Informatique — MPSI/PCSI, 2023–2024

TP n° 8 – Quelques algorithmes de tri

Un problème de tri est constitué en entrée d'une séquence[1] de n nombres $[a_1, a_2, \dots a_n]$ que l'on souhaite permuter, c'est-à-dire réorganiser, de façon à obtenir en sortie une autre séquence $[b_1, b_2, \dots b_n]$ vérifiant $b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n$. La séquence en entrée s'appelle une **instance** du problème de tri. Les nombres à trier sont parfois appelés **clés**.

Le tri est une opération majeure en informatique (maints programmes l'emploient comme phase intermédiaire), ce qui explique que l'on ait inventé un grand nombre d'algorithmes de tri. L'algorithme optimal pour une application donnée dépend, entre autres facteurs, du nombre d'éléments à trier, de la façon dont les éléments sont plus ou moins triés initialement, des restrictions potentielles concernant les valeurs des éléments, ainsi que du type de périphérique de stockage à utiliser : mémoire principale ou disques.

Un algorithme est dit correct si, pour chaque instance en entrée, il se termine en produisant la bonne sortie. L'on dit qu'un algorithme correct résout le problème donné. Pour déterminer si un algorithme est correct, il est nécessaire d'utiliser la notion d'**invariant de boucle** pour lequel nous devons montrer :

Initialisation Il est vrai avant la première itération de la boucle.

Conservation S'il est vrai avant une itération de la boucle, il le reste avant l'itération suivante.

Terminaison Une fois terminée la boucle, l'invariant fournit une propriété utile qui aide à montrer la validité de l'algorithme.

1. La notion de séquence en informatique désigne différents types d'objets, qui partagent le même genre de structure (à peu de choses près il s'agit d'une structure de vecteur, au sens mathématique du terme, c'est-à-dire un ensemble ordonné d'objets homogènes). En python, une séquence désignera, au choix, une liste, un tuple ou un tableau 1D. Dans ce TP, on se limitera à des listes.

1 Tri à bulles

Le fonctionnement du tri à bulles s'appuie sur des permutations répétées d'éléments contigus qui ne sont pas dans le bon ordre. Le principe est parcourir la séquence, à partir de la fin, en comparant deux éléments et en les mettant éventuellement dans le bon ordre (voir figure 1). Le parcours est répété tant qu'il y a eu au moins une inversion dans le parcours précédent. L'absence d'inversion assure que chaque élément est inférieur ou égal au suivant, et donc que la séquence (de gauche) est triée.

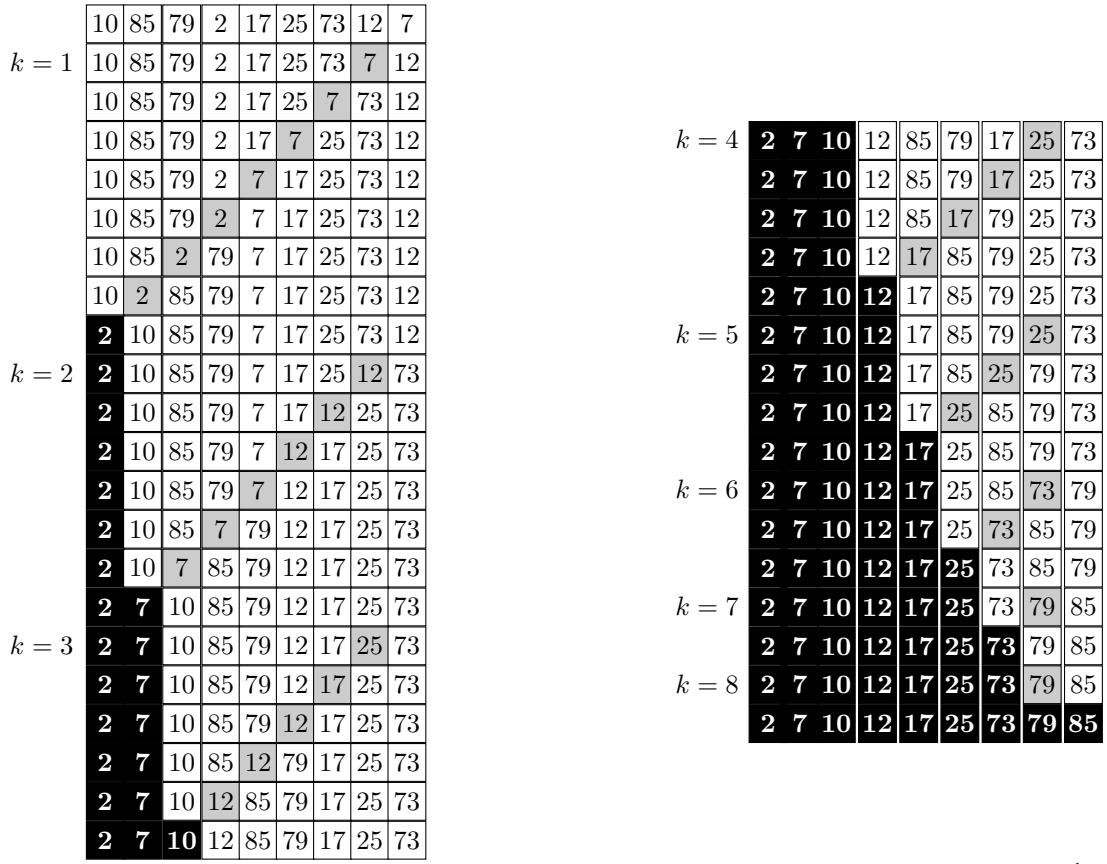

FIGURE 1 – Fonctionnement du tri à bulles avec la séquence $[10, 85, 79, 2, 17, 25, 73, 12, 7]$. À chaque itération $k \in [\![1, 8]\!]$, la case grise représente le résultat de la comparaison des couples, en noir la portion de séquence déjà triée.

Exercice 8.1 (Tri à bulles)

1. Écrire une fonction `TriBulles` qui prend comme argument une liste `L` et qui trie la liste « sur place » avec l'algorithme du tri à bulles.
2. Donner l'ordre de grandeur du nombre d'opération réalisées pour une liste de longueur n dans le meilleur et le pire des cas.

2 Tri par insertion

Le tri par insertion s'inspire de la manière dont la plupart des gens trient des cartes à jouer. Prenant successivement des cartes au dessus d'une pile sur la table, il s'agit de les insérer « au bon endroit » dans le jeu tenu (en main gauche figure 2). Si le joueur saisit ses cartes sur une table alors pour savoir où placer une nouvelle, il la compare avec les autres, de droite à gauche. À chaque instant, le jeu tenu est trié. C'est un algorithme efficace quand il s'agit de trier un petit nombre d'éléments.

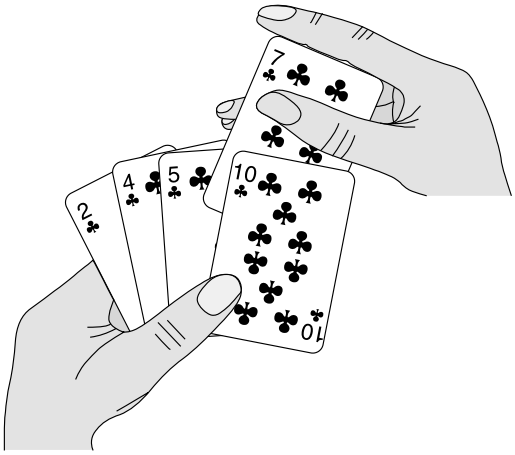

FIGURE 2 – Tri par insertion de cartes à jouer.

Le tri par insertion prend en entrée une séquence $L = [a_0, a_1, \ldots, a_{n-1}]$ de longueur n. On note $L_0 = [a_0]$ la première séquence triée. Pour tout $k \in [\![1, n-1]\!]$, il s'agit d'insérer l'élément a_k dans la séquence précédemment triée $L_{k-1} = [b_0, b_1, \ldots, b_{k-1}]$ en procédant par comparaison puis décalage vers la gauche si nécessaire de sorte que L_k se retrouve triée. On illustre ce processus avec la séquence $[5, 2, 4, 6, 1, 3]$ sur la figure 3.

$k=1$	5	**2**	4	6	1	3
	2	5	4	6	1	3
$k=2$	2	5	**4**	6	1	3
	2	**4**	5	6	1	3
$k=3$	2	4	5	**6**	1	3

$k=4$	2	4	5	6	**1**	3
	2	4	5	**1**	6	3
	2	4	**1**	5	6	3
	2	**1**	4	5	6	3
	1	2	4	5	6	3

$k=5$	1	2	4	5	6	**3**
	1	2	4	5	**3**	6
	1	2	4	**3**	5	6
	1	2	**3**	4	5	6

FIGURE 3 – Fonctionnement du tri par insertion avec la séquence $[5, 2, 4, 6, 1, 3]$. À chaque itération $k \in [\![1, 5]\!]$, la case noire représente la clé, comparée aux cases grises de la portion de séquence déjà triée.

Exercice 8.2 (Tri par insertion)

1. Écrire une fonction `TriInsertion` qui prend comme argument une séquence `L` est qui la trie « sur place » par insertion.
2. Déterminer la complexité au pire puis au mieux.
3. Montrer que l'invariant de boucle est ici que la séquence $L_k = [a_0, a_1, \ldots, a_k]$, pour $0 \leqslant k \leqslant n-1$ est triée.

On peut remarquer que le tri par insertion revient à **insérer** un élément en position $k \in [\![1, n-1]\!]$ dans une séquence L_{k-1} triée et que l'insertion elle-même peut s'écrire :

$$\forall k \in [\![1, n-1]\!],\ \textsc{Inserer}\,(L_{k-1}, a_k) = \begin{cases} \textsc{Inserer}\,(L_{k-2}, a_{k-1}) \circ \textsc{Permuter}\,(a_{k-1}, a_k) & \text{si } a_k < a_{k-1} \\ L_k & \text{sinon} \end{cases}$$

4. Écrire une fonction `Permuter` qui prend comme arguments une séquence `L` et deux indices d'objets à permuter `i` et `j` et qui réalise « sur place » la permutation $L_i \leftrightarrow L_j$.
5. Écrire une fonction `Inserer` qui prend comme arguments une séquence `L` et un indice `k` de l'élément à insérer et procède à son insertion dans la sous-séquence L_{k-1} par appels récursifs.
6. Écrire une fonction `TriInsRec` qui prend comme argument une séquence `L` et comme argument optionnel un indice associée au numéro de l'itération (1 par défaut) et trie la séquence `L` par insertion, « sur place » et par appels récursifs.

3 Tri par sélection

Le tri par sélection d'une séquence $L[i..j]$ à n éléments consiste à déterminer l'élément minimum de la séquence, à échanger cet élément avec le premier élément de la séquence et à réaliser récursivement ces opérations sur la séquence $L[i+1..j]$.

ℹ *On pourra librement utiliser la fonction* `Permuter` *définie dans l'exercice précédent.*

Exercice 8.3 (Tri par sélection)

1. Écrire une fonction `TriSelection` qui prend comme argument une séquence à trier `L` et qui la trie « sur place ».
2. Déterminer la complexité du tri par sélection.
3. Écrire une fonction `TriSelRec` qui prend comme arguments une séquence `L` de longueur n et un indice `k` (par défaut à 0) et qui procède au tri par sélection de la séquence $(L_j)_{k \leqslant j < n}$ par appels récursifs.

On peut évidemment procéder de façon similaire avec les maximums.

4. Écrire une fonction `TriSelection` qui prend comme argument une séquence à trier `L` et qui la trie « sur place » avec les maximums.

4 Tri par partition-fusion

Le principe du tri par partition-fusion ou tri fusion est de couper une séquences en deux morceaux toujours égaux (à un élément près) et de répéter l'opération sur les sous-séquences jusqu'à obtenir des séquences de longueur 0 ou 1 (déjà triées donc). Il reste ensuite l'étape délicate appelée fusion. Elle consiste à fusionner deux séquences triées de telle sorte qu'on obtienne une séquence triée.

```
[10  85  79  2  17  25  73  12  7]
[10  85  79  2][17  25  73  12  7]
[10  85][79  2][17  25][73][12  7]
[10][85][79][2][17][25][73][12][7]
[10  85][2  79][17  25][73][7  12]
[2  10  79  85][17  25][7  12  73]
[2  7  10  12  17  25  73  79  85]
```

FIGURE 4 – État successifs de la séquence donnée en argument.

L'étape clé est celle de fusion. Partant de deux séquences triées, il s'agit de les fusionner en une seule. Pour ce faire, on compare les éléments de chacune des deux séquences et on déplace le plus petit dans une nouvelle séquence. Quand une des deux séquence est vide, on déplace les éléments restants de la seconde séquence.

```
[3,5,8,9]    [1,2,6,10]   []
[3,5,8,9]    [2,6,10]     [1]
[3,5,8,9]    [6,10]       [1,2]
[5,8,9]      [6,10]       [1,2,3]
[8,9]        [6,10]       [1,2,3,5]
[8,9]        [10]         [1,2,3,5,6]
[9]          [10]         [1,2,3,5,6,8]
[]           [10]         [1,2,3,5,6,8,9]
[]           []           [1,2,3,5,6,8,9,10]
```

FIGURE 5 – Exemple de fusion.

Exercice 8.4 (Tri par partition-fusion)

1. Écrire une fonction `Fusion` qui prend comme arguments deux séquences triées `M` et `N` et qui renvoie une séquence triée issue de leur fusion.
2. Écrire une fonction `TriFusion` qui prend comme argument une séquence à trier `L` et qui renvoie une copie de cette séquence triée.
3. Estimer la complexité de cet algorithme.

5 Tri rapide

L'idée du tri rapide (ou *quick sort*) est à nouveau un partage en deux analogue au tri par partition-fusion mais au lieu d'une partition dichotomique systématique de la séquence, qui oblige à une fusion délicate et à l'emploi de listes temporaires, le tri rapide partitionne la séquence autour d'une valeur charnière, appelée **pivot**. Les appels récursifs sont alors lancés jusqu'à tomber sur des listes de taille 1 ou 0, donc déjà triées.

Pour le choix du pivot, plusieurs stratégies sont possibles ; de ce choix dépendra la complexité de l'algorithme. Une stratégie « classique » est de considérer le premier élément de la séquence comme pivot.

ℹ *On pourra librement utiliser la fonction* `Permuter` *définie dans l'exercice 2.*

Exercice 8.5 (Tri rapide)

1. Écrire une fonction `Partitionnement` qui prend comme arguments une séquence `L` et deux entiers `i` et `j` et qui partitionne la liste `L[i:j+1]` suivant le pivot `L[i]` et renvoie la nouvelle position du pivot.
2. Écrire une fonction `TriRapide` qui prend comme argument une séquence à trier `L` et qui renvoie une copie de cette séquence triée.
   ℹ On pourra utiliser des arguments optionnels sous la forme :

   ```
   def TriRapide(L, i=0, j=None):
       ...
   ```

3. Estimer la complexité de cet algorithme.

6 Tri par comptage

Le principe du tri par comptage (ou tri comptage, ou tri par dénombrement) repose sur la construction de l'histogramme des données (nombres entiers), puis le balayage de celui-ci de façon croissante, afin de reconstruire les données triées.

Exercice 8.6 (Tri par comptage)

Soit N un entier naturel non nul. On cherche à trier une liste `L` d'entiers naturels strictement inférieurs à N.

1. Écrire une fonction `comptage` qui prend comme arguments une séquence `L` et un entier `N` et qui renvoie une liste dont le k-ième élément désigne le nombre d'occurrences de l'entier k dans la liste `L`.
2. Écrire une fonction `TriComptage` qui prend comme arguments une séquence `L` et un entier `N` et qui trie la liste `L` « sur place ».
3. Estimer la complexité de cet algorithme.

* *
*