CHAPITRE 7 : Algorithmique 1

1 Parcours séquentiel d'un tableau 2
1.1 Histoire 2
1.2 Coût d'un algorithme 3
1.3 Complexité globale d'un algorithme 4
1.4 Algorithme de parcours 5
1.5 Preuve de correction 6
1.6 Algorithme de recherche du minimum et du maximum 8
1.7 Algorithme de calcul de la moyenne 8
2 Tri par sélection, tri par insertion 8
2.1 Tri d'un tableau par sélection 8
2.2 Tri par insertion 11

CHAPITRE 7 : Algorithmique 1

1 Parcours séquentiel d'un tableau

1.1 Histoire

Au cours de l'histoire sont d'abord apparus le calcul, puis les algorithmes et enfin les programmes.

- 1800	• En Mésopotamie (Irak actuel) **on sait calculer** sur les fractions, résoudre des équations du 2e ou du 3e degré, calculer les longueurs dans un triangle rectangle.
- 300	• Invention d'un algorithme pour calculer le plus grand diviseur commun (PGCD) de deux entiers a et b. C'est **l'algorithme d'Euclide**[1].
820	• **Al-Kwarizmi**[2] détaille toutes les étapes de calcul d'une racine d'un polynôme. Il s'intéresse aussi à prouver qu'un algorithme se termine, c'est à dire qu'il ne boucle pas une infinité de fois. C'est ce qu'on appelle **la preuve de terminaison** d'un algorithme.
1843	• **Ada de Lovelace**[3] s'intéresse à la *machine analytique* de son compatriote et contemporain **Charles Babbage**. Cette machine, jamais réalisée à cause de difficultés techniques de l'époque, devait pouvoir exécuter des calculs grâce à un programme enregistré sur de cartes en carton perforées. Elle réalise le **premier programme informatique** constitué d'opérations successives, définies à l'avance.

- **Algorithme**

C'est une procédure pour réaliser un calcul à partir d'actions simples. En entrée, il prend des valeurs et en renvoie d'autres en sortie.

- **Pseudo-code**

C'est l'écriture d'un algorithme en langage naturel. Il ne se réfère pas à un langage informatique particulier.

- **Programme**

C'est l'écriture d'un algorithme en un langage informatique précis comme Python, PHP ou JavaSript.

---

[1] **Euclide d'Alexandrie** vers 300 avant Jésus-Christ, est un mathématicien de la Grèce antique, auteur d'un traité de mathématiques. On retrouve son nom dans "division euclidienne", "distance euclidienne", "géométrie euclidienne".

[2] **Muhammad Inn Musa Al-Kwarizmi** né vers 780, mort vers 850 à Bagdad, est un mathématicien, géographe, astrologue et astronome persan.

[3] **Ada Lovelace**, de son nom complet Augusta Ada King, comtesse de Lovelace, née Ada Byron. Elle est née en 1815 et morte en 1852. Elle est une mathématicienne britannique, pionnière de la science informatique.

1.2 Coût d'un algorithme

- **Donald Knuth** [4] a introduit en 1962 la notion de *coût d'un algorithme* à la fois en *espace mémoire* utilisé sur la machine et en *temps de calcul*. C'est un **indicateur indépendant de la machine utilisée** (rapidité, type de mémoire, type de processeur etc.). Ainsi le temps de calcul n'est pas exprimé en secondes mais en **ordre de grandeur** du nombre d'opérations à faire pour réaliser tel algorithme sur un tableau de longueur n éléments par exemple. L'ordre de grandeur **est noté de façon abrégée $\boldsymbol{O}(\,)$.**

On parle de coût d'un algorithme. On dit aussi *complexité* d'un algorithme. On distingue donc :

- **La complexité spatiale** qui s'intéresse à la quantité de mémoire nécessaire pour exécuter un algorithme. Celle-ci est particulièrement critique pour les systèmes embarqués.
- **La complexité temporelle** qui sera exprimée en fonction de la taille n des données à traiter. Il s'agit de l'ordre du nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques, appels de fonctions) pour mener à bien un calcul.

Dans la suite de ce cours, on ne s'intéressera qu'à la complexité temporelle.

Exemples : On dispose d'un paquet de n cartes à jouer.

Un algorithme qui compare la valeur de chaque carte avec la 1re carte du paquet a une complexité de l'ordre de n, notée $O(n)$. Cela signifie que si on double la taille du paquet alors on double aussi la durée d'exécution. Si on multiplie par 10 la taille du paquet alors on multiplie par 10 la durée.

Un algorithme qui compare la valeur de chaque carte avec toutes les autres doit effectuer $n \times n = n^2$ comparaisons. Dans ce cas sa complexité est de l'ordre de n^2. On note la complexité $O(n^2)$. Dans ce cas si on double la taille du paquet alors on quadruple le temps d'exécution. Si on multiplie par 10 la taille du paquet, alors on multiplie par $10^2 = 100$ le temps d'exécution.

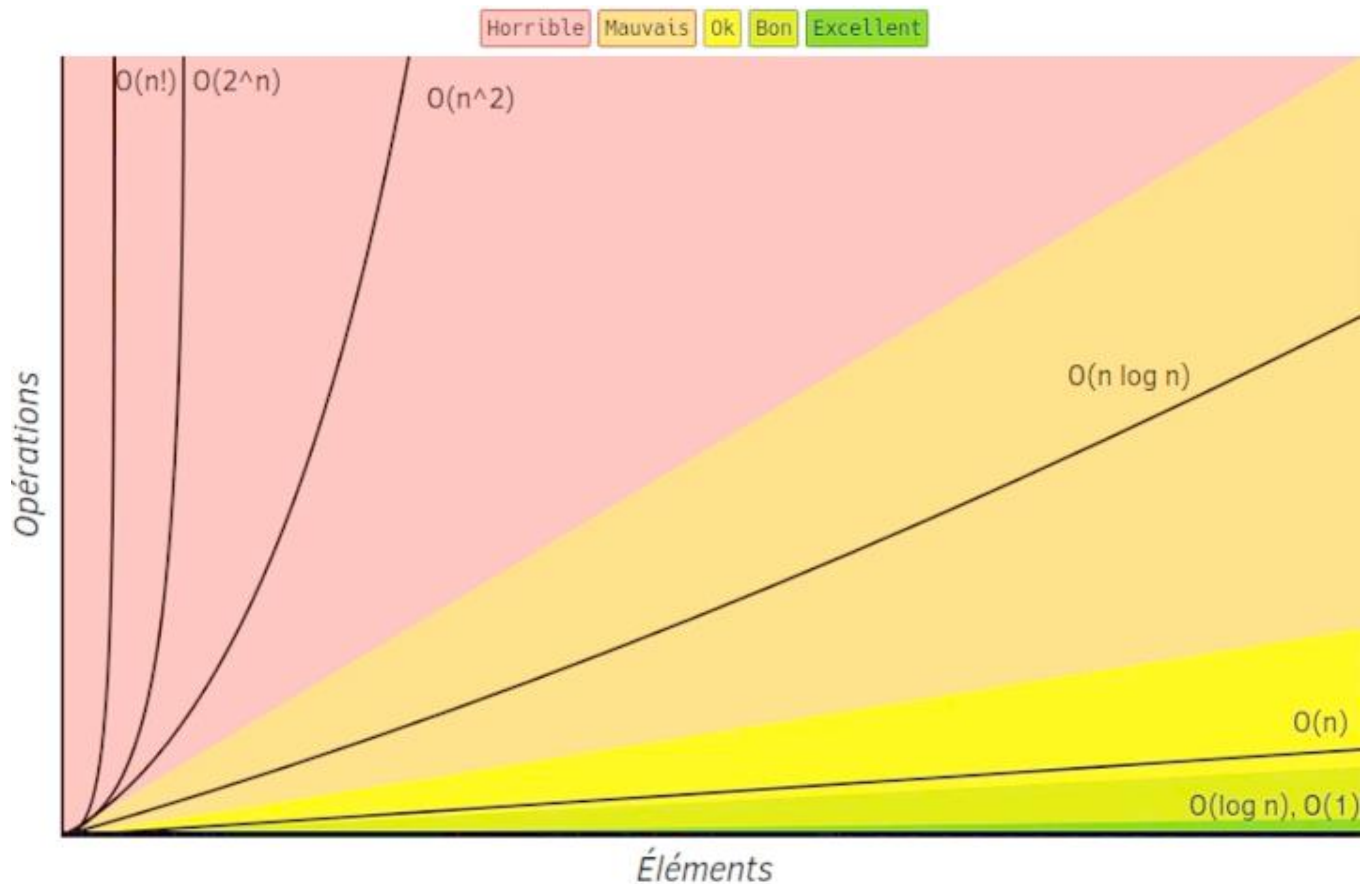
- Classement des algorithmes

On rencontre souvent des algorithmes de complexité temporelle de l'ordre de n ou de $n\log(n)$ où $\log$ est la fonction logarithme présente sur la calculatrice (croissance très lente). Dans ce cas l'algorithme est assez rapide (cas d'une complexité en $O(n)$).

On rencontre aussi des algorithmes de complexité temporelle de l'ordre de n^2 ou de 2^n . Dans ce cas l'algorithme est très lent.

---

[4] **Donald Knuth :** né en 1938, est un informaticien et mathématicien américain. Il est un des pionniers de l'algorithmique et a fait de nombreuses contributions dans plusieurs branches de l'informatique théorique.



Classement des algorithmes

Remarque

Plusieurs cas peuvent se présenter pour traiter n valeurs dans un algorithme :

S'il faut par exemple $4n + 2$ étapes alors l'expression est affine. Quand n devient très grand, c'est la contribution de $4n$ qui domine dans la somme. On retiendra que l'ordre de grandeur est n. On notera que la complexité est $C(n) = O(n)$. On dit que c'est une *complexité linéaire.*

S'il faut $3n^2 + 5n + 2$ étapes alors l'expression est du second degré. Quand n devient très grand, c'est la contribution de $3n^2$ qui domine dans la somme. On retiendra que l'ordre de grandeur est n^2. On notera que la complexité est $C(n) = O(n^2)$. On dit que c'est une *complexité quadratique*.

1.3 Complexité globale d'un algorithme

Il se peut qu'un algorithme soit composé de plusieurs blocs d'instructions. Dans ce cas, c'est la "plus mauvaise" complexité qui l'emporte.

Exemple

Un premier bloc a une complexité $C_1(n) = O(n^2)$ et un deuxième a une complexité $C_2(n) = O(n)$.

Alors, pour des valeurs de n élevées, c'est la complexité en $O(n^2)$ qui domine et donc la complexité globale de l'algorithme est aussi en $O(n^2)$.

1.4 Algorithme de parcours

Quand on souhaite parcourir une structure de données de manière itérative (élément par élément) pour rechercher un élément ou compter un nombre d'éléments, on peut choisir le **parcours total** ou le **parcours partiel**.

- **Parcours total**

Le parcours total est par exemple mis en œuvre lors d'un **comptage d'occurrence** (on compte combien de fois telle lettre se trouve dans un texte par exemple) la **boucle 'pour'** est à privilégier.

Exemple

En pseudo code on a :

```
Algorithme : parcours(tableau, valeur)
	compteur ← 0
	pour i allant de 0 à longueur(tableau) - 1
		si tableau[i] = valeur alors
			compteur = compteur + 1
	renvoyer compteur
```

- Complexité temporelle

On compte le nombre d'opérations élémentaires :

En dehors de la boucle il y a **1** affectation de valeur à la variable `compteur`.

Dans la boucle, à chaque tour de boucle il y a :

- 1 incrémentation du compteur de boucle `pour`
- 1 affectation d'une valeur du tableau à la variable `tableau[i]`.
- 1 comparaison dans le test de la condition du si.
- Dans le "pire des cas" (le cas où la liste est pleine de valeurs égales à la valeur recherchée) , 1 addition `compteur + 1` et 1 affectation dans l'instruction `compteur = compteur + 1`.

Donc cela fait $1 + n(1 + 1 + 1 + 2) = 5n + 1$ opérations élémentaires pour traiter un tableau de longueur n éléments.

On retiendra que l'ordre de grandeur de la complexité temporelle est n c'est à dire $O(n)$.

- **Parcours partiel**

Le parcours partiel est par exemple mis en œuvre lors d'une **recherche de l'existence** d'une occurrence (on veut savoir si une certaine structure comme un tableau contient un certain élément). Dans ce cas

la **boucle 'tant que'** est à privilégier. En effet dès qu'on a trouvé l'élément on arrête de chercher et on sait que l'élément est présent.

Exemple

En pseudo code on a :

```
Algorithme : cherche(tableau, valeur)
      trouve ← Faux
      i ← 0  #  Initialisation du compteur de boucle while
      tant que trouve = Faux et i < longueur(tableau)
            si tableau[i] = valeur alors
                  trouve ← Vrai
            i ← i + 1  #  Incrémentation du compteur de boucle while
      renvoyer trouve
```

- Complexité temporelle

On compte le nombre d'opérations élémentaires :

On peut avoir *le pire cas* (l'élément est absent, auquel cas, toute la structure doit être parcourue avant de savoir que l'élément ne s'y trouve pas).

On peut avoir *le meilleur cas* (l'élément est trouvé dans la première "case" de la structure, auquel cas, on arrête la recherche et on sait que l'élément cherché se trouve dans la structure de données. Mais comme ce n'est pas du tout certain d'avoir le meilleur cas, ni un cas favorable alors :

On retiendra la complexité dans *le pire cas.* Cela donne le maximum de nombre d'opérations auquel on peut s'attendre. La complexité dans *le pire cas* donne une bonne indication si on veut comparer plusieurs algorithmes entre eux.

On retiendra que, dans le cas où on parcourt une fois avec la boucle "tant que" tout le tableau de longueur n, alors la complexité temporelle de la recherche d'une occurrence est $O(n)$.

1.5 Preuve de correction

Afin de prouver qu'un algorithme est correct (donner une preuve de correction), on doit passer en revue deux points :

1. Prouver qu'il ne boucle pas de façon infinie : c'est **la preuve de terminaison**.
2. Prouver qu'il fournit le résultat attendu : c'est **la preuve de correction partielle**.

Exemples de preuve de terminaison Prouvez que les deux algorithmes suivants se terminent :

1)

```
Algorithme : parcours(tableau, valeur)
        compteur ← 0
        pour i allant de 0 à longueur(tableau) - 1
                si tableau[i] = valeur alors
                        compteur = compteur + 1
        renvoyer compteur
```

Réponse

C'est une boucle `pour` donc elle se termine toujours.

2)

```
Algorithme : cherche(tableau, valeur)
        trouve ← Faux
        i ← 0  #  Initialisation du compteur de boucle
        tant que trouve = Faux et i < longueur(tableau)
                si tableau[i] = valeur alors
                        trouve ← Vrai
                i ← i + 1  #  Incrémentation du compteur
        renvoyer trouve
```

Réponse

C'est une boucle `tant que` donc on doit examiner la condition de bouclage pour voir si elle *devient fausse* en un nombre *fini* de tours de boucle.

Avant l'entrée dans la boucle :

trouve vaut `Faux` et `i` vaut `0`

donc

`trouve = Faux` est `Vrai` et **`longueur(tableau) - i`** > 0 est `Vrai`

Donc la condition de maintien dans la boucle "tant que" a la valeur booléenne `Vrai.`

A chaque tour de boucle, `i` est incrémenté de 1 grâce à l'instruction `i ← i + 1` , donc

`longueur(tableau) - i` décroit strictement

Quand la condition **`longueur(tableau) - i`** > 0 devient fausse, la boucle se termine.

Vocabulaire

L'entier naturel **`longueur(tableau) – i`** décroît strictement à chaque tour de boucle.

Il est appelé **variant de boucle.**

1.6 Algorithme de recherche du minimum et du maximum

Exemple

```
Algorithme : recherche_maxi(tableau)
      maxi ← tableau[0]
      pour i allant de 1 à longueur(tableau) - 1
            si élément > maxi alors
                  maxi ← élément
      renvoyer maxi
```

- Terminaison : l'algorithme contient une boucle pour, donc il se termine.
- Coût : C'est celui d'un parcours total donc $O(n)$.

1.7 Algorithme de calcul de la moyenne

Il s'agit de parcourir toute la structure. On accumule dans une variable somme toutes les valeurs rencontrées. On utilisera donc une boucle `pour`. Donc l'algorithme se termine et sa complexité est égale à $O(n)$. Autrement dit **son coût est linéaire**.

2 Tri par sélection, tri par insertion

2.1 Tri d'un tableau par sélection

- **Méthode**

- Le tri par sélection consiste à partir du premier élément d'indice $i = 0$ (flèche rouge).
- Provisoirement, l'indice du minimum prend la valeur de i
- On compare les autres éléments pour j (flèche jaune) allant de $i + 1$ à longueur(tableau) -1.
- Si tableau[j] <tableau[indice du minimum] alors indice du minimum (flèche bleue) prend la valeur de *j*.

est plus petit que , mise à jour de

← Étape 8 sur 53 →

indique la carte en cours de traitement.

indique la comparaison en cours.

indique la carte candidate à l'échange.

- Enfin si indice du minimum est différent de i alors **on sélectionne tableau[indice du minimum]** et on l'échange avec tableau[i].

A ce stade, on est certain que le 1er élément du tableau est le plus petit. Le début du tableau (qui ne compte qu'un élément pour le moment) est trié.

- On recommence le parcours du tableau en prenant le 2e élément d'indice $i = 1$ (flèche rouge).

On a donc une boucle `pour` extérieure pour i allant de 0 à longueur du tableau -2 (i va de 0 à 6 dans l'exemple de 8 cartes).

A l'intérieur de cette boucle `pour` est imbriquée une deuxième boucle pour j allant de $i + 1$ à longueur du tableau -1 (c'est à dire la fin du tableau).

A chaque tour de boucle `pour` i ..., la partie triée du tableau située à gauche augmente de 1 élément.

A la fin du tour où i vaut longueur(tableau) – 1, le tableau est entièrement trié par ordre croissant.

```
Algorithme : tri_par_selection(tableau)
      pour i allant de 0 à longueur(tableau) - 2
            indice du minimum ← i
            pour j allant de i + 1 à longueur(tableau) - 1
                  si tableau[j] < tableau[indice du minimum]  alors
                        indice du minimum ← j
            échanger tableau[i] et tableau[indice du minimum]
      renvoyer tableau
```

- **Coût :** C'est celui d'une boucle `pour` imbriquée dans une boucle `pour` donc $O(n^2)$. Le coût est "quadratique", c'est à dire que si on double la longueur de la liste à trier, alors la durée du tri est multipliée par $2^2 = 4$. Si on multiplie par 10 la longueur de la liste à trier, alors la durée du tri est multipliée par $10^2 = 100$ etc. On est donc dans la catégorie des coûts très élevés !

- **Implémentation en Python**

Remarque : Pour parler des "tableaux" les concepteurs de Python ont choisi d'utiliser le terme de "list" ("liste" en français). Dans ce qui suit, la variable `tableau_a_trier` est donc du type list.

```python
def tri_selection(tableau_a_trier):
    for i in range(0, len(tableau_a_trier)-1):
        indice_mini = i
        for j in range(i+1, len(tableau_a_trier)):
            if tableau_a_trier[j] < tableau_a_trier[indice_mini]:
                indice_mini = j
        tableau_a_trier[i], tableau_a_trier[indice_mini] = tableau_a_trier[indice_mini], tableau_a_trier[i]
    return tableau_a_trier
```

- Preuves de correction

Les preuves de corrections sont en deux parties : 1) la terminaison et 2) la correction partielle grâce à une propriété **invariante** au cours du déroulement de l'algorithme.

<table>
<tr><th colspan="2">Correction totale</th></tr>
<tr><th>Terminaison</th><th>Correction partielle</th></tr>
<tr><td>

- L'algorithme contient deux boucles pour, donc il se termine.

</td><td>

Soit la propriété :
"A la fin du n ième tour de boucle, les n premiers éléments du tableau sont triés et sont plus petits que ceux qui restent à droite du tableau".

- *Montrons que la propriété est vraie au premier rang*

A la fin du premier tour de la boucle `pour` le sous-tableau ayant pour indice [0] contient un seul élément.
Donc on peut dire qu'il est trié puisqu'il n'a qu'un élément. Et cet élément est plus petit que ceux qui restent.
Donc la propriété est
vraie à la fin du premier tour de boucle.

- *Montrons que la propriété est héréditaire*

Supposons qu'à la fin du k ième tour de boucle, les k premiers éléments du tableau soient triés.

L'étape suivante consiste à trouver le plus petit élément dans ceux qui restent et de l'amener juste à la suite du sous-tableau trié.

Donc à la fin du $k+1$ ième tour de boucle, les $k+1$ premiers éléments du tableau sont triés et sont plus petits que ceux qui restent.

- *Conclusion* :

La propriété est vraie après la première étape.
De plus elle est héréditaire.
Donc elle vraie quel que soit le nombre de tours de boucles (elle est **invariante**)
En particulier, elle est vraie à la fin de l'algorithme près $n - 1$ tours de boucle.
Donc tout le tableau de taille n est trié en fin d'algorithme.

</td></tr>
</table>

Remarque

La preuve de correction partielle utilise **un invariant de boucle**. Il est souvent difficile de trouver la propriété invariante qui permet de prouver qu'un algorithme est correct.

2.2 Tri par insertion

- **Méthode**

- Le tri par insertion consiste à partir du 2e *élément* du tableau donc d'indice $i = 1$ (flèche rouge).
- Le sous-tableau ayant pour indices $[0, 1, \dots, i-1]$ est provisoirement trié. Donc ici le sous-tableau comportant la seule carte "As" est trié.
- Le but est d'insérer tableau[i] (ici le valet de pique) que nous appelons "clé" à sa place dans la partie provisoirement triée.

Sous-tableau provisoirement trié.

État initial

← Étape 1 sur 39 →

↑ indique la carte à insérer. C'est la "clé" x.

indique la comparaison en cours.

- Pour effectuer la première comparaison, nous affectons à j la valeur de i donc ici la valeur 1.

Sous-tableau provisoirement trié.

État initial

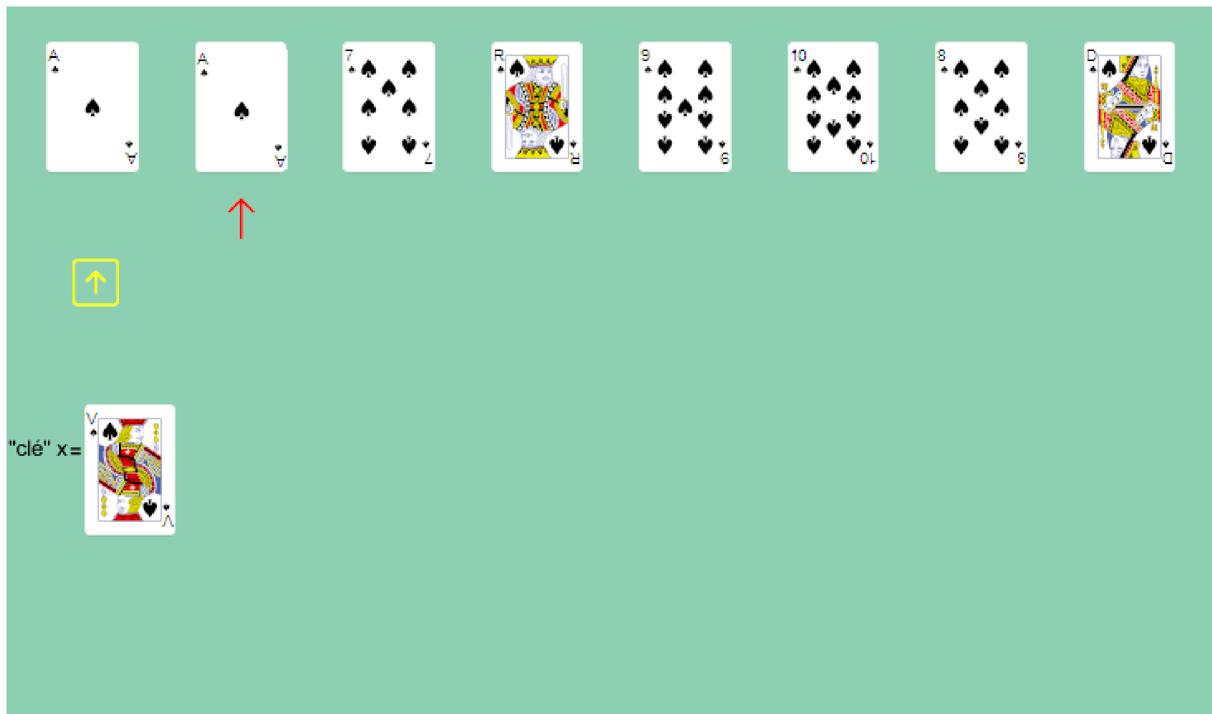
"clé" x=

← Étape 1 sur 39 →

↑ indique la carte à insérer. C'est la "clé" x.

indique la comparaison en cours.

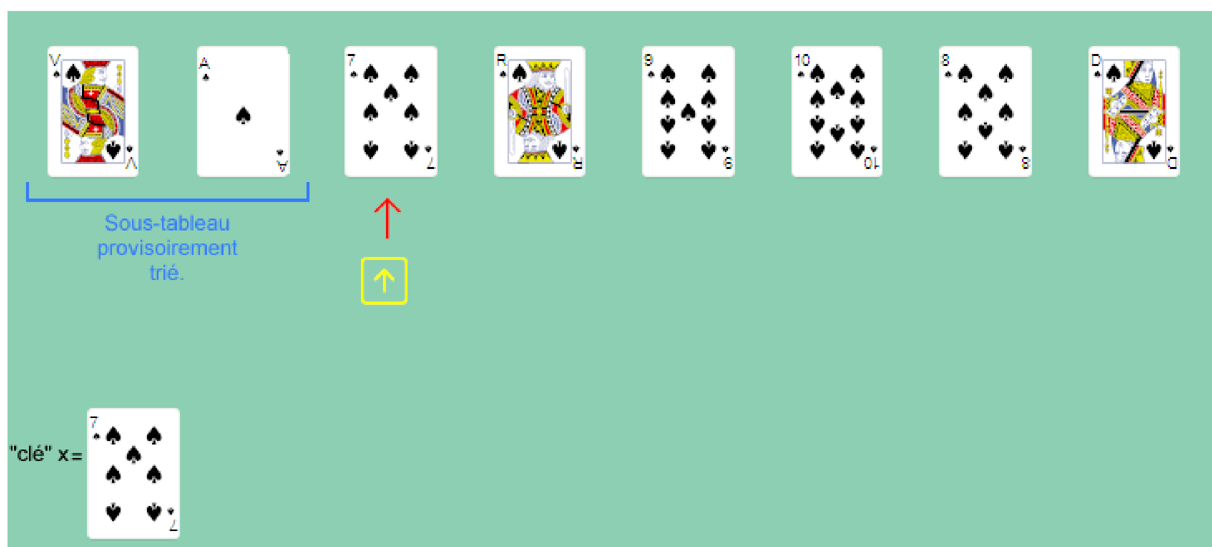
- On examine tableau[$j-1$]. Tant que $j > 0$ et tableau[$j-1$] $> x$, on recopie tableau [$j-1$] sur tableau[j] et on **décrémente** j d'une unité (on recule dans le tableau).



- Les valeurs du sous-tableau provisoirement trié qui sont strictement supérieures à la clé x se retrouvent donc décalées à droite d'un rang.

- Quand j atteint la valeur 0 (aucune valeur plus petite que la clé n'a été trouvée dans le sous-tableau) ou que la valeur de tableau[$j-1$] n'est pas supérieure à x, alors on sort du Tant que et tableau[j] prend la valeur de la clé x.

- A ce stade, le sous-tableau a vu son effectif augmenté d'une valeur. Cette valeur est la clé x. Et elle est à sa place. Donc le sous-tableau est provisoirement trié.



A chaque tour de boucle pour i ..., le sous-tableau déjà trié situé à gauche augmente de 1 élément.

Le tableau est entièrement trié à la fin de la boucle pour quand *i* pris la valeur longueur(tableau)-1.

Donc l'algorithme est composé d'une boucle pour principale et à l'intérieur se trouve imbriquée une boucle tant que. Les indices des éléments du tableau vont de 0 à longueur du tableau – 1.

```
Algorithme : tri_par_insertion(tableau)
    pour i allant de 1 à longueur(tableau) - 1  #  i = rang après le sous-tableau trié.
          j ← i  #  Initialisation de j avec i.
          x ← tableau[j]   #  La "clé" tableau[j] est stockée dans x.
          tant que j > 0 et x < tableau[j-1]
                tableau[j] ← tableau[j-1]  # Décalage à droite de tableau[j-1].
                j ← j - 1  # Décrémentation de j.
          tableau[j] ← x #  La "clé" est mise à sa place dans le sous-tableau trié.
    renvoyer tableau
```

- **Coût :** C'est celui d'une boucle tant que imbriquée dans une boucle pour donc $O(n^2)$. Le coût est "quadratique". Donc on est dans la catégorie des coûts très élevés.

Mais par rapport au tri par sélection, on remarque que si le tableau de départ est "presque trié", la boucle tant que sera exécutée un petit nombre de fois seulement. Donc ce tri est plus efficace. Il est souvent considéré comme le plus efficace sur des tableaux de petite taille ($n < 50$).

- **Implémentation en Python :**

```python
def tri_insertion(tableau_a_trier):
    for i in range(1, len(tableau_a_trier)):  #  i est le rang après le sous-tableau trié.
        j = i  #  Initialisation de j avec i.
        x = tableau_a_trier[j]  #  La "clé" tableau[j] est stockée dans x.
        while j > 0 and x < tableau_a_trier[j-1]:
            tableau_a_trier[j] = tableau_a_trier[j-1] # Décalage à droite de tableau[j-1].
            j = j - 1 # Décrémentation de j.
        tableau_a_trier[j] = x  #  La "clé" est mise à sa place dans le sous-tableau trié.
    return tableau_a_trier
```

- Preuves de correction

Les preuves de corrections sont en deux parties : 1) la terminaison et 2) la correction partielle grâce à une propriété **invariante** au cours du déroulement de l'algorithme.

Correction totale	
Terminaison	**Correction partielle**
• L'algorithme contient une boucle pour, donc qui se termine. • La boucle tant que contient **le variant de boucle *j*** qui est initialisé à i puis qui décroit strictement à chaque tour de boucle par l'instruction `j ← j - 1` La condition de maintien dans la boucle `j > 0` devient fausse après un certain nombre de tours de boucle. L'algorithme se termine.	Soit la propriété : "A la fin du n ième tour de boucle, les $n+1$ premiers éléments du tableau sont triés ". • *Montrons que la propriété est vraie au rang zéro.* Avant de faire le premier tour de boucle, le sous-tableau constitué du premier élément est considéré comme trié (puisqu'il est tout seul). Donc la propriété est **vraie au rang 0.** • *Montrons que la propriété est héréditaire* Supposons qu'à la fin du k ième tour de boucle, les $k+1$ premiers éléments du tableau soient triés. L'étape suivante consiste à prendre le $k+2^e$ élément (le premier de la partie non triée qu'on appelle "clé" et à le ranger à sa place dans le sous-tableau déjà trié de $k+1$ éléments. Donc après cela, les $k+2$ premiers éléments du tableau sont triés. • *Conclusion* : La propriété est vraie quel que soit le nombre de tours de boucles. En particulier, elle est vraie à la fin de l'algorithme près $n-1$ tours de boucle. Donc "A la fin du $n-1$ ième tour de boucle, les n éléments du tableau sont triés ". On a prouvé la correction partielle grâce à **l'invariant de boucle** "A la fin du n ième tour de boucle, les $n+1$ premiers éléments du tableau sont triés ".