Informatique PCSI
Prérequis TP 8 : algorithmes de tri

Algorithmes de tri quadratiques

Tri par sélection

Le principe

On dispose de n données. On cherche la plus petite donnée et on la place en première position, puis on cherche la plus petite donnée parmi les données restantes et on la place en deuxième position, et ainsi de suite. Cet algorithme est souvent utilisé pour trier à la main des objets, comme des cartes ou des livres.

Si les données sont les éléments d'une liste `liste`, l'algorithme consiste donc à faire varier un indice `i` de 0 à $n-2$. Pour chaque valeur de `i`, on cherche dans la tranche `liste[i:n]` le plus petit élément et on l'échange avec `liste[i]`. On répète la recherche d'un minimum.

Écrire un algorithme du tri sélection consiste à insérer dans une boucle, où `i` varie de 0 à $n-2$, un algorithme de recherche du plus petit élément dans une liste, et pour chaque valeur de `i` à faire l'échange de `liste[i]` avec `liste[i_mini]`.

La donnée en entrée est une liste de n éléments. Il n'y a pas de résultat renvoyé en sortie, la liste est modifiée en place. On dit que *le tri sélection est un tri en place*.

```
def tri_selection(liste):
    for i in range(len(liste)-1):
        i_mini = i # indice du minimum
        mini = liste[i]
        for j in range(i+1, len(liste)):
            if liste[j] < mini:
                i_mini = j
                mini = liste[j]
        liste[i], liste[i_mini] = liste[i_mini], liste[i]
```

Exemple avec la liste `[4, 4, 3, 2, 6, 5]` et les éléments échangés.

Pour `i` égal à 0 :	`[2, 4, 3, 4, 6, 5]`	après échange de 2 et 4.
Pour `i` égal à 1 :	`[2, 3, 4, 4, 6, 5]`	après échange de 3 et 4.
Pour `i` égal à 2 :	`[2, 3, 4, 4, 6, 5]`	après aucun échange.
Pour `i` égal à 3 :	`[2, 3, 4, 4, 6, 5]`	après aucun échange.
Pour `i` égal à 4 :	`[2, 3, 4, 4, 5, 6]`	après échange de 5 et 6.

On remarque que le 4 qui était en début de liste se retrouve après le 4 qui était en deuxième position. On dit que *le tri sélection n'est pas stable*.

Pour utiliser cette fonction il suffit d'écrire l'instruction `tri_selection(liste)`.

Si nous ne voulons pas modifier la liste passée en paramètre il faut en faire une copie et ensuite appliquer l'algorithme de tri à cette nouvelle liste qui est renvoyée à la fin.

Validité de l'algorithme

Il est intéressant de noter qu'après k passages dans la boucle externe, les k premiers éléments de la nouvelle liste sont à leur place définitive.

- **Terminaison**
  La terminaison est simple à prouver. Nous avons deux boucles `for` imbriquées et le nombre de passages dans ces deux boucles est parfaitement déterminé et il est évidemment fini.

- **Correction**
  Nous prouvons la correction en utilisant un invariant de boucle : "pour chaque `i`, la liste est une permutation de la liste initiale, la liste `liste[0:i+1]` est triée et tous les éléments de la liste `liste[i+1:n]` sont supérieurs à tous les éléments de la liste `liste[0:i+1]` ".
  Pour chaque valeur de `i`, au plus une permutation de deux éléments distincts a lieu et elle a lieu seulement si `liste[i]` n'est pas le minimum de `liste[i:n]`. C'est pourquoi après chaque passage dans la boucle externe, la nouvelle liste est une permutation de la liste initiale.
  Après le premier passage dans la boucle, pour `i` égal à 0, la liste `liste[0:1]` ne contient qu'un élément, le minimum de la liste, qui est inférieur à tous les éléments de la liste. La propriété est donc vraie pour `i` égal à 0.
  Si après un passage pour `i` égal à un k quelconque, la liste `liste[0:k+1]` est triée et tous les éléments de `liste[k+1:n]` sont supérieurs à tous les éléments de `liste[0:k+1]`, alors au passage suivant le minimum de la liste `liste[k+1:n]` est placé en position d'indice `k+1`. Ce minimum est supérieur à tous les éléments de la liste `liste[0:k+1]` et inférieur à tous les éléments de la liste `liste[k+2:n]`. La propriété est donc vraie pour `i` égal à $k+1$.

  La propriété est donc encore vraie après le dernier passage, pour `i` égal à $n-2$. Donc la liste `liste[0:n-1]` est triée et l'élément d'indice `n-1`, le dernier de la liste, est supérieur à tous les éléments de la liste `liste[0:n-1]`. Donc la liste `liste[0:n]`, soit toute la liste, est triée.

Coût de l'algorithme

Quels que soient les éléments d'une liste de longueur `n`, pour chaque valeur de `i`, `j` prend les valeurs de `i+1` à `n-1`, soit `n-i-1` valeurs. Et pour chaque valeur de `j`, une unique comparaison est effectuée. Donc, pour chaque valeur de `i`, nous avons exactement `n-i-1` comparaisons.

Au total, nous obtenons : `(n-1)+(n-2)+...+2+1` comparaisons, soit `n(n-1)/2` comparaisons. Le coût est donc de l'ordre de n^2 quelle que soit la liste de longueur `n`, même si elle est déjà triée. Cela signifie que le tri par sélection n'est pas très efficace. Il est cependant simple à programmer et utile dans le cas de listes ne comptant pas plus de 10^4 éléments.

Tri par insertion

Le principe

On dispose de n données. À chaque étape, on suppose que les k premières données sont triées et on insère une donnée supplémentaire à la bonne place parmi ces k données.

Si les données sont les éléments d'une liste, l'algorithme consiste donc à faire varier un indice `i` de 0 à $n-2$. Pour chaque valeur de `i`, on cherche dans la liste `liste[0:i+1]` à quelle place doit être inséré l'élément `liste[i+1]` qu'on appelle la clé. Pour cela on compare la clé successivement aux données précédentes, en commençant par la donnée d'indice `i` puis en remontant dans la liste jusqu'à trouver la bonne place, c'est-à-dire entre deux données successives, l'une étant plus petite et l'autre plus grande que la clé. Si la clé est plus petite que toutes les données précédentes, elle se place en premier. Pour ce faire, on décale d'une place vers la droite les données plus grandes que la clé après chaque comparaison.

La donnée en entrée est une liste de n éléments. Il n'y a pas de résultat renvoyé en sortie, la liste est modifiée en place. On dit que *le tri insertion est un tri en place*.

```
def tri_insertion(liste):
    for i in range(len(liste)-1):
        k = i + 1 # indice de la cle
        cle = liste[k]
        while k > 0 and cle < liste[k-1]:
            liste[k] = liste[k -1]
            k = k - 1
        liste[k] = cle
```

Exemple avec la liste `[4, 4, 3, 2, 6, 5]` et les clés successives.

Pour `i` égal à 0 avec la clé 4 : `[4, 4, 3, 2, 6, 5]`.
Pour `i` égal à 1 avec la clé 3 : `[3, 4, 4, 2, 6, 5]`.
Pour `i` égal à 2 avec la clé 2 : `[2, 3, 4, 4, 6, 5]`.
Pour `i` égal à 3 avec la clé 6 : `[2, 3, 4, 4, 6, 5]`.
Pour `i` égal à 4 avec la clé 5 : `[2, 3, 4, 4, 5, 6]`.

Un examen approfondi de l'algorithme montre que le *tri insertion est stable*. Deux éléments de même valeur placés dans un certain ordre avant le tri restent dans le même ordre après le tri.

Pour utiliser cette fonction il suffit d'écrire l'instruction `tri_insertion(liste)`.

Si nous ne voulons pas modifier la liste passée en paramètre, il faut en faire une copie, trier cette nouvelle liste et ensuite la renvoyer.

On peut noter ici qu'après k passages dans la boucle, les k premiers éléments de la liste sont triés. Mais ils ne sont pas, à priori, à leur place définitive.

Validité de l'algorithme

- **Terminaison**
  La boucle externe est une boucle `for` donc le nombre de passages est déterminé et fini. La boucle interne est une boucle `while`. Les valeurs prises par le variant `k` constituent une suite d'entiers strictement décroissante incluse dans la suite des entiers de `i+1` à 0. Il y a donc, pour chaque `i`, au plus `i+1` passages dans la boucle `while`.

- **Correction**
  Pour prouver la correction nous utilisons un invariant de boucle : "pour chaque `i`, la liste est une permutation de la liste initiale et la liste `liste[0:i+2]` est triée".
  Le principe de l'insertion assure que pour chaque `i`, la liste modifiée est une permutation de la liste initiale.
  Après le premier passage dans la boucle, pour `i` égal à 0, l'élément `liste[0]` et la première clé, d'indice 1, sont rangés dans l'ordre. Donc la liste `liste[0:2]` est triée. La propriété est donc vraie pour `i` égal à 0.
  Si après un passage pour `i` égal à un k quelconque, la liste `liste[0:k+2]` est triée, alors au passage suivant l'élément `liste[k+2]` est inséré à la bonne place parmi les éléments de la liste `liste[0:k+2]` ou reste à sa place. Donc la liste `liste[0:k+3]` est triée. La propriété est donc vraie pour `i` égal à $k+1$.

  La propriété est donc encore vraie après le dernier passage, pour `i` égal à $n-2$. À ce moment la liste `liste[0:n]`, c'est-à-dire la liste `liste`, est triée.

Coût de l'algorithme

Nous avons deux boucles imbriquées. Pour une liste de longueur `n`, le nombre de comparaisons peut être différent suivant la liste.

Si la liste est déjà triée, pour chaque valeur de `i`, `k` prend la valeur de `i+1` et il y a une seule comparaison, le test `cle < liste[k-1]`. La variable `i` prenant `n-1` valeurs, cela nous fait un total de `n-1` comparaisons. Le coût de l'algorithme est donc de l'ordre de `n`.

Si par contre les éléments de la liste sont rangés dans l'ordre décroissant, alors pour chaque valeur de `i`, `k` prend les valeurs de `i+1` à 1 soit `i+1` valeurs et donc `i+1` comparaisons.

Au total, nous avons donc : $1+2+\ldots+(n-2)+(n-1)$ comparaisons. Un calcul mathématique nous donne `n(n-1)/2` comparaisons. Le coût est donc de l'ordre de n^2 comparaisons.

On peut montrer qu'en moyenne, le coût est de l'ordre de n^2 comparaisons, comme pour le tri par sélection. Mais le tri par insertion est très intéressant si la liste est "presque triée".