Le JEU pour REVISER les bases du PYTHON

Récapitulatif des cours présents dans le jeu • **Chapitre 5**

Table des matières

Les algorithmes de Première

Partie 2

Les **algorithmes** sont des ensembles d'instructions composants les programmes informatiques.

Certains algorithmes ont été étudiés et travaillés pour répondre à des problématiques précises. En Première, vous avez dû voir les 5 algorithmes suivants :

- Le tri par **sélection** d'une liste
- Le tri par **insertion** d'une liste
- La recherche **dichotomique** dans une liste
- L'algorithme **glouton**
- L'algorithme **k plus proches voisins**

3. Les algorithmes gloutons

Les algorithmes gloutons permettent d'obtenir (de manière optimisée) un résultat optimal.

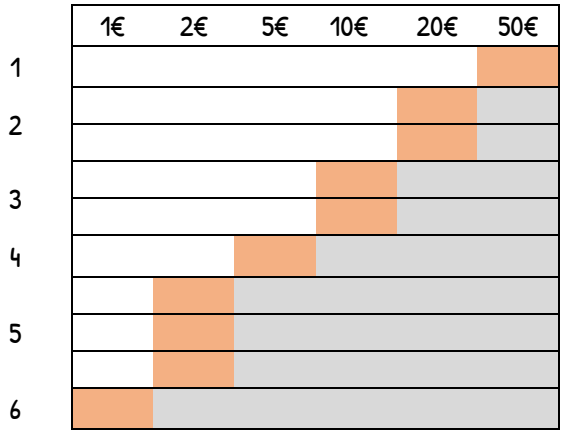
L'utilisation la plus connue de l'algorithme glouton est le rendu de monnaie.

Tester toutes les possibilités de pièces/billets pour rendre la monnaie est un algorithme « naïf », peu efficace.

L'algorithme consiste à rendre la pièce (ou le billet) le plus élevée et inférieure à la somme à rendre. On répète le processus tant que la pièce est inférieure à la somme, sinon on répète avec une pièce plus petite.

Exemple : On cherche à rendre 34€.

On sait que le résultat optimal est un billet de 20€, un de 10€ et deux pièces de 2€.



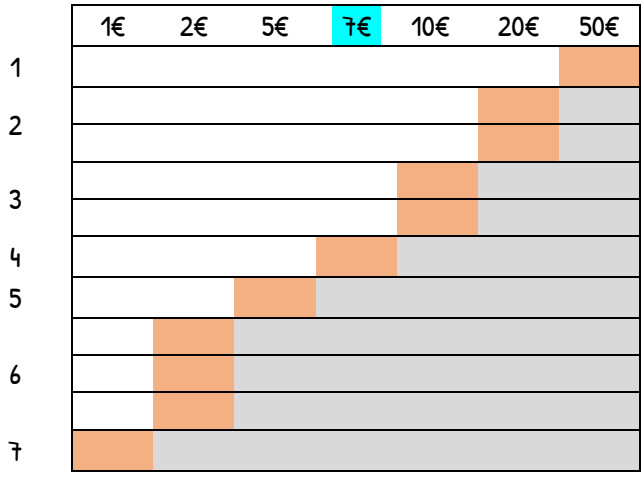
Pièce	Prendre	Reste à rendre
50€	non	34€
20€	oui	14€
	non	14€
10€	oui	4€
	non	4€
5€	non	4€
2€	oui	2€
	oui	0€
	non	0€
1€	non	0€

Le rendu de monnaie donne toujours le résultat optimal. C'est grâce aux valeurs des pièces et billets.

Que ce passe-t-il si on ajoute un billet de 7€ ?

Exemple : On cherche à rendre 34€.

On sait que le résultat optimal est un billet de 20€ et deux billets de 7€.



Pièce	Prendre	Reste à rendre
50€	non	34€
20€	oui	14€
	non	14€
10€	oui	4€
	non	4€
7€	non	4€
5€	non	4€
2€	oui	2€
	oui	0€
	non	0€
1€	non	0€

Le fait que l'on prenne toujours la plus grande valeur d'abord ne permet pas d'obtenir le résultat optimal. L'avantage de cet algorithme est qu'il est plus rapide que de tester toutes les possibilités.

Passons à un autre exemple très connu : Le problème de **Sac à dos**.

Vous devez remplir un sac avec des objets. Chaque objet a un poids et un prix. Le sac à un poids maximum et votre but est d'avoir un sac avec un prix total d'objets le plus élevé possible.

L'algorithme consiste à remplir le sac avec l'objet ayant le ratio prix/poids le plus élevé et dont le poids est inférieur au poids maximal du sac. On répète le processus tant que le poids de l'objet est inférieur au poids maximum, sinon on répète avec un objet dont le ratio est le plus petit.

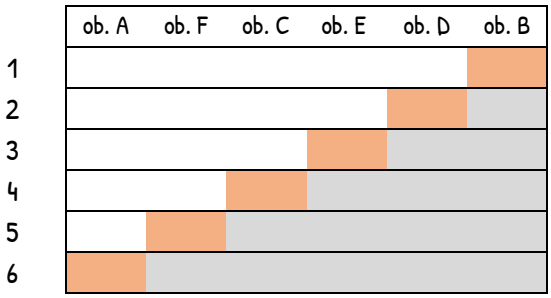
Exemple : On cherche à remplir un sac de 20kg maximum.

La liste des objets :

	Prix	Poids	Ratios
Objet A	1€	10kg	0,1
Objet B	50€	2kg	25
Objet C	30€	30kg	1
Objet D	20€	2kg	10
Objet E	3€	1kg	3
Objet F	10€	20kg	0,5

Tri
n° 6
n° 1
n° 4
n° 2
n° 3
n°5

Pour l'exemple, on supposera que chaque objet est unique, donc si l'objet sélectionné, on le supprime.



Poids	Prendre	Prix total	Poids restant
2kg	oui	50€	18kg
2kg	oui	+ 20€	16kg
1kg	oui	+ 3€	15kg
30kg	non		15kg
20kg	non		15kg
10kg	oui	+ 1€	5kg
		74€	

Le fait que l'on prenne toujours la plus grande valeur d'abord ne permet pas toujours d'obtenir le résultat optimal. Tout dépend des objets, de leur ratio et du poids maximum du sac.

Voyons comment généraliser les étapes des algorithmes gloutons !

Les principales étapes d'un algorithme glouton :

	Etapes	Exemples	
		Rendu de monnaie	Sac à dos
1	Sélectionner les élément	Les pièces (ou billets) ont une seule caractéristique : une valeur	Les objets ont 2 caractéristiques : un prixet un poids
2	Sélectionner un objectif	La somme à rendre	Le poids maximum du sac
3	Si les éléments ont plusieurs caractéristiques, on calcule un ratio	X	Calcul : prix/poids
4	Trier les éléments dans l'ordre décroissant pour avoir les éléments préférables en premiers	Tri des pièces par valeur	Tri des objets par ratio (si le calcul précédent est inversé poids/valeur, il faut trier par ordre croissant) [1]
5	Initialiser une liste pour lister les éléments utilisés.		
6	Prendre le premier élément	La première pièce	Le premier objet
7	Si la valeur de l'élément est inférieure ou égale à l'objectif : soustraire la valeur à l'objectif et on ajoute l'élément à la liste (*et si l'élément est unique, on passe au suivant*). Sinon, on passe à l'élément suivant.	Si la valeur de la pièce est inférieure ou égale à la somme à rendre : soustraire la valeur à la somme et on ajoute la pièce à la liste (*et si la pièce est unique, on passe à la suivante*). Sinon, on passe à la pièce suivante.	Si le poids de l'objet est inférieur ou égal au poids maximum : soustraire le poids de l'objet au poids max et on ajoute l'objet à la liste (*et si l'objet est unique, on passe au suivant*). Sinon, on passe à l'objet suivant.
8	Recommencer l'étape 6 tant que l'objectif n'a pas atteint 0 et qu'il reste des éléments à parcourir (*on stoppe le programme si l'une des deux conditions est fausse*).	Recommencer l'étape 6 tant que la somme n'a pas atteint 0 et qu'il reste des pièces à parcourir.	Recommencer l'étape 6 tant que le poids maximum n'a pas atteint 0 et qu'il reste des éléments à parcourir.

Il ne s'agit que des étapes principales. Certains algorithmes plus complexes utilisant le principe des algorithmes gloutons ont davantage d'étapes, ainsi que des variations.

Passons maintenant aux codes Python de ces deux exemples !

[1] Dans notre exemple, le poids est une contrainte, donc on divise la valeur par le ratio.
Explications : L'objet A vaut 1€ et pèse 10kg (valeur/poids = 0,1) ; L'objet B vaut 50€ et pèse 2kg (valeur/poids = 25). L'objet B est préférable à l'objet A et son ratio est le plus élevé (sauf si le calcul est inversé).

Rendu de monnaie

Etapes

1
2
3
6
8
7

```
monnaie = [500, 200, 100, 50, 20, 10, 5, 2, 1,
      0.5, 0.2, 0.1, 0.05, 0.02, 0.01]
somme_a_rendre = 34.81
pieces_rendu = []
i = 0
while somme_a_rendre > 0 and i < len(monnaie):
    if monnaie[i] <= somme_a_rendre:
        piece = monnaie[i]
        somme_a_rendre = somme_a_rendre - piece
        pieces_rendu.append(piece)
    else:
        i += 1

print(pieces_rendu)
```

Sac à Dos

Etapes

1
2
3
4
5
6
8
7

```
objets = [{"nom": "ObjetA", "Prix": 1, "Poids": 10},
          {"nom": "ObjetB", "Prix": 50, "Poids": 2},
          {"nom": "ObjetC", "Prix": 30, "Poids": 30},
          {"nom": "ObjetD", "Prix": 20, "Poids": 2},
          {"nom": "ObjetE", "Prix": 3, "Poids": 1},
         {"nom": "ObjetF", "Prix": 10, "Poids": 20}  ]
poids_max = 20
objets_selec = []
prix_total = 0

# Calcul du ratio
for elm in objets:
    elm["Ratio"] = elm["Prix"] / elm["Poids"]

# Tri par ratio
objets = sorted(objets, key=lambda x: x["Ratio"], reverse=True)
i = 0
while poids_max > 0 and i < len(objets):
    if objets[i]["Poids"] <= poids_max:
        poids_max = poids_max - objets[i]["Poids"]
        prix_total = prix_total + objets[i]["Prix"]
        objets_selec.append(objets[i])

    i += 1

print(poids_max, prix_total)
print(objets_selec)
```

La fonction **sorted** est une fonction Python permettant de trier des listes.

(cela ne vous empêche pas de connaître les algorithmes de tri vus précédemment !)

Sans entrer dans les détails :

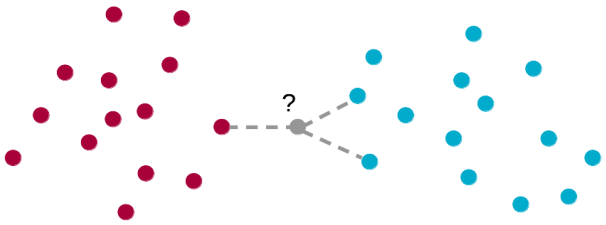
- `key=lambda x: x["Ratio"]` indique que le tri s'effectuera en fonction de la colonne "Ratio" de chaque dictionnaire de la ligne.
- `reverse=True` indique que le tri sera décroissant.

4. Les algorithmes k plus proches voisins

Les algorithmes k plus proches voisins sont des algorithmes d'apprentissage automatique (machine learning)[2].

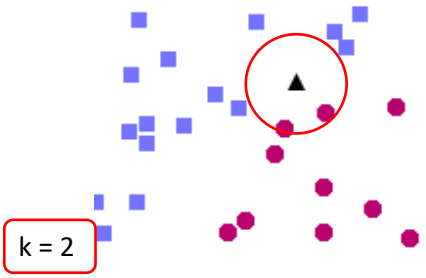
Ces algorithmes utilisent la proximité d'un élément avec d'autres pour classifier ou prédire.

Le principe est de regarder les k points les plus proches d'un point donné. Si une majorité de ces points appartiennent à une catégorie, on peut supposer que ce point appartient à la même catégorie.

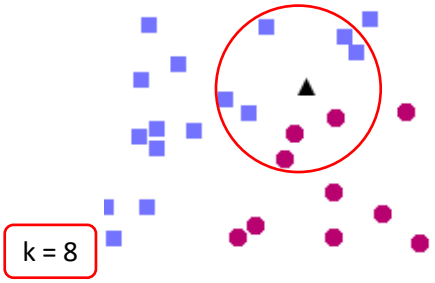


Tout comme l'algorithme glouton, un algorithme k plus proches voisins ne permet pas toujours d'obtenir le résultat optimal. Plusieurs facteurs sont à prendre en compte :

- Le nombre des points déjà catégorisés => Peu de points Beaucoup de points
- La qualité de ces points => Une catégorie surreprésentée Catégories équilibrées
- Le k utilisé => Le nombre choisi peu tout changé :



Catégorie majoritaire : Violet



Catégorie majoritaire : Bleu

[2] Un champ d'étude de l'Intelligence Artificielle

Prenons un exemple : On cherche l'espèce d'une iris.

On dispose des caractéristiques de la fleur (la taille, la largeur et la longueur des pétales, etc.) et ces mêmes caractéristiques pour toutes les iris dont l'espèce est connue.

Toutes ses caractéristiques nous permettent de comparer l'iris inconnue à chaque iris connue.

Pour connaître les iris plus proches, on calcule la distance[3] entre la fleur et chaque iris.

Calcul d'une distance : *On note A,B, C, etc. les caractéristiques de la fleur et d'une iris connue.*

Distance = |fleurA - irisA| + |fleurB - irisB| + |fleurC - irisC| + …

L'**espèce** majoritaire parmi k plus proches iris de la fleur inconnue sera considérée comme l'**espèce** de cette fleur.

Les principales étapes d'un algorithme k plus proches voisins :

	Etapes	Le problème de l'iris
1	Sélectionner les données connues	Les iris connues (et leurs caractéristiques)
2	Sélectionner les catégories présentes parmi ces données	Sélectionner les espèce d'iris connues
3	Sélectionner une donnée « mystère »	L'iris inconnue (et ses caractéristiques)
4	Sélectionner un entier k	Sélectionner un entier k
5	Calculer la distance entre la donnée (2) et chaque donnée connue (1) en fonction de leurs caractéristiques communes	Calculer la distance entre l'iris inconnue et chaque iris connue en fonction de leurs caractéristiques communes
6	Trier les distances calculées pour chaque donnée connue par ordre croissant	Trier les iris connues par ordre croissant en fonction de la distance calculées
7	Sélectionner les k premières données triées	Sélectionner les k premières iris
8	Parmi ces k données, trouver la catégorie majoritaire	Parmi ces k iris, trouver l'espèce majoritaire

Il ne s'agit que des étapes principales. Certains algorithmes plus complexes utilisant le principe des algorithmes k plus proches voisins ont davantage d'étapes, ainsi que des variations.

Passons maintenant au code **Python** de cet exemple !

[3] Il existe plusieurs façons de calculer une distance. Pour ce cours, on utilisera la distance de Manhattan.

Pour plus de lisibilité, certaines parties du programme seront placées dans des fonctions.

Etapes

5

```
from copy import deepcopy

def calculs_distances(liste_iris, nouvelle_iris):
      L = deepcopy(liste_iris)
      # Rappel : il est toujours préférable de copier une liste dans une fonction
      for iris in L:
            # Distance de Manhattan
            a = abs(nouvelle_iris["Longueur"] - iris["Longueur"])
            b = abs(nouvelle_iris["Largeur"] - iris["Largeur"])
            iris["Distance"] = a + b # Ajout au dictionnaire de chaque iris

      return L
```

8

```
def majoritaire(liste_iris, especes):
      # Initialiser un compteur de chaque espèce
      nb_especes = {}
      for e in especes:
            nb_especes[e] = 0

      maj_espece = especes[0]
      # Parcours des k iris
      for iris in liste_iris:
            e = iris["Especes"]
            nb_especes[e] += 1  # Mise à jour du compteur de l'espèce

            # Si le compteur de l'espèce est plus élevé que celui de l'espèce
            #  majoritaire, l'espèce devient l'espèce majoritaire
            if nb_especes[e] > nb_especes[maj_espece]:
                  maj_espece = e

      return maj_espece
```

1

```
iris_connues = [ {'Longueur': 1.4, 'Largeur': 0.2, 'Especes': 'Setosa'},
                 {'Longueur': 1.4, 'Largeur': 0.2, 'Especes': 'Setosa'},
                 ...
                 {'Longueur': 4.7, 'Largeur': 1.4, 'Especes': 'Virginica'},
                 {'Longueur': 4.5, 'Largeur': 1.5, 'Especes': 'Virginica'},
                 ...
                 {'Longueur': 6.0, 'Largeur': 2.5, 'Especes': 'Versicolor'},
                 {'Longueur': 5.1, 'Largeur': 1.8, 'Especes': 'Versicolor'}]
```

2

```
especes = ["Setosa", "Virginica", "Versicolor"]
```

3

```
iris_mystere = {"Longueur": 3.2, "Largeur": 0.8}
```

4

```
k = 7
```

5

```
iris_avec_dist = calculs_distances(iris_connues, iris_mystere)
```

6

```
iris_triees = sorted(iris_avec_dist, key=lambda x: x["Distance"])
```

7

```
selection = iris_triees[:k]
```

8

```
espece = majoritaire(selection, especes)
```

En savoir plus sur les fonctions

- https://fr.wikipedia.org/wiki/Algorithme_glouton
- https://fr.wikipedia.org/wiki/M%C3%A9thode_des_k_plus_proches_voisins
- Ou demande à ton/ta prof de NSI :)