Algorithmes de tri

Capacités attendues

- ✓ Écrire un algorithme de tri.
- ✓ Décrire un invariant de boucle qui prouve la correction des tris par insertion, par sélection.

On s'intéresse dans ce chapitre à des algorithmes permettant de trier (dans l'ordre croissant) les éléments d'une liste. Cela suppose que ces éléments sont de même type et qu'ils sont **comparables** : c'est le cas des nombres (type `int` ou `float`), mais aussi des chaînes de caractères (ordre lexicographique).

1 Le tri par sélection

Principe

Le tri par sélection consiste à :

- rechercher le élément de la liste, et l'échanger avec le élément ;
- rechercher le élément de la liste, et l'échanger avec le élément ;
- continuer sur le même principe jusqu'à ce que la liste soit triée.

Exercice 1 Dérouler le tri par sélection sur l'exemple ci-dessous : compléter à chaque étape l'état de la liste.

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

Définition

L'algorithme de tri par sélection est le suivant (on suppose que l'indice du premier élément est 0).

```
fonction tri_selection(tableau t) :
    n ← taille du tableau t
    pour i variant entre 0 et n − 1 :
        i_min ← i
        pour j variant entre i + 1 et n − 1 :
            si t[j]<t[i_min] :
                i_min ← j
        échanger t[i_min] et t[i]
```

Remarque : la fonction tri_selection trie le tableau t **en place**.

Proposition

Le tri par sélection se termine, est correct, et son coût en comparaisons est quadratique : si le tableau est de taille n, le nombre de comparaisons est de l'ordre de n^2.

Définition

On appelle **invariant de boucle** une propriété qui est vraie avant et après chaque itération.

Démonstration

- **Terminaison**

  L'algorithme utilise deux boucles bornées (imbriquées), donc il finira par se terminer.

- **Correction**

  On considère l'invariant de boucle : après k tours de la première boucle bornée (i variant entre 0 et $n-1$), les k premiers éléments de t sont les plus petits, triés dans l'ordre croissant.

  En effet, au premier tour de boucle, i_{min} est l'indice du plus petit élément donc, après échange de t$[i_{\text{min}}]$ et t$[i]$, le premier élément de t est le plus petit. Supposons maintenant que l'invariant de boucle est vrai après k tours de boucle. Au tour de boucle suivant, i_{min} est l'indice du plus petit élément parmi les éléments d'indice compris entre k et $n-1$. Comme t$[i_{\text{min}}]$ et t$[i]$ sont ensuite échangés, les $k+1$ éléments sont bien les plus petits, et triés dans l'ordre croissant.

- **Coût**

  Il y a n tours de la boucle principale (i variant entre 0 et $n-1$), contenant chacun $n-i-1$ tours de boucle secondaire (j variant entre $i+1$ et $n-1$). Cela donne un nombre de comparaisons égal à :

$$\sum_{i=0}^{n-1}(n-1-i) = (n-1)+(n-2)+\cdots+2+1 = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}\,. \qquad \square$$

Consigne importante

Pour les **exercices qui suivent** et dans lesquels il est demandé un programme Python, on se basera sur le fichier `tris.py` téléchargeable sur notre site et contenant du code Python à compléter et des tests adéquats. Avant de compléter le programme, supprimer `pass` dans les fonctions.

Exercice 2

Compléter et tester la fonction `tri_selection(lst)` qui prend en argument une liste et la trie **en place** selon le principe du tri par sélection.

2 Le tri par insertion

Principe

Le tri par insertion consiste à parcourir un tableau et à au fur et à mesure les éléments à leur place dans la partie .

Exercice 3

Dérouler le principe du tri par insertion sur l'exemple ci-dessous : compléter à chaque étape l'état de la liste.

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

Définition

L'algorithme de tri par insertion est le suivant (on suppose que l'indice du premier élément est 0).

```
fonction tri_insertion(tableau t) :
    n ← taille du tableau t
    pour i variant entre 1 et n − 1 :
        el_courant ← t[i]
        j ← i
        tant que j > 0 et el_courant <t[j − 1] :
            t[j] ← t[j − 1]
            j ← j − 1
        t[j] ← el_courant
```

Remarque : la fonction tri_insertion trie le tableau t **en place**.

Exercice 4

Compléter et tester la fonction `tri_insertion(lst)` qui prend en argument une liste et la trie **en place** selon le principe du tri par insertion.

Exercice 5

Dans quelle situation le nombre de comparaisons est-il le plus petit possible ? Le plus grand possible ?

Exercice 6

Le module `random` de Python fournit la fonction `randint(a, b)` qui prend en argument deux entiers `a` et `b` et renvoie un nombre entier au hasard dans l'intervalle $[\texttt{a}\,;\,\texttt{b}]$.

1. Compléter et tester la fonction `liste_au_hasard(n)` qui renvoie une liste composée de `n` entiers tirés au hasard compris entre 1 et 10^6 (on rappelle que 10^6 =`10**6` en Python).
2. On considère la liste renvoyée par l'appel `liste_au_hasard(10**3)`.
   (a) Quel temps faut-il pour trier cette liste avec l'algorithme de tri par insertion ?

   (Voir exemple dans le fichier téléchargeable sur notre site `exemple_mesure_temps.py`)

   (b) Comparer avec la méthode `sort` de Python.
3. Mêmes questions avec la liste renvoyée par par l'appel `liste_au_hasard(10**4)`.

Proposition

Le tri par insertion se termine, est correct, et son coût en comparaisons est quadratique dans le pire des cas : si le tableau est de taille n, le nombre de comparaisons est de l'ordre de n^2 dans le pire des cas.

Démonstration

- **Terminaison**
  À chaque tour de la boucle principale (qui est bornée), la boucle non bornée se termine. En effet, la variable j est constamment décrémentée et le test $j > 0$ finit donc par être faux.
- **Correction**
  On considère l'invariant de boucle : après k tours de la première boucle bornée (i variant entre 1 et $n-1$), les k premiers éléments de t sont triés dans l'ordre croissant.

  Au second tour de boucle, le premier élément de t est le plus petit. Supposons maintenant que l'invariant de boucle est vrai après k tours de boucle. Au tour de boucle suivant, l'élément d'indice k est inséré parmi les éléments d'indice compris entre 0 et $k-1$ de sorte que les $k+1$ premiers éléments soient bien triés dans l'ordre croissant.
- **Coût**
  Il y a $n-1$ tours de la boucle principale (i variant entre 1 et $n-1$) et, dans le pire des cas, i tours de la boucle "tant que" (car j varie entre i et 1). Cela donne un nombre de comparaisons égal à :

$$\sum_{i=1}^{n-1} i = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}\,. \qquad \square$$

Remarques

- On peut remarquer que le test "$j > 0$ et $e <$t$[j-1]$" comprend deux comparaisons, mais cela ne change pas l'ordre de grandeur du coût, qui reste quadratique.
- Le coût dans le meilleur des cas est par contre linéaire (de l'ordre de n), ce qui le rend plus performant que certains des algorithmes les plus rapides (tri fusion, tri rapide, dont le coût est de l'ordre de $n \log n$).
  En Python, la méthode `sort` utilise le *Timsort*, qui conjugue les principes du tri par insertion et du tri fusion.