TP 10 bonus : complément sur la complexité en moyenne du tri rapide, et sur une implémentation du tri rapide dans le pire cas en $\mathcal{O}(n \log n)$

Dans tous ces exercices, on ne va traiter que le cas de listes d'éléments deux à deux distincts. Cependant, il est souvent possible d'adapter les algorithmes pour traiter le cas général.

Exercice 1 – Tri rapide en place

On peut montrer que la complexité spatiale de la fonction `tri_rapide` écrite dans le TP 10, c'est-à-dire la quantité de mémoire auxiliaire utilisée pour calculer `tri_rapide(L)`, est en $\mathcal{O}(n)$. Contrairement au tri par partition fusion, il est relativement facile d'implémenter le tri rapide *en place* en implémentant la fonction `partition` de manière plus astucieuse : elle prend en entrée la liste `L` et les indices `deb` et `fin` désignant une tranche de `L` à partitionner, elle partitionne cette tranche en place en fonction du pivot `p = L[fin-1]` et renvoie l'indice `i` du pivot dans la tranche.
La stratégie dite de *Lomuto* pour partitionner une tranche de `L` en place est d'écrire une boucle dont la propriété suivante est un invariant :

- $\forall y \in T[deb : i], y \leq x$
- $\forall y \in T[i : j], y > x$,
- le reste n'a pas encore été touché.

▶ **Question 1** Écrire une fonction `partition(L, deb, fin)` comprenant une boucle qui partitionne la tranche `L[deb:fin]` en place en utilisant cet invariant.

Si l'on trie récursivement les deux tranches obtenues (`L[deb:i]` et `L[i+1:fin]`), la tranche entière est triée : l'étape de reconstruction est donc vide ici.

▶ **Question 2** En déduire une fonction `tri_rapide(L)` triant la liste `L` par tri rapide en place.

▶ **Question 3** En utilisant la fonction `temps_calcul`, comparer les temps de calcul du tri rapide en place, hors place et par partition fusion.

★ **Question 4** Implémenter également le tri rapide en place avec la stratégie de *Hoare*, décrit ici : `https://fr.wikipedia.org/wiki/Tri_rapide#Algorithme_de_partitionnement_alternatif`. Comparer le nombre d'accès à la mémoire effectués par les deux stratégies.

Exercice 2 – Complexité du tri rapide dans le pire et le meilleur cas

▶ **Question 1** Par une méthode similaire à celle adoptée pour le tri fusion dans le TP 10, montrer que la complexité de `tri_rapide` est en $\mathcal{O}(n \log n)$ dans le meilleur cas (où le pivot est toujours médian dans la tranche à trier) et en $\mathcal{O}(n^2)$ dans le pire cas (où le pivot est toujours l'élément le plus petit ou le plus grand dans la tranche à trier).

Exercice 3 – 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.

Supposons qu'avant de partitionner le tableau, on choisisse un pivot aléatoirement parmi les éléments du tableau. On peut alors montrer que le nombre de comparaisons effectuées par le tri rapide est en moyenne en $\mathcal{O}(n \log n)$.

Python
```python
def partition_randomisee(L, deb, fin):
    pos = random.choice(range(deb, fin))
    L[pos], L[fin-1] = L[fin-1], L[pos]
    // Après avoir choisi un pivot aléatoirement, on
     ↪ partitionne comme avant
    partition(L, deb, fin)
```

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 4 – Algorithme de sélection rapide

L'algorithme du tri rapide peut être adapté en un *algorithme de sélection rapide* qui, étant donné un tableau `L` et un entier k, renvoie le k-ème plus petit élément de `L`.

▶ **Question 1** En utilisant `tri_rapide` ou la fonction `sorted`, implémenter une fonction `selection(L, k)` qui renvoie le k-ème plus petit élément de `L`. *Sa complexité sera en $\mathcal{O}(n \log n)$.*

On peut améliorer cette complexité en utilisant une version modifiée du tri rapide, dans laquelle on ne trie pas les deux sous-tableaux mais seulement celui qui contient l'élément recherché. Par exemple, si l'on partitionne la liste `L` en `L[:i]` et `L[i+1:]`, alors :

— si $i = k$, alors `L[i]` est le k-ème plus petit élément de `L` ;
— si $i < k$, alors le k-ème plus petit élément de `L` est le $k-i$-ème plus petit élément de `L[i+1:]` ;
— si $i > k$, alors le k-ème plus petit élément de `L` est le k-ème plus petit élément de `L[:i]`.

▶ **Question 2** Implémenter une fonction `selection_rapide(L, k)` qui renvoie le k-ème plus petit élément de `L` en utilisant cette idée.

▶ **Question 3** Par une méthode similaire à celle adoptée pour le tri fusion dans le TP 10, montrer que la complexité de `selection_rapide` est en $\mathcal{O}(n)$ dans le meilleur cas (où le pivot est toujours médian) et en $\mathcal{O}(n^2)$ dans le pire cas (où le pivot est toujours l'élément le plus petit ou le plus grand).

En utilisant cette fonction de sélection rapide, on peut