Feuille de TD n°2

Exercice 1. Le but de cet exercice est de majorer la complexité en moyenne du tri rapide. Pour cela, on suppose que la tableau de données à trier est une permutation des n entiers de 0 à $n-1$, choisie équiprobablement parmis les $n!$ permutation possibles. Les pivot choisi est l'élément de plus petit indice dans le sous-tableau considéré, et l'algorithme de partition utilisé est celui donné en cours, que l'on rappelle ici :

```
PARTITION:  T = tableau indicé de a à b-1, p = indice du pivot choisi

a) Échanger T[a] et T[p].
b) Faire i <- a+1, j <- b
c) Tant que i < j et T[i] <= T[a]
     faire i <- i+1.
d) Tant que i < j et T[j-1] >= T[a]
     faire j <- j-1.
e) Si i < j
     échanger T[i] et T[j-1]
     faire i <- i+1, j <- j-1
     aller en c).
f) Échanger T[a] et T[i-1] et terminer.
   (Le pivot est en m=i-1).
```

1. Montrer qu'initialement, les n valeurs possibles pour le pivot $T[0]$ sont équiprobables.
2. Montrer que l'algorithme de partition ne change pas les cases de $T[a]$, ..., $T[m-1]$ dont le contenu est inférieur au pivot, et qu'il ne change pas les cases de $T[m+1]$, ..., $T[b-1]$ dont le contenu est supérieur au pivot.
3. Montrer que les positions respectives des entiers de $T[m+1]$, ..., $T[b-1]$ qui sont inférieurs au pivot sont renversées par l'algorithme de partition.
4. On pose $u = m-a$ le nombre de cases de contenu inférieur au pivot, et $v = b-m-1$ le nombre de cases de contenu supérieur au pivot, de sorte qu'il y a $u!v!$ ordres possibles après partition pour un pivot donné. Montrer que le nombre de permutations avant partition menant à l'un de ces ordres est
$$\sum_{k=0}^{\min(u,v)} \binom{u}{k}\binom{v}{k}.$$
5. En considérant le cœfficient de X^u dans $(1+X)^u(1+X)^v = (1+X)^{u+v}$, montrer que
$$\sum_{k=0}^{\min(u,v)} \binom{u}{k}\binom{v}{k} = \binom{u+v}{u} = \frac{(b-a-1)!}{u!v!}.$$
6. En déduire que les $u!$ (respectivement $v!$) permutations possibles du sous-tableau inférieur (respectivement supérieur) après partition sont équiprobables.
7. On note E_n l'espérance du nombre d'opérations nécessaires pour trier un tableau de taille n, sous les hypothèses précédentes. Montrer que
$$E_n \leqslant \frac{2}{n}\sum_{i=0}^{n-1} E_i + c_0 n + c_1 \qquad (\forall n > 1),$$
pour des constantes positives c_0 et c_1.

8. Soit $I : \mathbb{R}_+ \to \mathbb{R}_+$ une fonction continue croissante, C^1 sur $]1, +\infty[$ et solution sur $]1, +\infty[$ de l'équation

$$I(x) = \frac{2}{x} \int_0^x I(t)\, dt + c_0 x + c_1, \tag{1}$$

dont on suppose qu'elle vérifie $I(0) \geqslant E_0$ et $I(1) \geqslant E_1$. Montrer que $E_n \leqslant I(n)$ pour tout $n \geqslant 0$.

9. En dérivant l'équation (1) et en résolvant l'équation différentielle ainsi obtenue, montrer que les solutions de (1) sont de la forme

$$I(x) = 2c_0 x \log x - c_1 + c_2 x \qquad (\forall x > 1),$$

pour une certaine constante c_2.

10. En déduire que $E_n \underset{n\to\infty}{=} O(n \log n)$.

Exercice 2. Comparer la complexité des trois algorithmes suivants de sélection des m plus grands éléments parmi n :

- le tri par sélection partiel, où l'on arrête l'algorithme après avoir trouvé les m plus grands élements,
- le tri par tas partiel,
- l'algorithme de sélection rapide.

Exercice 3. Dans cet exercice, on écrira des algorithmes en pseudo-code en supposant que l'on dispose d'une structure `arbre` pour décrire un *arbre binaire de recherche*, et d'un type `noeud`, un arbre ayant tous ses nœuds étiquetés par des entiers. On suppose également que l'on dispose des fonctions suivantes :

- `valeur` qui, étant donné un `noeud`, nous renvoie l'entier qui l'étiquette,
- `fils_gauche` qui, étant donné un `noeud`, nous renvoie le `noeud` qui est son fils gauche,
- `fils_droit`, défini de manière similaire,
- `racine` qui, étant donné un `arbre`, nous renvoie sa racine (de type `noeud`),
- `sous-arbre_gauche` et `sous-arbre_droit` qui, étant donné un `arbre`, nous renvoient respectivement son sous-arbre gauche ou son sous-arbre droit (de type `arbre`).

Il existe un arbre vide, qui ne contient aucun nœud. En outre, par convention, `fils_gauche` et `fils_droit` renvoient `NULL` si le nœud donné en argument n'a pas de fils gauche, resp. droit.

1. Écrire en pseudo-code une fonction `recherche`, qui prend en argument un `arbre` et un `entier`, et renvoie le `noeud` étiqueté par cet entier, ou `NULL` si l'entier n'est pas dans l'arbre.

2. Écrire une fonction `imm_inf` qui, étant donné un `noeud` n_1, renvoie le `noeud` n_2 dont la valeur est immédiatement inférieure à celle de n_1.

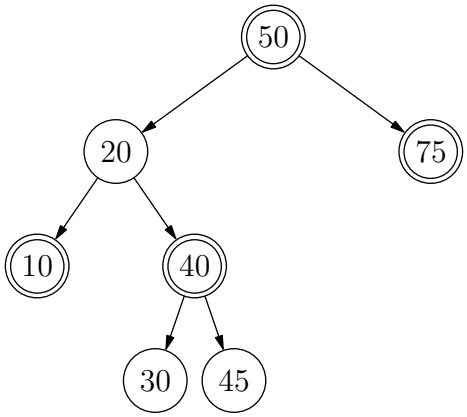
3. Montrer que n_2 ne peut pas avoir de fils droit.

4. Écrire une fonction `supprimer` qui prend en argument un arbre et un `entier`, et qui, si l'entier se trouve dans l'arbre, supprime le `noeud` sur lequel il se trouve selon les règles suivantes :

- si le nœud n'a pas de fils, la suppression est immédiate,
- si le nœud n'a qu'un seul fils, on le remplace par celui-ci,
- si le nœud a deux fils, on le remplace par le nœud dont la valeur est immédiatement inférieure.

5. Montrer qu'après suppression d'un élément selon les règles ci-dessus, l'arbre obtenu est toujours un arbre binaire de recherche.

Exercice 4. On considère l'arbre rouge-noir suivant. (Les nœuds noirs sont représentés avec un trait double, les nœuds rouges avec un trait simple).



1. Vérifier que cet arbre respecte bien les propriétés définissant les arbres rouge-noir, et que c'est bien un arbre binaire de recherche.
2. Insérer 35 dans l'arbre ci-dessus. On détaillera les opérations effectuées pour équilibrer l'arbre.