Algorithmes de recherche

Dans cette partie, nous n'utiliserons QUE DES TABLEAUX 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 :

```python
# afficher les impairs jusqu'à 10
i = 1
while i != 10:
    print(i)
    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 :

```python
lst = [3, 7, 10, 12, 13, 24]
for el in lst:
    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 :

```python
data = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
for i in range(3): # -> 3 étapes
    for j in range(3): # -> 3 étapes
        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 un tableau de type `list` nommé `lst` et une valeur `val`, et renvoie le tableau 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. Combien de tours de boucle faut-il pour trouver la valeur 14 ?

2. On cherche 208, non présent dans `lst` : combien faut-il de tours de boucle pour constater qu'on ne trouve pas cette valeur ?
3. Dans quel cas de complexité est-on ? Pourquoi ?
4. 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 nnombre d'utilisateurs d'Instagram qui se compte en milliards en 2023.)

Exercice 3

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 précéemment, 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 4

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 ?

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

   (a)

   (b)

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 :

```
fonction recherche_dichotomique(tableau t, valeur cible):
    indice_gauche ← indice du 1er element de t
    indice_droit ← indice du dernier element de t

    tant que indice_gauche ≤ indice_droit:
        indice_milieu ← (indice_gauche + indice_droit) // 2
        si t[indice_milieu] vaut cible:
            renvoyer indice_milieu
        si cible < t[indice_milieu]:
            indice_droit = indice_milieu - 1
        sinon:
            indice_gauche = indice_milieu + 1

    renvoyer -1
```

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 5

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

On considère que :

- `m` est `indice_milieu` ;
- `g` est `indice_gauche` ;
- `d` est `indice_droit`.

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 ?

3. En combien d'étapes trouve-t-on l'élément recherché ?

4. Que renvoie la fonction ?

5. En combien d'étapes aurions-nous trouvé cet élément avec l'algorithme de recherche séquentielle ?

Exercice 6

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 ?

3. En combien d'étapes trouve-t-on (ou pas) l'élément recherché ?

4. Que renvoie la fonction ?

5. En combien d'étapes aurions-nous trouvé (ou pas) cet élément en utilisant l'algorithme de recherche séquentielle ?

Exercice 7

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	$g <= d$	m	t$[m]$
1					

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

Tours de boucle	g	d	$g <= d$	m	t$[m]$
1					
2					
3					
4					

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

Tours de boucle	g	d	$g <= d$	m	t$[m]$
1					
2					
3					
4					
5					

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	n
Recherche séquentielle	1	2	4	8	16	32	64	128	n
Recherche dichotomique	1	2	3	4	5	6	7	8	$log_2(n)$

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

Démonstration rapide :

La dichotomie revient à couper en deux la liste jusqu'à ce qu'il n'y ait plus qu'un élément dedans. La question qu'on se pose pour évaluer la complexité est donc : combien de fois doit-on couper la liste en deux pour arriver à un seul élément ? Donc combien de fois doit-on diviser `n` par 2 ?

Si on traduit rapidement ceci en mathématiques, on cherche `a` dans $\frac{n}{2^a}$.

$\frac{n}{2^a} \Leftrightarrow n = 2^a$

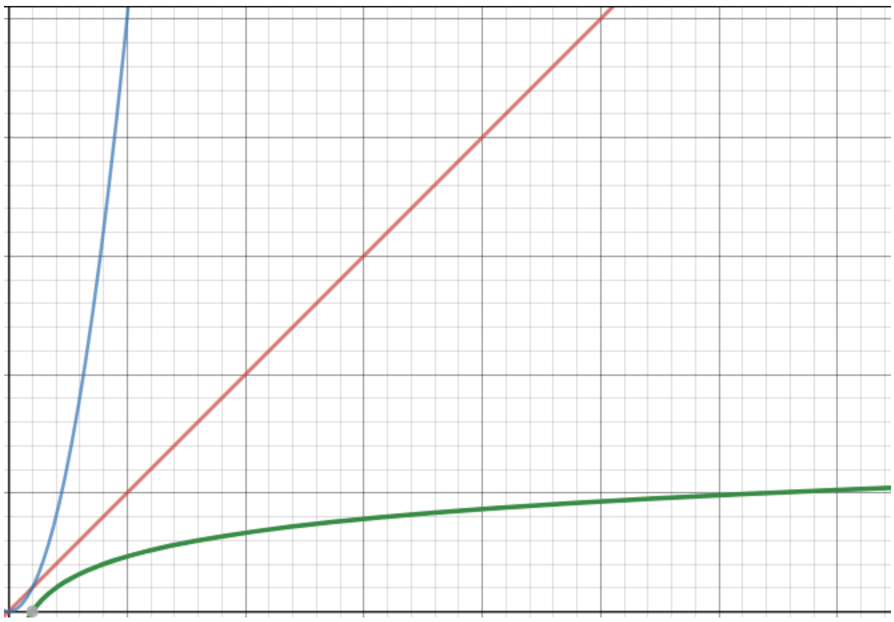
Pour information la fonction logarithme (programme de math en terminale) s'écrit $log_2(x)$ et est la fonction réciproque de 2^x.

$\Leftrightarrow log_2(n) = log_2(2^a)$

$\Leftrightarrow log_2(n) = a$ (voir propriété des fonctions réciproques)

Exercice 8

Associer chaque complexité vue (linéaire, quadratique et logarithmique) à une des courbes suivantes :



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 :

$$\texttt{indice\_droit - indice\_gauche} \geq 0$$

(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.

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

- 1ère structure conditionnelle :
  si $t[m] == cible$ alors on renvoie $m \Rightarrow$ terminaison assurée ;
- 2è 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 9

1. Télécharger le fichier `dicho_seq_display.py` sur notre site.
2. Programmer la fonction `recherche_sequentielle` en Python. **Les tests ne doivent pas êtres modifiés et doivent fonctionner !**
3. Programmer la fonction `recherche_dichotomique` en Python. **Les tests ne doivent pas êtres modifiés et doivent fonctionner !**
4. Lorsque les tests passent, décommenter la dernière ligne `plt.show()`, exécuter à nouveau le programme et observer le résultat. Que peut-on conclure ?