

Dans de nombreux domaines, lorsqu'on dispose de données rangées dans un tableau, il peut être intéressant de savoir trier celles-ci dans un certain ordre. On s'intéresse donc aux algorithmes permettant de trier un tableau.

1. Quels sont ces algorithmes ?
2. Peut-on garantir que ces algorithmes font bien ce qu'on veut ? (tests et preuves)

Les données pouvant être très nombreuses, un enjeu s'ajoute :

3. Prend-il longtemps à s'exécuter / occupe-t-il beaucoup de mémoire ? (complexité)

Pour illustrer ces notions, on s'intéresse à deux algorithmes « classiques » de tri.

1 Tris par insertion, par sélection

1.1 Tri par insertion

Voici un exemple d'application du tri par insertion :

Étape 0 : 12 42 11 95 22 3
 Étape 1 : 12 42 11 95 22 3
 Étape 2 : 12 42 11 95 22 3
 Étape 3 : 11 12 42 95 22 3
 Étape 4 : 11 12 42 95 22 3
 Étape 5 : 11 12 22 42 95 3
 Étape 6 : 3 11 22 12 42 95

Écrivons l'algorithme :

Algorithme 1 : Tri par insertion

Données : t un tableau de taille n

Résultat : aucun, le tableau t doit être trié

Idee : On a une partie du tableau triée (initialement vide) et on insère chaque élément du tableau à la bonne place dans cette partie

début

```

  pour  $i = 1$  à  $n - 1$  faire
     $x \leftarrow t[i]$  ;
     $j \leftarrow i$  ;
    tant que  $j > 0$  et  $x < t[j - 1]$  faire
       $t[j] \leftarrow t[j - 1]$  ;
       $j \leftarrow j - 1$ 
    fin
     $t[j] \leftarrow x$ 
  fin
fin
```

1.2 Tri par sélection

Voici un exemple d'application du tri par sélection :

Étape 0 : 12 42 11 95 22 3
 Étape 1 : 3 42 11 95 22 12
 Étape 2 : 3 11 42 95 22 12
 Étape 3 : 3 11 12 95 22 42
 Étape 4 : 3 11 12 22 95 42
 Étape 5 : 3 11 12 22 42 95
 Étape 6 : 3 11 12 22 42 95

Écrivons l'algorithme :

Algorithme 2 : Tri par sélection**Données** : t un tableau de taille n **Résultat** : aucun, le tableau t doit être trié**Idée** : On recherche le minimum du sous tableau $t[i] \dots t[n-1]$ pour le placer en position i **début**

```

  pour  $i = 0$  à  $n - 1$  faire
     $min \leftarrow t[i]$  ;
     $i_{min} \leftarrow i$  ;
    pour  $j = i + 1$  à  $n - 1$  faire
      if  $t[j] < min$  then
         $min \leftarrow t[j]$  ;
         $i_{min} \leftarrow j$ 
      end
    fin
    échanger  $t[i]$  et  $t[i_{min}]$ 
  fin
fin

```

2 Tests et preuves

2.1 Tests

Pour s'assurer de la correction (le fait que celui-ci fait bien ce que l'on veut) d'un algorithme, on peut effectuer des tests.

1. Pour des petits algos simples, on peut se contenter de tester à la main sur quelques exemples bien choisis.
2. Pour des algorithmes plus complexes, on pourra avoir recours à un jeu de tests automatisés.

Lors du développement d'un programme, il est extrêmement recommandé de tester les algorithmes implémentés au fur et à mesure et non à la fin du développement.

Attention cependant : comme en maths, des tests, même nombreux n'ont aucune valeur de preuve. Ils peuvent cependant augmenter la confiance qu'on donne à un algorithme.

Algorithme 3 : Vérifier qu'un tableau est trié**Données** : t un tableau de taille n **Résultat** : aucun, si l'algorithme termine, tout va bien**Idée** : On vérifie que $t[i] < t[i + 1]$ pour tout i entre 0 et $n - 2$ **début**

```

  pour  $i = 0$  à  $n - 2$  faire
    | Vérifier que  $t[i] \leq t[i + 1]$ 
  fin
fin

```

Remarque : Ce test nous dit si le tableau est trié ou non, mais pas forcément si un tableau donné a bien été trié. Par exemple, si on veut trier le tableau $[1, 2, 0]$ et que la réponse donnée par un algo (faux) est $[1, 2, 3]$, ce test ne détectera pas d'erreur.

Une fois une bonne fonction de test implémentée, on peut vérifier sur plusieurs exemples que les fonctions de tris fonctionnent.

Quelques pistes pour savoir comment tester un algo :

1. Tester les cas triviaux : ici, tableau vide, tableaux déjà triés...
2. Tester avec des cas "difficiles" pour l'algo : ici, un tableau totalement à l'envers par exemple.
3. Tester avec des données importantes : ici, des tableaux de grandes tailles par exemple.

Algorithme 4 : Tester une fonction de tri**Données** : aucune, si l'algorithme termine, tout va bien**Résultat** : aucun**Idée** : On génère différents tableaux aléatoire, qu'on mélange pour vérifier que le tri fonctionne**début**

```

    pour n = 0 à 100 faire
        générer un tableau  $t_0$  de taille  $n$  rempli de 0 ;
        générer un tableau  $t_1$  de taille  $n$  trié rempli de nombres aléatoires avec doublons ;
        générer un tableau  $t_2$  de taille  $n$  trié rempli de nombres aléatoires sans doublons ;
         $copie_0, copie_1, copie_2 \leftarrow t_0, t_1, t_2$  ;
        appliquer la fonction de tri à  $copie_0, copie_1, copie_2$  ;
        vérifier que  $copie_0 = t_0, copie_1 = t_1, copie_2 = t_2$ 
    fin
fin
```

2.2 Preuve de correction : invariant de boucle

Même si un algorithme testé a l'air de fonctionner, on ne peut en être certain qu'à condition de prouver sa correction.

Un méthode revenant régulièrement dans les preuves de correction d'algo est l'utilisation de ce qu'on appelle un **invariant de boucle**.

Définition : Un invariant de boucle est une propriété portant sur l'état des variables d'un programme et qui, si elle est vrai à l'entrée d'un passage de la boucle considérée (**for** ou **while**), sera également vraie au à l'entrée du suivant. Voyons cela sur un exemple simple.

On suppose qu'une variable x a été initialisée.

```

for i in range(100):
    x = 2*x+5
```

Considérons la propriété suivante :

$$\mathcal{P}(x) : x > 0$$

Supposons qu'au début d'un passage de la boucle $\mathcal{P}(x)$ soit vraie, autrement dit $x > 0$. Dans ce cas, $2 * x > 0$ et donc $2 * x + 5 > 0$. Ainsi, à la fin du passage (donc au début du suivant), on a bien toujours $x > 0$ et donc $\mathcal{P}(x)$ est toujours vraie.

$\mathcal{P}(x)$ est un invariant de la boucle considérée.

Propriété : Si \mathcal{P} est un invariant d'une boucle \mathcal{B} d'un algorithme et si \mathcal{P} est vraie en entrée de \mathcal{B} , alors \mathcal{P} est également vraie en sortie de \mathcal{B} .

Cette propriété assure que si la variable x est positive avant la boucle, elle sera positive après la boucle.

2.3 Correction du tri par insertion

Reprenons l'algorithme de tri par insertion :

Algorithme 1 : Tri par insertion**début**

```

    pour i = 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
        t[j] ← x
    fin
fin
```

L'algorithme est constitué de deux boucles imbriquées :

1. la boucle principale (sur i), disons A ;

2. le boucle interne (sur j), disons B .

Considérons la propriété suivante :

\mathcal{P} : Le sous tableau $[t[0] \dots t[i-1]]$ est trié au tour i de la boucle A .

Nous allons montrer que \mathcal{P} est un invariant de la boucle A .

Supposons qu'au tour i de la boucle A , le sous tableau $[t[0] \dots t[i-1]]$ est trié et montrons qu'à la boucle suivante, le sous tableau $[t[0] \dots t[i]]$ est encore trié.

La valeur de i est d'abord stockée dans une variable x puis on entre dans la boucle B (**while**). Dans cette boucle, on décale les valeurs du tableau à gauche de i d'une place vers la droite jusqu'à trouver une valeur inférieure à x ou qu'on atteigne le premier élément de t . En sortie de B , on a donc ou bien $j=0$ ou bien $t[j-1] \leq x$.

Dans les deux cas et si $i \neq j$, $t[j+1] > x$ car cela correspond au dernier test positif du **while**. D'où :

En sortie de B , si $j \neq 0$ et $j \neq i$ on a $t[j-1] \leq x < t[j+1]$

On place alors x dans $t[j]$.

Bilan :

- La partie de t à gauche de j (si elle n'est pas vide) n'ayant pas bougé au cour de la boucle B , est par hypothèse toujours triée.
- La partie de t à droite de j jusque i (si elle n'est pas vide) correspond aux valeurs du sous tableau trié, mais décalées d'un rang et et donc toujours triée.
- $t[j-1] \leq t[j] = x < t[j+1]$ en ne prenant pas en compte les parties impliquées si $j=0$ ou $j=i$

On en déduit qu'à la fin de la boucle A , le sous tableau $[t[0] \dots t[i]]$ est encore trié. \mathcal{P} est donc bien un invariant de la boucle A .

Or au premier passage de la boucle A , \mathcal{P} est vrai puisque $[t[0] \dots t[i-1]] = [t[0]]$ est un tableau à un élément donc trié. \mathcal{P} est donc toujours vrai en sortie de A , soit $[t[0] \dots t[i-1]] = [t[0] \dots t[n-1]] = t$ trié.

Ce qui prouve la correction de l'algorithme.

2.4 Correction du tri par sélection

Reprenons l'algorithme de tri par sélection :

Algorithme 2 : Tri par sélection

```

début
  pour  $i = 0$  à  $n - 1$  faire
     $min \leftarrow t[i]$  ;
     $i_{min} \leftarrow i$  ;
    pour  $j = i + 1$  à  $n - 1$  faire
      if  $t[j] < min$  then
         $min \leftarrow t[j]$  ;
         $i_{min} \leftarrow j$ 
      end
    fin
    échanger  $t[i]$  et  $t[i_{min}]$ 
  fin
fin

```

L'algorithme est constitué de deux boucles imbriquées :

1. la boucle principale (sur i), disons A ;
2. le boucle interne (sur j), disons B .

Considérons la propriété suivante :

\mathcal{P} : Le sous tableau $[t[0] \dots t[i-1]]$ contient les i plus petites valeurs de t dans l'ordre croissant.

Nous allons montrer que \mathcal{P} est un invariant de la boucle A .

Supposons que le sous tableau $[t[0] \dots t[i-1]]$ contient les i plus petites valeurs de t dans l'ordre croissant à l'entrée i de la boucle A .

Dans ce cas, la $i+1$ -ème plus petite valeur de t peut être trouvée en cherchant la plus petite valeur du sous tableau restant $[t[i] \dots t[n-1]]$.

C'est précisément ce que fait la boucle A , où l'on reconnaît un algo de recherche de minimum (jusque la fin de la boucle `while`).

Ce minimum est alors placé en place i du tableau, ce qui garanti qu'après la boucle A , le sous-tableau $[t[0]...t[i]]$ est toujours trié.

\mathcal{P} est donc bien un invariant de A qui de surcroît est vrai au premier passage (le tableau vide étant trié). \mathcal{P} est donc vrai en sortie de A , et \mathbf{t} est donc bien trié à la fin de l'algo.

3 Complexité

3.1 Compter les opérations

La complexité d'un algorithme vise à mesurer l'efficacité de ce dernier. Cela rassemble plusieurs notions différentes :

- La **complexité temporelle** mesure le nombre d'opérations effectuées par l'algo.
- La **complexité spatiale** (ou mémoire) mesure le nombre de variables utilisées par l'algo.

Dans le deux cas on peut distinguer plusieurs mesures :

- Les complexités **moyennes**.
- Les complexités **dans le pire cas**.

Dans ce cours on s'intéressera surtout à la complexité temporelle dans le pire cas, c'est à cela qu'on fera allusion en écrivant "complexité" dans ce cours.

Définition Une opération élémentaire d'un algorithme est :

- une opération arithmétique ou booléenne ;
- un test ($<$, $>$, $==$, $!=$, $<=$, $>=$) sur des types de base (pas de tableaux / chaînes de caractères, etc) ;
- une affectation.

Pour déterminer la complexité d'un algorithme, on compte ou on majore le nombre d'opérations élémentaires.

Prenons un exemple simple :

Algorithme 5 : Un exemple simple

```

début
  |  $x \leftarrow 2 + 3$  ;
  | si  $x > 2$  et  $x > 1$  alors
  | |  $x \leftarrow 4$ 
  | fin
fin

```

Cet algorithme comporte :

1. 1 somme et 1 affectation ligne 1 ;
2. 2 test et 1 opération ligne 2 ;
3. 1 affectation ligne 3.

Cela donne un total de 6 opérations élémentaires.

Remarque : puisqu'on raisonne "dans le pire cas", on considérera toujours qu'une partie de l'algorithme à l'intérieur d'un `if` est exécutée.

3.2 Complexité du tri par insertion

Reprenons l'algorithme de tri par insertion :

Algorithme 1 : Tri par insertion

```

début
  | pour  $i = 1$  à  $n - 1$  faire
  | |  $x \leftarrow t[i]$  ;
  | |  $j \leftarrow i$  ;
  | | tant que  $j > 0$  et  $x < t[j - 1]$  faire
  | | |  $t[j] \leftarrow t[j - 1]$ ;
  | | |  $j \leftarrow j - 1$ 
  | | fin
  | |  $t[j] \leftarrow x$ 
  | fin
fin

```

La boucle B effectue 3 opérations arithmétiques, 2 affectations, 2 tests, 1 opération booléenne et est effectuée dans le pire cas $(i - 1)$ fois pour un total de $8(i - 1)$ opérations élémentaires. La boucle A effectue 3 affectations en plus de la boucle B , soit un total de $8(i - 1) + 3 = 8i - 5$ opérations élémentaires au i -ème passage.

On en déduit que le nombre d'opérations élémentaires de l'algorithme vérifie :

$$\begin{aligned}
 c(n) &\leq 8 \times 1 - 5 \\
 &\quad + 8 \times 2 - 5 \\
 &\quad \dots \\
 &\quad + 8 \times (n - 1) - 5 \\
 &= 8 \times [1 + \dots + (n - 1)] - 5 \times (n - 1) \\
 &= 8 \times \frac{n(n - 1)}{2} - 5(n - 1) \\
 &= 4n^2 - 4n - 5n + 5 \\
 &= 4n^2 - 9n + 5
 \end{aligned}$$

Le tri par insertion est donc au plus quadratique : complexité en $\mathcal{O}(n^2)$.

Reprenons l'algorithme de tri par sélection :

Algorithme 2 : Tri par sélection

```

débüt
  pour i = 0 à n - 1 faire
    min ← t[i] ;
    i_min ← i ;
    pour j = i + 1 à n - 1 faire
      if t[j] < min then
        min ← t[j] ;
        i_min ← j
      end
    fin
    échanger t[i] et t[i_min]
  fin
fin

```

Dans la boucle B on effectue au pire 1 test et 2 affectations, le tout répété $(n - i)$ fois, pour un total de $3(n - i)$ opérations élémentaires. Dans la boucle A en plus de la boucle B on a 2 affectations plus la fonction d'échange, qui en effectue 3. D'où le coût d'un passage dans cette boucle : $3(n - i) + 5 = 3n - 3i + 5$ opérations.

On en déduit que le nombre d'opérations élémentaires de l'algorithme vérifie :

$$\begin{aligned}
 c(n) &\leq 3n - 3 \times 0 + 5 \\
 &\quad + 3n - 3 \times 1 + 5 \\
 &\quad \dots \\
 &\quad + 3n - 3 \times (n - 1) + 5 \\
 &= 3n \times n - 3 \times [0 + 1 + \dots + (n - 1)] + 5 \times n \\
 &= 3n^2 - 3 \times \frac{n(n - 1)}{2} + 5n \\
 &= 3n^2 - 1.5n^2 + 1.5n + 5n \\
 &= 1.5n^2 + 6.5n
 \end{aligned}$$

Le tri par sélection est donc au plus quadratique : complexité en $\mathcal{O}(n^2)$.