Spécifications pour les algorithmes de tri. On dispose d'un tableau `tab` de `n` éléments. L'algorithme, prenant en entrée le tableau `tab` doit renvoyer le tableau trié par ordre croissant.

1 Tri par sélection

1.1 Principe général

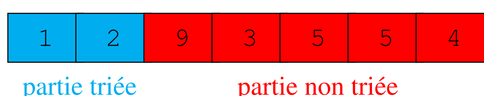
Le *tri par sélection* est un algorithme efficace lorsqu'il s'agit de trier un petit nombre d'éléments et particulièrement intuitif. Il s'inspire de la manière dont on pourrait classer des copies par ordre croissant de note :

- on choisit la copie dont la note est minimale et on la place face retournée sur la table ;
- on choisit ensuite parmi les copies restantes celle qui a la note la plus basse et on la place face retournée sur la précédente ;
- on réitère ce procédé jusqu'à ce qu'il n'y ait plus de copies à trier.

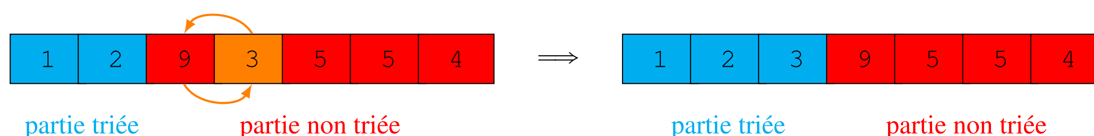
Ainsi, à tout moment, les copies retournées sur la table sont triées.

Dans le cas d'un tableau `tab` à trier, on procède de façon analogue. A chaque étape du tri :

1. le tableau `tab` est séparé en deux parties : une partie triée (à gauche) et une partie non triée que (à droite).



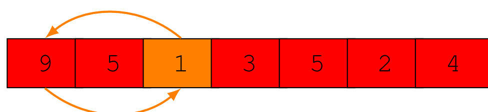
2. on choisit le plus petit élément de la partie non triée et on le place au début de la partie non triée de telle sorte que la partie triée soit augmentée d'un élément et la partie non triée soit diminuée d'un élément.



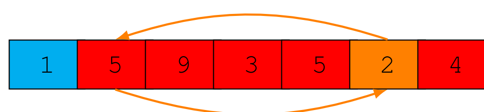
1.2 Un exemple pas à pas

L'exemple ci-dessous illustre le tri par sélection dans le cas du tableau `[9,5,1,3,5,2,4]`.

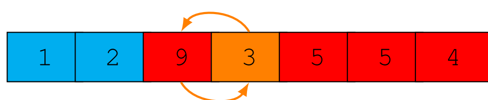
(a) Tableau initial



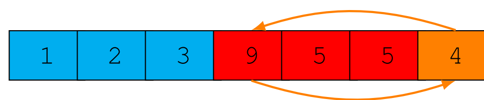
(b) Etape 1 : déplacement du minimum `1` en position 1



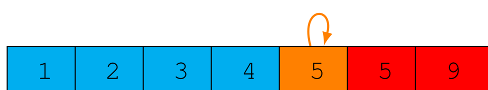
(c) Etape 2 : déplacement du minimum `2` en position 2



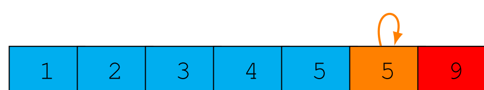
(d) Etape 3 : déplacement du minimum `3` en position 3



(e) Etape 4 : déplacement du minimum `4` en position 4



(f) Etape 5 : déplacement du minimum `5` en position 5



(g) Etape 6 : déplacement du minimum `5` en position 6

(h) Tableau final

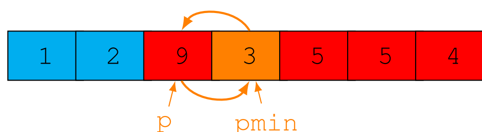
1.3 Algorithme

1.3.1 Enoncé

L'algorithme suivant permet d'effectuer un tri par sélection :

```
fonction tri_sélection(tab)
   n ← longueur de tab
   pour p allant de 1 à n-1
      pmin ← p
      pour j allant de p+1 à n
         si tab[j] < tab[pmin] alors
            pmin ← j
         fin si
      fin pour
      échanger tab[pmin] et tab[p]
   fin pour
   renvoyer tab
fin fonction
```

Le compteur `p` correspond à l'indice du premier élément de la partie non triée (position à laquelle il faut déplacer le plus petit élément de la partie non triée) et la variable `pmin` correspond à l'indice du plus petit élément de la partie non triée. On a ainsi le schéma suivant :



1.3.2 Preuve

Il s'agit ici de montrer, d'une part que l'algorithme se termine et d'autre part qu'il produit bien un tableau trié.

Terminaison : L'algorithme est composé de boucles `Pour`, on connaît le nombre de répétitions, donc l'algorithme se termine.

Correction partielle L'algorithme préserve l'invariant de boucle suivant : si $p \geqslant 1$, `tab` est trié entre les indices 1 et $p-1$, et tous les éléments restants sont supérieurs ou égaux à `tab[p-1]`.

1.3.3 Complexité

On choisit comme mesure élémentaire le nombre de comparaisons. Comme dans l'algorithme, on note n la longueur du tableau `tab`.

Théorème 1.1 *La complexité du tri par sélection est quadratique, c'est-à-dire de l'ordre de n^2. On dit qu'elle est en $O(n^2)$.*

Démonstration. Soit $p \in \{1, 2, ..., n-1\}$. Dans la boucle `Pour j allant de p+1 à n` on effectue $n-p$ comparaisons. Ainsi :

- Pour $p = 1$ on effectue $n-1$ comparaisons,
- Pour $p = 2$ on effectue $n-2$ comparaisons,
- ...
- Pour $p = n-2$ on effectue 2 comparaisons,
- Pour $p = n-1$ on effectue une seule comparaison.

L'algorithme effectue donc :

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

La complexité du tri par sélection est donc bien en $O(n^2)$.

■

2 Tri par insertion

2.1 Principe général

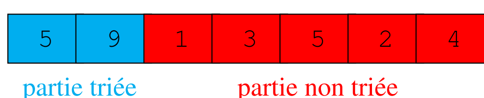
Le *tri par insertion* est un algorithme efficace lorsqu'il s'agit de trier un petit nombre d'éléments. Il s'inspire de la manière dont la plupart des gens tiennent des cartes à jouer :

- au début, la main gauche du joueur est vide et ses cartes sont posées en pile sur la table ;
- le joueur prend alors une par une les cartes situées sur la table, en choisissant toujours celle située sur le sommet de la pile, pour les placer dans sa main gauche ;
- pour savoir où placer une carte dans son jeu, le joueur la compare avec chacune des cartes déjà présentes dans sa main gauche pour l'insérer au bon endroit.

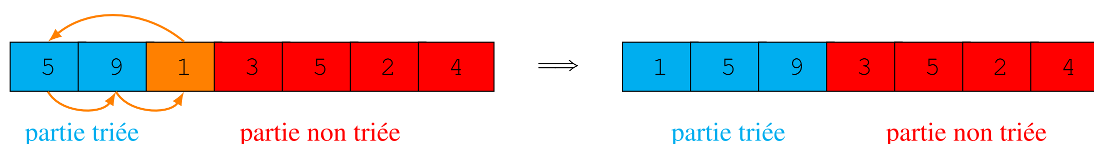
Ainsi, à tout moment, les cartes tenues par la main gauche sont triées et ces cartes étaient, à l'origine, les cartes situées au sommet de la pile sur la table.

Dans le cas d'un tableau `tab` à trier, on procède de façon analogue. A chaque étape du tri :

1. le tableau `tab` est séparé en deux parties : une partie triée que l'on suppose à gauche et une partie non triée que l'on suppose à droite ;

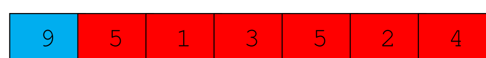


2. on insère le premier élément de la partie non triée dans la partie triée en décalant vers la droite tous les éléments de la partie triée qui lui sont supérieurs afin que la partie triée soit augmentée d'un élément et la partie non triée soit diminuée d'un élément.

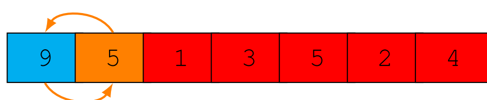


2.2 Un exemple pas à pas

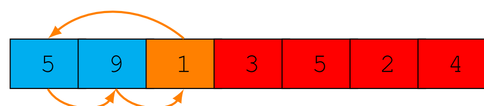
L'exemple ci-dessous illustre le tri par insertion dans le cas du tableau `[9,5,1,3,5,2,4]`.



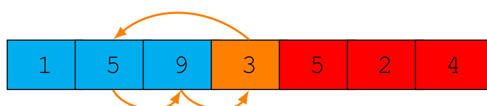
(a) Tableau initial



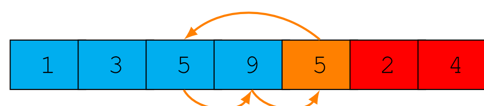
(b) Etape 1 : insertion de `5`



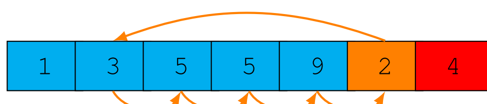
(c) Etape 2 : insertion de `1`



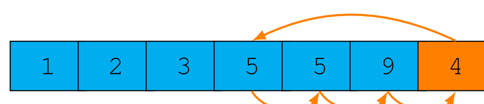
(d) Etape 3 : insertion de `3`



(e) Etape 4 : insertion de `5`



(f) Etape 5 : insertion de `2`



(g) Etape 6 : insertion de `4`

(h) Tableau final

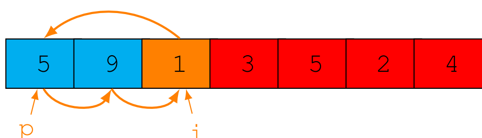
2.3 Algorithme

2.3.1 Enoncé

L'algorithme suivant permet d'effectuer un tri par insertion :

```
fonction tri_insertion(tab)
   n ← longueur de tab
   pour i allant de 2 à n
      temp ← tab[i]
      p ← i
      tant que p > 1 et tab[p-1] > temp
         tab[p] ← tab[p-1]
         p ← p - 1
      fin tant que
      tab[p] ← temp
   fin pour
   renvoyer tab
fin fonction
```

Le compteur `i` correspond à l'indice du premier élément de la liste non triée et la variable `p` correspond à l'indice de la position où il faut insérer l'élément `T[i]`. On a ainsi le schéma suivant :



2.3.2 Preuve

Il s'agit ici de montrer, d'une part que l'algorithme se termine et d'autre part qu'il produit bien un tableau trié.

Terminaison : La boucle `Pour` s'arrête (il y a $n-1$ répétitions), pour la boucle `Tant que` on utilise la variable p comme *variant de boucle* : p est strictement positive et décroissante, donc la boucle `Tant que` termine également.

Correction partielle L'algorithme préserve l'invariant de boucle suivant : si $i \geqslant 2$, `tab` est trié entre les indices 1 et $i-1$, et tous les autres éléments restent à trier.

2.3.3 Complexité

On choisit comme mesure élémentaire le nombre de comparaisons. Comme dans l'algorithme, on note n la longueur du tableau `tab`.

Théorème 2.1 *La complexité du tri par insertion est :*

- ⋆ *linéaire dans le meilleur des cas, c'est-à-dire de l'ordre de n. On dit qu'elle est en $O(n)$.*
- ⋆ *quadratique dans le pire des cas, c'est-à-dire de l'ordre de n^2. On dit qu'elle est en $O(n^2)$.*

Démonstration.

- ⋆ Dans le meilleur des cas, le tableau est déjà trié par ordre croissant : pour tout i variant de 2 à n, la boucle `Tant que` effectue exactement deux comparaisons (elle renvoie faux immédiatement).
  La boucle `Pour` nécessite une comparaison à chaque répétition pour l'incrémentation du compteur i. Elle est effectuée $n-1$ fois. Nous avons donc $3(n-1)$ comparaisons en tout. Ce qui donne bien une complexité linéaire.
- ⋆ Dans le pire des cas, le tableau est trié par ordre décroissant : la boucle `Pour` nécessite une comparaison à chaque répétition pour l'incrémentation du compteur i. Elle est effectuée $n-1$ fois.
  Soit $i \in \{2, 3, ..., n\}$. Dans la boucle `Tant que` on effectue $2i$ comparaisons (p varie de i à 1). Ainsi :
  - Pour $i = 2$ on effectue 4 comparaisons,
  - Pour $i = 3$ on effectue 6 comparaisons,
  - ...
  - Pour $i = n-1$ on effectue $2n-2$ comparaisons,
  - Pour $i = n$ on effectue $2n$ comparaisons.

  L'algorithme effectue donc :

$$4 + 6 + ... + (2n-2) + (2n) = 2(2 + 3 + .. + (n-1) + n) = 2 \times \frac{(n-1)(n+2)}{2} = n^2 + n - 2$$

  La complexité du tri par insertion est donc bien quadratique.

■