ALGORITHMES CLASSIQUES

I) Complexité d'un algorithme

Définition :

On appelle **complexité** d'un algorithme la façon dont le nombre d'*actions élémentaires* effectuées par cet algorithme évolue en fonction de la taille des données traitées. (On entend par action élémentaire : les affectations, comparaisons,…)

Ex : recherche d'un minimum dans une liste

```
def minimum(liste):
    min = liste[0]
    for item in liste:
        if item < min:
            min = item
    return min
```

Pour trouver le minimum, l'action élémentaire répétée un grand nombre de fois est la comparaison `item < min`. La fonction a fait autant de comparaisons qu'il y a d'éléments dans la liste. On dit que cet algorithme a une complexité **linéaire** : Si la liste est 10 fois plus grande, l'algorithme mettra environ 10 fois plus de temps à s'exécuter.

Vérification expérimentale :

En multipliant ci-dessous `taille_liste` par 10, 100 ou 1000, comment évolue la durée du script ?

```
import random, time

taille_liste = 10**5
liste = [random.randint(0,100) for i in range(taille_liste)]

début = time.time()
a = minimum(liste)
fin = time.time()

print(f"n = {taille_liste},  durée = {fin - début} secondes")
```

Remarques :

- On distingue parfois la complexité « en temps » et la complexité « en mémoire ». Nous ne considérerons cette année que la complexité en temps.
- Quand on cherche à déterminer la complexité d'un algorithme, on ne cherche pas à compter précisément le nombre d'actions élémentaires effectuées : un simple ordre de grandeur suffit.
- La complexité d'un algorithme peut être notamment :
  - constante (ne dépend pas de la taille n des données traitées)
  - linéaire (proportionnelle à n, par exemple lors d'une boucle simple)
  - quadratique (proportionnelle à n^2, par exemple s'il y a 2 boucles imbriquées)
- Une vidéo remarquable sur la question : https://www.youtube.com/watch?v=AgtOCNCejQ8

II) Recherche d'une valeur dans une liste

1) Cas d'une liste non triée : Recherche par « balayage »

On veut écrire une fonction **`recherche_balayage`** déterminant si une valeur est dans une liste ou non. Bien sûr, nous nous interdisons d'utiliser le mot clé « **`in`** » de Python qui ferait le travail à notre place ;-)

Principe :

Cette fonction va balayer la liste de gauche à droite à la recherche de la valeur demandée et s'arrêter dès que cette dernière est trouvée. A la fin du balayage, si la valeur demandée n'a pas été trouvée, alors cette valeur n'est pas dans la liste.

Code (à compléter) ***:***

```
def recherche_balayage(valeur, liste):
    for 
        if valeur == item:
            return 
    return 

liste = [3, 9, 5, 17, 12, 0, -4]
assert recherche_balayage(9, liste)
assert recherche_balayage(-4, liste)
assert not recherche_balayage(10, liste)
```

Complexité :

Quelle est la complexité de cet algorithme « dans le pire des cas » ?

2) Cas d'une liste triée : Recherche par dichotomie

Si la liste est triée, on peut résoudre le même problème de façon plus astucieuse.

Exemple : Le nombre 15 appartient-il à la liste triée ci-dessous ?

item	2	3	5	5	9	11	13	14	17	19	20
indice	0	1	2	3	4	5	6	7	8	9	10

Principe :

On va diviser par deux la plage de recherche à chaque étape en comparant 15 avec le terme qui est au milieu de cette plage.

	Étapes				
	1	2	3	4	5
Indice du début de plage : `a =`	0				
Indice de fin : `b =`	10				
Indice du milieu : `i =`	5				
On compare 15 avec :	11				

Code (à compléter) ***:***

```
def recherche_dichotomie(valeur, liste):
    a = 
    b = 
    while a <= b:
        i = (a+b)//2
        if valeur == liste[i]:
            
        elif valeur > liste[i]:
            
        else:
            
    return False

liste = [2,3,5,5,9,11,13,14,17,19,20]
assert not recherche_dichotomie(15, liste)
assert recherche_dichotomie(11, liste)
assert recherche_dichotomie(20, liste)
```

Complexité :

La complexité de cet algorithme est meilleure que celle de l'algorithme par balayage. En effet, même s'il y a plus d'actions élémentaires par passage dans la boucle, il y a beaucoup beaucoup moins de passages dans la boucle !

Observons le tableau ci-dessous :
D'une colonne à la suivante, on a multiplié par 10 la taille de la liste de départ.
D'une ligne à la suivante, on a divisé par deux la taille de la liste comme le fait la dichotomie.

100	1000	10000	100000	1000000	10000000	100000000	1000000000	10000000000	100000000000
50	500	5000	50000	500000	5000000	50000000	500000000	5000000000	50000000000
25	250	2500	25000	250000	2500000	25000000	250000000	2500000000	25000000000
12	125	1250	12500	125000	1250000	12500000	125000000	1250000000	12500000000
6	62	625	6250	62500	625000	6250000	62500000	625000000	6250000000
3	31	312	3125	31250	312500	3125000	31250000	312500000	3125000000
1	15	156	1562	15625	156250	1562500	15625000	156250000	1562500000
	7	78	781	7812	78125	781250	7812500	78125000	781250000
	3	39	390	3906	39062	390625	3906250	39062500	390625000
	1	19	195	1953	19531	195312	1953125	19531250	195312500
		9	97	976	9765	97656	976562	9765625	97656250
		4	48	488	4882	48828	488281	4882812	48828125
		2	24	244	2441	24414	244140	2441406	24414062
		1	12	122	1220	12207	122070	1220703	12207031
			6	61	610	6103	61035	610351	6103515
			3	30	305	3051	30517	305175	3051757
			1	15	152	1525	15258	152587	1525878
				7	76	762	7629	76293	762939
				3	38	381	3814	38146	381469
				1	19	190	1907	19073	190734
					9	95	953	9536	95367
					4	47	476	4768	47683
					2	23	238	2384	23841
					1	11	119	1192	11920
						5	59	596	5960
						2	29	298	2980
						1	14	149	1490
							7	74	745
							3	37	372
							1	18	186
								9	93
								4	46
								2	23
								1	11
									5
									2
									1
6	9	13	16	19	23	26	29	33	36

En dernière ligne, on a précisé le nombre d'étapes qui ont été nécessaires pour atteindre 1.

On remarque que quand on multiplie par 10 la taille de la liste, il faut ☐ au nombre d'étapes.

Trouverez-vous sur internet le nom de la fonction mathématique célèbre qui permet de faire ce genre de choses ?

On retiendra que la complexité de la recherche par dichotomie est ☐

Terminaison :

Dans cet algorithme, nous avons utilisé une boucle **`while`**. Or il y a toujours un risque que ce genre de boucle soit infinie ! On appelle **prouver la terminaison** d'un algorithme le fait de prouver que l'exécution de l'algorithme va s'arrêter quelles que soient les entrées.

Pour cela, nous allons montrer que **`b - a`** est un **variant de boucle**, c'est-à-dire un entier positif qui décroît strictement à chaque itération.

Ici l'algorithme s'arrête quand le variant de boucle **`b - a`** devient strictement négatif.
A chaque itération, on entre dans la boucle uniquement si **`a ≤ b`**,
puis, après modification de i, on obtient : **`a ≤ i ≤ b`**.
En fin de boucle :

- soit **`a = i + 1`**.
  Donc **`a`** est incrémenté d'au moins 1 et le variant de boucle est décrémenté d'au moins 1
- soit **`b = i - 1`**.
  Donc **`b`** est décrémenté d'au moins 1 et le variant de boucle est aussi décrémenté d'au moins 1

Bilan, à chaque itération le variant de boucle décroît d'au moins 1. Il finira donc bien par devenir négatif.

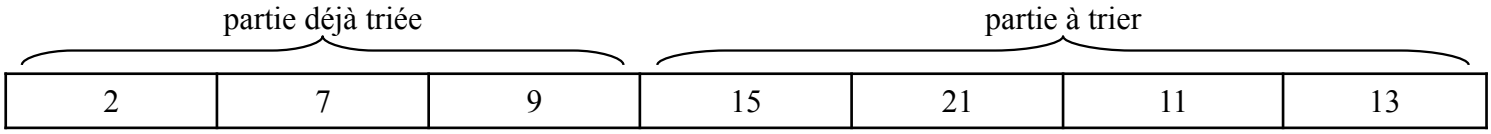
III) Algorithmes de Tri

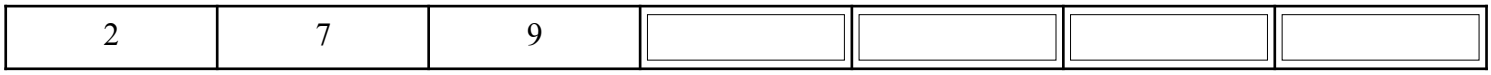
1) Tri par sélection

Principe :

A chaque étape d'un **tri par sélection**, la partie gauche de la liste est déjà triée. On parcourt alors la partie droite à la recherche du minimum que l'on échange avec le premier terme de cette partie droite.

Exemple d'étape de tri :





La même chose avec une animation visuelle : https://animations.interstices.info/methodes-tri/

Code (à compléter) ***:***

```
def tri_selection(liste):
    # Dans la partie restant à trier :
    #   i désigne l'indice du premier terme
    #   imin désigne l'indice du minimum

    for i in range

        imin =

        for j in range

            if liste[j] < liste[imin]:

                imin =

        # On permute le minimum de la partie à trier avec le premier terme

        liste[i], liste[imin] =

from random import randint
for i in range(10):
    l = [randint(-10, 10) for i in range(10)]
    tri_selection(l)
    assert l == sorted(l)
```

Complexité :

Si n est la taille de la liste, le nombre de fois où on exécute la boucle intérieure est :

$(n-1) + \ldots + 4 + 3 + 2 + 1 =$

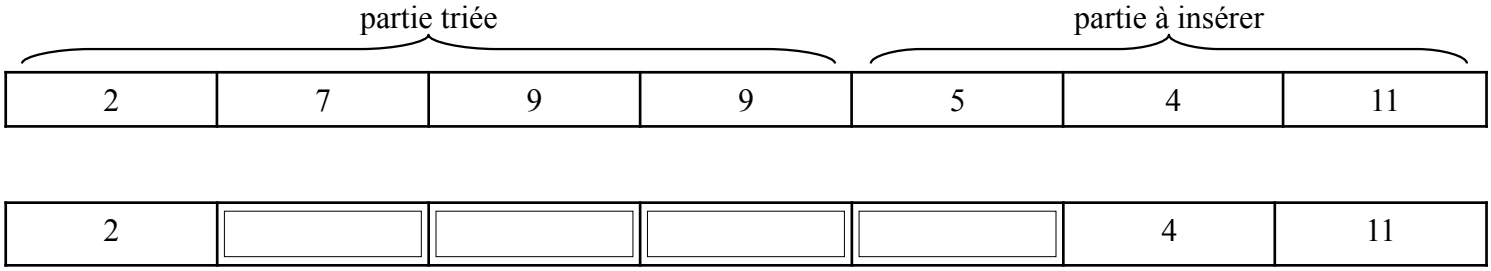
La complexité est donc

2) Tri par insertion

Principe :

A chaque étape d'un **tri par insertion**, la partie gauche de la liste est déjà triée et on cherche à y insérer au bon endroit le premier nombre de la partie droite. On sauvegarde donc ce nombre à insérer et on le compare successivement à chacun des nombres de la partie triée en commençant par les plus grands. Si le nombre de la partie triée est plus grand, on le décale d'une case à droite, sinon on écrase le nombre de droite par le nombre à insérer.

Exemple d'étape de tri :



Code (à compléter) ***:***

```
def tri_insertion(liste):
    # i désigne l'indice du terme à insérer (=le plus à gauche de la partie non triée)
    # j désigne l'indice du terme de la partie triée auquel on le compare
    for i in 
        terme_à_insérer = 
        j = 
        while j >= 0 and 
            liste[j+1] = 
            j -= 1
        liste[j+1] = 

from random import randint
for i in range(10):
    l = [randint(-10, 10) for i in range(10)]
    tri_insertion(l)
    assert l == sorted(l)
```

Complexité :

Si n est la taille de la liste, le nombre de fois où on exécute la boucle while est dans le pire des cas :

$1 + 2 + 3 + 4 + \ldots + (n - 1) =$ ☐ La complexité est donc ☐

Terminaison :

Il y a une boucle **`while`**. Choisissons **`j`** comme variant de boucle (première condition du **`while`**). A l'intérieur de la boucle, **`j`** est décrémenté donc ce variant finira donc bien par devenir négatif même si la deuxième condition du **`while`** reste toujours vraie.

IV) Algorithme des k plus proches voisins

L'exemple des Iris de Fisher

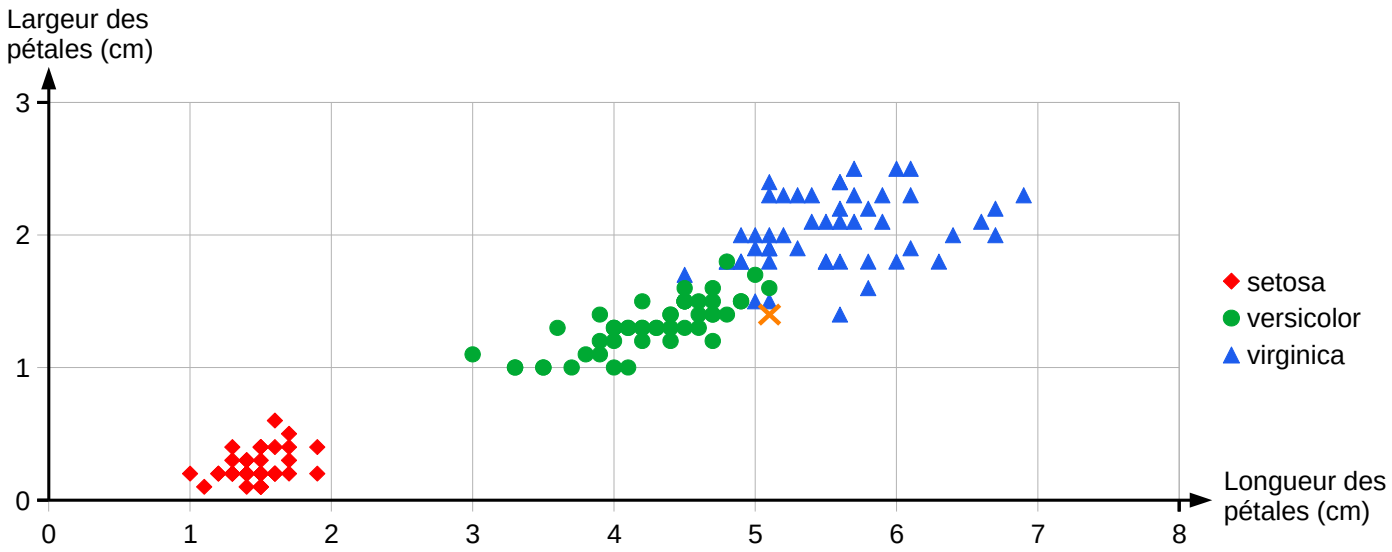
L'algorithme des **k plus proches voisins** appelé aussi **knn** (k nearest neighbors) est un algorithme de classification très simple qui fait partie de la famille des algorithmes d'apprentissage (machine learning).
Dans tout algorithme d'apprentissage, il faut un **jeu de données**. Nous utiliserons ci-dessous l'exemple très célèbre des « iris de Fisher ». Dans ce jeu de données, un botaniste a mesuré les pétales de 150 iris de trois espèces différentes : Les iris setosa, les iris versicolor et les iris virginica.

Iris setosa

Iris versicolor

Iris virginica

On a représenté les données mesurées par le botaniste dans la figure ci-dessous :



Principe de l'algorithme :

On vient de trouver un iris dans un champ et on cherche son espèce en s'appuyant sur notre jeu de données :

- On choisit une valeur de k appropriée (k = le nombre de voisins que l'on va considérer).
- On calcule les distances entre l'iris que l'on a trouvé et chaque iris du jeu de données.
- On identifie l'espèce majoritaire parmi les k plus proches voisins de l'iris trouvé.

Application :

Dimensions des pétales	Espèce si k = 2	Espèce si k = 5
1,5 cm × 0,3 cm		
4 cm × 1 cm		
7 cm × 2 cm		
iris symbolisé par une croix orange		

Remarques :

- Pour calculer les distances, nous avons utilisé ici la distance euclidienne : $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$.
  Rappel, pour visualiser une distance euclidienne sur un graphique, le repère doit être orthonormé !
- Selon les situations, on peut aussi utiliser d'autres types de distances : Manhattan, Hamming,...
- Le choix de k (nombre de voisins retenus) se fait souvent empiriquement en testant plusieurs valeurs.

Exercices :

1) Écrire un script Python qui :
- demande à l'utilisateur la longueur et la largeur d'un nouveau pétale d'iris.
- crée la liste des iris du jeu de données « iris.csv ». Chaque iris sera représenté par une sous-liste de la forme : [distance euclidienne entre le pétale de l'iris et le nouveau pétale, longueur du pétale de l'iris, largeur du pétale de l'iris, espèce de l'iris].
- trie cette liste de liste selon les distances entre le nouvel iris et les iris du jeu de données.
- Affiche les espèces des 5 plus proches voisins du nouvel iris.

2) Utiliser l'algorithme des k plus proches voisins pour aider le choixpeau magique de Poudlard à déterminer la maison d'Hermione.
- Jeu de données : « choixpeau-magique.csv ».
- Hermione a pour caractéristiques : Courage=8, Loyauté=6, Sagesse=6, Malice=6.
- k = 7
- Distance utilisée : Manhattan (faites une recherche sur le web pour savoir de quoi il s'agit ;-)

3) Utiliser l'algorithme des k plus proches voisins pour décider si un texte est écrit en français ou en anglais.
On s'appuiera pour cela sur le fait que la fréquence des « h » et celle des « y » sont assez différentes dans les deux langues et on créera un jeu de données en analysant ces fréquences dans quelques textes des deux langues.

V) Algorithmes gloutons

1) Problèmes d'optimisation

Un **problème d'optimisation** est un problème dont la difficulté n'est pas de trouver des solutions mais plutôt de trouver la « meilleure » solution.

Exemples :

- Rendre la monnaie avec un minimum de pièces.
- Mettre le plus de livres possible dans un carton.
- Trouver la route la plus rapide entre deux villes.

2) Algorithmes glouton

Pour résoudre un problème d'optimisation, on peut chercher toutes les solutions, puis définir ensuite la meilleure. Malheureusement, dans certains cas, trouver toutes les solutions peut être très long, même pour un ordinateur !
On peut alors essayer d'utiliser un **algorithme glouton** qui va procéder étape par étape, en faisant à chaque étape le meilleur choix possible à ce moment là.

Remarques :

- Le gros intérêt des algorithmes gloutons est qu'ils sont beaucoup plus rapides que la recherche de toutes les solutions.
- En revanche, on verra ci-dessous qu'un algorithme glouton ne trouve pas toujours la meilleure des solutions et même parfois peut ne pas trouver de solution du tout alors qu'il en existe !

3) Le rendu de monnaie

Commençons par un grand classique : Comment rendre la monnaie avec un nombre minimal de pièces.

a) Système de monnaie habituel

On doit rendre 162 euros en utilisant les pièces suivantes en nombre illimité : 100 ; 50 ; 20 ; 10 ; 5 ; 2 ; 1 €.

	Somme restant à rendre	Pièces rendues
Étape 0	162	aucune
Étape 1	62	100
Étape 2	12	100 ; 50
Étape 3	2	100 ; 50 ; 10
Étape 4	0	100 ; 50 ; 10 ; 2

Le principe est simple : à chaque étape, on rend la pièce la plus grosse possible et notre système de monnaie est fait de telle sorte que cet algorithme donnera toujours la meilleure solution.

b) Système de monnaie exotique

On doit rendre 162 euros en utilisant maintenant les pièces suivantes en nombre illimité : 50 ; 40 ; 3 ; 2 €.
En appliquant le même algorithme que ci-dessus, combien de pièces devra-t-on rendre ?
Peut-on trouver une solution permettant de ne rendre que 5 pièces ?
Avec ce nouveau système de monnaie, nous voyons que l'algorithme glouton permet de trouver une solution pour 162 euros mais celle-ci n'est pas optimale.
De même cet algorithme n'aurait pas permis de rendre 160 euros alors que 160 = 40 + 40 + 40 + 40.

c) Code (à compléter)

```
def rendu_glouton(somme_à_rendre, monnaie):
    """
    Paramètres :
        somme_à_rendre : int
        monnaie : tuple ou list trié dans l'ordre décroissant
    Valeur renvoyée :
        liste_pièces_à_rendre : list
        (renvoie une liste vide dans le cas où l'algorithme ne trouve pas de solution)
    """
    liste_pièces_à_rendre = []
    
        
            
    return liste_pièces_à_rendre

assert rendu_glouton(162, (100, 50, 20, 10, 5, 2, 1)) == [100, 50, 10, 2]
assert rendu_glouton(162, (50, 20, 10, 5, 2, 1)) == [50, 50, 50, 10, 2]
assert rendu_glouton(162, (50, 40, 3, 2)) == [50, 50, 50, 3, 3, 3, 3]
assert rendu_glouton(160, (50, 40, 3, 2)) == []
```

d) Conclusion

Si on veut dénombrer toutes les façons de rendre la monnaie pour choisir ensuite la meilleure, on trouve déjà plus de 30 000 possibilités dans le cas a) !
Certes notre algorithme glouton n'est optimal que si le système de monnaie est bien conçu mais il a quand même l'avantage de nous conduire directement à une bonne solution !

4) Le problème du sac à dos

Autre grand classique !
Vos parents vous envoient acheter ce qu'il faut pour l'apéritif. Vous avez le droit aux articles ci-dessous avec deux contraintes : Vous n'avez pas le droit de choisir deux fois le même article et tout doit tenir dans un sac ridicule qui contient au maximum 600 g !
Qu'allez vous choisir pour maximiser le prix du contenu du sac à dos sans dépasser la masse autorisée ?

Article	Chips bio	Pistaches	Curly	Noix de pécan	Saucisson
Masse (g)	100	300	100	100	300
Valeur (€)	2,4	4,55	1,30	3,80	6,25
Valeur/Masse	0,024	0,015	0,013	0,038	0,021

a) Premier essai :

Vous décidez de choisir en premier les articles les plus chers. Est-ce un algorithme glouton ?
Quels articles prenez-vous ? Quelle est la valeur du sac ?

b) Deuxième essai :

Vous décidez de choisir en premier les articles ayant le ratio « valeur/masse » le plus avantageux.
Est-ce un algorithme glouton ? Quels articles prenez-vous ?
Quelle est la valeur du sac ?

Code (à compléter)

```
def sac_à_dos_glouton(masse_max, liste_articles):
    articles = sorted(liste_articles, reverse = True)
    contenu_sac_à_dos =
    valeur_totale =
    masse_totale =
    for article in articles:
        
            
            
            
    return contenu_sac_à_dos, valeur_totale, masse_totale

liste_articles = [[0.024, 2.4, 100, "Chips bio"],
                  [0.015, 4.55, 300, "Pistaches"],
                  [0.013, 1.3, 100, "Curly"],
                  [0.038, 3.8, 100, "Noix de pécan"],
                  [0.021, 6.25, 300, "Saucisson"]]
assert sac_à_dos_glouton(600, liste_articles) == (['Noix de pécan', 'Chips bio',
'Saucisson', 'Curly'], 13.75, 600)
```

c) Conclusion :

Si on fait abstraction de la contrainte des 600 g, combien y a-t-il de façon de remplir le sac avec 5 articles ?
Même question avec 15 articles ?
On voit que l'étude de toutes les solutions est de complexité exponentielle alors que la complexité de l'algorithme glouton (au tri de la liste des articles près) est linéaire.

Pour approfondir la question : https://interstices.info/le-probleme-du-sac-a-dos/