TP : recherche de la médiane et algorithme de sélection

Le but de ce TP est de déterminer de façon efficace le k-ème plus grand élément d'une liste. Les algorithmes résolvant ce problème sont appelés « algorithmes de sélection ». On s'intéressera en particulier à la recherche de la médiane d'une liste de nombres, c'est-à-dire la valeur m de la liste pour laquelle il y a autant d'éléments de la liste plus petits que m que d'éléments plus grands que m (précisions plus bas).

On considère une liste `tab` contenant n nombres réels. Les indices vont ainsi de 0 à $n-1$.
On rappelle que si $0 \leqslant a \leqslant b \leqslant n$, `tab[a:b]` est la liste des éléments d'indice compris entre a et $b-1$, *i.e.* les éléments `tab[a]`, `tab[a+1]`, ..., `tab[b-1]`.
Dans la suite, les exemples seront explicités avec la liste de taille 11 suivante :

```
L = [3, 2, 5, 8, 1, 34, 21, 6, 9, 14, 8].
```

1 Plus petit et plus grand élément

Il est facile de déterminer le plus petit (ou le plus grand) élément d'une liste en la parcourant une seule fois.

1. Écrire une fonction d'en-tête

   ```
   indices_min_max(tab:list, a:int, b:int) -> (int, int):
   ```

   qui renvoie le couple formé des indices du plus petit et du plus grand élément parmi ceux de la liste `tab` d'indices compris entre `a` et `b`.
   Par exemple, `indices_min_max(L, 5, 10)` renverra `(7, 5)` car `L[7] = 6` et `L[5] = 34` sont respectivement le plus petit et le plus grand élément de `L[5:10] = [34, 21, 6, 9, 14]`.

2 Méthode naïve de recherche de la médiane

Dans cette partie, nous allons chercher à déterminer la médiane d'une liste en utilisant juste la définition de celle-ci. Commençons donc par rappeler la définition de la médiane.
Soient E un ensemble de nombres et t un réel. On note $E_{\leqslant t}$ (respectivement $E_{\geqslant t}$) l'ensemble des éléments de E inférieurs ou égaux (respectivement supérieurs ou égaux) à t. Par exemple, avec la liste `L` précédente, on a $\texttt{L}_{\leqslant 5} = \{1, 2, 3, 5\}$ et $\texttt{L}_{\geqslant 5} = \{5, 6, 8, 8, 9, 14, 21, 34\}$.
La valeur médiane m de l'ensemble E est un élément de E tel que les deux ensembles $E_{\leqslant m}$ et $E_{\geqslant m}$ vérifient

$$\text{Card}(E_{\leqslant m}) \geqslant \lfloor \frac{n+1}{2} \rfloor \quad \text{et} \quad \text{Card}(E_{\geqslant m}) \geqslant \lfloor \frac{n+1}{2} \rfloor$$

où $\lfloor \cdot \rfloor$ désigne la partie entière. Par exemple, la médiane de la liste `L` est 8 car $\text{Card}(\texttt{L}_{\leqslant 8}) = 7 \geqslant \lfloor \frac{11+1}{2} \rfloor$ et $\text{Card}(\texttt{L}_{\geqslant 8}) = 6 \geqslant \lfloor \frac{11+1}{2} \rfloor$.
Remarque : la partie entière se calcule via la fonction `floor` *du module* `math`.

2. Écrire une fonction d'en-tête

   ```
   nb_petits_grands(tab:list, a:int, b:int, val:float) -> (int, int):
   ```

   qui renvoie le couple ($\text{Card}(\texttt{tab[a:b]}_{\leqslant \texttt{val}})$, $\text{Card}(\texttt{tab[a:b]}_{\geqslant \texttt{val}})$).
   Par exemple `nb_petits_grands(L, 2, 10, 5)` renverra `(2,7)` et `nb_petits_grands(L, 2, 10, 8)` renverra `(4,5)`.
3. Écrire une fonction d'en-tête

   ```
   mediane_naive(tab:list, a:int, b:int) -> float:
   ```

qui renvoie la médiane de la liste `tab[a:b]` en utilisant la définition de la médiane et la fonction précédente.

4. Quelle est la complexité dans le pire cas de cet algorithme en fonction de la longueur n de la liste `tab` ?

Une méthode plus efficace consisterait à trier la liste par ordre croissant et à renvoyer la valeur du milieu, *i.e.* l'élément d'indice $\lfloor \frac{n+1}{2} \rfloor$ si la liste est de taille n. On sait que cette méthode requiert $O(n \ln n)$ opérations. La suite du sujet a pour but d'implémenter une méthode en temps linéaire $O(n)$ pour déterminer la k-ème plus grande valeur d'une liste de taille n.

3 Diviser pour régner

Pour accélérer la recherche, on va utiliser le paradigme « diviser pour régner » qui consiste à décomposer le problème en sous-problèmes plus petits et à regrouper ensuite les résultats.
À l'instar du tri rapide, le point clé est l'utilisation d'une fonction `partition(tab, a, b, indice_pivot)` qui prend en paramètre une liste de nombres `tab[a:b]` ainsi qu'un élément de la liste `tab[a:b]` appelé *pivot* repéré par son indice `indice_pivot`.
La fonction devra réordonner les éléments de `tab[a:b]` en trois groupes : elle positionnera les éléments strictement plus petits que la valeur *pivot* (c'est-à-dire ceux de $\texttt{tab}_{<pivot}$) au début, puis les éléments égaux à *pivot* et enfin ceux strictement plus grands que *pivot* (ceux de $\texttt{tab}_{>pivot}$). Par exemple, `partition(L, 0, 11, 3)` pourra donner `[3, 2, 4, 1, 6, 8, 8, 21, 34, 9, 14]` car le pivot `L[3]` valait 8. Notons que dans le résultat, les nombres strictement plus petits que le pivot peuvent se trouver dans n'importe quel ordre les uns par rapport aux autres, de même pour ceux strictement plus grands.

5. En s'inspirant du tri rapide, écrire la fonction d'en-tête

   ```
   partition(tab:list, a:int, b:int, indice_pivot:int) -> (int, int):
   ```

   Elle **modifiera** la liste `tab` et **renverra** le couple constitué par les indices respectifs du premier et du dernier élément égaux au *pivot*.

   Par exemple `partition(L, 0, len(L), 3)` renverra `(5,6)` car après l'exécution, la liste `L` aura été modifiée en `L = [3, 2, 5, 1, 6, 8, 8, 21, 34, 9, 14]` et le premier élément égal au pivot est celui d'indice 5 et le dernier celui d'indice 6.

Nous allons maintenant utiliser l'algorithme récursif suivant pour déterminer le k-ème plus grand élément d'une liste `tab[a:b]` :

- Si $k = 1$ et $a = b$ alors renvoyer `tab[a]`.
- Sinon, partitionner la liste `tab[a:b]` comme précédemment en utilisant comme pivot le premier élément de la liste, *i.e.* `tab[a]`. Notons (i, j) le couple renvoyé par la fonction `partition`.
  - Si $k - 1 < i - a$ (*i.e.* l'élément recherché se trouve dans le premier groupe), chercher le k-ème plus grand élément de `tab[a:i]` et renvoyer cet élément ;
  - Si $i - a \leqslant k - 1 \leqslant j - a$ (*i.e.* l'élément recherché se trouve dans le groupe du milieu), renvoyer le pivot ;
  - Si $k-1 > j-a$ (*i.e.* l'élément recherché se trouve dans le dernier groupe), chercher le $k-j+a-1$-ème plus grand élément de `tab[j+1:b]` et renvoyer cet élément.

6. Écrire une fonction d'en-tête

   ```
   element_k(tab:list, a:int, b:int, k:int) -> float:
   ```

   qui réalise l'algorithme de sélection du k-ème plus grand élément dans la liste `tab[a:b]` décrit ci-dessus et qui renvoie cet élément.

Malheureusement, si le choix du pivot est mauvais (par exemple le plus grand élément de la liste), à chaque étape, la recherche se restreint à une sous-liste qui est de longueur trop proche de celle à l'étape précédente. Ainsi, on peut montrer que la complexité dans le pire cas reste en $O(n^2)$, on ne semble donc pas avoir améliorer le calcul de la médiane.

Pour remédier à cela, on va améliorer le choix du pivot en le prenant égal à une valeur appelée *pseudo-médiane*. Pour calculer la pseudo-médiane d'une liste, on la divise en groupes de cinq éléments (plus éventuellement un groupe plus petit) et on calcule la médiane de chaque paquet de 5 (et de l'éventuel groupe plus petit) avec la fonction `element_k` précédente. On applique alors récursivement ce calcul sur la sous-liste des $n/5$ médianes jusqu'à obtenir une valeur qui est la *pseudo-médiane*.
Par exemple, on a

$$L = [\underbrace{3,2,5,8,1}_{3},\underbrace{34,21,6,9,14}_{14},\underbrace{8}_{8}]$$

donc la pseudo-médiane de `L` est la médiane de $[3, 14, 8]$, c'est-à-dire 8.

7. Écrire une fonction d'en-tête

   ```
   mediane_des_medianes(tab:list) -> float:
   ```

   qui renvoie la valeur de la pseudo-médiane de la liste `tab` comme décrit ci-dessus.

8. Écrire une fonction d'en-tête

   ```
   recherche(tab:list, val:float) -> int:
   ```

   qui renvoie le plus petit indice `i` tel que `tab[i]` soit égal à `val`. On pourra lui faire renvoyer -1 si aucun élément de `tab` ne vaut `val`.

9. Écrire une fonction d'en-tête

   ```
   selection(tab:list, a:int, b:int, k:int) -> float:
   ```

   qui renvoie le k-ème plus grand élément de la liste `tab[a:b]` en procédant de manière analogue à `element_k` sauf que le pivot choisi est la pseudo-médiane.

On peut montrer que cet algorithme permet de trouver le k-ème élément d'une liste en $O(n)$ opérations où n est la longueur de la liste. Cette efficacité repose sur le fait qu'à chaque étape au moins 30 % des éléments sont éliminés car le pivot est la pseudo-médiane calculée sur des groupes de cinq éléments.

4 Comparaison des méthodes

Le but de cette dernière partie est de comparer le temps d'exécution des deux fonctions `mediane_naive` et `selection` pour le calcul de la médiane d'une liste contenant un grand nombre d'éléments.
Pour calculer le temps d'exécution d'une fonction, on va utiliser la fonction `timeit` du module `timeit`. Celle-ci doit prendre en argument une fonction sans argument, on va donc inclure l'exécution des fonctions de calcul de médianes dans une fonction sans argument.

10. Pour que les deux fonctions permettant de calculer la médiane d'une liste prennent la même entrée, écrire deux fonctions d'en-têtes

    ```
    med_naive(tab:list) -> float:
    ```

    et

    ```
    med_des_med(tab:list) -> float:
    ```

    qui renvoient la médiane de la liste `tab` en utilisant respectivement les fonctions `mediane_naive` et `selection`.

11. Recopier la fonction suivante :

    ```
    def fct_sans_arg(f:function, tab:list) -> function:
        def fonc():
            return f(tab)
        return fonc
    ```

    La sortie étant une fonction sans argument, on pourra calculer son temps d'exécution comme décrit précédemment. L'appel se fera sous la forme

    ```
    from timeit import timeit
    timeit(fct_sans_arg(med_naive, [1,2,3])
    ```

Télécharger le fichier `test20000.txt` depuis `http://demeslay.maths.free.fr/#Info` et le placer dans le même dossier que le fichier `.py` de votre TP. Il contient vingt mille lignes, chacune contenant un nombre entre 0 et 1000.

12. Écrire une fonction d'en-tête

    ```
    recup_liste() -> list:
    ```

    qui renvoie la liste des nombres contenus dans le fichier précédemment téléchargé.

13. Écrire une fonction d'en-tête

    ```
    duree(f:function, fichier:str) -> float:
    ```

    qui renvoie le temps d'exécution de la fonction `f` sur la liste des nombres contenus dans le `fichier`.

14. Comparer les temps d'exécution de `med_des_med` et `med_naive` sur la liste des nombres du fichier téléchargé précédemment.