque trié / trié, et un tableau généré aléatoirement. Que remarque-t-on ?

a. Pour lever cette restriction, on pourrait simplement trier selon la clé `(elt, indice_initial_elt)` avec l'ordre lexicographique. Cette transformation permet aussi de rendre n'importe quel algorithme de tri stable, c'est-à-dire que deux éléments égaux pour l'ordre de tri gardent leur ordre relatif dans le tableau trié.

b. Vous avez déjà vu les indicatrices ? Si oui, c'est $X_{i,j} = \mathbb{1}_{\{\text{“}x_i \text{ est comparé avec } x_j\text{”}\}}$, une indicatrice sur un évènement.

Exercice 5 – ★ Algorithme de sélection rapide

Il est plus connu sous le nom d'algorithme *Quickselect*, par sa similarité avec le tri rapide. Il est fait pour résoudre le problème de calcul suivant :

Sélection du $k$ième élément le plus petit

Entrée : T un tableau de n éléments, $k \in [\![0, n-1]\!]$
Sortie : le $k$ième élément le plus petit du tableau, noté $\texttt{select}(T, k)$ [a].

a. Comme on est en info, on commence à compter à zéro.

▶ **Question 1** Déterminer $\texttt{select}(T, 0)$ et $\texttt{select}(T, n-1)$.

▶ **Question 2** Exprimer la médiane du tableau T (la médiane de ses éléments) en fonction de valeurs de $\texttt{select}(T, k)$ bien choisies.

▶ **Question 3** Proposer une stratégie trouvant le $k$ième élément le plus petit d'un tableau de taille n en $\mathcal{O}(n \log n)$.

Le principe de l'algorithme de sélection rapide vient du constat suivant : si l'on partitionne une tranche $T[deb : fin]$ dans lequel on cherche le $k$ième élément le plus petit dans cette tranche, et qu'à la fin du partitionnement le pivot se trouve à l'indice piv, on a trois cas :

— Si $deb + k = piv$, on sait que le pivot se trouve exactement à la même position à la fin du partitionnement que si l'on finissait de trier le tableau ensuite : tous les éléments inférieurs ou égaux à $T[piv]$ sont à gauche de piv et tous les éléments strictement supérieurs sont à droite. Ainsi, on peut simplement renvoyer $T[piv]$.
— Si $deb + k < piv$, on cherche le $k$ième élément le plus petit de la tranche $T[deb : fin]$: puisqu'on a $piv - deb > k$ éléments inférieurs ou égaux à $T[piv]$, le $k$ième élément le plus petit de la tranche est exactement le $k$ième élément le plus petit de la tranche $T[deb : piv]$: il suffit alors d'y faire un appel récursif, sans avoir à toucher la seconde tranche.
— De même, si $deb + k > piv$, l'élément recherché est le $(k - (piv - deb + 1))$ième élément le plus petit de la tranche $T[piv + 1 : fin]$, et il suffit d'y faire un appel récursif.

▶ **Question 4** Implémenter cette stratégie en une fonction `int quickselect_aux(int* t, int deb, int fin, int k)` puis une fonction `int quickselect(int* t, int n, int k)`. On autorise le fait que le tableau soit modifié par des permutations comme effet secondaire d'un appel à `quickselect`.

▶ **Question 5** Déterminer la complexité temporelle et spatiale dans le pire et le meilleur cas de l'algorithme `quickselect`. Comparer avec la stratégie précédente. *Comme dans l'exercice de ce matin, choisir le pivot aléatoirement dans la tranche permet d'obtenir une meilleure complexité en moyenne, en l'occurrence en $\mathcal{O}(n)$.*

★ **Question 6** Il est possible d'améliorer la complexité de `quickselect` dans le pire cas, en utilisant la stratégie suivante pour choisir un pivot.

1. On divise le tableau en $\lfloor \frac{n}{5} \rfloor$ tranches de taille 5, laissant éventuellement une dernière tranche de taille ≤ 5 ;
2. on choisit dans chacune de ces $\left\lceil \frac{n}{5} \right\rceil$ la médiane (par exemple, en triant chaque tranche d'abord) ;
3. on choisit comme pivot la médiane de ces $\left\lceil \frac{n}{5} \right\rceil$ éléments, choisis en utilisant...un appel récursif à `quickselect`. [b]

Il y a beaucoup de façons de se tromper en termes de complexité spatiale et temporelle pour appliquer cette stratégie. Une bonne façon de l'implémenter (sans se tromper) est de placer ces $\left\lceil \frac{n}{5} \right\rceil$ médianes dans la tranche $T\left[deb : deb + \left\lceil \frac{n}{5} \right\rceil\right]$ (et en plaçant les valeurs dans cette tranche dans les places laissées). La suite, je vous la laisse : l'écrire, la tester, montrer que cette stratégie de choix de pivot donne une complexité de Quickselect dans le pire cas en $\mathcal{O}(n)$ ... [c]

b. Si on séparait les fonctions de choix de pivot et de sélection en deux fonctions distinctes, cela nous donnerait des fonctions mutuellement récursives ! On va plutôt intégrer le choix du pivot comme un prétraitement de la sélection.

c. Et oui ! un indice : montrer que chacune des partitions créées en utilisant ce pivot contiennent au moins $\frac{3n}{10}$ éléments et au plus $\frac{7n}{10}$. On pourrait aussi montrer qu'en divisant en des sous-tableaux de taille $\neq 5$, tout s'écroule et on retrouve un pire cas en $\mathcal{O}(n^2)$. Bref, Quickselect avec pivot par médiane des médianes est une petite chose fragile. La même stratégie de choix de pivot garantit aussi une complexité dans le pire cas en $\mathcal{O}(n \log n)$ pour le tri rapide : cependant, choisir un pivot aléatoire sera bien plus efficace en moyenne.

Exercice 6 – Paire la plus proche dans un plan

On va se sortir de nos algos de tri, de médiane, de recherche dans les tableaux et étudier l'un des problèmes le plus concret que l'on a vu pour l'instant : on considère une suite de n points du plan $\mathbb{R}^2$ et l'on cherche la distance minimale entre deux de ces points. On utilisera le type suivant pour représenter un élément :

```OCaml
type position = { x : float; y : float }
```

▶ **Question 1** Écrire une fonction `distance : position -> position -> float` renvoyant la distance entre deux points du plan.

▶ **Question 2** Quelle complexité peut-on attendre d'un algorithme naïf pour résoudre ce problème ?

On va essayer d'appliquer le principe Diviser-pour-Régner à ce problème :

- Les cas de base sont quand le nuage de points en contient au plus deux.
- Sinon, on sépare le nuage de points en deux nuages de même taille selon la coordonnée `x` (on met à gauche la moitié contenant les éléments de plus petit `x`, à droite l'autre moitié)
- on résout récursivement le problème sur ces deux sous-nuages,
- pou reconstruire la solution, il faut comparer les deux distances minimales trouvées à gauche et à droite avec la distance minimale entre deux éléments des deux moitiés.

Pour la division, on va simplement trier les points par `x` croissant à l'aide de la fonction `List.sort`.

▶ **Question 3** Regarder la documentation de la fonction `List.sort`[a], et en déduire une fonction `trier_selon_x : position list -> position list`. On admettra que sa complexité est en $\mathcal{O}(n \log n)$ dans le pire cas, avec `n` la longueur de la liste.

▶ **Question 4** Écrire une fonction `dmin_deux_listes : position list -> position list -> float` tel que l'appel `dmin_deux_listes g d` évalue la distance minimale entre un point de `g` et un point de `d`.

▶ **Question 5** Écrire une fonction `dmin_naif_dpr : position list -> float` implémentant la stratégie ci-dessus. *Pour diviser une liste en deux parties de longueur presque égales, on l'a déjà implémentée...*

▶ **Question 6** Évaluer sa complexité.

Pour l'améliorer, il faut réussir à améliorer l'étape de reconstruction qui est trop coûteuse pour l'instant. Pour cela, on va essayer d'éviter de calculer des distances entre éléments des deux nuages si l'on n'en a pas besoin. Supposons que l'on a déjà séparé trié et séparé notre nuage de points en une partie à gauche E_1 et une partie à droite E_2. Soit x_m l'abscisse du point le plus à gauche de E_2, $d_{1/2}$ la plus petite distance entre deux points de $E_{1/2}$. Notons $d = \min(d_1, d_2)$.

▶ **Question 7** Justifier que l'on peut se limiter aux points dont l'abscisse x vérifie $x_m - d \leq x \leq x_m + d$[b].

▶ **Question 8** Dans la bande entre les abscisses $x_m - d_{g,d}$ et $x_m + d_{g,d}$, supposons que l'on a trié les données dans l'ordre croissant des ordonnées. Montrer que l'on peut se limiter à comparer chaque point de cette bande avec les 7 points suivants.

▶ **Question 9** En déduire une fonction `dmin_dpr : position array -> float` calculant la distance minimale entre deux points de l'entrée en utilisant la stratégie ci-dessus pour la reconstruction. Ici, on utilisera plutôt un tableau de positions plutôt qu'une liste.

▶ **Question 10** Montrer qu'elle est de complexité en $\mathcal{O}\big(n(\log n)^2\big)$.

★ **Question 11** Il est en fait possible de faire un algorithme en $\mathcal{O}(n \log n)$ en essayant d'être un peu plus intelligent (i.e. en évitant de trier à chaque appel récursif selon y). Bon courage !

a. par exemple, sur Codium on peut l'écrire dans un fichier `.ml` faire `Ctrl+click` dessus et ensuite cliquer sur le bouton `ml/mli` en haut à droite.

b. On peut commencer par un dessin pour le justifier, mais il faut ensuite énoncer formellement votre justification avec vos connaissances en géométrie dans $\mathbb{R}^2$.