Algorithmes de tri par insertion et de tri par sélection [RÉSUMÉ]

Dernière mise à jour le : 03/01/2024

> **Attention** : Ceci n'est qu'un résumé de cours et ne remplace en rien l'étude complète des deux algorithmes qui a été faite dans l'activité et dans les exercices.

On veut trier dans l'ordre croissant un tableau `T` d'entiers non vide.

Spécification d'un algorithme de tri

- **Entrée/Sortie** : un tableau `T` de taille n constitué d'entiers (les indices variant de `0` à `n-1`)
- **Rôle** : trier `T` par ordre croissant
- **Précondition** : `T` non vide
- **Postcondition** : `T` est trié c'est-à-dire que chaque élément de `T` est inférieur ou égal à l'élément suivant : pour tout entier i compris entre 0 et $n-2$, on a $T[i] \leq T[i+1]$

Tri par insertion

Algorithme

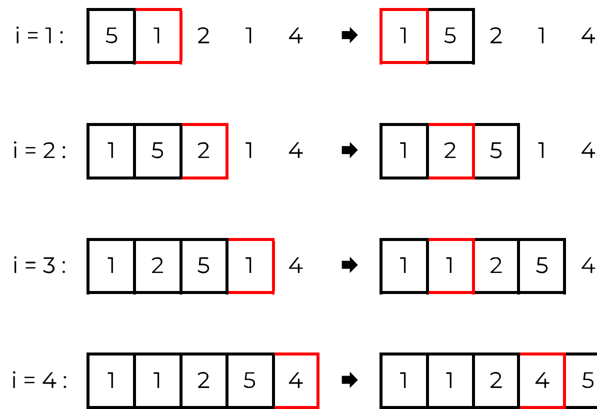
> **Idée de l'algorithme de tri par insertion** :
>
> - Prendre le deuxième élément du tableau et l'insérer à sa place parmi les éléments qui le précèdent
> - Prendre le troisième élément du tableau et l'insérer à sa place parmi les éléments qui le précèdent
> - Continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

> C'est l'algorithme que l'on applique pour trier des cartes que l'on aurait en main.

Algorithme

```
pour i de 1 à n-1 faire
    x ← T[i]
    j ← i
    tant que j > 0 et x < T[j-1] faire
        T[j] ← T[j-1]
        j ← j - 1
    fin tant que
    T[j] ← x
fin pour
```

Exemple : On veut trier le tableau `T = [5, 1, 2, 1, 4]`.



En Python

```python
def tri_insertion(T):
    for i in range(1, len(T)):
        x = T[i]
        j = i
        while j > 0 and x < T[j-1]:
            T[j] = T[j-1]
            j = j - 1
        T[j] = x
```

```python
>>> t1 = [5, 1, 2, 1, 4]
>>> tri_insertion(t1)
>>> t1
[1, 1, 2, 4, 5]
```

Coût de l'algorithme

Le coût en temps de l'algorithme définit son efficacité. Déterminer le coût d'un algorithme revient à évaluer le *nombre d'opérations élémentaires* dans le **pire cas**.

Pour simplifier l'étude, on compte uniquement les comparaisons entre deux éléments du tableau (correspond à la ligne `x < T[j-1]`).

> Le pire cas pour le tri par insertion est un tableau trié dans l'ordre décroissant car il faut aller insérer chaque élément en première position : cela implique de le comparer à tous les éléments qui le précèdent.

Pour un tableau de taille n trié par ordre décroissante (pire cas), on a :

- Au premier passage dans la boucle : il y a 1 comparaison à faire (`T[1]` est comparé à `T[0]`) ;
- Au deuxième passage : il y a 2 comparaisons à faire (`T[2]` est comparé successivement à `T[1]` et `T[0]`) ;
- Ainsi de suite ;
- Au dernier passage (`i` = n−1) : il y a n−1 comparaisons à faire (`T[n-1]` est comparé successivement à `T[n-2]`, `T[n-3]`, ..., `T[0]`).

Bilan : Il y a donc en tout $1+2+3+\cdots+(n-2)+(n-1)$ comparaisons dans le pire cas.

Or, $1+2+3+\cdots+(n-2)+(n-1)=\dfrac{n(n-1)}{2}=\dfrac{n^2}{2}-\dfrac{n}{2}$, on peut alors dire que le coût de l'algorithme est **de l'ordre de n^2** (on prend la plus grande puissance de n) et on parle de **coût quadratique**.

Le tri par insertion a donc un *coût quadratique* dans le pire cas. Cela signifie que *si on multiplie le taille du tableau à trier par k alors le temps pour le tri est multiplié par k^2* (cette propriété est illustrée expérimentalement dans le dernier paragraphe).

Tri par sélection

Algorithme

Idée de l'algorithme de tri par sélection :

- Rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0 ;
- Rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 1 ;
- Continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

Cela revient donc à faire à chaque étape une recherche du minimum dans la zone du tableau de droite pas encore triée.

Algorithme

```
pour i de 0 à n-2 faire
    ind_min ← i
    pour j de i+1 à n-1 faire
        si T[j] < T[ind_min] alors
            ind_min ← j
        fin si
    fin pour
    echange(T, i, ind_min)   # échange T[i] et T[ind_min]
fin pour
```

Exemple : On veut trier le tableau `T = [5, 1, 2, 1, 4]`.

i = 1 : **5** 1 2 1 4 ➡ 1 **5** 2 1 4

i = 2 : 1 **5** 2 1 4 ➡ 1 1 2 **5** 4

i = 3 : 1 1 **2** 5 4 ➡ 1 1 **2** 5 4

i = 4 : 1 1 2 **5** 4 ➡ 1 1 2 4 **5**

En Python

```python
def echange(T, i, j):
    """échange T[i] et T[j] dans le tableau T"""
    temp = T[i]
    T[i] = T[j]
    T[j] = temp

def tri_selection(T):
    for i in range(len(T)-1):
        ind_min = i
        for j in range(i+1, len(T)):
            if T[j] < T[ind_min]:
                ind_min = j
        echange(T, i, ind_min)
```

```
>>> t1 = [5, 1, 2, 1, 4]
>>> tri_selection(t1)
>>> t1
[1, 1, 2, 4, 5]
```

Coût de l'algorithme

Comme précédemment, on n'étudie que le nombre de comparaisons (ligne `T[j] < T[ind_min]`) dans le pire cas.

Quel que soit le tableau de départ, il y a toujours le même nombre de comparaisons à faire, car on sait le nombre de tours de boucle à faire pour qu'il s'agit de deux boucles `for`. Il n'y a donc pas de pire ou de meilleur cas dans le cas du tri par insertion (ou tous les cas sont des pires cas).

Plus précisément, pour un tableau `T` de taille n :

- Au 1er passage dans la boucle (pour `i` = 0), il y a $n-1$ comparaisons dans la recherche du plus petit élément (recherche du minimum parmi les $n-1$ dernières valeurs) ;
- Au 2ème passage (pour `i` = 1), il y en a $n-2$ comparaisons (recherche du minimum parmi les $n-2$ dernières valeurs);
- Ainsi de suite ;
- Au dernier passage (pour `i` = $n-2$), il n'y a plus que 1 comparaison (recherche du minimum parmi les 2 dernières valeurs).

Bilan : Il y a donc en tout $(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}$ comparaisons dans le pire cas (soit le même nombre que pour le tri par insertion), donc le coût est aussi **de l'ordre de n^2**.

Le tri par sélection a donc un *coût quadratique* dans le pire cas. Cela signifie que *si on multiplie le taille du tableau à trier par k alors le temps pour le tri est multiplié par k^2* (cette propriété est illustrée expérimentalement dans le dernier paragraphe).

Les tris offerts par Python

Il existe d'autres algorithmes de tri et certains sont beaucoup plus efficaces (pour information, leur coût est de l'ordre de $n\log_2(n)$ qui est largement inférieur à n^2.)

Python fournit deux fonctions pour trier de manière efficace est un tableau selon que l'on veuille ou non modifier le tableau initial.

La fonction `sorted` prend en argument un tableau et renvoie un *nouveau* tableau, trié, contenant les mêmes éléments.

```
>>> t = [12, 5, 3, 6, 8, 10]
>>> sorted(t)
[3, 5, 6, 8, 10, 12]
>>> t
[12, 5, 3, 6, 8, 10]  # le tableau t de départ n’a pas été modifié !
```

La construction `t.sort()`, en revanche, modifie le tableau `t` pour le trier sans rien renvoyer.

```
>>> t = [12, 5, 3, 6, 8, 10]
>>> t.sort()
>>> t
[3, 5, 6, 8, 10, 12]
```

Efficacité des algorithmes

On peut mesurer expérimentalement l'efficacité des algorithmes de tri par insertion, par sélection ainsi que des tris offerts par Python.

Protocole de mesure

Les valeurs de temps mesurées ci-dessous dépendent évidemment de la puissance de l'ordinateur qui exécute les algorithmes. Mais l'idée générale et les ordres de grandeurs du coût restent identiques avec les ordinateurs classiques actuels.

On crée une fonction pour construire un tableau décroissant qui sera un pire cas.

```python
def fabrique_tableau_decroissant(taille):
    tab = [taille - k for k in range(taille)]
    return tab
```

```
>>> fabrique_tableau_decroissant(10)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Le protocole de mesure du temps est proposé ci-dessous :

```python
import time

def mesures_pire_cas(taille):
    # tri par sélection
    tab = fabrique_tableau_decroissant(taille)
    t0 = time.time()
    tri_selection(tab)
    t1 = time.time()
    tps_tri_sel = t1 - t0

    # tri par insertion
    tab = fabrique_tableau_decroissant(taille)
    t2 = time.time()
    tri_insertion(tab)
    t3 = time.time()
    tps_tri_ins = t3 - t2

    print(f"temps du tri par sélection d'un tab de taille {taille} : {tps_tri_sel}")
    print(f"temps du tri par insertion d'un tab de taille {taille} : {tps_tri_ins}")
```

Essais

```
>>> mesures_pire_cas(100)
temps du tri par sélection d'un tab de taille 100 : 0.0009999275207519531
temps du tri par insertion d'un tab de taille 100 : 0.002000093460083008
```

```
>>> mesures_pire_cas(1000)
temps du tri par sélection d'un tab de taille 1000 : 0.10900020599365234
temps du tri par insertion d'un tab de taille 1000 : 0.22899985313415527
```

> On voit qu'en multipliant la taille par 10, le temps pour le tri est multiplié par environ $10^2 = 100$.

```
>>> mesures_pire_cas(2000)
temps du tri par sélection d'un tab de taille 2000 : 0.43900012969970703
temps du tri par insertion d'un tab de taille 2000 : 0.9839999675750732
```

> On multiplié la taille par 2 par rapport à l'exemple précédent et le temps est environ multiplié par $2^2 = 4$.

Les limites des tris par insertion et par sélection

Pour illustrer le propos, on reprend la valeur précédente, à savoir qu'il faut environ 0,1 sec pour trier par sélection un tableau de taille 1000.

Le coût de l'algorithme étant quadratique, pour trier un tableau de taille 1 million (10^6) soit 1000 fois plus grand, il faut multiplier le temps par $1000^2 = 10^6$ donc il faudrait $0,1 \times 10^6 = 10^5$ secondes, soit 1 jour 3 heures 46 minutes et 20 secondes.

> Et pour un tableau de taille 10^9 il faudrait plus de 3170 ans...

On voit donc qu'un alorithme de tri qui a un coût quadratique n'est donc pas efficace. D'autant plus, que les tris offerts par Python le sont. En effet, il ne faut que quelques millisecondes pour trier un tableau de taille 10^6 :

```
def mesures_pire_cas_sort(taille):
    tab = fabrique_tableau_decroissant(taille)

    # tri du tableau avec les méthodes offertes par Python et mesure du temps
    t0 = time.time()
    tab.sort()  # ou sorted(tab)
    t1 = time.time()
    tps_tri_python = t1 - t0

    print(f"temps du tri avec les fonctions offertes par Python : {tps_tri_python}")
```

```
>>> mesures_pire_cas_sort(1_000_000)
temps du tri avec les fonctions offertes par Python : 0.009999990463256836
```

Bilan

- Le **tri par sélection** et le **tri par insertion** sont deux algorithmes de tri élémentaires, qui peuvent être utilisés pour trier des tableaux.
- Ces deux algorithmes ont un **coût quadratique** dans le pire cas (et même en moyenne), donc de l'ordre de n^2 où n est la taille du tableau : cela signifie que si on multiplie la taille du tableau par k alors le temps de calcul (dans le pire cas) est multiplié par k^2.
- Les deux restent peu efficaces dès que les tableaux contiennent plusieurs milliers d'éléments. Il existe de meilleurs algorithmes de tri, plus complexes, dont celui offert par Python avec les fonctions `sort` et `sorted`.
- Nous reviendrons sur cela lorsque nous aborderons les tris de tables de données.

Références :

- Documents ressources de l'équipe éducative du DIU EIL, Université de Nantes, Christophe JERMANN et Christophe DECLERCQ.
- Livre Numérique et Sciences Informatiques, niveau Première, T. BALABONSKI, S. CONCHON, J.-C. FILLIATRE, K. NGUYEN, éditions Ellipses

---

Germain BECKER & Sébastien POINT, Lycée Mounier, ANGERS

Voir en ligne : info-mounier.fr/premiere_nsi/algorithmique/tris-insertion-selection