TP INFO 5 | Algorithmes de tri

Nous exposerons ici quelques algorithmes de tri classiques (en version itérative ou récursive) avant d'aborder quelques exercices.

1. Introduction

Avant de traiter des données, il peut être judicieux de les trier. Par exemple, si le traitement comporte de nombreuses recherches, il est astucieux de trier au préalable les données afin d'utiliser l'algorithme de recherche dichotomique plutôt qu'une recherche linéaire. Nous avons également vu dans le TP 3 qu'un tri judicieux ouvrait la voie à des heuristiques gloutonnes exactes.

Trier pour traiter plus efficacement

Structurer des données au moyen d'une relation d'ordre permet dans de nombreux cas d'en accélérer le traitement.

D'un point de vue académique, l'étude des algorithme de tris est aussi intéressante par la richesse des stratégies rencontrées (itération, récursion, diviser pour régner, comparaison des performances des diffférentes méthodes, etc.).

Nous commencerons par un peu de vocabulaire.

Tri en place

Un algorithme de tri est dit en place si la liste est triée par le seul effet de permutations de ses termes.

L'avantage des tris en place est de ne pas encombrer la mémoire. Cela sera particulièrement important voire critique dans le cas de listes très longues.

Nous allons étudier en particulier quatre tris classiques et quelques autres (voire des variantes) en exercice :

- Le tri par sélection (selection sort) ;
- Le tri par insertion (Insertion sort) ;
- Le tri rapide (Quick sort) ;
- Le tri fusion (Merge sort).

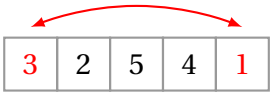
Les deux premières méthodes sont naturellement itératives alors que les deux autres sont récursives.

2. Le tri par sélection

Le principe du tri par sélection pour une liste de termes numérotés de 0 à $n-1$ est le suivant :

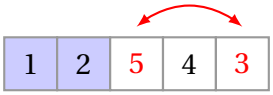
- rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0 ;
- rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 1 ;
- continuer ainsi jusqu'à ce que le tableau soit entièrement trié.

Il s'agit d'un tri en place. En voici une illustration :



On permute `t[0]` et le minimum de `t[0:]`

On permute `t[1]` et le minimum de `t[1:]`



On permute `t[2]` et le minimum de `t[2:]`

On permute `t[3]` et le minimum de `t[3:]`

La liste est triée

On note n la longueur de `t`.

On effectue deux boucles `for` imbriquées.

La première pour faire varier i de 0 à $n-1$, et la seconde pour rechercher la position du minimum de `t[i:]` (par l'algorithme classique vu en première année) avant d'effectuer la permutation de `t[i]` et de ce minimum.

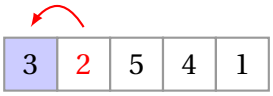
À vous de jouer :

1 *Tri par sélection*

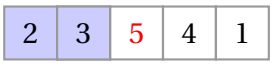
Écrire une fonction `triSelection` prenant en argument une liste d'entiers `t` et la triant en place selon cet algorithme.

3. Le tri par insertion

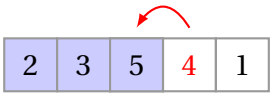
Cet algorithme est utilisé naturellement par la plupart des personnes pour trier des cartes : prendre les cartes mélangées une à une sur la table, et former une main en insérant chaque carte à sa place. Il s'effectue en place.



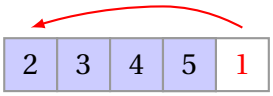
Insertion de 2 dans `t[0:1]`



Insertion de 5 dans `t[0:2]`



Insertion de 4 dans `t[0:3]`



Insertion de 1 dans `t[0:4]`

La liste est triée

On note n la longueur de la liste.

L'algorithme est le suivant : pour tout indice $i \in [\![1, n]\!]$, on insère *à sa place* `t[i]` dans `t[0:i]`.

On effectue donc d'abord une boucle `for`.

L'insertion de `t[i]` peut se faire par permutations successives ou par décalages (cette méthode étant moins coûteuse, cf. ci-dessus). Puisque la position finale de `t[i]` n'est pas connue à l'avance, il faudra programmer une boucle `while` dans la boucle `for`.

Il est temps de passer à la pratique :

2 *Tri par insertion f*

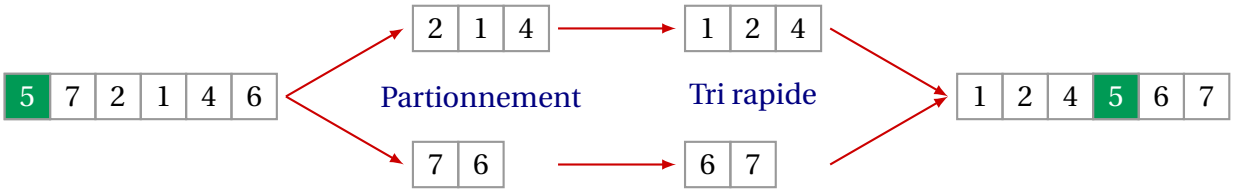
Écrire une fonction `triInsertion` prenant en argument une liste d'entiers `t` et la triant selon cet algorithme.

4. Le tri rapide

Il s'agit d'un exemple d'application du paradigme *diviser pour régner* que nous avons déjà rencontré dans la recherche dichotomique dans une liste triée : on coupe en deux la liste afin d'appliquer récursivement l'algorithme à chacun des deux morceaux puis combiner les deux réponses pour obtenir le tri final :

- On crée la liste `tinf` des termes de la forme `t[i]` tels que $i > 0$ et $t[i] <= t[0]$ puis la liste `tsup` des termes de la forme `t[i]` tels que $i > 0$ et $t[i] > t[0]$.
- On applique récursivement l'algorithme à `tinf` et `tsup` : on note `ti` et `ts` ces deux listes triées dans l'ordre croissant.
- On renvoie la liste `ti+[t[0]]+ts` (on met le terme `t[0]` à sa place pour obtenir le tri complet de la liste initiale).

Voici le principe sur un exemple :



À vos claviers !

3 *Tri rapide ff*

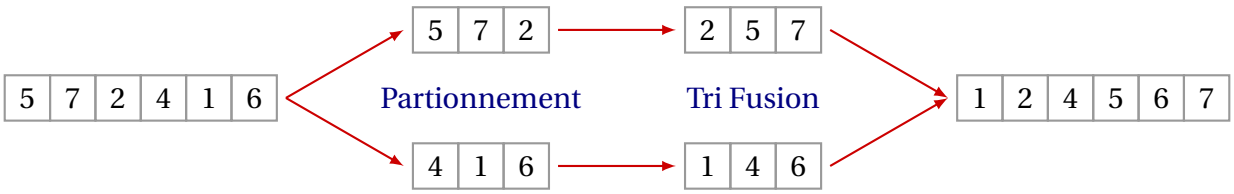
Écrire une fonction `triRapide` prenant en argument une liste d'entiers `t` et renvoyant sa version triée en suivant cet algorithme. On ne demande pas un tri en place, celui-ci est plus technique et sera étudié plus loin dans un exercice.

5. Le tri par fusion

Le tri par fusion est un autre exemple d'application du paradigme *diviser pour régner*. Il consiste à :

- Couper la liste en deux parties à peu près égales.
- Trier récursivement chaque sous-liste.
- Fusionner les deux sous-listes triées en une liste triée.

Voici le principe sur un exemple :



Il reste à coder tout cela :

4 — *Le tri fusion fff*

1. Écrire une fonction `fusion` prenant en argument deux listes `t1` et `t2` triées dans l'ordre croissant et renvoyant leur fusion en une seule liste croissante. On donnera deux solutions : une itérative et une autre récursive.
2. En déduire une fonction `triFusion` prenant en argument une liste d'entiers `t` et renvoyant sa version triée suivant cet algorithme. On ne demande pas une version en place, trop technique à mettre en œuvre.

6. Exercices

5 — *Le tri bulle*

Le tri bulle est *un tri en place* fondé sur le principe suivant :

- ⇛ On parcourt la liste du premier au dernier terme en permutant les paires de termes consécutifs qui ne sont pas rangés dans l'ordre croissant.
- ⇛ On parcourt à nouveau la liste du premier à l'avant-dernier terme en permutant les paires de termes consécutifs qui ne sont pas rangés dans l'ordre croissant.
- ⇛ Et ainsi de suite jusqu'à épuisement des termes.

Après un premier parcours de la liste, le maximum se retrouve donc en dernière position (il a remonté la liste comme une bulle de champagne). Au second passage, c'est le maximum de la liste privée de son maximum qui arrive en avant dernière position. Et ainsi de suite.

1. Mettre en œuvre cet algorithme *à la main* sur `t=[6,-3,7,5,1,0]`.
2. Écrire une fonction *non récursive* `triBulle` prenant en argument une liste `t` d'entier et la triant en place selon cette méthode.

6 — *Tri par dénombrement f*

Soit $N \in \mathbb{N}^*$. On suppose dans cet exercice que les listes sont constituées d'entiers compris entre 0 et $N-1$. Écrire une fonction `triParDenombrement` d'arguments une liste d'entiers `t` et `N` qui renvoie la liste `t` triée. On pourra utiliser une liste auxiliaire `tcomptage` qui compte les occurrences de chaque entier de $[\![0, N-1]\!]$.

7 — *Versions récursives des tris par sélection et par insertion* ***ff***

1. Écrire une fonction récursive `triSelection` prenant en argument une liste `t` d'entiers et `i` un indice valide de `t` et qui trie `t` de l'indice `i` à l'indice `len(t)-1` selon l'algorithme de tri par sélection en place.
2. Écrire une fonction récursive `triInsertion` prenant en argument une liste `t` d'entiers et `i` un indice valide de `t` et qui trie `t` de l'indice `0` à l'indice `i` selon l'algorithme de tri par insertion en place.

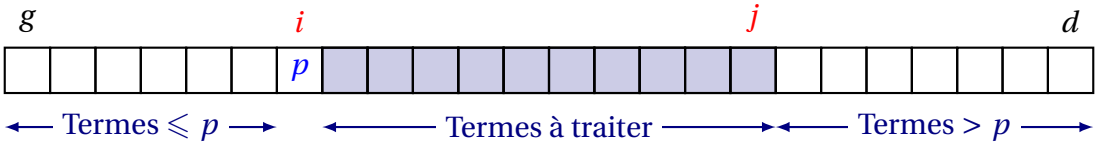
8 — *Version en place du tri rapide* ***fff***

Afin d'améliorer le tri rapide étudié ci-dessus, nous allons mettre en œuvre un algorithme de segmentation. *Segmenter une liste* `t` *entre les indices* `g` *et* `d` *selon un pivot* p signifie la transformer par permutations successives en une nouvelle liste `t'` telle qu'il existe un indice i vérifiant $t'[i] = p$ et pour tout $\ell \in [\![g, i[\![$, $t'[\ell] \leqslant p$ et $\forall \ell \in]\!]i, d]\!]$, $t'[\ell] > p$.

Pour effectuer cette *segmentation* en place, l'idée est d'effectuer un seul parcours de la liste entre les indices `g` et `d` en préservant la propriété suivante :

$$t[i] = p, \quad \forall \ell \in [\![g, i[\![, \ t[\ell] \leqslant t[i] \quad \text{et} \quad \forall \ell \in]\!]j, d]\!], \ t[\ell] > t[i]$$

On utilise ainsi deux indices i et j vérifiant la propriété précédente. On initialise par $(i, j) = (g, d)$. En fin d'algorithme $j = i$, et la segmentation est accomplie.



En fin de segmentation, le pivot p est à la bonne place, ie est à la position qu'il occupe dans la liste triée.

1. En déduire une fonction récursive `triSeg(t,d,f)` (où `t` est une une liste et `d`, `f` sont deux indices valable de `t` vérifiant `d<=f`) qui trie dans l'ordre croissant la portion de `t` comprise entre les indices `d` (comme début) et `f` (comme fin), en n'effectuant que des permutations de termes de `t`.
2. Écrire une fonction `triRapide` prenant en argument une liste `t` d'entier et la triant dans l'ordre croissant par permutations successives selon l'algorithme du tri rapide en place.

9 — *Algorithme de Quick select* ***fff***

Écrire une fonction `quickSelect` prenant en argument une liste `t` d'entiers et `k` un indice valide de `t` et qui renvoie le terme en position k dans la liste `t` triée. On proposera une méthode plus efficace que celle consistant à *trier toute la liste* puis renvoyer le terme souhaité.

10 — *Tri tricolore* ***fff***

On s'intéresse au tri d'une liste t à valeurs dans $[\![0, 2]\!]$. Proposer puis implémenter un algorithme de *tri en place* de t ne réalisant qu'un seul passage sur la liste `t`.

7. Indications

1

On peut structurer l'algorithme par une double boucle `for`.

2

On peut effectuer l'insertion au moyen d'une boucle `while` par permutations successives ou décalage vers la droite.

3

Le cas d'une liste vide doit faire partie des cas de base.

4

Pour la version itérative, on pourra initialiser une liste `aux` avec des zéro de taille égale à la liste fusionnée puis la mettre à jour en piochant dans `t1` ou `t2` dans l'ordre croissant. jusqu'à remplissage complet de `aux`

5

Il suffit de coder une double boucle `for`.

6

Une fois obtenue la liste `comptage`, il suffit de concaténer les liste de la forme `[i]*comptage[i]`.

7

Il suffit à chaque fois de réaliser une itération de l'algorithme de tri puis les autres récursivement.

8

Pour trier entre les indices d et f, on commence par segmenter entre ces indices puis, en notant i l'indice final du premier terme dans la liste segmentée, on appelle récursivement le tri entre les indices d et $i-1$ puis entre les indices $i+1$ et f. Attention à bien identifier le cas de base afin d'éviter une récursion infinie.

9

S'aider de l'algorithme de segmentation utilisé dans la version en place du tri rapide.

10

Utiliser le principe de la segmentation utilisé dans la version en place du tri rapide : utiliser trois indices afin de délimiter quatre zones dans la liste.