Les algorithmes de tri

Il est temps de faire le tri

Lorsqu'on manipule des tableaux de valeurs, ou des listes, il est souvent très utile de les trier. Cela permet, par exemple, de trouver la médiane, de faire des recherches plus rapidement, d'essayer des solutions possibles par ordre de priorité, de comparer facilement deux tableaux, d'afficher des noms par ordre alphabétique…
Trier des données, cela peut sembler simple, mais ça ne l'est pas tant que cela. Il y a de nombreuses façons de s'y prendre, avec des algorithmes plus ou moins performants. Nous allons considérer des algorithmes qui trient un tableau "sur place", c'est-à-dire qui échangent les éléments du tableau deux par deux jusqu'à ce que le tableau soit trié.
Dans cette feuille, nous allons considérer un tableau T de n valeurs, numérotées de 1 à n. Nous noterons "Échanger(T[i], T[j])" pour dire qu'on échange les valeurs des cases i et j du tableau T. Nous allons considérer 3 algorithmes, dits "naturels", puisqu'ils peuvent correspondre à des méthodes de tri utilisées "dans la vraie vie".

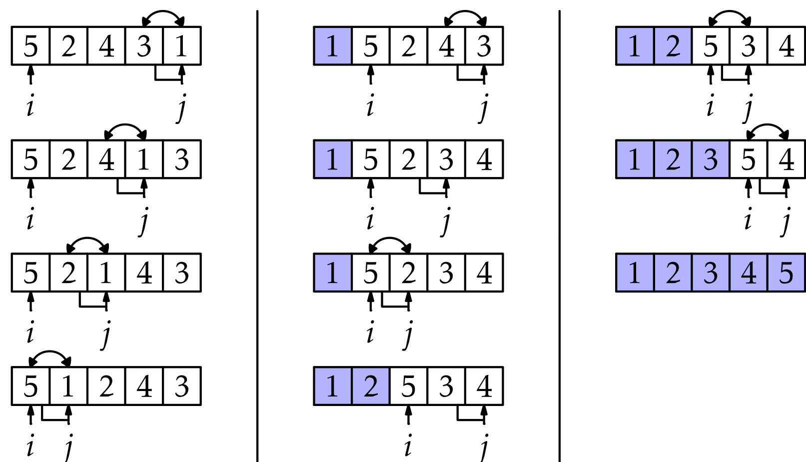
Le tri à bulles

Le tri a bulles, comme son nom l'indique, consiste à faire remonter les valeurs les plus faibles, comme les bulles remontent à la surface d'un liquide. L'algorithme est donné ci-contre. C'est probablement l'algorithme de tri le plus simple à écrire.

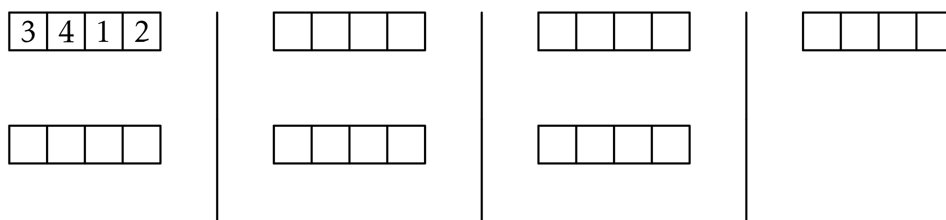
Algorithme du tri à bulles :

```
Pour i allant de 1 à n − 1 :
    Pour j allant de n à i + 1 :
        Si T[j − 1] > T[j] :
            Échanger(T[j − 1], T[j])
```

Intuitivement, il consiste à partir de la fin du tableau et à remonter vers le premier élément et à chaque fois que 2 valeurs consécutives sont dans le mauvais ordre, on les inverse. Ainsi, on fait remonter la plus petite valeur rencontrée jusqu'à rencontrer une plus petite valeur, qui continuera à remonter.
Après un premier parcours du tableau, le minimum se trouve en première position. On recommence alors en partant de la fin en allant jusquà la deuxième position. C'est alors le deuxième minimum qui se retrouve en bonne position. Voici un schéma qui illustre le tri d'un tableau avec cet algorithme.



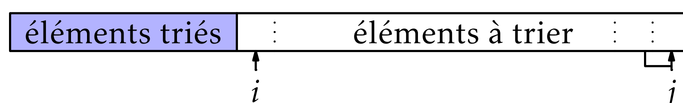
Exercice 1 : Donner les étapes permettant de trier le tableau ci-dessous :



Preuve de correction

Puisque ce tableau utilise des boucles bornées, sa terminaison est évidente. Mais est-ce qu'il fait bien ce qu'on attend ? Pour cela, nous allons utiliser un invariant de boucle pour la boucle "externe" (celle avec i). On peut remarquer qu'avant chaque tour de cette boucle, les éléments en position 1 à $i-1$ sont triés et en bonne position.

État du tableau au début d'un tour de la boucle externe :



À la fin de l'exécution de la boucle "interne" (celle avec j), le plus petit élement entre ceux en position i à n se retrouve en position i. En effet, à chaque comparaison, le plus petit des deux éléments se retrouve à gauche. À la fin, c'est forcément le plus petit des éléments parcourus qui est le plus à gauche, donc en position i.
Avant la toute dernière itération de la boucle externe, les éléments en position 1 à $n-2$ sont triés. On a $i = n-1$ et $j = n$. Il n'y a plus que les 2 derniers éléments qui peuvent être mal placés. Si c'est le cas, il suffit de les inverser. Dans tous les cas, à la fin de l'exécution, la boucle est bien triée.

Complexité

Pour déterminer la complexité, il faut regarder combien de tests sont effectués. Le nombre d'échanges est compris entre 0 et le nombre de tests. Les échanges sont faits en temps constant, donc c'est le nombre de tests qui donnera une idée du pire des cas.
Comme il y a 2 boucles dans lesquelles on parcourt le tableau, on peut en déduire que la complexité est, au pire, $O(n^2)$. Mais on peut donner une valeur plus précise. Pour la boucle externe, il y a $n-1$ itérations. Pour la boucle interne, il y en a $n-1$ tests, puis $n-2$, puis $n-3$ et ainsi de suite jusqu'à la dernière itérations où il n'y en a qu'un. Le nombre de tests est donc :

$$S = (n-1) + (n-2) + (n-3) + \ldots + 1$$

Pour faire cette somme, on peut utiliser une technique simple.
On remarque que $(n-1)+1 = n$, $(n-2)+2 = n$, ...,$1 + (n-1) = n$. On a donc :

$$2S = (n-1)+1+(n-2)+2+(n-3)+3+\ldots+1+(n-1) = (n-1)\times n \iff S = \frac{n(n-1)}{2}$$

Ce qui confirme que la complexité est bien $O(n^2)$.

Exercice 2 : Dans chaque cas, déterminer le nombre de tests effectués pour trier un tableau de taille n.

1) $n = 10$ 2) $n = 20$ 3) $n = 100$ 4) $n = 10\,000$

Exercice 3 :

1) Donner un exemple de tableau de longueur 2 qui nécessite le nombre maximum d'échanges possible.
2) Donner un exemple de tableau de longueur 3 qui nécessite le nombre maximum d'échanges possible.
3) Donner un exemple de tableau de longueur 4 qui nécessite le nombre maximum d'échanges possible.
4) Dans le cas général, quel type de tableau va nécessiter le nombre maximum d'échanges ?

En pratique, le tableau est souvent trié avant la dernière itération. Dans ce cas, lors des dernières itérations, aucun échange n'est effectué. Il est possible de rajouter une optimisation qui améliore la complexité. Il suffit d'avoir une variable booléenne qui vaut Faux au début de la boucle externe et passe à Vrai dès qu'un échange est effectué. Si à la fin de l'exécution de la boucle interne la variable vaut toujours Faux, on arrête l'exécution de la fonction.
Avec cette optimisation, si le tableau est déjà trié, il ne sera fait qu'un seul parcours du tableau. La complexité est alors linéaire.

Dans le pire des cas, il y aura quand même $\frac{n(n-1)}{2}$ échanges.

Sur un tableau choisi au hasard, le nombre moyen d'échanges est de $\frac{n(n-1)}{4}$. Ce qui est la moitié de la version non optimisée.
Lorsqu'on parle de la complexité d'un algorithme, on parle en général de sa complexité dans le pire des cas. Mais parfois, on précise sa complexité moyenne lorsque les “pires” des cas sont rares ou ont une faible probabilité d'arriver.

D'autres types de tris

Dans l'exemple nous avons considéré que “trier” voulait dire “ranger des nombres dans l'ordre croissant”. On peut tout à fait imaginer trier dans l'ordre décroissant, en mettant le maximum au début du tableau. Mais on peut aussi trier des mots dans l'ordre alphabétique, comme dans un dictionnaire. On peut également trier des n-uplets. Dans ce cas, on tri d'abord par rapport aux premiers éléments, puis en cas d'égalité en fonction des deuxièmes éléments et ainsi de suite. On parle alors d'**ordre lexicographique**.

Exercice 4 : Trier les tableaux suivants en utilisant l'ordre demandé :

1) `[5, 1, 5, 9, 91]` dans l'ordre décroissant.
2) `["baton", "avion", "ami", "bateau", "coq"]` dans l'ordre alphabétique.
3) `[(7, 1), (3, 9), (3, 1), (6, 15), (6, 7)]` dans l'ordre lexicographique.

Il est également possible d'imaginer des tris avec d'autre types d'ordres. Par exemple, trier les mots d'abord par longueur puis par ordre alphabétique. Ainsi `"ami"` serait avant `"coq"` qui serait avant `"avion"`.
On peut aussi trier les n-uplets en spécifiant l'ordre des colonnes que l'on regarde. Par exemple pour des dates notées `(31, 1, 2020)`, on commencerait par trier par rapport aux années (la 3e colonne), puis les mois (la 2e colonne), puis par rapport au jour.
Il est également possible d'inventer des tris qui sont basés par rapport à une formule mathématiques. On peut, par exemple, trier des points en fonction de leur distance par rapport à l'origine du repère.

Efficacité des algorithmes de tri

Les algorithmes de tri “naturels” ont généralement une complexité quadratique, que ce soit dans le pire des cas ou en moyenne. Par contre, il existe des algorithmes de tri qui ont une complexité en $O(n\log(n))$. C'est-à-dire presque linéaire. C'est notamment le cas du tri fusion qui consiste à découper le tableau en petits tableaux qui sont triés indépendamment. Le tableau est alors reconstruit au fur et à mesure, car il est facile de fusionner des tableaux déjà triés.

Le tri par sélection

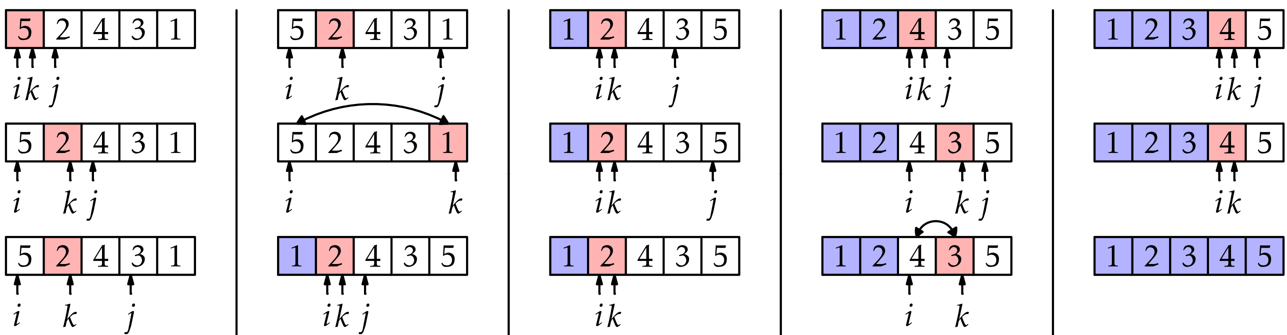
L'algorithme de tri à bulles, est simple à programmer, mais peu efficace car il effectue un grand nombre de comparaisons et d'échanges.

L'algorithme de **tri par sélection** lui, cherche à minimiser le nombre d'échanges nécessaires. L'idée est de chercher le plus petit élément et de le mettre en première position, puis le deuxième plus petit et le mettre en deuxième position et ainsi de suite.

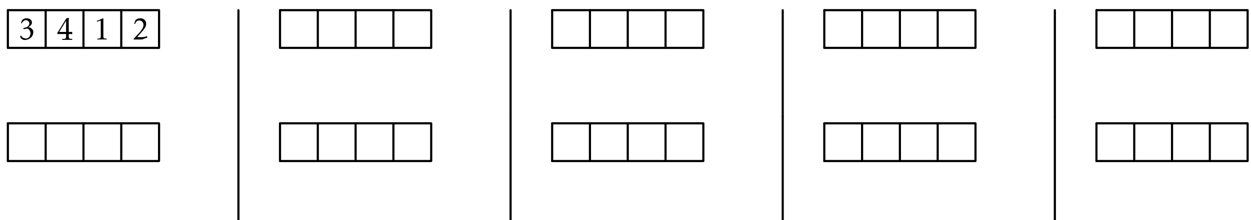
L'algorithme est ci-contre. La boucle externe est composée de 2 parties. Tout d'abord, on cherche la position k du plus petit élément entre i et n. Une fois trouvé, on inverse les éléments en position i et k. Voici un exemple d'exécution :

Algorithme du tri par sélection :

Pour i allant de 1 à $n-1$:
- $k \leftarrow i$
- **Pour** j allant de $i+1$ à n :
  - **Si** $T[j] < T[k]$:
    - $k \leftarrow j$
- Échanger($T[i]$, $T[k]$)



Exercice 5 : Donner les étapes correspondant au tri du tableau ci-dessous :

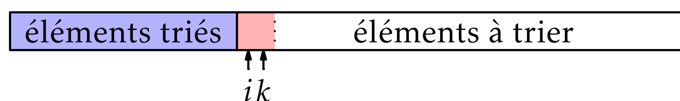


Correction et complexité du tri par sélection

La terminaison de l'algorithme ne pose pas de problème puisque les boucles sont toutes les deux bornées.

Pour la correction, on remarque qu'avant chaque tour de la boucle externe, les éléments en position 1 à $i-1$ sont triés et en bonne position. Ce sera notre invariant de boucle.

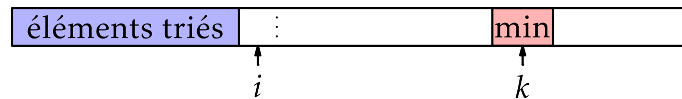
État du tableau au début d'un tour de la boucle externe :



Pour démontrer que cette propriété est bien un invariant de boucle, il faut également remarquer qu'à la fin de l'exécution de la boucle interne, le minimum des éléments entre i et n est

en position k. Lorsqu'on permute les éléments en position i et k, on s'assure que l'élément en position i est bien placé pour la prochaine itération.

État du tableau à la fin de la boucle interne :



Pour la complexité, comme pour le tri à bulles, le nombre de tests à effectuer est :

$$(n-1)+(n-2)+\ldots+1=\frac{n(n-1)}{2}$$

La complexité de cet algorithme de tri est donc quadratique. Par contre, le nombre d'échanges, lui est toujours $n-1$. Il y a donc un nombre linéaire d'échanges. Cet algorithme de tri est avantageux si l'échange de valeurs est coûteux en temps ou en mémoire.
Il est également possible de l'améliorer en rajoutant un test pour s'assurer que $i \neq k$ avant de faire l'échange. Ainsi, si le minimum est déjà en bonne position, pas besoin de faire d'échange. Dans ce cas, le nombre d'échanges va de 0 en $n-1$ en fonction du tableau.

Exercice 6 : On suppose qu'on a rajouté l'optimisation pour éviter de faire l'échange si $i = k$. Donner des exemples de tableaux nécessitant le nombre maximum d'échanges, avec 2, 3, 4 et n éléments.

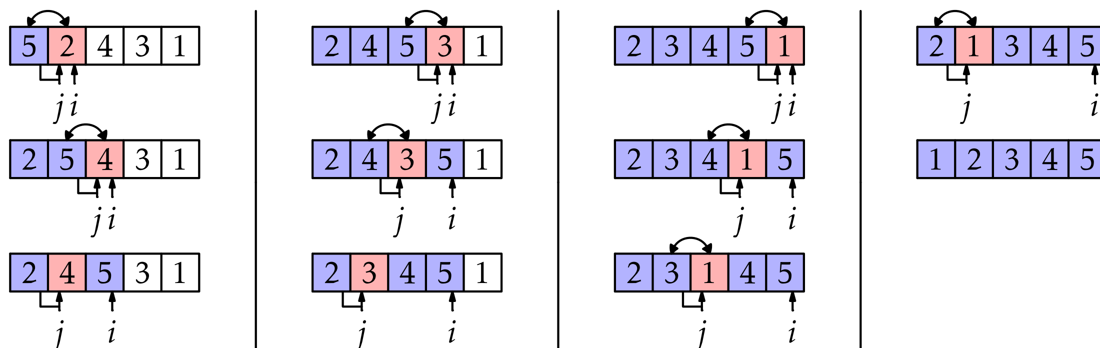
Le tri par insertion

Le **tri par insertion** a pour but de minimiser le nombre de comparaisons nécessaires. L'idée consiste à parcourir la liste de gauche à droite en plaçant le nouvel élément au bon endroit parmi ceux de gauche. En gros, cela revient à trier les éléments au fur et à mesure qu'on les voit.

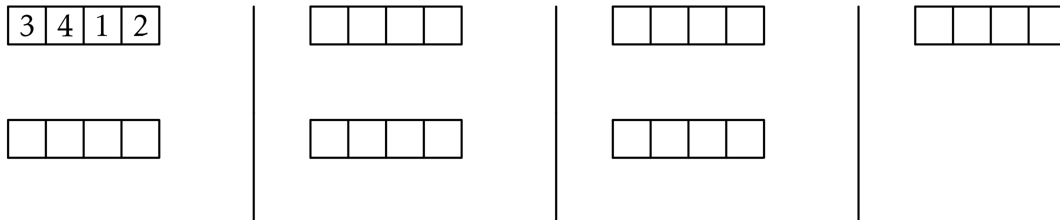
Algorithme du tri par insertion :

Pour i allant de 2 à n **:**
  $j \leftarrow i$
  Tant que $j > 1$ et $T[j-1] > T[j]$ **:**
    Échanger($T[j-1]$, $T[j]$)
    $j \leftarrow j-1$

Contrairement aux tris à bulles ou à sélection, les éléments déjà parcourus ne sont pas forcément dans leur position finale. Les prochains éléments viendront s'intercaler au fur et à mesure.
Voici un exemple d'utilisation :



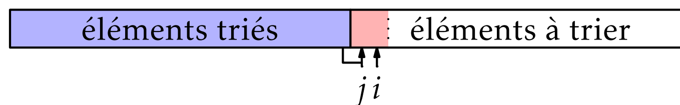
Exercice 7 : Donner les étapes correspondant au tri du tableau ci-dessous :



Correction et complexité du tri par insertion

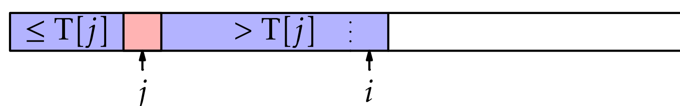
Puisqu'il y a une boucle non bornée, il faut s'assurer qu'elle termine forcément. Dans le cas du tri par sélection, puisque j décroit à chaque itération et que la boucle s'arrête dès qu'il est inférieur ou égal à 1, la boucle se termine forcément.
Pour la correction, il faut remarquer qu'avant chaque itération de la boucle externe, tous les éléments en position 1 à $i-1$ sont classés dans l'ordre croissant, même s'ils ne sont pas forcément en position définitive.

État du tableau au début d'un tour de la boucle externe :



Pour s'assurer que cette propriété est bien un invariant de boucle, il faut s'assurer que la boucle interne met l'élément en position i au bon endroit parmi les premiers éléments. Si le début du tableau est bien trié, cet élément se déplacera vers la droite en s'arrêtant soit en première position si c'est le minimum, soit après le premier élément qui est inférieur. Il est lui-même inférieur à tous les éléments à droite qui sont déjà triés. La liste des i premiers éléments est donc bien triée.

État du tableau à la fin de la boucle interne :



La complexité de cet algorithme est beaucoup plus difficile à estimer que pour les autres, à cause de la boucle non bornée. Dans le pire des cas, on peut remarquer qu'il y a, encore une fois, $\dfrac{n(n-1)}{2}$ tests et échanges. Pour déterminer précisément ce nombre, il suffit de compter pour chaque élément du tableau combien sont supérieur et se trouvent avant lui. Le nombre d'échanges nécessaires est la somme de ces nombres. Le nombre de tests est lui égal à la somme du nombre d'échanges et de $n-1$.
Au minimum, si la liste est déjà triée, il y a donc $n-1$ tests et aucun échanges. En fait, si le tableau est “presque trié”, le coût est linéaire, ce qui en fait un des meilleurs algorithmes de tris dans ces cas là.
La complexité de cet algorithme varie donc entre linéaire et quadratique selon les cas.

Exercice 8 :

1) Déterminer le nombre d'échanges à faire pour le tableau suivant : `[7, 9, 1, 15, 3, 47]`
2) Déterminer le nombre d'échanges à faire pour le tableau suivant : `[87, 11, 7, 4, 8, 22]`
3) Quel type de tableaux correspond au pire des cas pour cet algorithme ?