TD 2

Exercice 1. *Implantation en place de* QUICKSORT

On propose les implantations suivantes en C de l'algorithme de PARTITION commun à QUICKSORT et QUICKSELECT. On suppose que `swap(T, i, j)` échange les éléments d'indices `i` et `j` du tableau `T`. L'objectif des deux algorithmes est le même : étant donné un tableau T et des indices b, h et p vérifiant $0 \leq b \leq p \leq h < |T|$, le sous-tableau $T[b:h]$ (éléments d'indices b à h, inclus) est partitionné en utilisant l'élément $T[p]$ comme pivot. La partition est effectuée *en place*, c'est-à-dire que le sous-tableau $T[b:h]$ contient après exécution les mêmes éléments qu'avant, mais les éléments inférieurs au pivot sont *au début* du sous-tableau, et les autres à la fin. L'entier renvoyé permet de connaître la limite entre les deux sous-parties de $T[b:h]$. *On suppose dans l'exercice que T ne contient que des éléments distincts deux à deux.*

```
int partition_lomuto(int* T, int b, int h, int p) {
    int pivot = T[p];
    swap(T, h, p);
    int i = b-1;
    for(int j = b; j < h; j++)
        if (T[j] <= pivot) swap(T, ++i, j);
    swap(T, ++i, h);
    return i;
}

int partition_hoare(int* T, int b, int h, int p) {
    int pivot = T[p];
    swap(T, b, p);
    int i = b - 1, j = h + 1;
    while(1) {
        do i++; while(T[i] < pivot);
        do j--; while(T[j] > pivot);
        if (i >= j) return j;
        swap(T, i, j);
    }
}
```

1. Montrer la correction de `partition_lomuto` et `partition_hoare`.
2. Analyser la complexité dans le pire cas des algorithmes, en nombre d'échanges (appels à `swap`).
3. On suppose qu'on exécute `partition_lomuto` sur un tableau T, avec $b = 0$, $h = n-1$ (où $n = |T|$) et un indice de pivot p choisi de manière aléatoire uniforme entre 0 et $n-1$. Calculer l'espérance du nombre d'échanges effectués.
4. *(Bonus, à la maison !)* Donner des implantations complètes en place de QUICKSORT utilisant chacun des deux algorithmes et comparer leurs performances en pratique.

Exercice 2. RANDMINCUT *amélioré*

Le but de cet exercice est d'étudier une variante de l'algorithme RANDMINCUT qui a une bien meilleure complexité. L'idée est que les problèmes arrivent plutôt quand il reste peu de sommets : le risque de contracter deux sommets *par erreur* devient plus grand.

1. On fixe une coupe minimum d'un graphe G à n sommets, et on contracte aléatoirement des arêtes de G jusqu'à ce que G ait n_0 sommets. Quelle est la probabilité qu'une arête de la coupe minimale ait été contractée ?
2. Montrer que si $n_0 \geq 1 + n/\sqrt{2}$, avec probabilité au moins $1/2$, aucune arête de la coupe minimum n'a été contractée.

On note CONTRACTION(G) l'algorithme qui contracte aléatoirement des arêtes de G tant que G a plus de $(1 + n/\sqrt{2})$ sommets. L'algorithme BETTERMINCUT est le suivant.

BETTERMINCUT(G) :

1. Si G a moins de 8 sommets, renvoyer une coupe minimale ;
2. $C_1 \leftarrow$ BETTERMINCUT(CONTRACTION(G)) ;
3. $C_2 \leftarrow$ BETTERMINCUT(CONTRACTION(G)) ;
4. Renvoyer $\min(C_1, C_2)$.

On suppose qu'on a un algorithme calculant une coupe minimale exacte pour un graphe de moins de 8 sommets (par exemple par recherche exhaustive).

3. Montrer que la complexité de BETTERMINCUT est $O(n^2 \log n)$.

4. En notant p_n la probabilité que BETTERMINCUT renvoie une coupe minimale quand le graphe en entrée a n sommets, borner p_n en fonction de $p_{n/\sqrt{2}+1}$.

5. *On admet que* $p_n = \Omega(1/\log n)$. Montrer qu'avec $O(\log^2 n)$ répétitions de l'algorithme, sa probabilité de succès est au moins $1 - 1/n^c$ pour une certaine constante c. Quelle est la complexité globale obtenue ?