Algorithmique	Chap 05	Tris et algorithmes classiques

5.1 Les algorithmes classiques de tris

5.1.1 Tri par sélection

Idée générale

Le tri par sélection repose sur une idée simple :

- 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 suivant :

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

À l'étape i, on cherche le minimum dans la partie $i \ldots n-1$, puis on l'échange avec la case i.

6	2	8	3	1	7	Départ
1	2	8	3	6	7	$i = 0$: minimum = 1, échange avec la case 0
1	2	8	3	6	7	$i = 1$: minimum = 2, rien à changer
1	2	3	8	6	7	$i = 2$: minimum = 3, échange avec la case 2
1	2	3	6	8	7	$i = 3$: minimum = 6, échange avec la case 3
1	2	3	6	7	8	Fin : tableau trié

Exercice 1 — Implémentation en Python

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

Exercice 2 — 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 3 — 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.

5.1.2 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 (exemple pas à pas)

On part du tableau :

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

À l'étape i, on **insère** $tab[i]$ dans la partie gauche $tab[0..i-1]$ supposée triée.

6	2	8	3	1	7	Départ
2	6	8	3	1	7	$i = 1$: insertion de 2 dans [6]
2	6	8	3	1	7	$i = 2$: insertion de 8 dans [2,6]
2	3	6	8	1	7	$i = 3$: insertion de 3 dans [2,6,8]
1	2	3	6	8	7	$i = 4$: insertion de 1 dans [2,3,6,8]
1	2	3	6	7	8	Fin : tableau trié

Exercice 4 — Implémentation en Python

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

Exercice 5 — 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 6 — Complexité

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

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

5.1.3 Tri rapide

Idée générale

Le tri rapide repose sur le principe **diviser pour régner** :

- on choisit un **pivot** dans le tableau ;
- on **partitionne** le tableau : les éléments plus petits que le pivot sont placés à gauche, les plus grands à droite ;
- le pivot est alors à sa **position finale** ;
- on applique récursivement le même procédé aux deux sous-tableaux.

Algorithme — Tri rapide

Input : Un tableau tab et deux indices g et d avec $0 \leq g \leq d < n$
Output : La partie $tab[g..d]$ est triée dans l'ordre croissant
if $g \geq d$ **then**
  arrêter
choisir un pivot (par exemple $tab[d]$)
$i \leftarrow g$
Pour j allant de g à $d - 1$:
  Si $tab[j] \leq pivot$ alors
    échanger $tab[i]$ et $tab[j]$
    $i \leftarrow i + 1$
échanger $tab[i]$ et $tab[d]$
tri_rapide($tab, g, i - 1$)
tri_rapide($tab, i + 1, d$)

Schéma — Tri rapide en place : partition (Lomuto)

On part de :

$$tab = [6,\ 2,\ 8,\ 3,\ 1,\ 7], \qquad \text{pivot} = 7\ (\text{dernier élément})$$

La barre verticale indique la frontière i : à gauche de i, on place progressivement les valeurs $\leq$ pivot.

						pivot	
$i = 0$	6	2	8	3	1	7	Départ
$i = 1$	6	2	8	3	1	7	$6 \leq 7$ (on avance i)
$i = 2$	6	2	8	3	1	7	$2 \leq 7$ (on avance i)
$i = 2$	6	2	8	3	1	7	$8 > 7$ (on ne change rien)
$i = 3$	6	2	3	8	1	7	$3 \leq 7$ (échange 3 et 8, on avance i)
$i = 4$	6	2	3	1	8	7	$1 \leq 7$ (échange 1 et 8, on avance i)
$i = 4$	6	2	3	1	7	8	Échange final : pivot à sa position définitive

À la fin de la partition :

- tous les éléments à gauche du pivot sont $\leq$ pivot ;
- tous les éléments à droite sont $>$ pivot ;
- le pivot est à sa position finale.

Exercice 7 — Implémentation en Python

Écrire une fonction récursive `tri_rapide(tab, g, d)` qui trie `tab[g..d]` dans l'ordre croissant.

Exercice 8 — Correction et terminaison

1. Expliquer pourquoi, après la phase de partition, le pivot est à sa position définitive.

2. En supposant les appels récursifs corrects, expliquer pourquoi le tableau est trié après leur exécution.

3. Comment pourrait-on justifier la terminaison de l'algorithme ?

Exercice 9 — Complexité

1. Quel est le coût de la phase de partition sur un tableau de taille n ?

2. Si le pivot coupe le tableau en deux parties de tailles comparables, quelle est la complexité totale de l'algorithme ?

3. Donner un exemple de choix de pivot menant au pire cas.

4. Quelle est alors la complexité dans ce cas ?

5.2 Algorithmes classiques de Première

5.2.1 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 — Schéma (parcours et mise à jour)

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

6	2	8	3	1	7	Départ : $m = 6$
6	2	8	3	1	7	$i = 1$, $2 \leq 6$ donc m inchangé
6	2	8	3	1	7	$i = 2$, $8 > 6$ donc $m \leftarrow 8$
6	2	8	3	1	7	$i = 3$, $3 \leq 8$ donc m inchangé
6	2	8	3	1	7	$i = 4$, $1 \leq 8$ donc m inchangé
6	2	8	3	1	7	$i = 5$, $7 \leq 8$ donc m inchangé

Fin : maximum = 8

Exercice 10 — Implémentation en Python

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

Exercice 11 — 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 12 — Complexité

1. Combien de comparaisons effectue-t-on pour un tableau de taille n ?
2. Conclure en notation $O(\cdot)$.

5.2.2 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 — Schéma (réduction de l'intervalle)

Exemple : $tab = [1, 2, 3, 6, 7, 8]$ et on cherche $x = 7$.

1	2	3	6	7	8	$g = 0,\ d = 5$ milieu $m = 2$, $tab[m] = 3 < 7$ donc $g \leftarrow 3$
1	2	3	6	7	8	$g = 3,\ d = 5$ milieu $m = 4$, $tab[m] = 7$ trouvé

Exercice 13 — Implémentation en Python

Écrire une fonction `dichotomie(tab, x)` qui renvoie `True` ou `False`.

Exercice 14 — Correction et terminaison

1. Proposer un invariant portant sur l'intervalle $[g, d]$: si x est dans le tableau, où peut-il encore se trouver ?

2. Proposer un variant montrant que la boucle termine.

Exercice 15 — 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 $O(\cdot)$.