Tris efficaces, tri par tas

1 Visualisation des tris

Télécharger le fichier `animation_eleves.py` sur mon site web, pour visualiser (et réviser) les tris. Respectez les instructions !

2 Le tri par tas

On étudie dans ce problème un algorithme de tri différent de ceux vus en cours. Il est plus compliqué à mettre en œuvre, mais est assez élégant, et de complexité $O(n \log n)$ dans le pire cas, tout en s'effectuant en place [1] Il est basé sur la structure de *tas*. La figure 1 montre comment on peut interpréter une liste comme un *arbre binaire complet à gauche* :

- à chaque indice de la liste est associé ce qu'on appelle un *noeud* de l'arbre (voir figure 1). Les *étiquettes* des noeuds correspondent aux éléments de la liste.
- L'indice 0 est associé au noeud tout en haut de l'arbre et s'appelle la *racine* de l'arbre. (En informatique, les arbres poussent de haut en bas.)
- Chaque noeud possède zéro, un ou deux fils. Les noeuds ayant un ou deux fils sont appelés des *noeuds internes*, les autres des *feuilles*. Pour un noeud interne on parlera de fils gauche et de fils droit. Si un noeud interne n'a qu'un seul fils, on dira que c'est un fils gauche.
- Inversement, chaque noeud autre que la racine possède un unique *parent*.
- De noeuds qui sont reliés le sont par des *arcs* (l'un est nécessairement le parent de l'autre).
- L'arbre est rempli du haut vers le bas, et de gauche à droite. La racine a l'étiquette `L[0]` et les feuilles ont pour étiquettes les derniers éléments de la liste. Tous les niveaux de l'arbre sont remplis au maximum, excepté peut-être le dernier ou les éléments sont le plus à gauche possible.

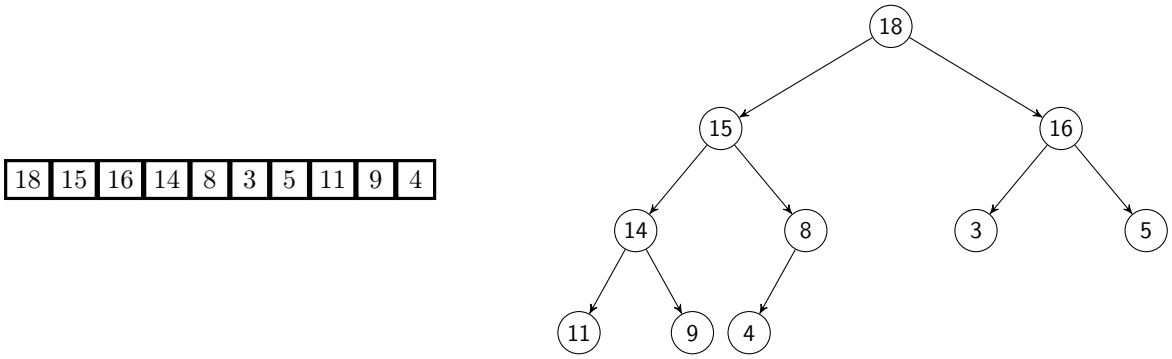

FIGURE 1: une liste et l'arbre associé

On montre facilement, et on l'admettra dans la suite, que si i est l'indice d'un noeud d'une liste de taille n, les indices de ses fils sont donnés par $2i+1$ et $2i+2$ (les fils en question existent si ces indices sont strictement inférieurs a n). Inversement, si $i \neq 0$ est l'indice d'un élément de la liste autre que la racine, l'indice de son parent est donné par la partie entière $\left\lfloor \frac{i-1}{2} \right\rfloor$. On pourra définir dans la suite et utiliser librement les 3 fonctions Python suivantes :

```
def pere(i):
    return((i-1)//2)
```

```
def fg(i):
    return(2*i+1)
```

```
def fd(i):
    return(2*i+2)
```

Venons en maintenant à la définition d'un *tas* : on dit que la liste `L` de taille n vérifie la propriété de tas si pour tout $i \in \{0, \ldots, n-1\}$, `L[i]` est supérieur ou égal à `L[f]`, avec f un fils de i.

1. En quelque sorte, il cumule les avantages du tri rapide et du tri fusion !

La liste donnée en exemple vérifie la propriété de tas, comme il est facile de le voir sur l'arbre associé : l'étiquette d'un noeud interne est plus grande que celles de ses fils.

Le tri par tas fonctionne de la façon suivante :

— transformation de la liste en tas ;

— tri de la liste à partir du tas.

2.1 Fonction entasser

Le but ici est d'écrire une fonction utile dans la suite, qui nous servira à la fois à transformer la liste en tas puis dans un second temps à trier la liste. Les fonctions prennent toutes en paramètre un entier p : c'est la taille de la portion de liste sur laquelle on travaille.

Question 1. Écrire une fonction `imax(L,i,p)` prenant en entrée une liste `L` de taille n, un indice $i < n$, un entier p vérifiant $i < p \leq n$ et renvoyant l'indice $j \in \{i, 2i+1, 2i+2\}$ tel que `L[j]` est le maximum parmi les $\{$`L[f]` $\mid f \in \{i, 2i+1, 2i+2\}, f < p\}$.

```
>>> L=[16, 4, 10, 14, 7, 9, 3, 2, 8, 1]
>>> imax(L,1,10)
3
```

Question 2. On travaille avec une liste dont les éléments d'indice compris dans $[\![i+1, p[\![$ vérifient la propriété de tas : pour $k \in [\![i+1, p[\![$, on a `L[k]` $\geq$ `L[f]` avec f un fils de k d'indice au plus $p-1$ (s'il en existe un). À l'aide de la fonction précédente, écrire une fonction récursive `entasser(L,i,p)` qui rétablit la structure de tas également en l'indice i. Pour cela, on calcule l'indice `m=imax(L,i,p)` :

— s'il vaut i, il n'y a rien à faire ;

— sinon, on échange les valeurs de `L[i]` et `L[m]`, et on rappelle récursivement `entasser` en l'indice m.

La fonction ainsi écrite est de complexité logarithmique en n (et même en p) : en effet, elle est linéaire en la hauteur de l'arbre (dont on vérifie facilement qu'elle est en $O(\log n)$).

```
>>> L=[16, 4, 10, 14, 7, 9, 3, 2, 8, 1]
>>> entasser(L,1,10)
>>> L
[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
```

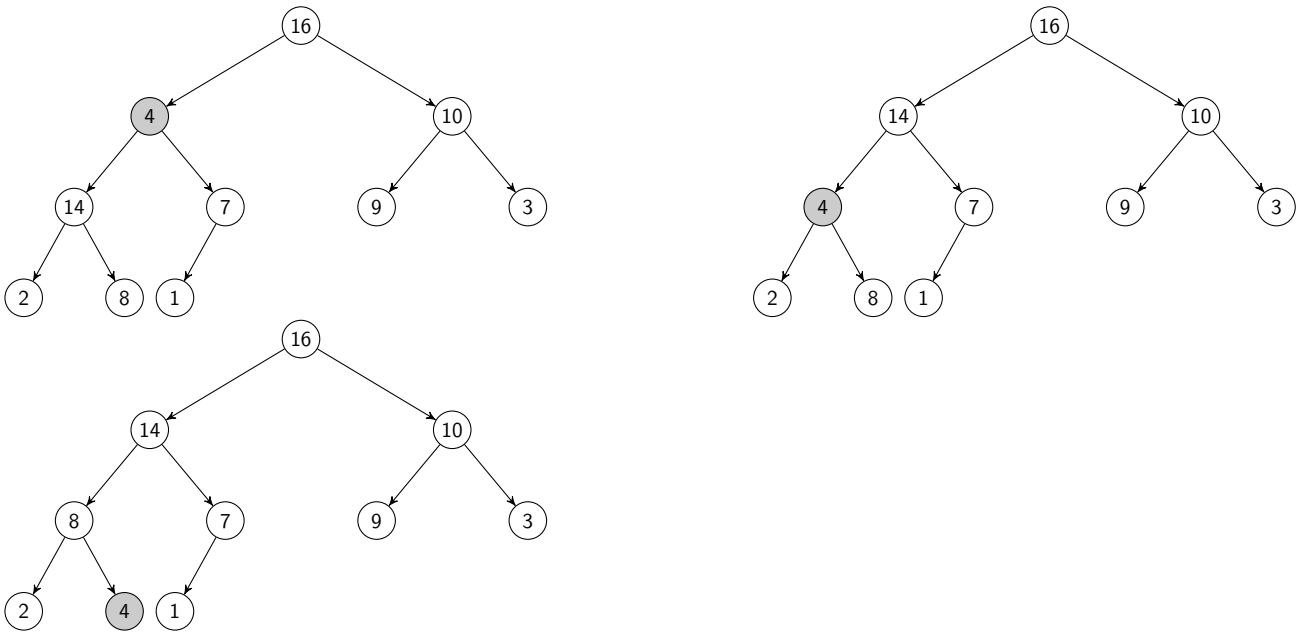

FIGURE 2: L'effet de `entasser(L,1,10)`

2.2 Construction de tas

Le but est de transformer une liste en tas.

Question 3. Écrire une fonction `construit_tas(L)` transformant la liste en tas, de complexité $O(n \log n)$ avec n la taille de la liste. Indication : il suffit d'appeler la fonction précédente avec $p = n$, pour tous les indices i *décroissants*. On commence à construire le tas depuis les feuilles jusqu'à la racine.

```
>>> L=list(range(10))
>>> construit_tas(L)
>>> L
[9, 8, 6, 7, 4, 5, 2, 0, 3, 1]
```

2.3 Tri de la liste

Voyons maintenant comment trier la liste `L`, qu'on suppose transformée en tas. La séquence d'instructions :

```
n=len(L)
L[0], L[n-1] = L[n-1], L[0]
entasser(L,0,n-1)
```

place le plus grand élément de la liste à la fin, et rétablit la structure de tas *sur toute la liste sauf son dernier élément*. On peut avec cette idée trier la liste en plaçant d'abord les plus grands éléments à leurs positions définitives : il suffit de réitérer une opération similaire qui placera successivement le deuxième plus grand élément en avant-dernière position, le troisième plus grand en avant-avant-dernière position, etc...

Question 4. Écrire une fonction `tri_par_tas(L)` prenant en entrée une liste quelconque, la transforme en tas, puis la tri en suivant l'idée précédente. Tester.

3 Quelques modifications des tris

Les exercices suivants effectuent quelques variations autour des tris du cours, que vous pouvez télécharger sur le site web (fichier `tri_cours.py`). Le tri fusion présenté n'est pas tout à fait le même que celui du cours.

Exercice 1. *Un tri rapide en quelques lignes.* Écrire une fonction effectuant un tri « rapide » non en place : il renvoie une copie triée de la liste. Dans l'idée, cela sera similaire au tri fusion : on écrit une fonction `partition(L)` qui renvoie trois morceaux : le pivot, la portion de liste constituée des éléments $<$ pivot, la portion de liste constituée des éléments $\geq$ pivot (sans le pivot !), puis une fonction de tri rapide en quelques lignes.

Exercice 2. *Modification du tri rapide.* Que se passe-t-il sur des listes où il y a beaucoup d'éléments égaux avec le tri rapide (comme par exemple des listes de bits ?). Écrire une fonction `partition2(L,g,d)` qui répartit la portion en 3 morceaux : les éléments strictement inférieurs au pivot, les éléments égaux au pivot, les éléments strictement supérieurs, et renvoie les deux indices de début des deux dernières portions. Raisonner d'abord au brouillon en ayant une idée claire de l'invariant de boucle à maintenir. Vérifier qu'avec cette fonction de partition, le tri est très efficace sur des listes avec peu d'éléments distincts (qu'on pourra générer aléatoirement avec `randint` du module `random`), contrairement au tri standard.

Exercice 3. *Modification du tri rapide (2).* Lorsque les partitions se passent mal (par exemple sur des listes de bits, ou des listes presque triées lorsqu'on ne choisit pas le pivot au hasard), le nombre d'appels récursifs imbriqués devient trop important pour Python. Modifier le tri rapide pour obtenir une fonction `tri_rapide2`, qui transforme l'appel récursif à `aux` sur la plus grande portion en boucle `while`.

Exercice 4. *Réécriture du tri rapide sans récursivité.* À l'aide d'une pile, réécrire le tri rapide pour qu'il n'utilise plus du tout la récursivité.

Exercice 5. *Comparaisons des tris.* Le but est de comparer les tris sur des listes générés aléatoirement. Travaillez avec le fichier `tri_cours.py`. La fonction `randint` du module `random` permet d'obtenir des entiers aléatoires : `randint(a,b)` fournit un entier aléatoire de $[\![a, b]\!]$.

```
>>> from random import randint
>>> randint(0,10000)
8310
```

La fonction `clock` du module `time` permet de mesurer le temps. Elle s'utilise comme suit pour mesurer le temps d'exécution d'un script :

```
from time import clock
t=clock()
[script]
t2=clock()-t #t2 contient le temps d'exécution du script.
```

Enfin, la fonction `plot` de `matplotlib.pyplot` permet de tracer des graphe, mais vous le saviez déja. Pour chacun des 5 tris du cours, et pour toute taille de liste t entre 10 et 200 par pas de 10, mesurer la moyenne sur 10 essais du temps d'exécution de chacun des tris sur une liste d'entiers aléatoires de $[\![0, 10000]\!]$ de taille t, et tracer le résultat. Voici un squelette pour faire ceci efficacement.

```
tris=[tri_selection, tri_insertion, tri_bulles, tri_rapide, tri_fusion]
noms_tris=["selection", "insertion", "bulles", "rapide", "fusion"]
temps_execution=[ [] for _ in range(5)] #liste contenant des listes de temps d'exécution
tailles=list(range(10,210,10)) #de 10 à 200 par pas de 10
for t in taille:
   for i in range(5):
       [...] #calcul des temps d'exécution
for i in range(5):
    plt.plot(tailles, temps_execution[i], label=noms_tris[i]) #tracé
plt.legend(loc="upper left") #positionnement de la légende.
plt.show()
```

Attention à faire une copie de la liste à trier avant de lancer un tri : celui-ci s'effectue en place et la liste sera triée à la fin.