Idées à la base des algorithmes de tris naïfs

Michel Billaud (michel.billaud@u-bordeaux.fr, michel.billaud@laposte.net)

30 mai 2020

Table des matières

1 Motivation

Voir d'où viennent les idées pour les algorithmes de tris naïfs qu'on présente souvent en premier aux débutants.

Dans tout ce qui suit, on suppose qu'on veut ordonner un tableau en ordre croissant.

2 Le tri par sélection

Avant d'en arriver là, on est certainement passé par un exercice classique : l'écriture d'une fonction pour trouver **la plus petite** (ou la plus grande) valeur contenue dans un tableau. (Voir code en annexe).

2.1 Début d'idée : le plus petit va au début

L'idée, c'est que si un tableau contient au départ des valeurs

```
 0    1    2    3    4    5
+----+----+----+----+----+----+  au départ
| 66 | 22 | 33 | 11 | 55 | 44 |
+----+----+----+----+----+----+
```

à la fin, la **première case** sera occupée par **la plus petite valeur** :

```
+----+----+----+----+----+----+  à la fin
| 11 | __ | __ | __ | __ | __ |
+----+----+----+----+----+----+
```

- Trouver la plus petite valeur : on sait faire.
- Et ensuite, il n'y aura plus qu'à faire pareil pour le second (le plus petit du reste), le troisième etc.

2.2 Échange pour ne pas perdre les valeurs

Un petit souci : ne pas perdre la valeur 66 qui occupait la première case.

Pour cela, une idée simple est de la placer dans la case qui a été libérée par la valeur 11. On aura donc réalisé un **échange**, dans le tableau, entre la position 0 et la position où se trouve le minimum :

```
 0    1    2    3    4    5
+----+----+----+----+----+----+  au départ
| 66 | 22 | 33 | 11 | 55 | 44 |
+----+----+----+----+----+----+

+----+----+----+----+----+----+  après échange des positions 0 et 3
| 11 | 22 | 33 | 66 | 55 | 44 |
+----+----+----+----+----+----+
```

2.3 Mais où est le minimum ?

Qui plus est, il faut adapter la fonction de recherche, parce qu'on a besoin de sa **position** dans le tableau.

```
Pour placer le premier élément :
- chercher la position p de la plus petite valeur
- échanger les contenus des positions 0 et p
```

2.4 Généraliser aux suivants

L'idée c'est de faire la même chose pour le second élément. Mais cette fois-ci, on cherchera la plus petite valeur **à partir de la position 1**, et non 0. Pour le troisième, on partira de la position 2. Etc.

Il faut donc généraliser la fonction de recherche pour qu'elle commence à une certaine **position de départ**, qui n'est pas toujours 0.

À faire : modifier le code de la fonction pour qu'elle tienne compte de ce paramètre supplémentaire :

```
int position_minimum(int depart, int taille, int tableau[])
{
    ...
}
```

Les deux nombres `depart`, `taille` fournissent un *intervalle de recherche* dans le tableau. C'est un intervalle "semi-ouvert" qui contient `depart` mais pas `taille`.

2.5 Algorithme général

L'algorithme général consiste à placer successivement les bonnes valeurs aux positions 0, 1, jusqu'à `taille-1` :

```
pour i de 0 à taille-1 {
   placer la bonne valeur à la position i
}
```

Détaillons : pour placer la bonne valeur, il faut trouver sa position, puis échanger

```
pour i de 0 à taille-1 {
   trouver la position p de la plus petite valeur
           de l'intervalle i .. taille (non compris)
   échanger les contenus des positions i et p
}
```

Une remarque comme on procède par échanges, si on a placé les taille-1 premiers éléments, le dernier est arrivé à sa place. On peut donc s'arrêter à l'avant dernier.

3 Le tri par échanges successifs

L'idée est très voisine du tri par sélection :

- on veut remplir la première case avec la plus petit valeur ;

- pour cela, on parcourt le tableau, et si on trouve une valeur plus petite que ce qu'on a dans la première case, on fait un échange.
- donc, à la fin, la première case contiendra la plus petite valeur.

```
pour i de 1 taille - 1 {
   si tableau[i] < tableau[0] {
      échanger les contenus des positions 0 et i ;
   }
}
```

Ensuite, il faut faire la même chose pour les positions 1, 2, etc.

Comme pour le précédent, la boucle extérieure met définitivement les valeurs à leur place (on ne touche plus jamais à la position 0 après le premier tour).

4 Le tri par insertion

L'idée est différente : on insère les éléments un à un, dans une **séquence ordonnée** située au début du tableau.

4.1 L'algorithme général

On place successivement les éléments :

```
pour i de 0 à taille - 1 {
   insérer l'élément qui est en position i
}
```

Illustration :

```
 0    1    2    3    4    5
+----+----+----+----+----+----+ le tableau
| 66 | 22 | 33 | 11 | 55 | 44 |
+----+----+----+----+----+----+

+----+----+----+----+----+----+ la séquence, au départ
|    |    |    |    |    |    |
+----+----+----+----+----+----+ insérer 66
| 66 |    |    |    |    |    |
+----+----+----+----+----+----+ insérer 22
| 22 | 66 |    |    |    |    |
+----+----+----+----+----+----+ insérer 33
| 22 | 33 | 66 |    |    |    |
+----+----+----+----+----+----+ insérer 11
 ...
```

- pour placer 22, on l'a échangé avec l'élément qui est à sa gauche (66) qui est plus grand. Et on est arrivé au bord, donc on s'est arrêté.
- Pour placer le 33, on l'a échangé avec l'élément de gauche (66). Et comme 22 est plus petit, on s'est arrêté là.
- pour placer 11, on échangera avec 66, 33, et 22.

4.2 L'insertion

En première approche, pour insérer un élément en position `p`, sachant que ce qui est à gauche est déjà ordonné

- on compare avec ce qui est à gauche (en position `p-1`)
- si besoin est, on échange, et on recommence en position `p-1`.

L'expression "si besoin est" veut dire - si on n'est pas arrivé en position 0 (le bord du tableau) - si ce qui est à gauche est plus grand

Pour insérer en position p :

```
q = p
tant que q > 0 et tableau[q] > tableau[p]
faire {
    échanger les contenus des positions q et q-1
    q = q - 1
}
```

4.3 Algorithme complet

En intégrant les deux boucles

```
pour p de 0 à taille -1 faire {
    q = p
    tant que q > 0 et tableau[q] > tableau[p]
    faire {
        échanger les contenus des positions q et q-1
        q = q - 1
    }
}
```

4.4 Insertion améliorée : décalage

On peut améliorer un peu

- on partant de la position 1 (puisque une séquence d'un seul élément est déjà ordonnée),
- en remplaçant les échanges par des décalages.

Exemple et illustration : quand on veut insérer 11 dans la séquence qui contient 22, 33 et 66 : on met de côté la valeur 11 dans une variable temporaire.

```
 0    1    2    3 q                   temp
+----+----+----+----+----+----+       +----+
| 22 | 33 | 66 | 11 |    |    |       | 11 |
+----+----+----+----+----+----+       +----+
```

Le 66 n'étant pas à la bonne place, on le décale à la place du 11

```
                  q
+----+----+----+----+----+----+       +----+
| 22 | 33 | 66 | 66 |    |    |       | 11 |
+----+----+----+----+----+----+       +----+
```

pareil pour le 33 et le 22

```
            q
+----+----+----+----+----+----+       +----+
| 22 | 33 | 66 | 66 |    |    |       | 11 |
+----+----+----+----+----+----+       +----+

       q
+----+----+----+----+----+----+       +----+
| 22 | 22 | 66 | 66 |    |    |       | 11 |
+----+----+----+----+----+----+       +----+
```

Et comme on a fini de décaler (ici parce qu'on est arrivé au bord), il ne reste plus qu'à recaser le 11

```
 q
+----+----+----+----+----+----+       +----+
| 11 | 22 | 66 | 66 |    |    |       | 11 |
+----+----+----+----+----+----+       +----+
```

Exercice

- écrire les deux variantes du tri par insertion
- comparer les temps d'exécution.

5 Le tri à bulles

L'idée du tri à bulles vient de l'équivalence de deux définitions de ce qu'est un tableau `t` ordonné (dans l'ordre croissant)

1. pour tous i et j, si `t[i] < t[j]` alors `i < j`
2. pour tout i, `t[i-1] < t[i]`

La quantification correcte des indices est laissée au lecteur.

La première définition est "globale" : les cases que l'on compare peuvent être n'importe où. La seconde est locale, c'est un critère sur des cases voisines.

5.1 De la vérification à la rectification

Vérifier si un tableau est ordonné est donc plus facile en se basant sur la seconde définition : on parcourt le tableau, et on vérifie que chaque élément est en bon ordre avec son voisin.

Ce qui amène à une idée pour le tri : si deux voisins ne sont pas dans le bon ordre, on arrange ça en les permutant.

```
 0    1    2    3    4    5
+----+----+----+----+----+----+  au départ
| 66 | 22 | 33 | 11 | 55 | 44 |
+----+----+----+----+----+----+
| 22 | 66 | 33 | 11 | 55 | 44 |
+----+----+----+----+----+----+
| 22 | 33 | 11 | 66 | 55 | 44 |
```

```
+----+----+----+----+----+----+
| 22 | 33 | 11 | 55 | 66 | 44 |
+----+----+----+----+----+----+
| 22 | 33 | 11 | 55 | 44 | 66 |
+----+----+----+----+----+----+
```

remarquez que

- ça n'a pas complètement trié le tableau
- mais maintenant, l'élément le plus grand (66) est arrivé à sa place définitive : à la fin du tableau

Si on refait un passage, le 55 arrivera en avant-dernier, d'où il ne devrait plus bouger ensuite, et les autres éléments se seront rapprochés de leurs positions définitives.

5.2 Variantes du tri à bulles

En faisant plusieurs passages, de plus en plus courts, on finit donc par ordonner le tableau.

À partir de là, on peut construire plusieurs variantes du tri à bulles

1. faire autant de passages qu'il y a d'éléments (moins 1 pour les raisons habituelles),

La version classique (la plus mauvaise !)

```
pour i de taille à 1 (en descendant) {
   pour j de 1 à i-1 {
      si tableau[i] > tableau[j] {
          échanger les positions i et j
      }
  }
}
```

C'est la plus mauvaise parce qu'elle effectue le même nombre de comparaisons indépendamment du fait que le tableau soit déjà ordonné ou pas.

2. faire des passages tant qu'il y eu des échanges.

Dans ce cas, le premier passage sert à constater que le tableau est déjà ordonné, et le travail s'arrête là.

5.3 Optimisation

D'autres optimisations complémentaires sont possibles :

- faire commencer/finir un passage là où le passage précédent a commencé/terminé de faire des échanges.

6 Annexes

6.1 Recherche de la plus petite valeur d'un tableau

En C, ce serait quelque chose comme

```
int plus_petite_valeur(int taille, int tableau[])
{
   int minimum = tableau[0];
   for (int i = 1; i < taille; i++) {
       if (tableau[i] < minimum) {
          minimum = tableau[i];
       }
   }
   return minimum;
}
```

Questions :

- pourquoi ne commence-t-on pas avec `int minimum = 0` ?
- que se passerait-il si la boucle `for` commençait à 0 ?

6.2 Échange de deux éléments dans un tableau

```
void echanger(int position1, int position2, int tableau[])
{
    int temporaire = tableau[position1];
    tableau[position1] = tableau[position2];
    tableau[position2] = temporaire;
}
```

6.3 Recherche de la position de la plus petite valeur

À comparer avec le recherche de la plus petite valeur.

```
int position_minimum(int taille, int tableau[])
{
    int position = 0;
    int minimum = tableau[0];
    for (int i = 1; i < taille; i++) {
       if (tableau[i] < minimum) {
          position = i;
          minimum = tableau[i];

       }
   }
   return position;
}
```

6.4 Tester si un tableau est en ordre croissant

```
bool en_ordre_croissant(int taille, int tableau[])
{
```

```
    for (int i = 1; i < taille; i++) {
       if (tableau[i-1] > tableau[i]) {
          return false;       // on a trouvé un contre-exemple
       }
    }
    return true;
}
```

Question : pourquoi on ne commence pas à 0 ?