Algorithmes de tri

Julien Noyer

2018-09-26

Algorithme de trie *Organiser et trier des données*

Un **algorithme de tri** en informatique permet d'organiser une collection de données selon un ordre déterminé au préalable comme par *ordonner des entiers du plus grand au plus petit*. **Les algorithmes de tri** sont utilisés dans de très nombreuses situations et ils sont en particulier utiles à de nombreux algorithmes plus complexes comme certains *algorithmes de recherche* ou dans des *réseaux de neurones* informatiques. Ils peuvent également servir pour mettre en forme des données afin de les rendre plus lisibles pour l'utilisateur.

Le tri a bulle

Figure 1 – Tri à bulle : https://www.youtube.com/watch?v=lyZQPjUT5B4

Le tri à bulle consiste à parcourir le tableau en permutant toutes les paire d'éléments consécutifs non ordonnés. Après le premier parcours, le plus grand élément se retrouve dans la deuxième case du tableau, il reste donc à appliquer la même procédure sur les autres données en paçant les plus grandes données vers la droite.

```
Tableau au début [12, 0, 7, 29, 3]

[0, 12, 7, 29, 3]
[0, 7, 12, 29, 3]
[0, 7, 12, 3, 29]
[0, 7, 3, 12, 29]

Tableau à la fin [0, 3, 7, 12, 29]
```

Cette algorithme de tri est très simple à comprendre, on peut voir son fonctionnement qui permet de faire « descendre » le plus petit chiffre vers la gauche. Nous allons voir à présent comment écrire cette algorithme en pseudo code.

Pseudo-code

```
ALGO bubbleSorting
  data: ARRAY<NUMBER>
  dataSize: NUMBER
  sorted: BOOLEAN
  i: NUMBER
  tmp: NUMBER

START
  FUNCTION sort(data)
      dataSize <- data.LENGTH
      sorted <- False

  START
    WHILE sorted == False
      sorted <- True

      FOR i FROM 0 TO dataSize [ i <- i + 1 ]
        IF data[i] > data[i + 1] THEN
          tmp <- data[1 +1]
          data[i + 1] <- data[i]
          data[i] <- tmp
          sorted <- False
        END IF
      END FOR

      dataSize <- dataSize - 1
    END WHILE
  END FUNCTION

END ALGO
```

— Estimez-vous cette algorithme rapide ? Très rapide ?
— Cette algorithme est-il efficace sur des grandes collections de données ?
— Pourquoi prendre la dance pour illustrer un algorithme ?

Complexité

Dans ce calcule nous recherchons une valeur pour le traitement de l'algorithme de **tri à bulle** en concidérant qu'une action élémentaire vaut 1.

Début de la function :

- dataSize <- data.LENGTH = 1
- sorted <- False = 1
- sorted <- True = 1
- **Total**: 3

Boucle :

- i FROM 0 TO dataSize = 2
- i + 1 = 1
- data[i] > data[i + 1] = 2
- tmp <- data[1 +1] = 2
- data[i + 1] <- data[i] = 2
- data[i] <- tmp = 1
- sorted <- False = 1
- dataSize <- dataSize - 1 = 2
- **Total** = 13 x dataSize

Nous pouvons à présent calculer une valeur pour le traitement du tableau suivnat :

```
[6, 1, 12, 0, 25, 7, 29, 3, 45]
```

- Formule : **Val(n) = 13n + 3**
- Résultat : 81 x 18 + 3 = **120**

Le tri par selection

Figure 2 – Tri à bulle : https://www.youtube.com/watch?v=Ns4TPTC8whw

Le tri par sélection consiste à trouver dans le tableau le numéro de l'élément le plus petit, c'est-à-dire l'entier minminimum. Une fois ce numéro trouvé, les éléments sont échangés, cet échange nécessite, puis la même procédure est appliquée sur la suite d'éléments.

Pseudo-code

```
ALGO selectSorting
  data: ARRAY<NUMBER>
  dataSize: NUMBER
  sorted: BOOLEAN
  i: NUMBER
  j: NUMBER
  tmp: NUMBER

START
  FUNCTION sort(data)
      dataSize <- data.LENGTH
      sorted <- False
```

```
    START
      FOR i FROM 0 TO dataSize [ i <- i + 1 ]
        FOR j FROM i + 1 TO dataSize [ j <- j + 1 ]
          IF data[j] < data[i] THEN
            tmp <- data[i]
            data[i] <- data[j]
            data[j] <- tmp
          END IF
        END FOR
      END FOR
    END FUNCTION
END ALGO
```

— Cette algorithme vous semble-t-til plus efficace que le premier ?
— Quelle est votre niveau de compréhention sur le pseudo-code ci-dessus ?

Complexité

Dans ce calcule nous recherchons une valeur pour le traitement de l'algorithme de **tri par sélection** en concidérant qu'une action élémentaire vaut 1.

Début de la function :

— dataSize <- data.LENGTH = 1
— sorted <- False = 1
— **Total**: 3

Boucle i :

— i FROM 0 TO dataSize = 2
— i + 1 = 1
— **Total** = 2 x dataSize

Boucle j :

— j FROM i + 1 TO dataSize = 3
— j + 1 = 1
— data[j] < data[i] = 1

- tmp <- data[i] = 1
- data[i] <- data[j] = 1
- data[j] <- tmp = 1
- **Total** = 8 x dataSize

Nous pouvons à présent calculer une valeur pour traitement du tableau suivnat :

```
[6, 1, 12, 0, 25, 7, 29, 3, 45]
```

- Formule : **Val(n) = 8n x 2n + 3**
- Résultat : 81 x 18 + 3 = **1 461**

Le tri par insertion

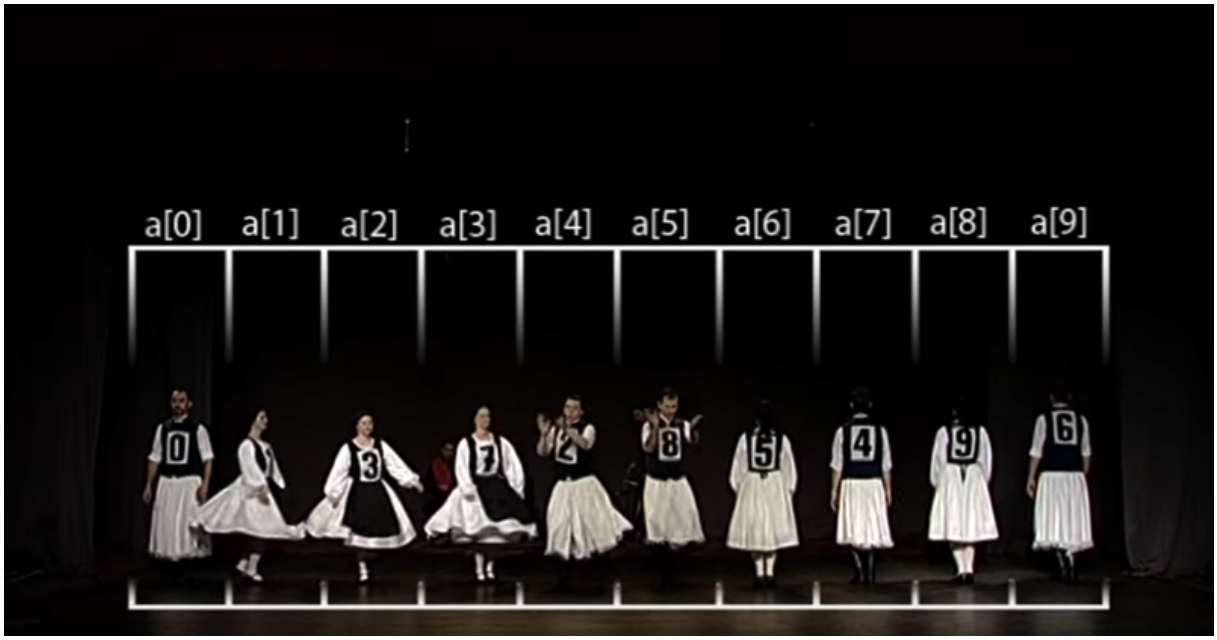

Figure 3 – Tri à bulle : https://www.youtube.com/watch?v=ROalU379l3U

Le tri par insertion est très différent de la méthode de tri par sélection et s'apparente à celle utilisée pour trier ses cartes dans un jeu : on prend une carte, puis la deuxième que l'on place en fonction de la première, ensuite la troisième que l'on insère à sa place en fonction des deux premières et ainsi de suite. Le principe général est donc de considérer que les premières cartes sont triées et de placer les suivantes à leur place parmis les carte déjà tirées.

Pseudo-code

```
ALGO insertSorting
  data: ARRAY<NUMBER>
  dataSize: NUMBER
  i: NUMBER
  j: NUMBER
  tmp: NUMBER

START
  FUNCTION sort(data)
      dataSize <- data.LENGTH

    START
      FOR i FROM 0 TO dataSize [ i <- i + 1 ]
        tmp <- data[i]
        j <- i - 1

        WHILE temp < data[j] && j >= 0 THEN
          data[i] <- data[j + 1]
          j <- j - 1
        END WHILE

        data[j + 1] <- tmp
      END FOR
    END FUNCTION
END ALGO
```

— Cette algorithme vous semble-t-til être le plus efficace ?
— Serait-il possible d'associer cette algorithme à un des deux autres ?

Complexité

Dans ce calcule nous recherchons une valeur pour le traitement de l'algorithme de **tri par insertion** en concidérant qu'une action élémentaire vaut 1.

Début de la function :

— dataSize <- data.LENGTH = 1
— **Total**: 1

Boucle :

— i FROM 0 TO dataSize = 2
— i + 1 = 1
— tmp <- data[i] = 1
— j <- i - 1 = 1
— data[j + 1] <- tmp = 1
— **Total** = 8 x dataSize

While :

— temp < data[j] && j >= 0 = 3
— data[i] <- data[j + 1] = 2
— j <- j - 1 = 2
— **Total** = 7 x dataSize

Nous pouvons à présent calculer une valeur pour traitement du tableau suivnat :

```
[6, 1, 12, 0, 25, 7, 29, 3, 45]
```

— Formule : **Val(n) = 7n x 8n + 1**
— Résultat : 63 x 81 + 1 = **5 014**

Le tri en fusiuon

Figure 4 – Tri fusion : https://www.youtube.com/watch?v=XaqR3G_NVoo

Le tri en fusion applique le principe de « diviser pour régner ». En effet, étant données deux collection de données triées et la longueur des collections, il est très facile d'obtenir une troisième collection de données triés de longueur égale à la taille des deux premières collections, par « interclassement » ou fusion des deux précédentes collections.

Pseudo-code

```
ALGO insertSorting
  data: ARRAY<NUMBER>
  dataSize: NUMBER
  rightData: ARRAY<NUMBER>
  leftData: ARRAY<NUMBER>
  fusionedArray: ARRAY<NUMBER>
  result: ARRAY<NUMBER>
  middle: NUMBER
  sorted: NUMBER

START
```

```
  FUNCTION bubbleSort(data)
      dataSize <- data.LENGTH
      sorted <- False

  START
    WHILE sorted == False
      sorted <- True

      FOR i FROM 0 TO dataSize [ i <- i + 1 ]
        IF data[i] > data[i + 1] THEN
          tmp <- data[1 +1]
          data[i + 1] <- data[i]
          data[i] <- tmp
          sorted <- False
        END IF
      END FOR

      dataSize <- dataSize - 1
    END WHILE

    RETURN data
  END FUNCTION

  FUNCTION fusionSort(data)
      middle <- data.LENGTH / 2

      leftData <- FROM data[middle + 1] TO data.LENGTH - 1
      rightData <- FROM data[0] TO data[middle]

      leftArray <- bubbleSort(leftData)
      rightArray <- bubbleSort(rightData)

      fusionedArray <- leftArray + rightArray

      RETURN bubbleSort(fusionedArray)
  END FUNCTION
END ALGO
```

— Quel est l'intêret de cumuler plusieur algorithmes de trie ?
— Cette méthode est-elle plus ou moins efficace que les autres ?
— Que ce passe-t-il pour un tableau de 2 données ?

Complexité

Dans ce calcule nous recherchons une valeur pour le traitement de l'algorithme de **tri en fusion** en concidérant qu'une action élémentaire vaut 1.

Trie à bulle :

— **Val(n) = 13n + 3**

Trie en fusion :

— middle <- data.LENGTH / 2 = 2
— leftData <- FROM data[middle + 1] TO data.LENGTH - 1 = 3
— rightData <- FROM data[0] TO data[middle] = 1
— leftArray <- bubbleSort(leftData) = 1
— rightArray <- bubbleSort(rightData) = 1
— fusionedArray <- leftArray + rightArray = 2
— **Total** = 10 x dataSize

Nous pouvons à présent calculer une valeur pour traitement du tableau suivnat :

```
[6, 1, 12, 0, 25, 7, 29, 3, 45]
```

— Formule : **Val(n) = 10n + (13n + 3)**
— Résultat : 63 x 81 + 1 = **210**

Conclusion

Nous venons de voir à travers *3 algorithmes de tri* comment il est possible d'**organiser des informations en définissant des étapes logiques** pour y arriver. Comme nous l'avons vu plusieurs algorithmes peuvent résoudre un même problème, vous pouvez à présent **tester vos connaissances** avec les *exercices* suivants et aller plus loins dans votre découvert des algorithme de tri avec les *ressources* ci-arpès.

Exercices

Les ébnoncés qui vont suivre définisse la problèmatique à résoudre et le résultat attendu. Pour chaque exercice vous devez réaliser le pseudo-code, le calcule de la complexité et un script Javascrip réalisant l'algorithme.

- **Ecrire un algorithme pour savoir si les nombres d'un tableau sont pair ou non.**
- Entrée = `[65, 2, 87, 30, 16, 5, 14, 67, 56, 8]`

- Sortie = `[65 != pair, 2 = pair, ...]`
- **Ecrire un algorithme qui permet de calculer la moyenne des chiffres d'un tableau.**
- Entrée = `[65, 2, 87, 30, 16, 5, 14, 67, 56, 8]`
- Sortie = `Moyenne 35 - Plus petit = 2, Plus grand = 87`
- **Ecrire un algorithme qui permet d'additionner les chiffres d'un nombre.**
- Entrée = `123456789`
- Sortie = `1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45`

Ressources

- Wikipedia : Algorithme de tri
- [Introduction aux algorithmes de tri][https://interstices.info/les-algorithmes-de-tri/]
- Sorting.at
- Algorithmes de tri visualisés en danses folkloriques
- Algorithme de tri en Javascript