

# Algorithmes et Pensée Computationnelle

## Algorithmes de recherche - Exercices avancés

Le but de cette séance est de se familiariser avec les algorithmes de recherche. Dans la série d'exercices, nous manipulerons des listes et collections en Java et Python. Nous reviendrons sur la notion de récursivité et découvrirons les arbres de recherche. Au terme de cette séance, l'étudiant sera en mesure d'effectuer des recherches de façon efficiente sur un ensemble de données.

Le code présenté dans les énoncés se trouve sur Moodle, dans le dossier **Ressources**.

## 1 Recherche binaire

**Question 1:** (🕒 10 minutes) Recherche binaire (Python)

Soient une liste d'entiers triés **L** et un entier **e**. Écrivez un programme qui retourne l'index de l'élément **e** de la liste **L** en utilisant une recherche binaire. Si **e** n'est pas dans **L**, retournez **-1**.

### >\_ Exemple

```
L = [123, 321, 328, 472, 549]
e = 328
```

Résultat attendu : 2

```
1 def recherche_binaire(liste,e):
2     first = 0
3     last = len(L)-1
4
5     #complétez ici
6
7
8 L = [123, 321, 328, 472, 549]
9 e = 328
10 recherche_binaire(L,e)
```

### 💡 Conseil

Inspirez-vous des conseils des exercices précédents (exercices basiques).

**Question 2:** (🕒 15 minutes) Recherche binaire - plus proche élément (Python)

Soient une liste d'entiers **triés** **L** et un entier **e**. Écrivez un programme retournant la valeur dans **L** la plus proche de **e** en utilisant une recherche binaire (binary search).

```
1 def plus_proche_binaire(liste,n):
2     #complétez ici
3
4
5 L = [1, 2, 5, 8, 12, 16, 24, 56, 58, 63]
6 e = 41
7 print(plus_proche_binaire(L,e))
8 # Résultat attendu : 56\\
```

### 💡 Conseil

Pensez à définir des variables **min** et **max** délimitant l'intervalle de recherche et une variable booléenne **found** initialisée à **false** et qui devient **true** lorsque l'algorithme trouve la valeur la plus proche de **e**.

**Question 3:** (🕒 20 minutes) Recherche dans une matrice (Python)

Considérez une matrice ordonnée **m** et un élément **l**.

Pour rappel, une matrice ordonnée répond aux critères suivants :

$m[i][j] \leq m[i+1][j]$  (une ligne va du plus petit au plus grand)

$m[i][j] \leq m[i][j+1]$  (une colonne va du plus petit au plus grand)

Écrivez un algorithme qui retourne la position de l'élément **l** dans **m**. Si **l** n'est pas présent dans **m** alors, votre programme retournera **(-1, -1)**.

>\_ Exemples

**Exemple 1** : si  $m = [[1,2,3,4],[4,5,7,8],[5,6,8,10],[6,7,9,11]]$  et que  $l=7$ . Nous souhaitons avoir la réponse **(1,2)** OU **(3,1)** (l'une des deux, pas besoin de retourner les deux résultats).

**Exemple 2** : si  $m = [[1,2],[3,4]]$  et que  $l=7$ . Nous souhaitons avoir la réponse **(-1,-1)** car 7 n'est pas dans la matrice **m**.

```
1 def recherche_matricielle(m,l):
2     #complétez ici
3
4
5     m=[[1,2,3,4],[4,5,7,8],[5,6,8,10],[6,7,9,11]]
6     l=7
7     recherche_matricielle(m,l)
```

💡 Conseil

Pour cet exercice, il est nécessaire d'utiliser des boucles **for** imbriquées, c'est-à-dire : une boucle **for** dans une autre boucle **for**. Cela permet de parcourir tous les éléments d'une liste (ou d'un tableau) à deux dimensions (dans notre cas une matrice).