Algorithmique

Là, ça devient intéressant.

Points abordés

Contenus	Capacités attendues	Commentaires
Parcours séquentiel d'un tableau	Écrire un algorithme de recherche d'une occurrence sur des valeurs de type quelconque. Écrire un algorithme de recherche d'un extremum, de calcul d'une moyenne.	On montre que le coût est linéaire.
Tris par insertion, par sélection	Écrire un algorithme de tri. Décrire un invariant de boucle qui prouve la correction des tris par insertion, par sélection.	La terminaison de ces algorithmes est à justifier. On montre que leur coût est quadratique dans le pire cas.

Plan

- Définition d’un **algorithme**
- Exemple d’algorithme : recherche d’un extremum.
- Introduction au concept de **complexité**
- **Tri** d’une liste
  - Tri par sélection
    - Concept **d’invariant de boucle**
  - Tri par insertion
- Recherche dichotomique

Un algorithme, c’est quoi ?

- Un algorithme est une **suite** précise, finie et détaillée **d’instructions** qui permet de **résoudre un problème** ou de réaliser une tâche précise.
- C’est une procédure qui peut être suivi pas à pas par une personne ou une machine.
- Typiquement, un algorithme prend une **entrée** (un problème) et renvoie une **sortie** (une solution).

Exemple d’algorithmes

Algorithme	Description	Entrée	Sortie
Algorithme du « Rubik’s Cube »	Détermine les mouvements à effectuer pour résoudre un Rubik’s Cube.	Un Rubik’s Cube quelconque.	Un Rubik’s Cube non-mélangé.
Algorithme du « Labyrinthe »	Détermine le chemin à emprunter pour sortir d’un labyrinthe.	Un labyrinthe.	Un chemin qui relie l’entrée à la sortie du labyrinthe.
Algorithme du Plus Court Chemin	Détermine le plus court chemin dans un graphe pondéré entre deux sommets.	Un graphe pondéré, un sommet de départ et un sommet d’arrivé.	Le plus court chemin entre les deux sommets.
Algorithme de recommandation	Détermine le « meilleur » contenu à suggérer à un utilisateur (tweet, tiktok, post, film, livre, musique etc.)	Préférences de l’utilisateur et son historique de consommation.	Un nouveau contenu à présenter à l’utilisateur.

Algorithme de recherche d'un extrémum

- On se place dans le cas de la recherche du **minimum**.
- **Entrée :** Une liste de nombres entiers.
- **Sortie :** Le plus petit nombre de la liste.

PSEUDOCODE

Définir m comme la 1ère valeur de la liste
Pour chaque valeur v de la liste **Faire**:
    Si v < m **Alors**:
        Affecter v à m
    Fin Si
Fin Pour
m est le minimum de la liste

CODE PYTHON

```python
def minimum(tab: list) -> int:
    minimum_courant = tab[0]
    for val in tab:
        if val < minimum_courant:
            minimum_courant = val
    return minimum_courant
```

Mesurer la performance d’un algorithme

- Souvent, il existe des algorithmes très différents pour résoudre le même problème. Comment déterminer lequel est le « meilleur » ?
- Il y a deux mesures classiques pour mesurer la « performance » d’un algorithme :
  - Son **temps** d’exécution
  - La quantité de **mémoire** qu’il utilise

- Evidemment, on compare les algorithmes suivant **la même entrée**, le même problème !

La taille d’une entrée

- Un problème est souvent décrit par une taille.
  - *Recherche du minimum* : longueur de la liste
  - *Plus court chemin dans un graphe* : nombre de sommets et d’arêtes
  - *Pour l’algorithme du Rubik’s Cube* : la taille du cube (3x3x3, 4x4x4 etc.)

- **Plus la taille du problème est grand, plus le temps d’exécution augmente.**

- On souhaite analyser un algorithme pour exprimer **son temps d’exécution** en fonction de la **taille de l’entrée**.
- Si n est la taille du problème, on voudrait connaître f(n) qui renvoie son temps d’exécution !

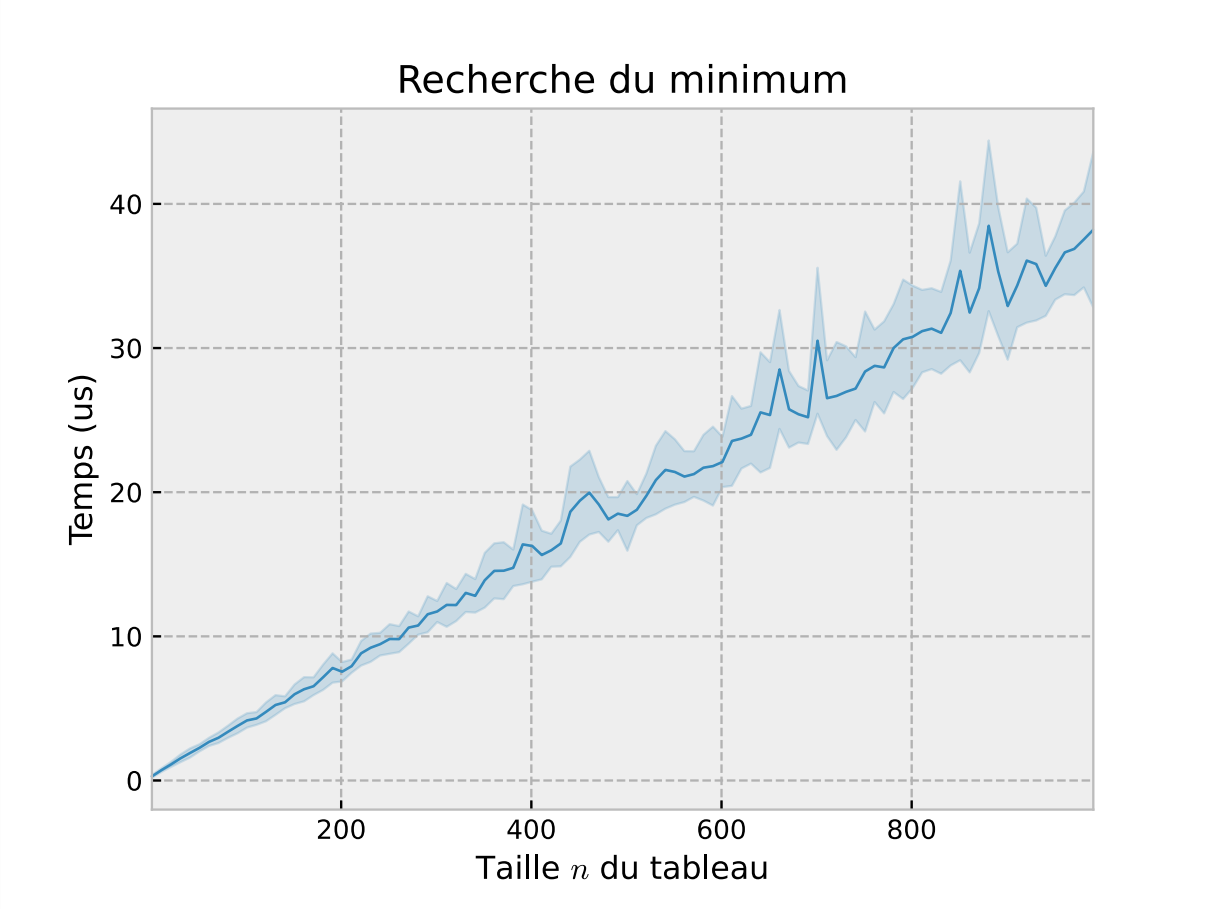
La taille d’une entrée – Exemple sur la recherche du minimum

- J’exécute le programme :
  - Quand le tableau est de taille 200, il s’exécute en 8 µs.
  - Quand le tableau est de taille 400, il s’exécute en 16 µs.
  - Quand le tableau est de taille 600, il s’exécute en 24 µs.
  - etc.

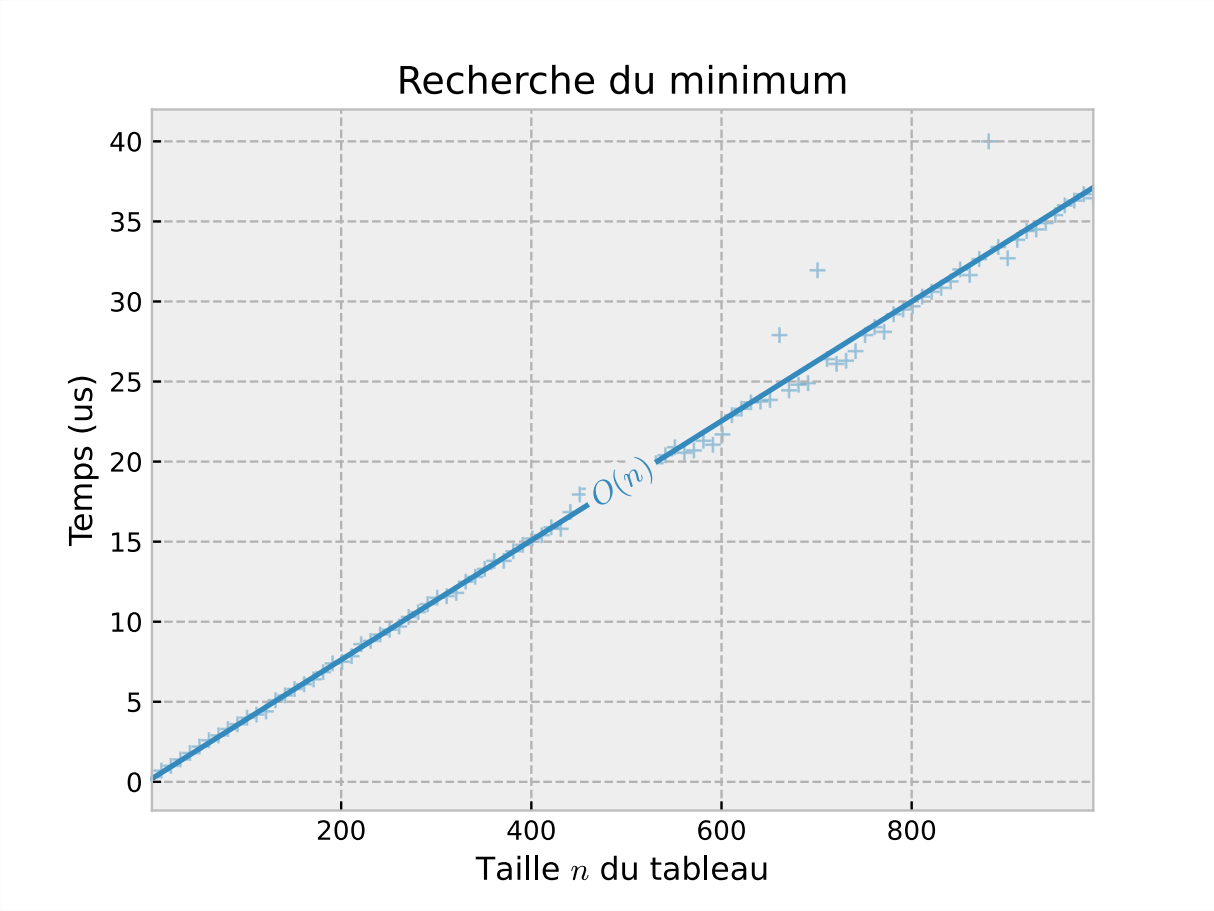
CODE PYTHON

```python
def minimum(tab: list) -> int:
    minimum_courant = tab[0]
    for val in tab:
        if val < minimum_courant:
            minimum_courant = val
    return minimum_courant
```

La taille d’une entrée – Exemple sur la recherche du minimum



La taille d’une entrée – Exemple sur la recherche du minimum



Pourquoi c’est linéaire ?

- Pourquoi ? Comptons le nombre d’instructions élémentaires exécutées en fonction de la taille de la liste !

Taille n de la liste	Nombre d’instructions exécutées
1	
2	
3	
4	

CODE PYTHON

```python
def minimum(tab: list) -> int:
    minimum_courant = tab[0]
    for val in tab:
        if val < minimum_courant:
            minimum_courant = val
    return minimum_courant
```

Pourquoi c’est linéaire ?

- Pourquoi ? Comptons le nombre d’instructions élémentaires exécutées en fonction de la taille de la liste !

Taille n de la liste	Nombre d’instructions exécutées
1	1 + 2 + 1 = 4
2	1 + 2 * 2 + 1 = 6
3	1 + 2 * 3 + 1 = 8
4	1 + 2 * 4 + 1 = 10

CODE PYTHON

```python
def minimum(tab: list) -> int:
    minimum_courant = tab[0]
    for val in tab:
        if val < minimum_courant:
            minimum_courant = val
    return minimum_courant
```

Pourquoi c’est linéaire ?

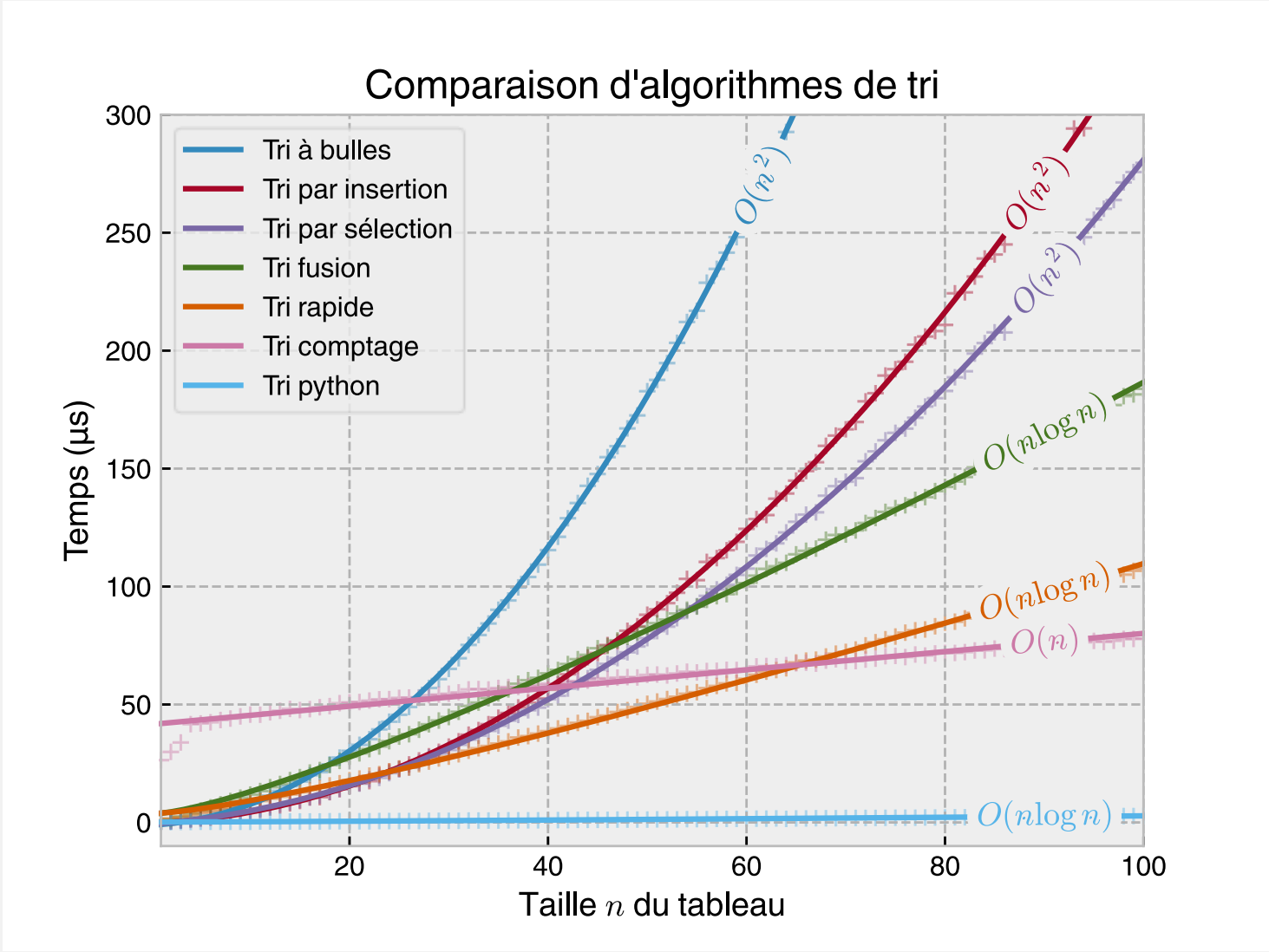
- Nb. d’instructions exécutées = 2 * n + 2
- Le nombre d’instructions est **linéaire** en la taille de la liste.
- Si on suppose que le temps d’exécution de chaque instruction est le **même**, alors :
- Le temps est proportionnelle à 2 * n + 2 donc est aussi **linéaire**.
- On dit que cet algorithme est de **coût linéaire.**

- Petite astuce pour déterminer la complexité : **Repérer les boucles !**

Simplifier l’écriture - Notation de Landau

- Au lieu d’écrire précisément le temps d’exécution en fonction de la taille l’entrée, on la simplifie avec la **notation de Landau**.
- La notation de Landau, O, permet de décrire *asymptotiquement* une fonction. C’est une fonction « tendance » sur comment croît la fonction.
- Par exemple. Soit **n** la taille de la liste, la **complexité en temps** de l’algorithme de recherche du minimum est **O(n)**.
- Les constantes, les facteurs multiplicatifs disparaissent ! On voir alors toute de suite le coût linéaire de l’algorithme !

Notation de Landau – Quelques exemples



Tri d’une liste

- Un **algorithme de tri** est un algorithme qui permet de **trier** une liste d'éléments selon **un certain critère**, par exemple leur ordre croissant ou décroissant.

- Par exemple, on peut considérer l’algorithme de tri suivant :
  - **Entrée** : une liste de nombres entiers
  - **Sortie** : la liste triée par ordre croissant

- L’algorithme appliqué à la liste [4, 7, 1, 3, 5] renvoie donc la liste [1, 3, 4, 5, 7].

Les différents algorithmes

Il existe **d'innombrables** algorithmes pour trier une liste :

- *Tri par sélection*
- *Tri fusion*
- *Tri rapide*
- *Tri par tas*
- *Introsort*
- *Tri arborescent*
- *Smoothsort*
- *Tri de Shell*
- *Tri à bulles*
- *Tri par insertion*
- *etc.*

Tri par sélection – L’idée

- **Constat** : Dans une liste triée par ordre croissant, qu’a de spécial la première valeur ?

  C’est le **minimum**.
- **Idée** :
  - Déterminer le minimum de la liste (**sélection**)
  - Le retirer de la liste
  - Redéterminer le minimum
  - Le retirer
  - etc.
- Les valeurs sont ainsi retirées par ordre croissant !

Tri par sélection – Un exemple

[4, 7, **1**, 3, 5]
→ 1
[4, 7, **3**, 5]
→ 3
[**4**, 7, 5]
→ 4
[7, **5**]
→ 5
[**7**]
→ 7
[ ]

On lit dans l’ordre 1, 3, 4, 5, 7. On a bien trié la liste.

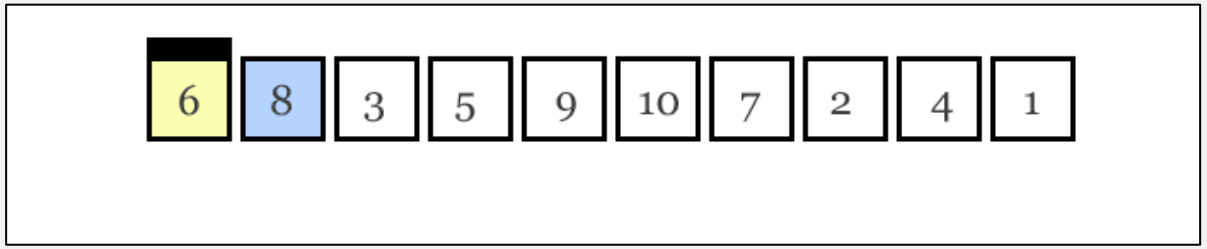
Tri par sélection – Un 1er pseudocode

Entrée : La liste L
Sortie : La listre triée $L_{triée}$

$L_{triée}$ = [ ]
Tant Que L n'est pas vide **Faire**
    Trouver le minimum de L
    Le retirer de L
    L'ajouter à $L_{triée}$
Fin Tant Que
Retourner $L_{triée}$

Tri par sélection – Un petit problème

- **Problème** : Retirer un élément d'une liste est de **coût linéaire** (dans le pire des cas, on copie toute la liste !).
- **Idée :** Déplacer les minimums successifs en début de liste grâce des permutations.



- Comme on peut le voir, **on scinde la liste en deux** : une partie triée (verte) et une partie non-triée (blanc).
- On modifie **en place** la liste, on la modifie directement sans créer de nouvelle liste.

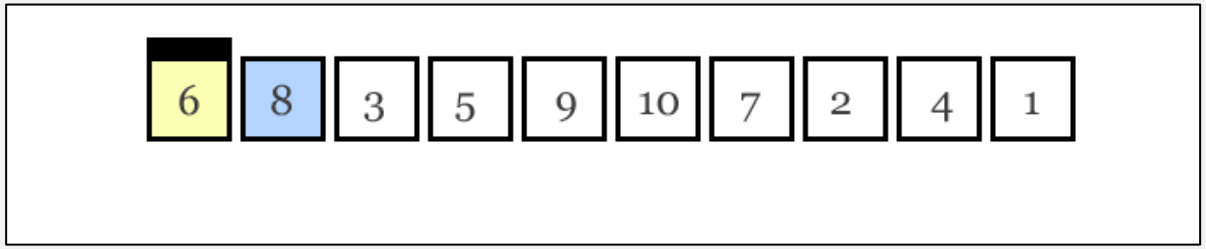
Tri par sélection – Un 2nd pseudocode

Pour i de 0 à n (exclus) **Faire** :

déterminer m l'indice du minimum entre i et n (exclus)

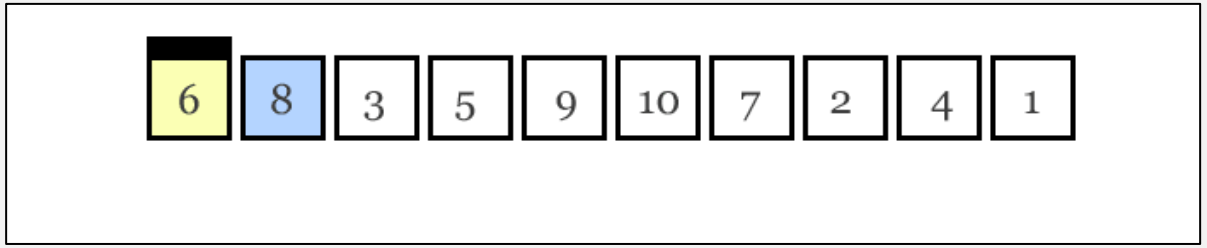
échanger la place de tab[m] et tab[i]

FinPour



Tri par sélection – A vous ! En Python !

- Définir la fonction `min_index(tab, debut, fin)` qui renvoie **l'indice du minimum** de la liste tab entre les indices debut et fin (exclus).
- Définir la fonction `echange(tab, i, j)` qui échange le contenu des valeurs tab[i] et tab[j].
- Enfin, écrire la fonction `tri_selection(tab)` qui tri le tableau donné en entrée.



Tri par sélection – Réponse

```
def min_index(tab, debut, fin):
    best = debut
    for i in range(debut + 1, fin):
        if tab[i] < tab[best]:
            best = i
    return best
```

Tri par sélection – Réponse

```
def min_index(tab, debut, fin):
    best = debut
    for i in range(debut + 1, fin):
        if tab[i] < tab[best]:
            best = i
    return best


def echange(tab, i, j):
    tab[i], tab[j] = tab[j], tab[i]
```

Tri par sélection – Réponse

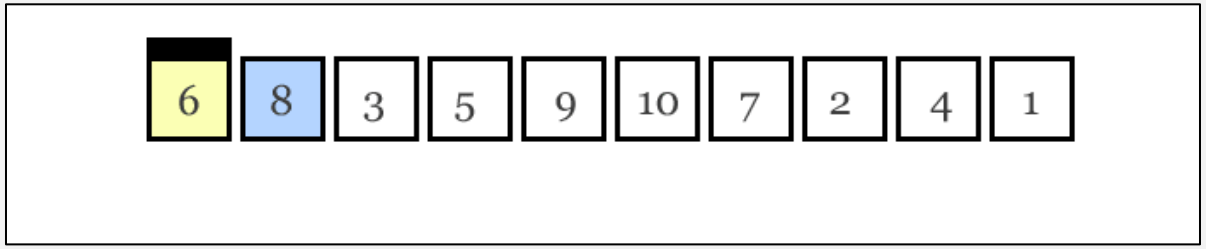
```
def min_index(tab, debut, fin):
    best = debut
    for i in range(debut + 1, fin):
        if tab[i] < tab[best]:
            best = i
    return best


def echange(tab, i, j):
    tab[i], tab[j] = tab[j], tab[i]


def tri_selection(tab):
    n = len(tab)
    for i in range(n - 1):
        j = min_index(tab, i, n)
        echange(tab, i, j)
```

Tri par sélection – Complexité

- Au début, on cherche le minimum dans une liste de taille n
- Puis de taille n – 1
- Puis de taille n – 2
- Etc. jusqu'à 1



Si l'on effectue O(N) instructions pour chercher le minimum dans une liste de taille N, le tri par sélection a pour complexité :

$$1 + 2 + 3 + \ldots + (n - 1) + n$$

$$= n * (n + 1) / 2 = \mathbf{O(n^2)}$$

C'est une **complexité quadratique.**

Tri par sélection – Par ordre décroissant

- Finalement, en changeant ce simple signe, on peut renverser l'ordre !

```
def min_index(tab, debut, fin):
    best = debut
    for i in range(debut + 1, fin):
        if tab[i] < tab[best]:
            best = i
    return best
```

```
def max_index(tab, debut, fin):
    best = debut
    for i in range(debut + 1, fin):
        if tab[i] > tab[best]:
            best = i
    return best
```

Tri par sélection – Projection

- Pour ceux qui ont vu la notion de projection. La projection est une fonction f (paramètre key) qui va s'appliquer aux éléments **juste avant la comparaison** :

```
def min_index(tab, debut, fin, f):
    best = debut
    for i in range(debut + 1, fin):
        if f(tab[i]) < f(tab[best]):
            best = i
    return best
```

- Très utile pour trier une liste d'éléments incomparables (comme des dictionnaires !)

Notion d’invariant de boucles

- Un **invariant de boucle** est une propriété qui est **vraie** pour tous les passages dans la boucle. L'invariant doit être vrai avant d'entrer dans la boucle (Initialisation) et rester vrai jusqu’à la fin de celle-ci (Héridité et Conclusion).!
- Trouver et prouver que l'on a un invariant de boucle permet de prouver la **correction**(validité) d'un algorithme.

Notion d’invariant de boucles
Exemple Recherche du minimum

- Pour rappel, l’algorithme est :

  initialiser *mini* à l'élément à l’indice 0 de la liste
  Pour chaque élément de la liste à partir du 2nd **Faire**
  Si l'élément est plus petit que *mini* **Alors**
  mettre *mini* à la valeur de élément
  Renvoyer *mini*

- L’**invariant** sera ici la propriété :

  "*mini* à l'étape i est le minimum de la liste d'éléments de 0 à i"

Notion d’invariant de boucles
Exemple Recherche du minimum

- Pour rappel, l’algorithme est :

  initialiser *mini* à l'élément à l’indice 0 de la liste
  Pour chaque élément de la liste à partir du 2nd **Faire**
  Si l'élément est plus petit que *mini* **Alors**
  mettre *mini* à la valeur de élément
  Renvoyer *mini*

- L’**invariant** sera ici la propriété :

"*mini* à l'étape i est le minimum de la liste d'éléments de 0 à i"

- Pour prouver cette invariant, on raisonne par **récurrence.** On montre que l'invariant est vrai à l'étape n induit qu'il est vrai à l'étape n+1 (hérédité) et qu'il est vrai **au départ** (initialisation).

Notion d’invariant de boucles
Exemple Recherche du minimum

- **Propriété : P(n) :** "*mini* à l'étape n est le minimum de la liste d'éléments de 0 à n"
- **Initialisation** : la propriété est vraie avant d’entrer dans la boucle puisque mini vaut l'élément à l’indice 0 de la liste. **P(0) vraie.**
- **Hérédité** : On suppose que la propriété est vraie au rang n, c’est-à-dire que mini est le minimum de la sous-liste de 0 à n.

Si la n+1ème valeur est plus petite que mini, alors elle devient le minimum de la sous-liste de 0 à n+1.

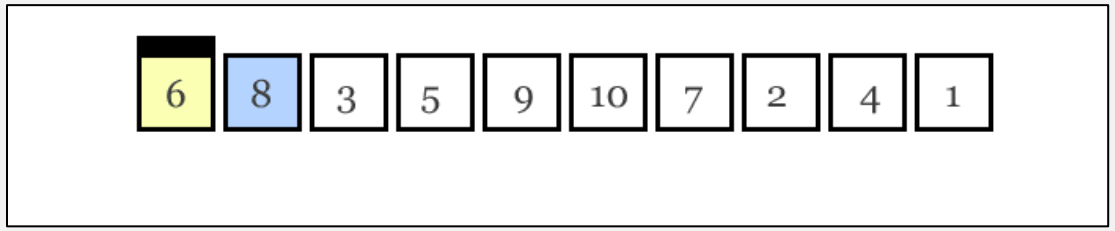
Sinon on garde l’ancien minimum qui reste toujours le minimum de la sous-liste de 0 à n+1.

Donc P(n+1) est vraie. L’hérédité est prouvé, **P(n) ⇒ P(n+1)**.

- **Conclusion** : Par principe de récurrence, la propriété est vraie. Donc l’invariant de boucle est vraie, donc l’algorithme est valide et se termine.

Notion d'invariant de boucles
Tri par sélection

```
def tri_selection(tab):
    n = len(tab)
    for i in range(n - 1):
        j = min_index(tab, i, n)
        echange(tab, i, j)
```



- Quel invariant choisir ici pour la boucle en jaune ? On souhaite bien prouver que la liste est bien triée à la fin de l'algorithme !

P(i) : « A la i-ème étape, la sous-liste de 0 à i (exclus) est triée. »

- Preuve en 3 étapes (initialisation/hérédité/conclusion) à faire !

Le tri par insertion – L’idée

- Le tri par insertion est le tri que vous utilisez pour ranger votre jeu de cartes ! Lorsqu’on prend une nouvelle carte, on l’insère directement à la bonne place dans le jeu trié de sa main.
- On conserve donc cette idée de **partition** en une **sous-liste triée** et une **sous-liste non-triée**.
- Au lieu de trouver le minimum dans la sous-liste non-triée (tri par sélection), on prend un élément de celle-ci et on l’insère au bon indice dans la sous-liste triée.

Tri par insertion – Un exemple

En bleu la sous-liste triée, en noir la sous-liste non triée.

[ ] [**4**, 7, 1, 3, 5]

[4] [**7**, 1, 3, 5]

[4, 7] [**1**, 3, 5]

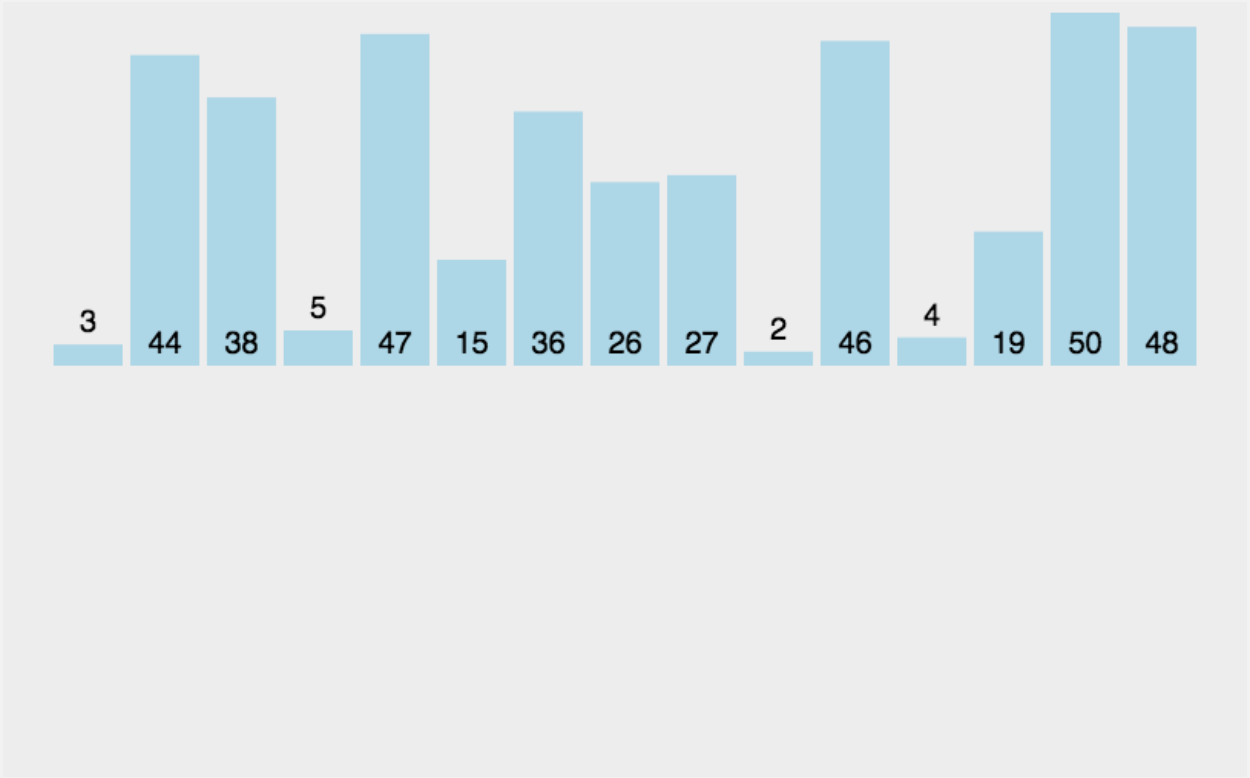
[1, 4, 7] [**3**, 5]

[1, 3, 4, 7] [**5**]

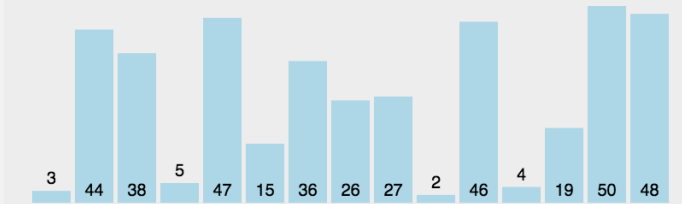
[1, 3, 4, 5, 7] [ ]

Tri par insertion – Modification en place

- Comme pour le tri par sélection, on peut travailler **en place** avec la liste : les éléments triés sont à gauche (ici en orange), les éléments non-triés à droite (ici en bleue).



Tri par insertion – Pseudocode



Pour i de 1 à n (exclus) **Faire** :

// Invariant de boucle : la sous-liste de 0 à i (exclu) est triée

Positionner correctement l'élément tab[i] dans la sous-liste de 0 à i.

FinPour

- L'idée est de faire remonter l'élément tab[i] (en rouge) grâce à des **échanges successifs**, tant que son voisin à gauche (en vert) est plus grand.

Tri par insertion – Pseudocode

Pour i de 1 à n (exclus) **Faire** :

    // Invariant de boucle : la sous-liste de 0 à i (exclu) est triée

    j = i

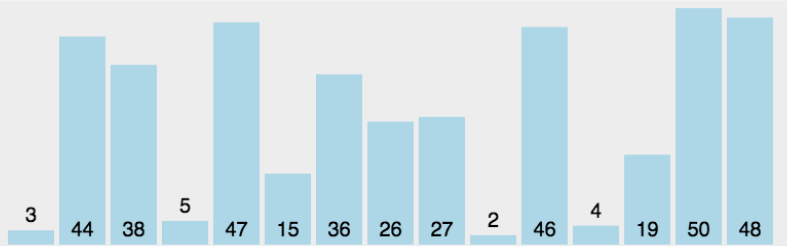
    TantQue j > 0 et tab[j – 1] > tab[j] **Faire**

        échanger les valeurs j et j – 1 de tab

        décrementer j

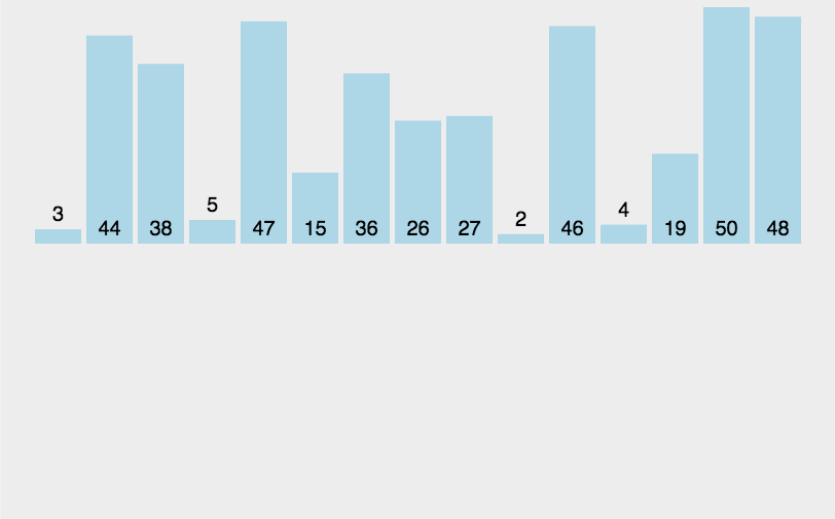
    FinTantQue

FinPour



Tri par insertion – Analyse

- Dans le pire des cas, la liste en entrée est triée par ordre décroissant.
- Combien de comparaisons/échanges se font par étape ?
- Que se passe-t-il si la liste en entrée est déjà triée ?
- Preuve par récurrence de l’invariant de boucle ?



Le recherche linéaire

- Imaginons qu’on ait une liste de nombres entiers.
- Comment déterminer si une certaine valeur **x** apparaît dans cette liste ?
- Ecrire un premier algorithme qui résous cette question.
- Quelle est sa complexité ?

Le recherche dichotomique

- On suppose maintenant qu'on ait une liste de nombres **triée par ordre croissant.**

  tab = [0, 1, 4, 9, 10, 12, 13, 15, 16, 18, 20, 30, 50, 52]

- Je cherche à savoir si la valeur 20 apparaît dans cette liste. Imaginons, je regarde la valeur à l'indice 5 :

  tab = [0, 1, 4, 9, 10, 12, 13, 15, 16, 18, 20, 30, 50, 52]

  0 4 5 6 13

- Que peut-on dire de la présence de la valeur 20 dans les deux sous-listes orange et bleu ?

Le recherche dichotomique

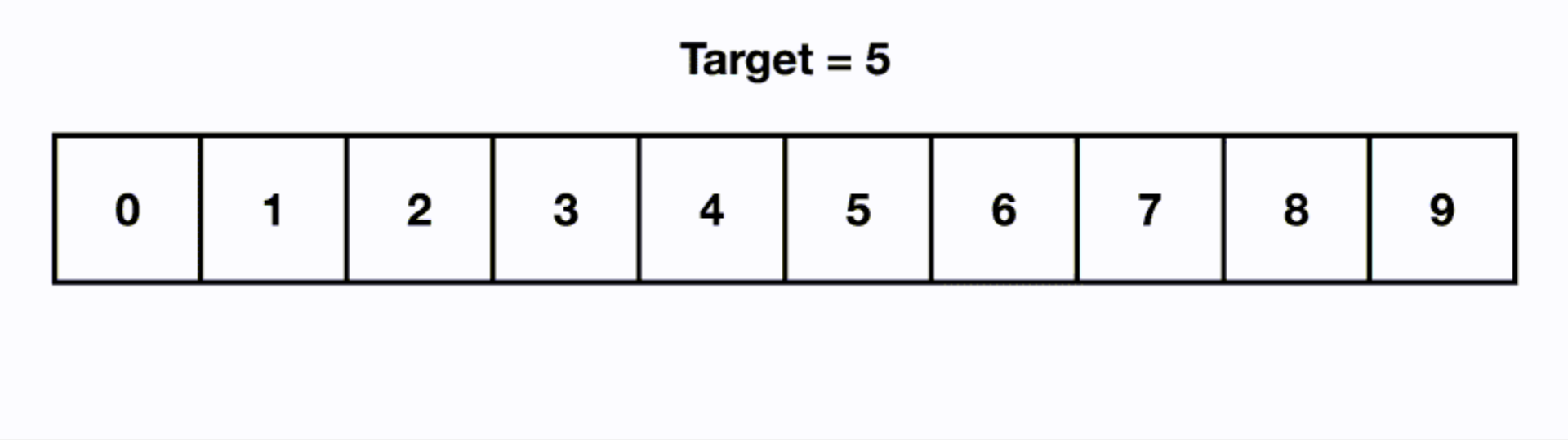
- Quel indice doit-on choisir pour **maximiser** la taille de la sous-liste à écarter ?
- **L'indice du milieu !**
- **Formuler** alors l'algorithme pour chercher rapidement une valeur dans une liste triée selon ce principe.
- Cet algorithme s'appelle la **recherche dichotomique** *(« couper en deux » en grec).*

Le recherche dichotomique - Pseudoalgo

- On compare la **valeur recherchée** à la **valeur du milieu** de la liste
  - Si elle est ==, retourner Vrai
  - Si elle est >, on écarte les valeurs du milieu jusqu'à la fin de la liste
  - Sinon (<), on écarte les valeurs du début de la liste jusqu'au milieu
- Si la nouvelle liste est **vide**, retourner Faux
- On **recommence** l'opération sur la nouvelle liste

Le recherche dichotomique - Python

- On imagine maintenant l’implémentation de cet algorithme.
- A-t-on vraiment besoin d’enlever des valeurs de la liste ?
- On utilise alors deux indices « **début** » et « **fin** » qui vont nous indiquer la tranche dans laquelle la valeur recherchée puisse exister.



Le recherche dichotomique - Python

```python
def dichotomie(tab, valeur):
    debut = 0
    fin = len(tab) - 1

    while debut <= fin:
        milieu = (debut + fin) // 2
        if tab[milieu] == valeur:
            return True
        elif tab[milieu] > valeur:
            fin = milieu - 1
        else:
            debut = milieu + 1

    return False
```

Le recherche dichotomique
Analyse Complexité

- A chaque étape, j'écarte la moitié de la liste.
- La liste a N éléments.
- Combien d'étapes dois-je faire pour arriver à 1 ?
- C'est-à-dire combien de fois par 2 dois-je diviser N pour obtenir 1 ?
- On doit résoudre l'équation (pour x) :

$$\frac{N}{2^x} = 1$$

- On a donc environ $x = \log_2 N$ étapes dans cet algorithme.
- L'algorithme est de complexité O(log N).

Le recherche dichotomique Analyse Complexité

n	log2(n)
1	0,00
10	3,32
100	6,64
1000	9,97
10000	13,29
100000	16,61
1000000	19,93

Le recherche dichotomique
Variant de boucle

- Un variant de boucle permet de prouver la **terminaison** d'un algorithme ; qu'il s'arrête peu importe l'entrée.
- Un variant de boucle est une valeur entière qui doit :
  - être **positive ou nulle.**
  - être **strictement décroissante**.
- Le variant de boucle de la recherche dichotomique serait

  fin – début

Le recherche dichotomique
Variant de boucle

- Le variant de boucle de la recherche dichotomique serait

  fin – début (la taille de la tranche considérée)
- On va montrer que la valeur « **fin – début** » décroit strictement à chaque itération.
- Si tab[milieu] == valeur, alors on sort de la boucle.
- Si tab[milieu] > valeur, alors **debut** devient **milieu+1**, donc le variant décroit strictement (la gauche du tableau se rapproche de la droite).
- Si tab[milieu] < valeur, alors **fin** devient **milieu–1**, donc le variant décroit strictement (la droite du tableau se rapproche de la gauche).

Le recherche dichotomique
Invariant de boucle

- On veut démontrer la correction de l'algorithme avec un invariant de boucle.
- *« Si **valeur** est dans **tab**, son indice est compris entre **début** et **fin**. »*
- *Démonstration :*
  - *Si la propriété est vraie en entrée de boucle, alors il n'y a que trois possibilités.*
  - *Si **tab[milieu] == valeur**, alors on sort de la boucle.*
  - *Si **tab[milieu] > valeur**, alors la recherche se poursuit de gauche à milieu−1, la propriété est encore vraie (car la liste est triée).*
  - *Si **tab[milieu] < valeur**, alors la recherche se poursuit de milieu+1 à droite, la propriété est encore vraie (car la liste est triée).*
- *On a donc bien un invariant de boucle et l'algorithme fait bien ce que l'on veut dans le cas où la recherche aboutit.*