1 Préliminaire : activité de tri « vivante »

Voir activité info débranchée : découvrir les tris.

2 Tri par sélection

L'intérêt d'avoir une collection triée pour effectuer des recherches dans cette collection semble assez évident (ex : chap. P7-4 recherche dichotomique). Il existe de nombreux algorithmes de tri (plus ou moins efficaces) et nous allons en étudier deux cette année (*pas les plus efficaces, mais instructifs*).

2.1 Présentation de l'algorithme

On désire trier un tableau par ordre croissant.

> On commence par rechercher l'élément de plus petite valeur (avec un parcours séquentiel du tableau) et on permute cet élément avec celui en 1ère position du tableau. Par la suite, on ne modifiera plus l'emplacement cet élément.
> On reprend alors un parcours séquentiel du reste du tableau à la recherche de l'élément de plus petite valeur parmi ceux non triés et on le permute avec le 2ème élément du tableau. Et ainsi de suite jusqu'à sélectionner et positionner chaque élément dans le tableau à son bon emplacement.
> **Au cours de la progression de l'algorithme, on aura donc toujours la partie gauche du tableau qui est triée, et la partie droite qui reste à trier.**

Voici l'algorithme écrit en pseudo-code :

Fonction TriSelection
pour *i* **de** *1* **à** *taille(t)-1* **faire**
  imini ← *i*
  pour *j* **de** *i+1* **à** *taille(t)* **faire**
    si *t[j]* < *t[imini]* **alors**
      imini ← j
    fin si
  fin pour
  Permuter *t[i]* et *t[j]*
fin pour

On constate que cet algorithme contient donc 2 boucles « *Pour* » imbriquées.

Si on définit une fonction annexe de recherche de minimum dans un tableau (cf chap. P7-1), on peut un peu alléger la présentation de l'algorithme, et le rendre plus modulaire.
Voici une version Python qui correspond à cette approche.

```
def mini(tab, i_deb):
  """ Recherche l'indice de l'élément minimum dans le tableau tab
   entre les indices i_deb (inclus) et la fin du tableau."""
  imini = i_deb
  for i in range(i_deb, len(tab)):
    if tab[i] < tab[imini]:
      imini = i
  return imini
```

```
def tri_selection(tab):
  for i in range(len(tab)-1):
    imini = mini(tab, i)
    tab[imini], tab[i] = tab[i], tab[imini]
```

2.2 Preuve de terminaison

(Revoir chap. P6-1§6 : terminaison / variant de boucle).
La preuve de la terminaison de l'algorithme est aisée puisqu'il fait intervenir seulement des boucles for bornées.
On note `n` la taille de `tab`.
Les deux **variants de boucles** sont simplement `((n-1)-1-i)` pour la boucle for principale et `(n-1-j)` pour la boucle for interne.

2.3 Preuve de correction

Un algorithme est **correct** s'il fait ce qu'on attend de lui ! L'algorithme de tri est donc correct s'il trie bien le tableau.

> Pour prouver la correction, il faut pouvoir exhiber un **invariant de boucle**, càd une **propriété en rapport avec le but de l'algorithme qui est vraie avant et après chaque itération de boucle**.

Pour l'algorithme de tri par sélection, l'invariant de boucle (pour la boucle *for* principale) est :
Le tableau est trié jusqu'à l'indice i (inclus), et tous les autres éléments sont plus grands.

1. On vérifie aisément que cette propriété est vraie au début pour $i = 0$ car à la fin de la boucle *for* principale, on a placé le plus petit élément du tableau à cet indice 0. Le tableau est donc trié pour cet unique élément et tous les autres éléments sont plus grands.
2. Supposons maintenant que l'invariant est vrai pour i quelconque et montrons qu'il reste vrai à la fin d'une nouvelle itération.
   Au cours d'une itération de la boucle *for* principale, la boucle *for* secondaire permet de trouver le plus petit élément restant non trié, et ensuite cet élément est placé à l'indice i.
   Le tableau devient donc bien trié jusqu'à l'indice i avec les autres éléments restants plus grands.
3. À la fin de la boucle for principale, $i = len(t) - 2 = n - 2$, et donc le tableau devient trié jusqu'à cet indice $n - 2$, càd l'avant dernier indice du tableau ; en outre le seul dernier élément est plus grand, et donc le tableau est bien trié dans son intégralité ! (preuve de correction terminée).

On peut représenter cela à l'aide du tableau suivant :

avant itération		**après itération**		**variant**
i	**invariant**	**i**	**invariant**	**(n-2) - i**
non défini	aucun élément trié	0	t trié jusqu'à indice 0 plus grands de indice 1 à n-1	n-2
0	t trié jusqu'à indice 0 plus grands de indice 1 à n-1	1	t trié jusqu'à indice 1 plus grands de indice 2 à n-1	n-3
... cas général ...				
i-1	t trié jusqu'à indice i-1 plus grands de indice i à n-1	i	t trié jusqu'à indice i plus grands de indice i+1 à n-1	(n-2) - i
... fin ...				
n-3	t trié jusqu'à indice n-3 plus grands de indice n-2 à n-1	n-2	t trié jusqu'à indice n-2 plus grands de indice n-1 à n-1	(n-2) - (n-2) = 0

Lorsque le variant s'annule, la boucle se termine et le tableau est trié pour les indices de 0 à n-2 et l'unique dernier élément d'indice n-1 est plus grand, donc le tableau est bien trié dans son intégralité.

2.4 Complexité

Dans cet algorithme pour un tableau de taille n, dans la boucle for principale, i varie de 0 à $(n-2)$, et pour chacune de ces valeurs de i, dans la boucle for secondaire, j varie de $(i+1)$ à $(n-1)$ [soit $(n-1-i)$ valeurs] à la recherche du minimum en effectuant des comparaisons.
Il y a donc $(n-1)$ comparaisons pour $i=0$ puis de moins en moins jusqu'à 1 seule comparaison pour $i=n-2$.
Le nombre total de comparaisons est donc : $(n-1)+(n-2)+...+1=\frac{n(n-1)}{2}=\frac{n^2-n}{2}$.

> La complexité d'un algorithme décrit l'évolution du temps d'exécution en fonction de la taille asymptotique des données (càd pour des données de grande taille).

Pour un tableau de **grande** taille n, n devient négligeable devant n^2 et le nombre de comparaisons (donc le temps d'exécution) progresse donc comme n^2.

> On dit alors que le **coût** de cet algorithme est **quadratique** (varie comme n^2).
> On écrit parfois que l'algorithme est en $O(n^2)$.

3 Tri par insertion

3.1 Présentation de l'algorithme

On cherche un autre algorithme pour trier un tableau de n éléments par ordre croissant de valeurs.

> La démarche repose sur le principe suivant : on suppose que les i premiers éléments sont déjà triés. On cherche alors à insérer le 1er élément non trié (le i+unième) à sa place parmi ceux déjà triés. Pour cela on décale d'un cran vers la droite tous les éléments triés dont la valeur est plus grande que celle de l'élément à insérer. Et finalement on place cet élément à insérer dans la case qui se retrouve vide.
> Ensuite on passe à l'insertion du 1er nouvel élément non trié, jusqu'à avoir ainsi traité tous les éléments du tableau.

Cette procédure s'applique donc en commençant par insérer à sa bonne place le 2ème élément du tableau (le premier est à sa place par défaut) puis en progressant pour tous les éléments du tableau en insérant les éléments à leur bonne place un par un au fur et à mesure.

Remarque : c'est souvent la manière dont les gens trient un jeu de cartes.

Voici l'algorithme écrit en pseudo-code :

```
Fonction TriInsertion
pour i de 2 à taille(t) faire
    element ← t[i]
    j ← i-1
    tant que t[j] > element ET j >= 1 faire
        t[j + 1] ← t[j]
        j ← j - 1
    fin tq
    t[j+1] = element
fin pour
```

Comme pour l'algorithme de tri par sélection, on peut rendre cet algorithme plus modulaire en définissant une fonction annexe qui gère l'insertion au bon endroit.
Voici ce qu'on pourrait coder en Python :

```
def insertion(tab, imax):
    """ Insère l'element tab[imax] dans la partie du tableau qui est triée jusqu'à
    ↪  (imax-1).
    A l'issu de l'insertion, le tableau doit être trié jusqu'à imax. """
    element = tab[imax]
    i = imax - 1
    while i >= 0 and tab[i] > element:
        tab[i+1] = tab[i]
        i = i - 1
    tab[i+1] = element

def tri_insertion(tab):
   for i in range(1, len(t)):
      insertion(tab, i)
```

Remarque : on verra en TP comment on peut s'assurer de la correction de ces fonctions avec des instructions `assert`.

3.2 Preuve de la terminaison

1. Comme pour l'algorithme de tri pas sélection, la boucle *for* termine toujours : son variant de boucle est simplement `(n-1-i)` (`n` est la taille du tableau).
2. La terminaison d'une boucle while est toujours plus délicate et importante à vérifier.
   On ne peut pas exhiber de variant de boucle en s'appuyant sur le test `tab[i] > element` ; en revanche la condition `i >= 0` permet de dégager le simple variant de boucle `i` : c'est bien une grandeur positive qui décroît à chaque itération, assurant ainsi la terminaison de la boucle while.

3.3 Preuve de correction

Pour prouver la correction de cet algorithme, nous allons à nouveau nous appuyer sur un invariant de boucle.
L'invariant de boucle est ici (pour la boucle *for*) : **Le tableau est trié jusqu'à l'indice i (inclus).**

1. On vérifie que cette propriété est vraie au début pour $i = 1$ car le tableau qui était forcément trié jusqu'à l'indice 0 (valeur unique, triée d'office) devient trié jusqu'à l'indice en cours de travail, càd i = 1. (deux éléments sont alors triés).
2. Supposons maintenant que l'invariant est vrai pour i quelconque et montrons qu'il reste vrai à la fin d'une itération.
   Avant d'aborder l'itération i, le tableau est trié jusqu'à l'indice (i-1).
   Au cours d'une itération de la boucle *for*, toutes les valeurs du tableau sont décalées d'un cran vers la droite jusqu'à l'indice i et on insère la valeur à trier au bon endroit (à un indice entre 0 et i inclus). Donc à la fin de l'itération, le tableau est devenu trié jusqu'à l'indice i inclus.
3. À l'abord de la dernière itération de la boucle *for*, `i = len(t)-1`, et donc le tableau est trié jusqu'à l'indice `len(t)-2` ; au cours de cette dernière itération le tableau est alors trié jusqu'à l'indice `len(t)-1`, càd le dernier indice du tableau, et donc le tableau est bien trié dans son intégralité !

On peut représenter cela à l'aide du tableau suivant :

avant itération		**après itération**		**variant**
i	**invariant**	**i**	**invariant**	**(n-1) - i**
non défini	t trié jusqu'à indice 0	1	t trié jusqu'à indice 1	n-2
1	t trié jusqu'à indice 1	2	t trié jusqu'à indice 2	n-3
... cas général ...				
i-1	t trié jusqu'à indice i-1	i	t trié jusqu'à indice i	(n-1) - i
... fin ...				
n-2	t trié jusqu'à indice n-2	n-1	t trié jusqu'à indice n-1	(n-1) - (n-1) = 0

Lorsque le variant s'annule, la boucle se termine et le tableau est trié pour les indices de 0 à n-1 (du 1er au dernier élément), donc le tableau est bien trié dans son intégralité.

3.4 Complexité

Le coût de cet algorithme dépend de l'état initial du tableau : s'il est trié par ordre décroissant, c'est le **cas le pire** et il faut à chaque itération décaler le nouvel élément (d'indice i) jusqu'à la position d'indice 0 et donc faire i décalages des autres éléments.

Il peut donc être nécessaire d'effectuer $1 + 2 + ... + (n-1) = \dfrac{n(n-1)}{2}$ opérations pour un tableau de taille n.

Comme pour l'algorithme de tri par sélection, pour un tableau de **grande** taille n, ce nombre est de l'ordre de n^2.

> Le coût de l'algorithme de tri par insertion est donc également **quadratique** (varie comme n^2), dans le pire des cas.

Remarque : Lorsqu'on s'intéresse la complexité d'un algorithme on cherche toujours à envisager le cas le pire. Mais on peut aussi calculer des coûts en moyenne.

4 D'autres tris

La complexité quadratique qu'on obtient pour les deux algorithmes présentés (sélection et insertion) est prohibitive pour des tableaux de trop grande taille.

Heureusement, il existe d'autres algorithmes de tris beaucoup plus performants avec un coût moindre.

Les meilleurs algorithmes ont une complexité en $O(n\log(n))$.

Ils sont par exemple implémentés dans les fonction de tri intégrées à Python.

Compléments d'animation :

Toptal : sorting algorithms animations

Interstices : algorithmes de tri