Algorithmes de tris

On l'a vu dans les séances précédentes, avoir des données triées peut être très intéressant algorithmiquement : la recherche dichotomique dans un tableau trié est de complexité logarithmique (comparée à linéaire dans le cas général), et certains algorithmes gloutons reposent sur la notion d'ordre.

Nous allons explorer dans cet séance quelques algorithmes de tri.

I. Tris quadratiques

1. Tri par sélection

Supposons que l'on a le tableau suivant :

```
t = [4, 7, 9, 6, 5, 2, 1, 8, 0, 3]
```

Question 1 Quelle est la valeur que l'on doit mettre dans la dernière case ?

Question 2 Écrire une fonction `position_plus_grand(t)` qui renvoie la position du plus grand élément de `t`.

Question 3 Modifier la fonction précédente en une fonction

```
position_plus_grand(t, n)
```

qui se limite aux `n` premières positions de `t`.

On peut alors programmer l'algorithme de *tri par sélection* qui procède ainsi :

1. On sélectionne le plus grand élément du tableau et on l'échange avec l'élément en dernière position. Il reste à classer le tableau privé du dernier élément.

2. On sélectionne le plus grand élément des valeurs qui restent à classer, on l'échange avec le dernier élément de ce qui reste à classer, et on continue...

Voici les premières étapes du tri par sélection du tableau précédent, où la barre verticale indique la position à partir de laquelle les plus grandes valeurs sont

triées et à leur place :

```
[4, 7, 9, 6, 5, 2, 1, 8, 0, 3]
[4, 7, 3, 6, 5, 2, 1, 8, 0,|9]
[4, 7, 3, 6, 5, 2, 1, 0,|8, 9]
[4, 0, 3, 6, 5, 2, 1,|7, 8, 9]
[4, 0, 3, 1, 5, 2,|6, 7, 8, 9]
                ...
```

Question 4 Écrire une fonction `tri_selection(t)` qui trie le tableau `t` passé en argument en appliquant l'algorithme précédent. Cette fonction ne renvoie rien (pas de `return`) mais modifie le contenu du tableau `t` qui, à la fin, doit être trié.

Question 5 Justifier que pour trier un tableau de taille n, l'algorithme effectue exactement $\frac{n(n-1)}{2}$ comparaisons[1] et en déduire que l'algorithme de tri par sélection est quadratique dans le pire comme dans le meilleur des cas.

2. Tri par insertion

On peut procéder autrement pour trier un tableau. Supposons que les k premières valeurs du tableau sont triées. Par exemple, pour $k = 4$:

```
t = [4, 6, 7, 9, 5, 2, 1, 8, 0, 3]
```

On doit maintenant mettre la valeur v en position k (ici, $v =$ `t[4]` $= 5$) au bon endroit pour que les $k+1 = 5$ premières valeurs soient triées. Pour cela, on regarde la valeur en position $k-1$ (donc la position juste à gauche de la valeur à classer, qui correspond à la plus grande des valeurs déjà classées) et si celle-ci est plus grand que v, on la décale d'une place vers la droite et on se déplace vers la gauche jusqu'à trouver une valeur strictement plus petite que v (ou arriver au début du tableau). La case juste à droite est alors la place où mettre v.

Voici l'insertion de $v = 5$ (initialement en position 4, où figure maintenant un symbole ✠ qui va se déplacer vers la gauche jusqu'à trouver la place où mettre v. Tant que la valeur soulignée est plus grande que 5, on l'échange avec ✠ :

```
[4, 6, 7, 9̲, ✠, 2, 1, 8, 0, 3]
[4, 6, 7̲, ✠, 9, 2, 1, 8, 0, 3]
[4, 6̲, ✠, 7, 9, 2, 1, 8, 0, 3]
[4̲, ✠, 6, 7, 9, 2, 1, 8, 0, 3]
```

[1] Pour les algorithmes de tri, le nombre de comparaisons effectuées est une bonne mesure de complexité.

À la fin, ✠ indique où insérer v :

```
[4, 5, 6, 7, 9, 2, 1, 8, 0, 3]
```

Bien sûr, le symbole ✠ sert uniquement à indiquer une position. En particulier, on ne se soucie pas de la valeur qui y figure.

Question 6 Écrire une fonction `decalage(t, v, p)` qui effectue la phase d'insertion. Elle prends pour arguments le tableau `t`, la valeur à insérer `v` ainsi que la position initiale `p` correspondant à ✠.

Question 7 Écrire la fonction `tri_par_insertion(t)` qui va trier le tableau `t` en le triant de gauche à droite via l'insertion successive des différentes valeurs. De même, cette fonction ne renvoie rien mais le tableau passé en argument doit être trié à la fin.

Question 8 Quel est, dans le pire des cas, le nombre de comparaisons nécessaires pour trier un tableau de taille n ? Donner un exemple de tableau pour lequel ce maximum est atteint.

Question 9 Quel est, dans le meilleur des cas, le nombre de comparaisons nécessaires pour trier un tableau de taille n ? Donner un exemple de tableau pour lequel ce minimum est atteint.

II. Tris efficaces

Les deux algorithmes que l'on vient de présenter ont une complexité quadratique. Nous allons maintenant présenter deux algorithmes plus efficaces.

1. Tri fusion

Le premier algorithme repose sur l'idée qu'il est facile de *fusionner* deux listes croissantes en une même liste croissante. En effet, on effectue pour cela un parcours simultané des deux liste de la gauche vers la droite.

Dans l'exemple suivant, on fusionne les deux premières listes, la troisième liste

étant le résultat en train d'être construit.

```
[1, 3, 4, 6, 7]      [0, 2, 5]      []
[1, 3, 4, 6, 7]      [0, 2, 5]      [0]
[1, 3, 4, 6, 7]      [0, 2, 5]      [0, 1]
[1, 3, 4, 6, 7]      [0, 2, 5]      [0, 1, 2]
[1, 3, 4, 6, 7]      [0, 2, 5]      [0, 1, 2, 3]
[1, 3, 4, 6, 7]      [0, 2, 5]      [0, 1, 2, 3, 4]
[1, 3, 4, 6, 7]      [0, 2, 5]_     [0, 1, 2, 3, 4, 5]
[1, 3, 4, 6, 7]      [0, 2, 5]_     [0, 1, 2, 3, 4, 5, 6]
[1, 3, 4, 6, 7]_     [0, 2, 5]_     [0, 1, 2, 3, 4, 5, 6, 7]
```

Concrètement, on utilise pour cela deux indices initialisés à 0, un par tableau, et tant que aucune indice n'est sorti du tableau correspondant, on compare les deux éléments pointés par les indices, on ajoute le plus petit au résultat en construction, et on décale l'indice correspondant vers la droite.

Question 10 Que faire lorsque l'un des indices « sort » de son tableau ?

Question 11 Écrire une fonction `fusion(t1, t2)` qui construit et renvoie le tableau obtenu en fusionnant `t1` et `t2`.

On peut maintenant définir le tri fusion : pour trier un tableau `t`],

1. on divise le tableau en deux sous-tableaux (concrètement, à partir de `t`, on considère `t[:len(t) // 2]` et `t[len(t) // 2:]`, essayez sur des exemples pour voir ce que cela fait) ;
2. on trie les deux sous-tableaux (comment ? grâce au génie de la récursion) ;
3. on fusionne les deux sous-tableaux triés.

Question 12 Il s'agit clairement d'un algorithme récursif. Quel est le cas de base ?

Question 13 Écrire une fonction `tri_fusion(t)` qui implémente cet algorithme. Contrairement aux algorithmes précédents, le tableau passé en argument n'est pas modifié et on renvoie un tableau correspondant à la version triée de `t` (il y a donc un **`return`**).

On peut montrer[2] que l'algorithme de tri fusion à une complexité linéarithmique dans le pire des cas, c'est-à-dire en $O(n \ln n)$ pour un tableau de taille n. On

[2] Et les élèves suivant l'option informatique le feront.

peut aussi montrer que cette complexité est optimale pour un algorithme de tri reposant sur la comparaison d'éléments.

2. Tri rapide

Nous allons pour finir étudier un dernier algorithme de tri basé sur la comparaison d'éléments deux à deux.

L'idée ici est d'utiliser une valeur *pivot* (qui sera, dans notre cas, la première valeur du tableau) et modifier le tableau de telle sorte qu'à la fin, les valeurs avant le pivot soient plus petites que celui-ci, et les valeurs après le pivot soient plus grandes, ce que l'on nomme **partitionner** le tableau. À l'issue de cette partition, le pivot est à la bonne place et il reste à trier les valeurs avant, et les valeurs après.

Pour illustrer cela, supposons que l'on veut trier le tableau :

```
[5, 6, 7, 9, 4, 2, 1, 8, 0, 3]
```

La valeur pivot est `5`, et comme pour le tri fusion, nous allons marquer son emplacement du signe ✠, indiquant une case du tableau dont on ne se soucie pas de la valeur. Nous allons ensuite parcourir le reste du tableau (c'est la case soulignée dans ce qui suit) en ayant les invariants suivants :

- ⋆ les valeurs situées avant ✠ sont strictement inférieures à la valeur pivot,
- ⋆ les valeurs situées entre ✠ et la case courante (celle qui est soulignée) sont supérieures ou égales au pivot.

Pour chaque valeur rencontrée lors du parcours, il faut donc la comparer à la valeur pivot, et modifier le tableau au besoin.

Les étapes du parcours du tableau précédent sont représentés dans la figure 1.

On remarque que si la valeur actuellement visitée est supérieure ou égale au pivot, on ne fait rien sinon passer à la valeur suivante, puisqu'elle se trouve à droite de ✠.

Par contre, si la valeur actuelle est plus petite que le pivot, c'est un peu plus compliqué puisqu'il faut décaler ✠ d'un cran vers la droite pour insérer la nouvelle valeur plus petite. On procède à un échange de la façon suivante :

1. On commence par copier la valeur actuelle à l'emplacement de ✠. La valeur de la case soulignée n'est plus importante (on vient de la copier ailleurs), on la y fait figurer `*` ;
2. On recopie la valeur à droite de ✠ à la case soulignée (on peut, il y a `*`) (elle reste à droite de ✠). On peut maintenant faire figurer `*` dans la case à droite de ✠.

```
[✠, 6, 7, 9, 4, 2, 1, 8, 0, 3]
[✠, 6, 7, 9, 4, 2, 1, 8, 0, 3]
[✠, 6, 7, 9, 4, 2, 1, 8, 0, 3]
[✠, 6, 7, 9, 4, 2, 1, 8, 0, 3]
[4, ✠, 7, 9, 6, 2, 1, 8, 0, 3]
[4, 2, ✠, 9, 6, 7, 1, 8, 0, 3]
[4, 2, 1, ✠, 6, 7, 9, 8, 0, 3]
[4, 2, 1, ✠, 6, 7, 9, 8, 0, 3]
[4, 2, 1, 0, ✠, 7, 9, 8, 6, 3]
[4, 2, 1, 0, 3, ✠, 9, 8, 6, 7]_
```

Figure 1: Exemple de partition.

3. On peut maintenant décaler ✠ vers la case où figure `*`, et passer à la case suivante dans le parcours.

Ces étapes sont illustrées figure 2.

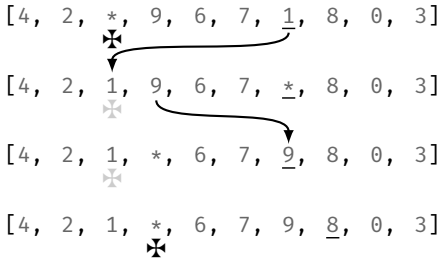

Figure 2: Traitement d'une valeur inférieure au pivot

Question 14 Écrire une fonction `partition(tab, debut, fin)` qui effectue la partition du tableau entre la position `debut` (incluse, c'est là que l'on va prendre la valeur pivot) et la position `fin` exclue. L'exécution de cette fonction aura pour

effet de modifier le tableau comme expliqué ci-dessus, et renverra la position finale de ✠.

Par exemple, on doit avoir le comportement suivant :

```
>>> t = [6, 7, 9, 1, 2, 1, 8, 0, 3]
>>> partition(t, 0, len(t))
5
>>> t
[1, 2, 1, 0, 3, 7, 8, 9, 7]
```

On voit que le tableau comporte deux valeurs `7`, mais c'est dû au fait que la première occurence correspond à la position de ✠, position qui est la valeur retournée par l'appel de la fonction. C'est là que l'on doit mettre la valeur pivot à la fin.

Question 15 Modifier la valeur précédente de telle sorte que l'on mette la valeur pivot dans la case marquée par ✠ avant de renvoyer sa position.

On doit avoir maintenant :

```
>>> t = [6, 7, 9, 1, 2, 1, 8, 0, 3]
>>> partition(t, 0, len(t))
5
>>> t
[1, 2, 1, 0, 3, 6, 8, 9, 7]
```

Question 16 Comment continuer le tri ?

Question 17 Puisqu'il s'agit d'un algorithme récursif, quel est le case de base ?

Question 18 En déduire une fonction `tri_rapide(t)` qui trie

III. Quelques questions supplémentaires

Nous allons commencer par parler de la complexité du tri rapide.

Question 19

1. Simuler, à la main, une étape de partitionnement avec le tableau

   `[1, 2, 3, 4, 5, 6]`.

2. Où est la position finale du pivot dans ce cas et, plus généralement, dans le cas où le tableau est trié par ordre croissant ?

3. Quels sont les appels récursifs suivants ?

4. En déduire que dans le cas d'un tableau trié par ordre croissant, la complexité du tri rapide est quadratique.

Si dans le pire des cas, le tri rapide est quadratique (il n'est donc pas rapide du tout comparé au tri fusion, par exemple), en moyenne, il est effectivement de complexité en $O(n \ln n)$ et comparativement plus rapide que le tri fusion.

Question 20 Reprendre les fonctions du TP6 sur la complexité pour mettre en évidence ce résultat.

Une façon de limiter le risque d'avoir un tri quadratique dans le cas d'un tableau ordonné, une variante fréquemment utilisée est d'utiliser comme pivot la médiane entre la première valeur du tableau, la dernière et la valeur au milieu. On commencera alors par faire, si besoin, un échange entre la première valeur et la valeur pivot retenue.

Question 21

1. Justifier que dans ce cas, le pivot retenu est au milieu du tableau.

On admettra qu'alors le tri est bien linéarithmique.

2. Écrire une fonction `choix_pivot(tab, debut, fin)` qui choisit le pivot comme on vient de l'expliquer, et fait les échanges nécessaires pour que la valeur pivot soit en position `debut` et modifier la fonction `tri_rapide` pour en tenir compte.

Cette méthode n'empêche cependant pas d'avoir dans le pire des cas un comportement quadratique. D'autres variantes du tri rapides le permettent, comme l'algorithme Introsort, mais cela nous emmènerait trop loin du programme.