# Méthodes algorithmiques

Sophie Pinchinat
sophie.pinchinat@irisa.fr

IRISA, Université de Rennes 1

UE ALG – année 2015-2016

# Table des Matières

# Conception et analyse des algorithmes

- Conception : formuler le problème à résoudre avec une précision mathématique suffisante de manière à poser une question concrète et définir un algorithme permettant de résoudre ce problème
- Analyse : montrer la correction de cet algorithme et donner une borne sur son temps d'exécution et/ou l'espace mémoire qu'il utilise pour établir son efficacité.
- Importance du choix d'un modèle technologique d'exécution.

  Nous prenons le modèle RAM (Random Access Memory) à processeur unique

En pratique on s'appuie sur quelques techniques de conception fondamentales, très utiles pour évaluer la complexité inhérente du problème et pour formuler un algorithme qui le résout.

# Techniques de conception

- La familiarisation avec ces techniques est un processus progressif
- Il faut une expérience pour reconnaître/flairer le “genre” du problème
- Savoir apprécier les nuances subtiles de la nature du problème qui induisent des effets déterminants sur la difficulté du problème.

**Exemple**

Circuit Eulérien (facile) vs. Voyageur de Commerce (difficile).

**Exercice**

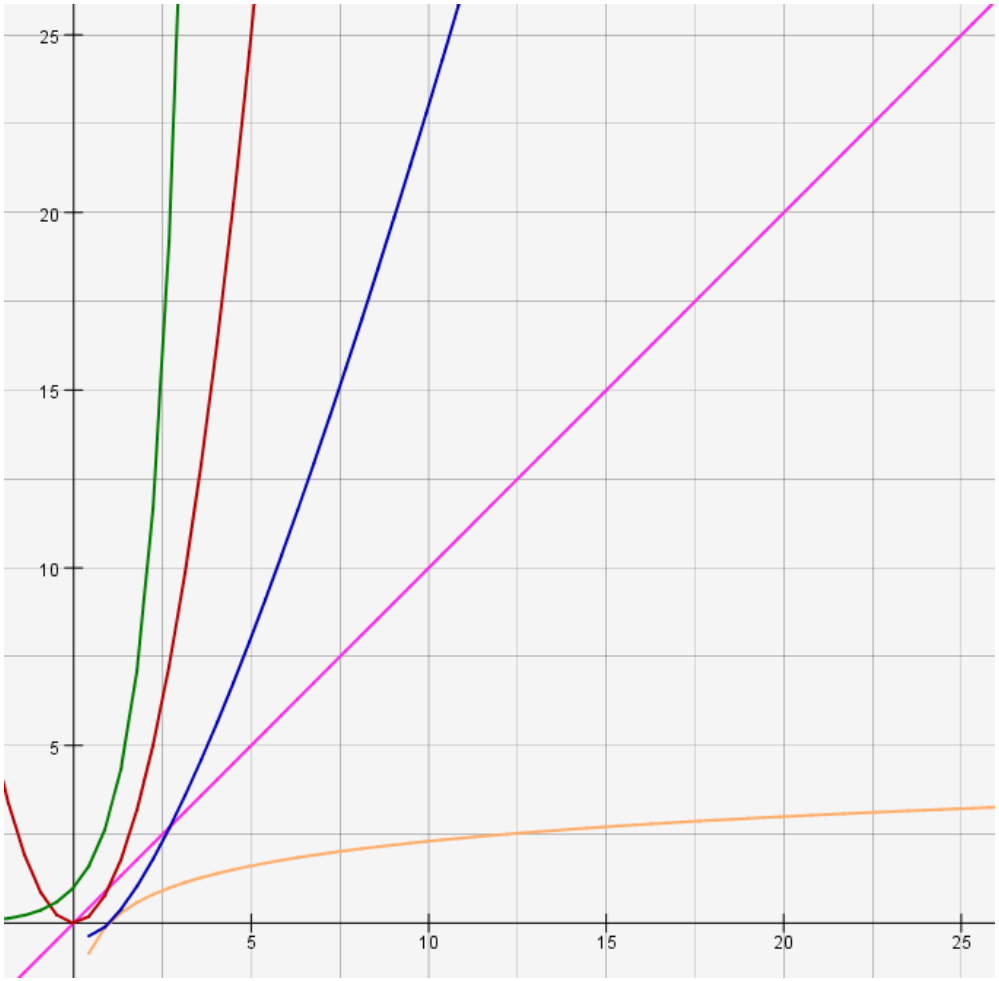
Trouver par exemple dans [DPV06] la définition de ces deux problèmes et ce qu’on sait à leur sujet.

# Plan du cours

- Diviser pour Régner (3)
- Algorithmes Gloutons (3)
- Programmation Linéaire (2)
- Programmation Dynamique (3)
- Essais Successifs et quelques notions d'heuristiques (1)
- Problèmes NP-complets (1)

# Table des Matières

# Quelques fonctions classiques




Complexités $\begin{cases} O(e^n) \\ O(n^2) \\ O(nlog(n)) \\ O(n) \\ O(log(n)) \end{cases}$

**Exercice**

Retrouver la bonne courbe.

# Ordre de grandeurs/Notations asymptotiques

Soit $g : \mathbb{N} \to \mathbb{R}^+$ une fonction donnée.

## Définition

$$\Theta(g(n)) = \{f(n) \mid \text{il existe des constantes positives } c_1, c_2 \text{ et } n_0 \text{ telles que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ pour tout } n \geq n_0\}$$

## Définition

$$O(g(n)) = \{f(n) \mid \text{il existe des constantes positives } c \text{ et } n_0 \text{ telles que } 0 \leq f(n) \leq c g(n) \text{ pour tout } n \geq n_0\}$$

## Définition

$$\Omega(g(n)) = \{f(n) \mid \text{il existe des constantes positives } c \text{ et } n_0 \text{ telles que } 0 \leq c g(n) \leq f(n) \text{ pour tout } n \geq n_0\}$$

# Ordres de grandeurs $\Theta, O, \Omega$

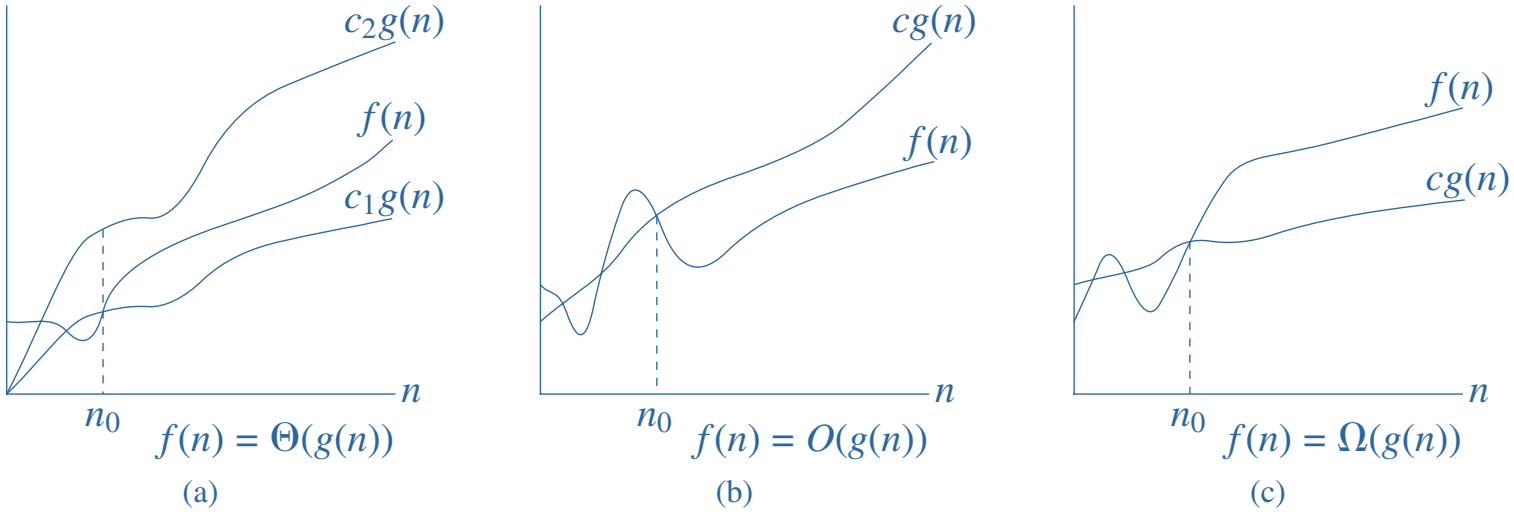



**Figure 3.1** Exemples de notations $\Theta$, $O$ et . Dans chaque partie, la valeur de $n_0$ est la valeur minimale possible ; n'importe quelle valeur supérieure ferait aussi l'affaire. **(a)** La notation $\Theta$ borne une fonction entre des facteurs constants. On écrit $f(n) = \Theta(g(n))$ s'il existe des constantes positives $n_0$, $c_1$ et $c_2$ telles que, à droite de $n_0$, la valeur de $f(n)$ soit toujours comprise entre $c_1g(n)$ et $c_2g(n)$ inclus. **(b)** La notation $O$ donne une borne supérieure pour une fonction à un facteur constant près. On écrit $f(n) = O(g(n))$ s'il existe des constantes positives $n_0$ et $c$ telles que, à droite de $n_0$, la valeur de $f(n)$ soit toujours inférieure ou égale à $cg(n)$. **(c)** La notation $\Omega$ donne une borne inférieure pour une fonction à un facteur constant près. On écrit $f(n) = \Omega(g(n))$ s'il existe des constantes positives $n_0$ et $c$ telles que, à droite de $n_0$, la valeur de $f(n)$ soit toujours supérieure ou égale à $cg(n)$.

# À vous de jouer !

## Exercice

Notez sur une feuille libre à votre nom les exercices que vous avez faits avec leur correction et leur référence bibliographique. Nous ramasserons en TD.

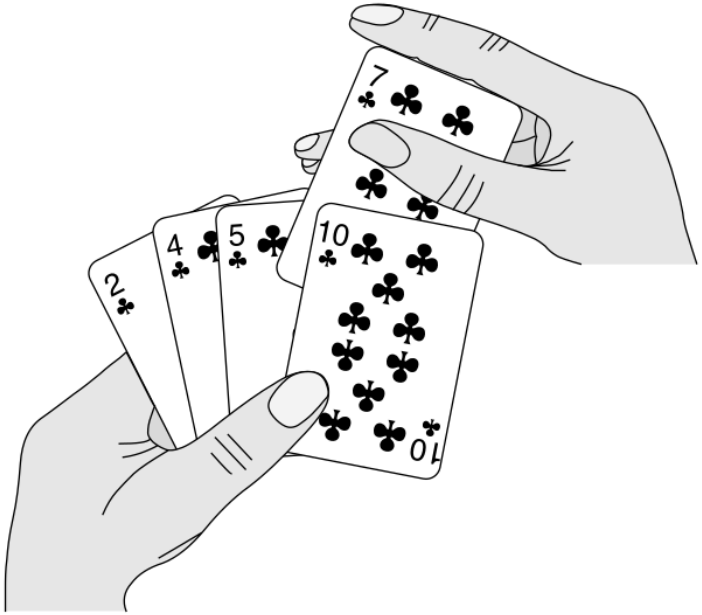
# Table des Matières

# Le problème du tri

## Problème (Le tri)

**Entrée :** un tableau $T$ d'entiers

**Sortie :** une permutation de $T$ triée



Une approche naturelle

# Un algorithme naturel : le tri par insertion

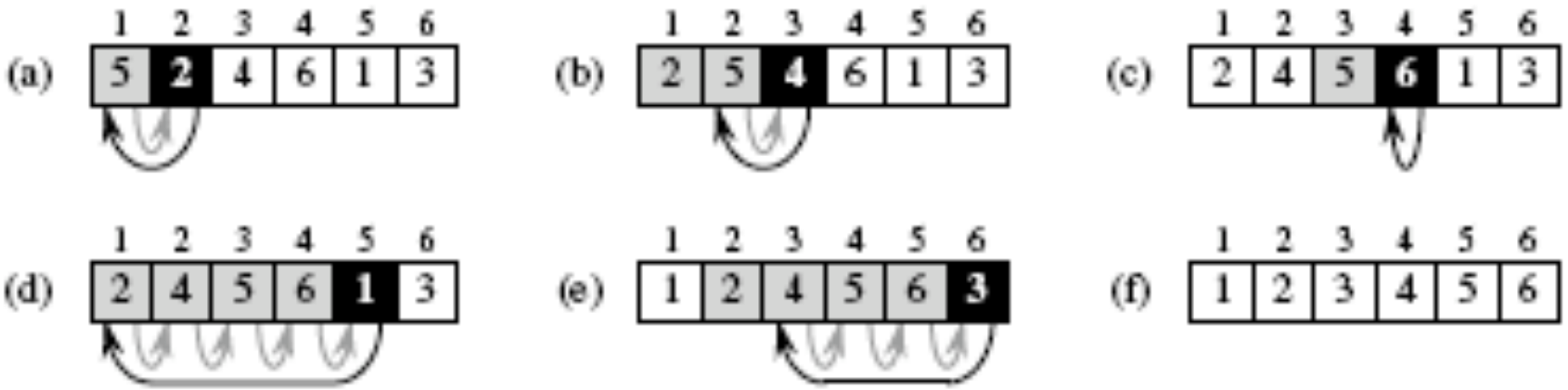



Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$ ; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

# Un algorithme naturel : le tri par insertion

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3           ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4           i ← j − 1
5           tant que i > 0 et A[i] > clé
6               faire A[i + 1] ← A[i]
7                    i ← i − 1
8           A[i + 1] ← clé
```

# Correction du tri par insertion par invariant de boucle

Les invariants de boucle sont des propriétés qui aident à comprendre/expliquer/justifier pourquoi un algorithme est correct. Une fois l'invariant énoncé, il faut montrer trois choses concernant l'invariant de boucle :

- Initialisation : il est vrai avant la première itération de la boucle.
- Conservation : s'il est vrai avant une itération de la boucle, il le reste avant l'itération suivante.
- Terminaison : une fois la boucle terminée( !), l'invariant fournit une propriété utile qui aide à montrer la correction de l'algorithme.

# Correction du tri par insertion par invariant de boucle

Les invariants de boucle sont des propriétés qui aident à comprendre/expliquer/justifier pourquoi un algorithme est correct. Une fois l'invariant énoncé, il faut montrer trois choses concernant l'invariant de boucle :

- Initialisation : il est vrai avant la première itération de la boucle.
- Conservation : s'il est vrai avant une itération de la boucle, il le reste avant l'itération suivante.
- Terminaison : une fois la boucle terminée( !), l'invariant fournit une propriété utile qui aide à montrer la correction de l'algorithme.

## Exemple

Pour le tri par insertion :

> *"Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau* $A[1..j-1]$ *se compose des éléments qui occupaient initialement les positions* 1 *à* $j-1$, *mais qui sont maintenant triés."*

# Correction du tri par insertion par invariant de boucle

Les invariants de boucle sont des propriétés qui aident à comprendre/expliquer/justifier pourquoi un algorithme est correct. Une fois l'invariant énoncé, il faut montrer trois choses concernant l'invariant de boucle :

- Initialisation : il est vrai avant la première itération de la boucle.
- Conservation : s'il est vrai avant une itération de la boucle, il le reste avant l'itération suivante.
- Terminaison : une fois la boucle terminée( !), l'invariant fournit une propriété utile qui aide à montrer la correction de l'algorithme.

## Exemple

Pour le tri par insertion :

> *"Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau* $A[1..j-1]$ *se compose des éléments qui occupaient initialement les positions* 1 *à* $j-1$, *mais qui sont maintenant triés."*

## Remarque

Pour les preuves de correction par invariants de boucle, il faut toujours prouver la terminaison à part : ici c'est une boucle for et pour la boucle tant que, c'est la valeur de $i$ qui diminue et finit donc par atteindre la valeur 0.

# Invariant de boucle du tri par insertion

*Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau $A[1..j-1]$ se compose des éléments qui occupaient initialement les positions $A[1..j-1]$ mais qui sont maintenant triés.*

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3          ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4          i ← j − 1
5          tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← clé
```

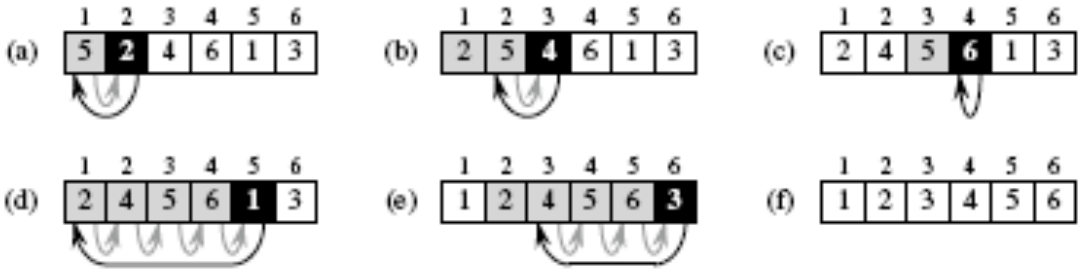



Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$ ; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

# Invariant de boucle du tri par insertion

*Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau $A[1..j-1]$ se compose des éléments qui occupaient initialement les positions $A[1..j-1]$ mais qui sont maintenant triés.*

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3           ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4           i ← j − 1
5           tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                   i ← i − 1
8           A[i + 1] ← clé
```

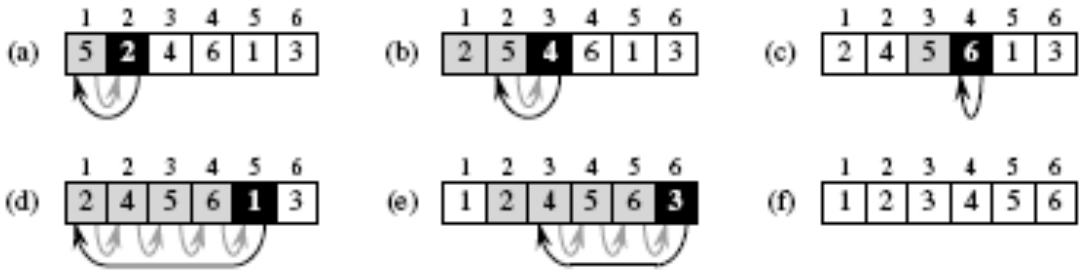



Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$ ; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

À l'initialisation j = 2 et le tableau $A[1..j-1]$ est $A[1]$ qui est trivialement trié.

# Invariant de boucle du tri par insertion

*Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau $A[1..j-1]$ se compose des éléments qui occupaient initialement les positions $A[1..j-1]$ mais qui sont maintenant triés.*

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3          ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4          i ← j − 1
5          tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← clé
```

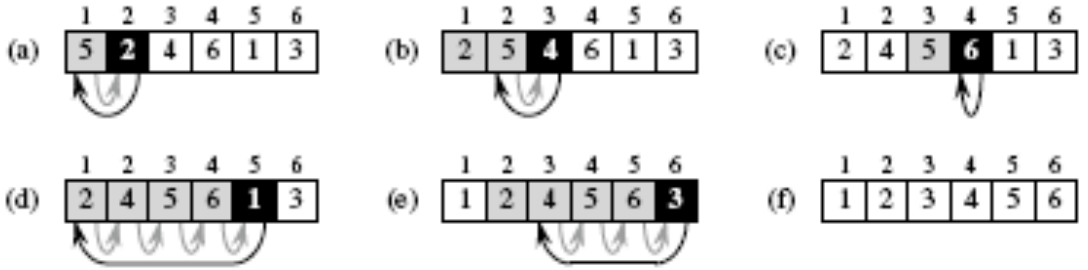



Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$ ; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

Conservation (on pourrait être plus formel).
Le corps de la boucle "pour" extérieure fonctionne en déplaçant $A[j-1]$, $A[j-2]$, $A[j-3]$, etc. d'une position vers la droite jusqu'à ce qu'on trouve la bonne position pour A[j] (lignes 4–7).
On insère alors la valeur de A[j] (ligne 8) à la bonne place.

# Invariant de boucle du tri par insertion

*Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau $A[1..j-1]$ se compose des éléments qui occupaient initialement les positions $A[1..j-1]$ mais qui sont maintenant triés.*

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3          ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4          i ← j − 1
5          tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← clé
```

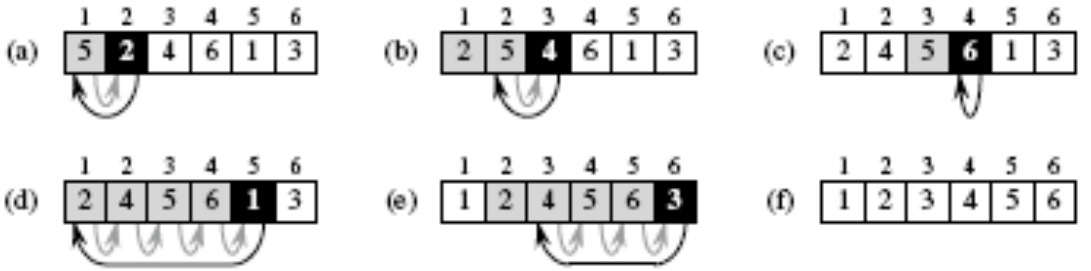



Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$ ; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

Terminaison : La boucle pour extérieure prend fin quand $j$ dépasse $n$, c'est-à-dire quand $j = n + 1$.

En substituant $n + 1$ à $j$ dans la formulation de l'invariant de boucle, on obtient :

*Le sous-tableau $A[1..n]$ se compose des éléments qui appartenaient originellement à $A[1..n]$ mais qui sont maintenant triés.*

Donc tout le tableau $A$ est trié ! Par conséquent l'algorithme est correct.