

Algorithmes de recherche

Dans cette partie, nous n'utiliserons QUE DES LISTES DE DONNÉES TRIÉES !

1 Terminaison et variant

Rappel : une boucle `while` est une boucle **non bornée** donc potentiellement **infinie** !

1.1 Terminaison d'un algorithme

- Vérifier la terminaison d'un algorithme, c'est s'assurer que l'algorithme va s'arrêter.
- Vérifier la terminaison d'un algorithme, **ce n'est pas** vérifier sa validité (appelée correction d'un algorithme).

Exemple :

```
1 # afficher les impairs jusqu'à 10
2 i = 1
3 while i != 10:
4     print(i)
5     i = i + 2
```

La boucle ne se **termine** jamais : on passe de $i = 9$ à $i = 11$ et comme $11 \neq 10$, la boucle continue.

Solution : écrire `while i < 10`.

1.2 Variant de boucle

On appelle **variant d'une boucle** une expression dont la valeur varie à chacune des itérations de la boucle.

Dans l'exemple précédent, `i` est un variant de la boucle car sa valeur est incrémentée de 2 à chacune des itérations.

Un variant de boucle bien choisi permet de prouver qu'une boucle non bornée se termine.

2 Complexité (temporelle)

La complexité (temporelle) d'un algorithme est le temps utilisé par cet algorithme en fonction de la taille des données en entrée.

Ce temps correspond au **nombre d'étapes de calcul** avant d'arriver à un résultat.
(Notons que ce temps est négligeable pour de petites quantités.)

Pour évaluer la complexité d'un algorithme, on **calcule le nombre d'étapes** pour n données : comparaisons, calculs, affectations, etc.

Exemple :

```
1 lst = [3, 7, 10, 12, 13, 24]
2 for el in lst:
3     print(el)
```

Il y a $n = 6$ éléments dans la liste en entrée et une boucle sur cette liste, donc il y a 6 étapes de calcul (en l'occurrence, les étapes sont les affectations de `el`).

Complexité linéaire

Dans l'exemple précédent, il y a autant d'étapes que le nombre de données en entrée. On dira alors que **le coût de cet algorithme est linéaire** ou encore que **sa complexité est d'ordre n , notée $O(n)$** .

Exemple :

```
1 data = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
2 for i in range(3): # -> 3 étapes
3     for j in range(3): # -> 3 étapes
4         data[i][j] = i + j
```

En entrée, `data` est une liste de 3 éléments, on a donc $n = 3$. La première boucle `for` se fait en 3 étapes, la deuxième se fait en 3 étapes *pour chaque étape de la première boucle*. On a donc $3 \times 3 = 9$ étapes dans cet algorithme, soit n^2 .

Complexité quadratique

Dans l'exemple précédent, il y a n^2 étapes pour n données en entrée. On dira alors que **le coût de cet algorithme est quadratique** ou encore que **sa complexité est d'ordre n^2 , notée $O(n^2)$** .

Ainsi, quand on écrit un algorithme, on évalue sa **performance grâce à sa complexité**.

On peut analyser cette dernière **dans le pire des cas** (un maximum d'étapes sont exécutées), dans le cas moyen ou dans le meilleur des cas (un minimum d'étapes sont exécutées).

Notons qu'on ne s'intéressera cette année qu'au pire des cas.

3 Algorithmes de recherche d'éléments dans des listes triées

Il existe plusieurs façons de chercher une valeur dans un tableau (ou liste en Python).

Exercice 1

Écrire en Python une fonction `recherche(lst, val)` qui prend en argument une liste `lst` et une valeur `val`, et renvoie la liste des indices correspondant à la valeur `val`. (PC + papier)

→ Soit le tableau suivant, **valable pour tout ce qui suit** :

lst = [1, 4, 10, 14, 21, 30, 76, 78, 99]

3.1 La recherche séquentielle

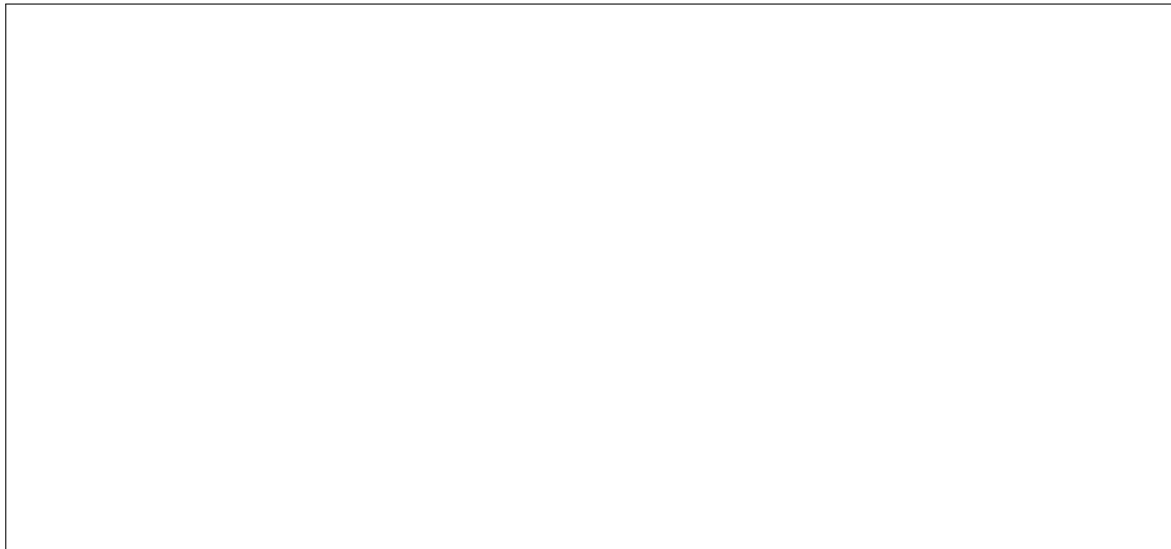
Il s'agit de parcourir toute la liste et de comparer l'élément courant à l'élément cherché à chaque itération.

Exercice 2

1. On cherche la valeur 14 dans la liste lst : il faut **4** tours de boucle pour la trouver ;
2. On cherche 208, non présent dans lst : il faut **9** tours de boucle pour constater qu'on ne trouve pas cette valeur (soit la taille de la liste en entrée). Concernant la complexité, on est donc dans le **pire des cas** car c'est le maximum d'itérations qu'on puisse faire avant d'avoir une réponse.
3. Quel élément faut-il chercher pour être dans le meilleur des cas concernant la complexité ?

On s'aperçoit que, si on a une très grande liste, cette façon de chercher devient rapidement très longue. (Souvenez-vous du nombre d'utilisateurs d'Instagram qui se compte en milliards en 2023.)

En **pseudo-code**, dit aussi **langage naturel**, l'algorithme de **recherche séquentielle** s'écrit de la manière suivante :

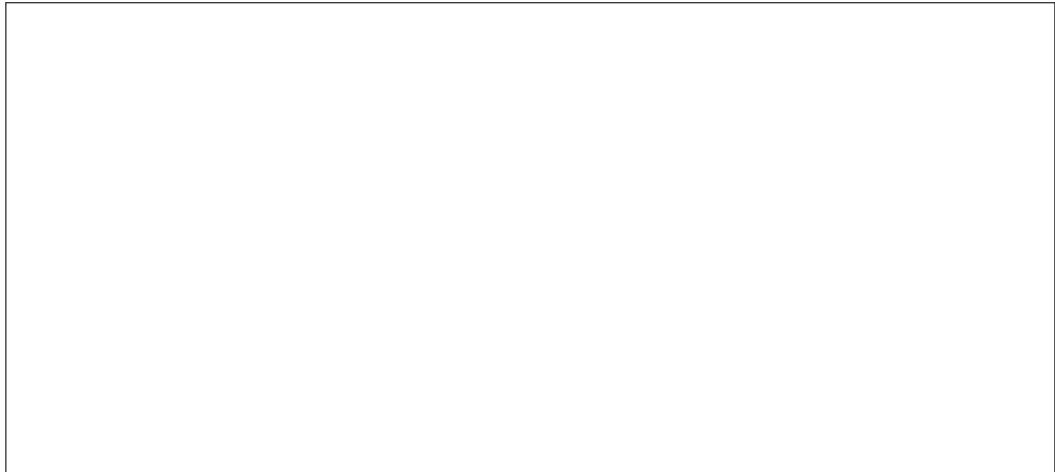


La fonction `recherche_sequentielle` prend en argument un tableau et une valeur cible, et renvoie l'indice de la valeur cible dans le tableau, -1 sinon.

Comme on l'a vu précédemment, on peut dire que la **complexité** de cet algorithme est **linéaire**, en $O(n)$, puisqu'il y a (environ) autant d'opérations que le nombre d'éléments en entrée.

Exercice 3

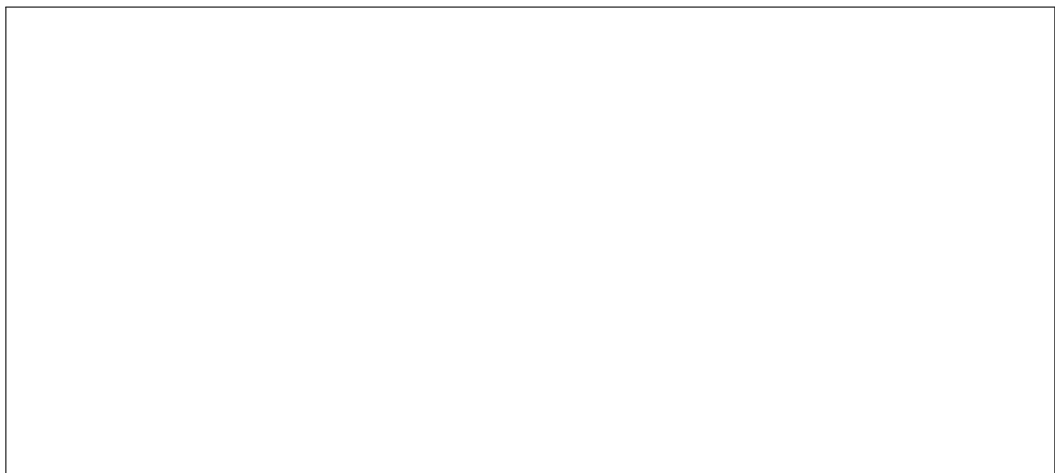
1. (a) Écrire en langage naturel l'algorithme de recherche séquentielle du plus grand élément d'un tableau.



- (b) Quel est le coût de cet algorithme ?

Coût linéaire, en $O(n)$

2. Mêmes questions pour l'algorithme de calcul de la moyenne des éléments du tableau.



(a)

- (b) Coût linéaire, en $O(n)$

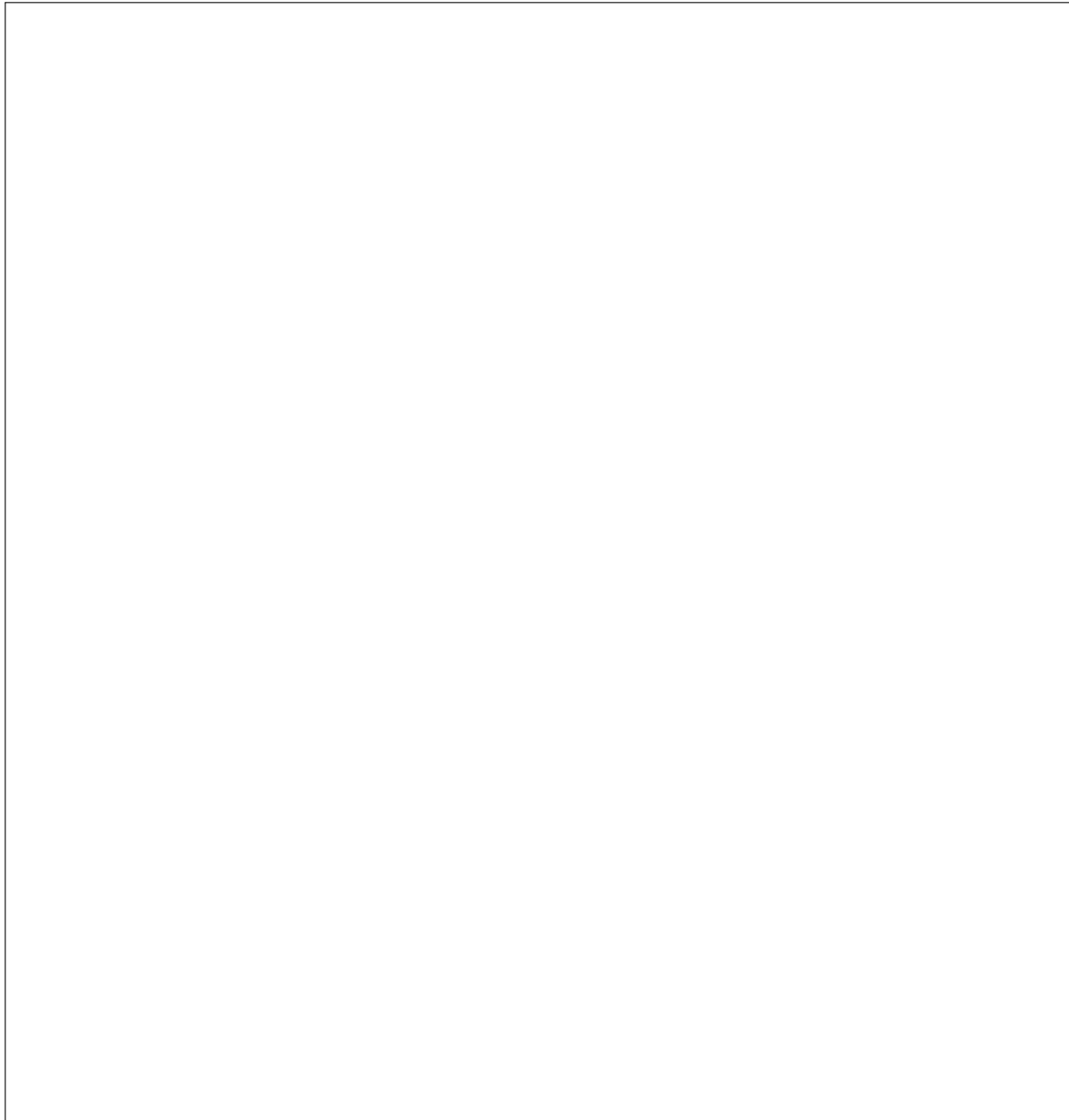
3.2 La recherche par dichotomie

Jeu : « Devine à quel nombre je pense. »

Assez intuitivement, on constate qu'une bonne façon de trouver le bon nombre dans le jeu précédent est de tester le nombre du milieu, ce qui réduit l'espace de recherche à la moitié de l'espace de recherche précédent, etc. jusqu'à trouver. C'est ainsi que fonctionne l'algorithme de recherche par dichotomie (du grec qui signifie littéralement « couper en deux »).

3.2.1 L'algorithme

Voici l'algorithme de recherche dichotomique en pseudo-code/langage naturel :



La fonction `recherche_dichotomique` prend en argument un tableau trié et une valeur cible, et renvoie l'indice de la valeur cible dans le tableau si celle-ci lui appartient, -1 sinon.

Exercice 4

On cherche la valeur 10 dans `lst` (définie précédemment, en tête de section) en appliquant l'algorithme de recherche dichotomique.

1. Compléter le tableau du déroulement de l'algorithme :

indices									
valeurs	1	4	10	14	21	30	76	78	99
tour 1	g				m				d
tour 2		m							

2. Combien y a-t-il d'éléments dans la liste ? **Il y a 9 éléments dans la liste.**
3. En combien d'étapes trouve-t-on l'élément recherché ? **On trouve en 3 étapes.**
4. En combien d'étapes aurions-nous trouvé cet élément en utilisant l'algorithme de recherche séquentielle ?
En 3 étapes aussi.

Exercice 5

On cherche la valeur 45 dans `lst` (définie précédemment, en tête de section) en appliquant l'algorithme de recherche dichotomique.

1. Compléter le tableau du déroulement de l'algorithme :

indices									
valeurs	1	4	10	14	21	30	76	78	99
tour 1									

2. Combien y a-t-il d'éléments dans la liste ? **Il y a 9 éléments dans la liste.**
3. En combien d'étapes trouve-t-on l'élément recherché ? **On trouve en 3 étapes.**
4. En combien d'étapes aurions-nous trouvé cet élément en utilisant l'algorithme de recherche séquentielle ?
En 9 étapes. Comme l'élément n'est pas présent dans la liste, on est obligé de tout regarder avant de pouvoir l'affirmer.

Exercice 6

Dérouler l'algorithme de recherche dichotomique avec les données ci-dessous.

- $t = [2, 5, 7, 8, 12, 16, 18, 20, 25, 30, 32]$ et $cible = 16$.

Tours de boucle	g	d	$g \leq d$	m	$t[m]$
1	0	10	True	5	16

- $t = [2, 5, 7, 8, 12, 16, 18, 20, 25, 30, 32]$ et $cible = 12$.

Tours de boucle	g	d	$g \leq d$	m	$t[m]$
1	0	10	True	5	16
2	0	4	True	2	7
3	3	4	True	3	8
4	4	4	True	4	12

- $t = [2, 5, 7, 8, 12, 16, 18, 20, 25, 30, 32]$ et $cible = 19$.

Tours de boucle	g	d	$g \leq d$	m	$t[m]$
1	0	10	True	5	16
2	6	10	True	8	25
3	6	7	True	6	18
4	7	7	True	7	20
5	7	6	False		

3.2.2 Complexité (de la recherche dichotomique)

Nombre de tours de boucle maximum :

Taille de la liste	1	2	4	8	16	32	64	128	
Recherche séquentielle	1	2	4	8	16	32	64	128	
Recherche dichotomique	1	2	3	4	5	6	7	8	

On dira alors que **le coût de cet algorithme est logarithmique** ou que **sa complexité est d'ordre $\log_2(n)$** , notée $O(\log_2(n))$.

($\log_2(x)$ est la fonction inverse de 2^x .)

Notons que **plus la liste en entrée est grande, plus l'algorithme de recherche dichotomique est efficace par rapport à la recherche séquentielle**.

3.2.3 Terminaison

Un (bon) variant de boucle est la quantité :

$indice_droit - indice_gauche \geq 0$ (voir ligne `while ind_g <= ind_d`)

Son évolution à chaque tour de la boucle `while` permet de montrer que celle-ci se termine.

Soit m l'indice du milieu, g l'indice gauche et d l'indice droit.

$$m = (g + d) // 2 \Rightarrow g \leq m \leq d$$

- 1ère structure conditionnelle :
si $t[m] == cible$ alors on renvoie $m \Rightarrow$ terminaison assurée ;

- 2^e structure conditionnelle :
si $cible < t[m]$ alors $d = m - 1 \Rightarrow$ le variant décroît ;
- Sinon :
 $g = m + 1 \Rightarrow$ le variant décroît ;
- Renvoie -1 si pas trouvé \Rightarrow terminaison assurée.

Donc la **terminaison est assurée dans tous les cas**.

3.2.4 Diviser pour régner

L'algorithme de recherche dichotomique est une illustration du paradigme de programmation **diviser pour régner** qui consiste à découper un problème initial en sous-problèmes plus simples. Ce paradigme fournit des algorithmes efficaces pour de nombreux problèmes, la dichotomie l'illustre bien.

Exercice 7

Programmer la fonction `recherche_dichotomique` en Python. (PC + papier pour mémoire)

