Informatique

TP 2

Algorithmes de tri

Instructions

Dans ce TP, on pourra utiliser indifféremment des listes Python ou des tableaux NumPy à une dimension.

1 Quelques tris simples

On s'intéresse à des tris de données sur lesquelles on suppose uniquement des **opérations de comparaison** (<=, <, >= ou >).

Tous les tris de ce §1 sont dits « **en place** ».

1.1 Tri par propagation (tri « à bulles », *bubble sort*)

Le « tri à bulles » est un algorithme de tri relativement naïf. Son principe est le suivant. On se donne une liste `x`. On peut déterminer si `x` est triée en utilisant une fonction vue au TP 1 [1]. Si `x` n'est pas triée, alors il existe un indice i de `x` tel que $\texttt{x}[i] > \texttt{x}[i+1]$. Si on échange les éléments d'indices i et $i+1$ de `x`, alors on obtient une liste qui est « mieux triée ». On peut à nouveau tester si cette liste « mieux triée » est réellement triée, et si ce n'est pas le cas, répéter le même processus.

Question 1. *Écrire une fonction* Python *qui prend argument une liste d'entiers* `x` *et qui renvoie une liste triée par ordre croissant en utilisant le principe ci-dessus. On pourra utiliser une fonction qui teste si une liste est triée (c.f. TP 1).*

Il existe un moyen d'améliorer le principe ci-dessus. Notons que chaque appel à la fonction qui teste si `x` est triée coûte de l'ordre de `len(x)` opérations. De plus, lorsque la liste `x` n'est pas triée, la recherche d'un indice i tel que $\texttt{x}[i] > \texttt{x}[i+1]$ peut coûter de l'ordre de `len(x)` opérations. Au lieu d'utiliser explicitement une fonction qui teste si `x` est triée, on peut en un seul parcours à la fois tester si `x` est triée, et effectuer les permutations des $\texttt{x}[i]$ et $\texttt{x}[i+1]$ tels que $\texttt{x}[i] > \texttt{x}[i+1]$.

D'autre part, l'ordre de parcours de `x` peut permettre d'économiser un certain nombre d'opérations.

Question 2. *Considérons la fonction suivante :*

```
def f(x):
  for i in range(1,len(x)) :
    if x[i] < x[i-1] :
      x[i],x[i-1] = x[i-1],x[i]
  return None
```

Après l'exécution de `f` *sur une liste* `y`*, que contient le dernier élément de* `y` *?*

Question 3. *Écrire une fonction* Python *qui prend argument une liste d'entiers* `x` *et qui renvoie une liste triée par ordre croissant en utilisant le principe ci-dessus,* **et qui n'utilise pas de fonction testant si une liste est triée**.

1. https://perso.ens-lyon.fr/colin.riba/teaching/cpes/tp/tp01.pdf

1.2 Tri par sélection (*selection sort*)

L'idée du tri par sélection est la suivante. Soit `v` un élément maximal d'une liste `x`. Alors la liste obtenue en échangeant `v` avec le dernier élément de `x` est « mieux triée » que `x`. On peut répéter le même procédé sur cette nouvelle liste. Mais attention, `v` étant l'élément maximal de `x`, une fois placé à la fin, on sait qu'il est à la bonne position. On ne doit donc pas chercher l'élément maximal de la nouvelle liste, mais l'élément maximal de la nouvelle liste **privée de son dernier élément**. Il est utile de se donner la variante suivante de la recherche d'élément maximal.

Question 4. *Écrire une fonction qui prend en arguments une liste* `x` *et un entier* `n` *et qui renvoie l'indice du plus grand élément de la liste* $[\texttt{x}[0]\ ,\ \ldots\ ,\ \texttt{x}[m-1]]$ *où* $m = \min(\texttt{n}, \texttt{len}(\texttt{x}))$.

Considérons une application de l'idée ci-dessus sur la liste

$$\texttt{x} = [\ 9\ ,\ 4\ ,\ 7\ ,\ 2\ ,\ 0\ ]$$

Notons i_max l'indice du plus grand élément de la portion de liste considérée. On a, au départ :

$$\begin{array}{l}\texttt{x} = [\ 9\ ,\ 4\ ,\ 7\ ,\ 2\ ,\ 0\ ] \\ \qquad\quad \uparrow \\ \qquad i\_max\end{array}$$

ce qui nous amène à la liste

$$\begin{array}{l}\texttt{x} = [\ 0\ ,\ 4\ ,\ 7\ ,\ 2\ ,\ 9\ ] \\ \qquad\qquad\quad \uparrow \\ \qquad\qquad i\_max\end{array}$$

Dans cette nouvelle liste, i_max est l'indice de l'élément maximal de la liste $[0, 4, 7, 2]$, c'est-à-dire de la partie de la liste `x` strictement à gauche de la flèche $\downarrow$.

En itérant, on obtient :

$$\begin{array}{l}[\ 0\ ,\ 4\ ,\ 2\ ,\ 7\ ,\ 9\ ] \\ \quad\ \uparrow \\ \quad i\_max\end{array}$$

L'étape suivante nous donne

$$\begin{array}{l}[\ 0\ ,\ 2\ ,\ 4\ ,\ 7\ ,\ 9\ ] \\ \quad\ \uparrow \\ \quad i\_max\end{array}$$

et enfin

$$\begin{array}{l}[\ 0\ ,\ 2\ ,\ 4\ ,\ 7\ ,\ 9\ ] \\ \ \uparrow \\ i\_max\end{array}$$

Question 5. *Écrire une fonction* Python *qui implémente un algorithme de tri selon le principe ci-dessus.*

1.3 Tri par insertion (*insertion sort*)

L'idée du tri par insertion est de parcourir la liste à trier, et de la modifier au fur et à mesure, de telle sorte que la partie à gauche de la position courante soit toujours triée. Pour cela, en parcourant la liste, on insère l'élément courant à sa bonne place dans la partie gauche de la liste.

Cette insertion est réalisée au moyen d'une boucle. Supposons que i soit la position courante dans `x` et que `v` soit l'élément de `x` à la position i. Pour $j < i$, si `v` $<$ `x`$[j]$ alors `v` doit être placé à gauche de la position d'indice j, et on continue à parcourir `x` vers la gauche, jusqu'à arriver soit à $j = 0$, soit à un j tel que `x`$[j] \leq$ `v`. Dans les deux cas, on a trouvé la bonne place pour `v`.

Question* 6. *Écrire une fonction* Python *qui implémente un algorithme de tri selon le principe ci-dessus.*

2 Calculs de coûts

Question 7.** *Pour chacunes des Questions 3 (tri à bulles), 5 (tri par sélection) et 6 (tri par insertion), donner, en fonction de la longueur de* `x`*, le nombre d'opérations effectuées dans chacuns des cas suivants.* [2]

(1) la liste `x` *est déjà triée,*

(2) la liste `x` *est triée dans l'ordre inverse (i.e.* `rev(x)` *est triée).*

2. On se basera sur les coûts des primitives Python donnés sur la page https://wiki.python.org/moin/TimeComplexity.