


Algorithmique

Chap 03


Tris et algorithmes classiques

3.1 Algorithmes classiques

3.1.1 Recherche du maximum

 **Idée générale**

Pour trouver un maximum dans un tableau, on parcourt les éléments de gauche à droite en conservant en mémoire la meilleure valeur rencontrée jusqu'ici (et éventuellement son indice). À chaque nouvel élément, on compare et on met à jour si nécessaire.

 **Algorithme — Recherche du maximum**

Input : Un tableau *tab* de taille *n* (avec $n \geq 1$)

Output : La valeur maximale de *tab*


$m \leftarrow tab[0]$

Pour *i* allant de 1 à $n - 1$:

 Si $tab[i] > m$ alors

$m \leftarrow tab[i]$

Renvoyer *m*

 **Schéma — Recherche du maximum**

Exemple : $tab = [6, 2, 8, 3, 1, 7]$.

$i = 0$	<table><tr><td>6</td><td>2</td><td>8</td><td>3</td><td>1</td><td>7</td></tr></table>	6	2	8	3	1	7	$m = 6$ Départ : on initialise $m \leftarrow tab[0]$
6	2	8	3	1	7			
$i = 1$	<table><tr><td>6</td><td>2</td><td>8</td><td>3</td><td>1</td><td>7</td></tr></table>	6	2	8	3	1	7	$m = 6$ $tab[1] = 2 \leq m$: on garde <i>m</i>
6	2	8	3	1	7			
$i = 2$	<table><tr><td>6</td><td>2</td><td>8</td><td>3</td><td>1</td><td>7</td></tr></table>	6	2	8	3	1	7	$m = 8$ $tab[2] = 8 > m$: mise à jour $m \leftarrow 8$
6	2	8	3	1	7			
$i = 3$	<table><tr><td>6</td><td>2</td><td>8</td><td>3</td><td>1</td><td>7</td></tr></table>	6	2	8	3	1	7	$m = 8$ $tab[3] = 3 \leq m$: on garde <i>m</i>
6	2	8	3	1	7			
$i = 4$	<table><tr><td>6</td><td>2</td><td>8</td><td>3</td><td>1</td><td>7</td></tr></table>	6	2	8	3	1	7	$m = 8$ $tab[4] = 1 \leq m$: on garde <i>m</i>
6	2	8	3	1	7			
$i = 5$	<table><tr><td>6</td><td>2</td><td>8</td><td>3</td><td>1</td><td>7</td></tr></table>	6	2	8	3	1	7	$m = 8$ $tab[5] = 7 \leq m$: on garde <i>m</i>
6	2	8	3	1	7			

Fin : maximum = 8.

Exercice 1 — Implémentation en Python

Écrire une fonction `maximum(tab)` qui renvoie la valeur maximale de `tab`.

Exercice 2 — Correction et terminaison

1. Proposer un invariant $P(i)$: que vaut m après avoir parcouru les indices $0..i$?

2. Justifier l'initialisation et l'hérédité de l'invariant.

3. Justifier la terminaison de l'algorithme.

Exercice 3 — Complexité

1. Combien de comparaisons effectue-t-on pour un tableau de taille n ?

2. Conclure en notation $O(\cdot)$.

3.1.2 Recherche dichotomique dans un tableau trié

Idée générale

Dans un tableau trié, on peut éliminer la moitié des valeurs à chaque étape : on compare la valeur cherchée à l'élément du milieu, puis on conserve uniquement la moitié où elle peut encore se trouver.

Algorithme — Recherche dichotomique

Input : Un tableau **trié** tab de taille n , une valeur x

Output : Vrai si x est dans tab , Faux sinon

$g \leftarrow 0, \quad d \leftarrow n - 1$

Tant que $g \leq d$:

$m \leftarrow \left\lfloor \frac{g + d}{2} \right\rfloor$

Si $tab[m] = x$ alors renvoyer Vrai

Sinon si $tab[m] < x$ alors $g \leftarrow m + 1$

Sinon $d \leftarrow m - 1$

Renvoyer Faux

Schéma — Dichotomie 1 : On trouve à la fin

Exemple : $tab = [2, 5, 9, 12, 18, 23, 27, 31, 38, 41, 49, 52, 60, 67, 74]$ et on cherche $x = 49$.

Légende : ■ zone conservée ■ zone éliminée ■ milieu m

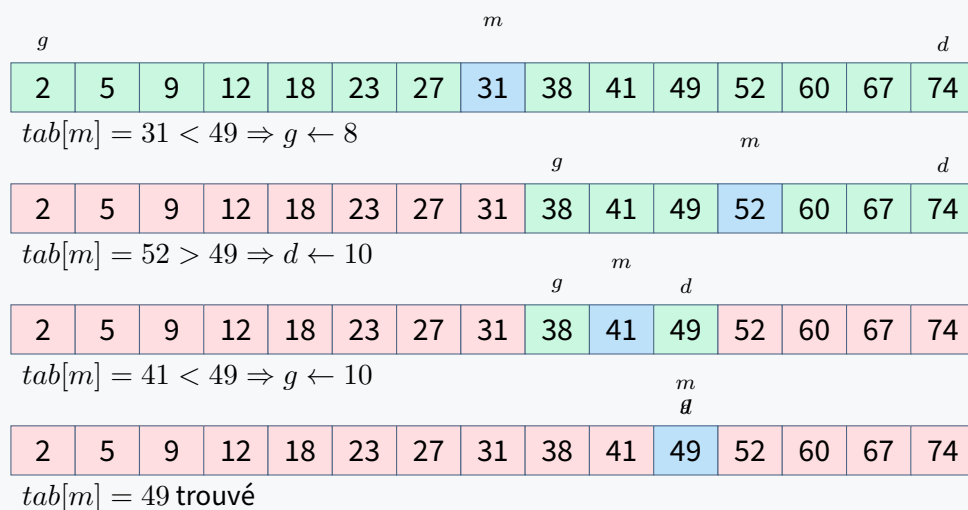




Schéma — Dichotomie 2 : On trouve en cours de route

Exemple : $tab = [2, 5, 9, 12, 18, 23, 27, 31, 38, 41, 49, 52, 60, 67, 74]$ et on cherche $x = 38$.

Légende : ■ zone conservée ■ zone éliminée ■ milieu m

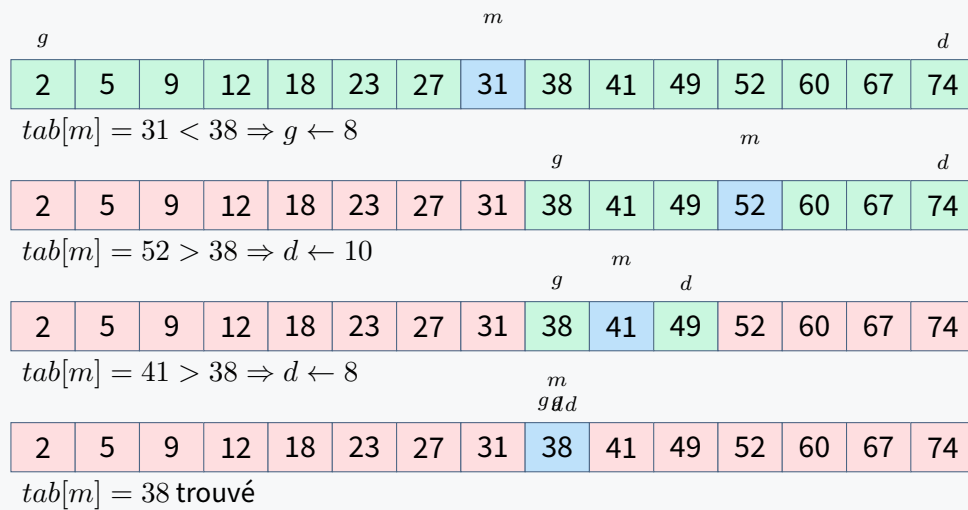
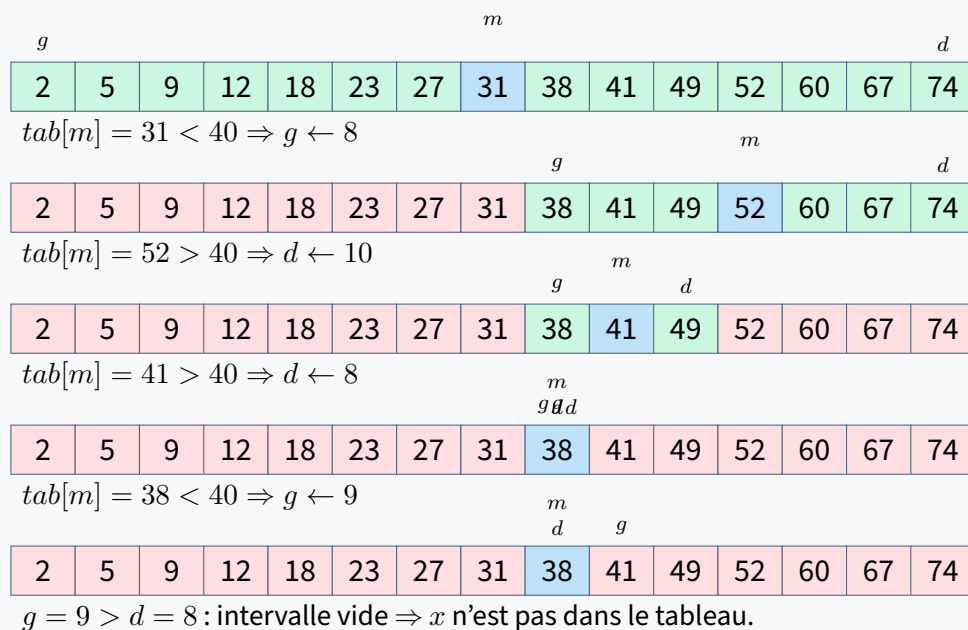


Schéma — Dichotomie 3 : On ne trouve pas

Exemple : $tab = [2, 5, 9, 12, 18, 23, 27, 31, 38, 41, 49, 52, 60, 67, 74]$ et on cherche $x = 40$.

Légende : ■ zone conservée ■ zone éliminée ■ milieu m



[illegible][illegible]

2. Proposer un variant montrant que la boucle termine.

Exercice 6 — Complexité

1. Combien de fois au maximum peut-on diviser la taille de l'intervalle par 2 avant d'arriver à 0?

2. Conclure en $O(\cdot)$.

3.2 Les algorithmes classiques de tris

3.2.1 Tri par sélection

Idée générale

Le tri par sélection repose sur l'idée suivante :

- on considère qu'une partie du tableau est déjà triée;
- à chaque étape, on cherche le **plus petit élément** de la partie non triée;
- on place cet élément à la fin de la partie triée.

La taille de la partie triée augmente d'une unité à chaque étape.

Algorithme — Tri par sélection

Input : Un tableau tab de taille n

Output : tab trié dans l'ordre croissant

Pour i allant de 0 à $n - 1$:

$imin \leftarrow i$

Pour j allant de $i + 1$ à $n - 1$:

Si $tab[j] < tab[imin]$ alors

$imin \leftarrow j$

échanger $tab[i]$ et $tab[imin]$

Schéma — Tri par sélection

On part du tableau :

$$tab = [6, 2, 8, 3, 1, 7]$$

Pour chaque étape i :

- **Avant** : on repère i_{\min} , l'indice du minimum dans $[i..n-1]$.
- **Après** : on échange $tab[i]$ et $tab[i_{\min}]$ (si $i_{\min} \neq i$).

Légende : ■ déjà trié ■ zone à traiter ■ case i ■ minimum i_{\min}



Exercice 7 — Implémentation en Python

Écrire en Python une fonction `tri_selection(tab)` qui trie la liste `tab` dans l'ordre croissant.

Exercice 8 — Correction et terminaison

1. Proposer une propriété $P(i)$ pouvant servir d'**invariant** pour la boucle principale.

2. **Initialisation** : expliquer pourquoi $P(0)$ est vraie.

3. **Hérédité** : supposer $P(i)$ vraie et expliquer pourquoi l'itération suivante permet d'obtenir $P(i + 1)$.

.....

.....

.....

.....

.....

.....

4. **Terminaison** : justifier que l'algorithme s'arrête toujours.

.....

5. Conclure quant à la **correction totale**.

.....

Exercice 9 — Complexité du tri par sélection

1. À l'étape i , combien de comparaisons sont effectuées pour chercher le minimum ?

.....

2. En déduire le nombre total de comparaisons effectuées par l'algorithme (sous forme de somme).

.....

.....

3. Donner l'ordre de grandeur de la complexité en notation $O(\cdot)$.

.....

4. Cette complexité dépend-elle de l'ordre initial du tableau ? Justifier.

.....

3.2.2 Tri par insertion

Idée générale

Le tri par insertion s'inspire de la manière dont on trie des cartes à la main :

- on parcourt le tableau de gauche à droite;
- on suppose que la partie gauche est déjà triée;
- on insère l'élément courant à la bonne position dans cette partie triée.

Algorithme — Tri par insertion

Input : Un tableau tab de taille n

Output : tab trié dans l'ordre croissant

Pour i allant de 1 à $n - 1$:

$x \leftarrow tab[i]$

$j \leftarrow i - 1$

 Tant que $j \geq 0$ et $tab[j] > x$:

$tab[j + 1] \leftarrow tab[j]$

$j \leftarrow j - 1$

$tab[j + 1] \leftarrow x$

Schéma — Tri par insertion

On part du tableau :

$$tab = [6, 2, 8, 3, 1, 7]$$

À l'étape i , la clé $x = tab[i]$ (case orange) est insérée dans la partie gauche déjà triée; après insertion, la case i (fin de la zone triée) est en bleu. La flèche indique le déplacement de la clé vers sa nouvelle position.

Légende : ■ déjà trié ■ clé (avant) ■ case i (après) → déplacement



Exercice 10 — Implémentation en Python

Écrire en Python une fonction `tri_insertion(tab)`.

Exercice 11 — Correction et terminaison

1. Proposer un invariant $P(i)$ pour la boucle principale.

2. Expliquer pourquoi l'étape d'insertion conserve la propriété $P(i)$.

3. Justifier la terminaison :

- de la boucle `for`,
- de la boucle `while` (donner un variant).

.....

.....

.....

.....

4. Conclure quant à la correction totale.

.....

Exercice 12 — Complexité

1. Donner la complexité dans le meilleur cas et le pire cas.

.....

.....

.....

2. Conclure en $O(\cdot)$.

.....

.....