ATgorithmique

ATgorithmes De Tri

1. Introduction

a. *Qu'est-ce que le tri ?*

On sait déjà que les tableaux permettent de stocker plusieurs valeurs sous la forme d'une séquence ordonnée dans laquelle chaque valeur est associée à un indice. Il peut alors être intéressant de ranger ces valeurs dans un ordre croissant ou décroissant. Trier un tableau, c'est donc ranger les éléments de ce tableau dans un ordre défini (croissant ou décroissant).
Le tri possède d'innombrables applications. Une fois que des données sont triées, il est très rapide de comparer deux tableaux par exemple. Cela peut nous permettre aussi par exemple de déterminer simplement la valeur min, la valeur max, la médiane etc...

b. *De quoi avons-nous besoin ?*

Pour fonctionner correctement, un algorithme de tri doit respecter la spécification suivante :
EN ENTRÉE : Ensemble d'éléments tous comparables deux à deux par un ordre $\leq$
EN SORTIE ; Éléments triés selon cet ordre du plus petit au plus grand

c. *Que peut-on trier ?*

Dans la plupart des cas, on triera des entiers dans un tableau par ordre croissant. C'est le cas le plus simple et celui que l'on va étudier. Cependant, les tris exposés ici sont généralisables à d'autres autres situations.
Par exemple, nous pourrions trier des chaînes de caractères :

"a" ≤ "b", "char" ≤ "charles", "char" ≤ "chat" ...

et pourquoi pas, par exemple, des cartes à jouer. Si je définis que ♣ ≤ ♦ ≤ ♠ ≤ ♥ , et
2 ≤ 3 ≤ 4 ≤ … ≤ 10 ≤ V ≤ D ≤ R ≤ As alors :
As♣ ≤ 4♦, V♠ ≤ D♠, 3♦ ≤ 2♥, ...

Nous allons commencer par 2 algorithmes "classiques" de tri : le tri par sélection et le tri par insertion.

2. Tri par sélection

a. *Qu'est-ce que le tri ?*

Le tri par sélection (ou tri par échange/extraction) est sans doute le tri le plus simple à imaginer.
Le principe est de chercher le plus petit élément du tableau et de le mettre en premier. Ensuite, repartir du second élément et chercher le plus petit élément du tableau restant pour le mettre en second, etc..

b. *Explication :*

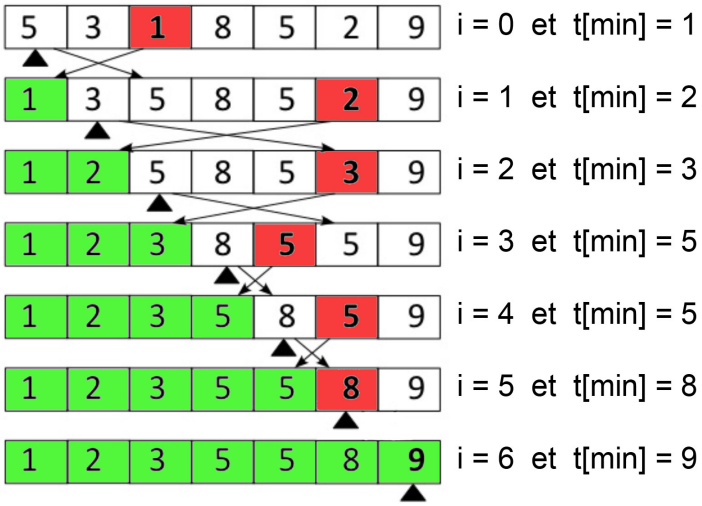
Le tri par sélection parcourt le tableau de la gauche vers la droite, en maintenant sur la gauche une partie déjà triée et à sa place définitive :

déjà trié	pas encore trié

A chaque étape, on cherche le plus petit élément dans la partie droite non triée, puis on l'échange avec l'élément le plus à gauche de la partie non triée. Ainsi, la première étape va déterminer le plus petit élément et le placer tout à gauche du tableau. Puis la deuxième étape va déterminer le deuxième plus petit élément et le placer dans la deuxième case du tableau et ainsi de suite.
Par construction, la partir gauche déjà triée ne contient que des éléments inférieurs ou égaux à ceux de la partie de droite restant à trier.

	0	*i*
t	éléments déjà triés	éléments plus grands non triés

Sur ce dessin, on matérialise bien le fait que l'élément d'indice *i* n'est pas encore examiné et ne fait donc pas partie des éléments déjà triés. En particulier, à la première itération de la boucle, *i* vaut 0 et la partie gauche est donc vide.



c. *Algorithme*

```
def tri_selection(tab):
    """
    Donnée : prend un tableau tab de longueur n
    Résultat : renvoie le tableau trié par ordre croissant
    """
    for i in range(len(tab) - 1):
        indice_min = i

        for j in range(i+1,len(tab)) :
            if tab[j] < tab[indice_min] :
                indice_min = j
        if indice_min != i :
            temp = tab[i]
            tab[i] = tab[indice_min]
            tab[indice_min] = temp
    return tab

# Test
print(tri_selection([0,8,3,4,9,50,2]))
```

```
[0, 2, 3, 4, 8, 9, 50]
```

Il est légitime de se demander si c'est là une façon efficace de trier un tableau.

d. *Complexité*

i. Notion d'efficacité

La notion de complexité d'un algorithme va être directement liée à l'efficacité de cet algorithme. Pour un même problème, par exemple trier un tableau, il existe plusieurs algorithmes, certains algorithmes sont plus efficaces que d'autres (par exemple un algorithme A mettra moins de temps qu'un algorithme B pour résoudre exactement le même problème, sur la même machine).
Il existe 2 types de complexité : une complexité en temps et une complexité en mémoire.

Nous nous intéresserons ici uniquement à la complexité en temps. La complexité en temps est directement liée au nombre d'opérations élémentaires qui doivent être exécutées afin de résoudre un problème donné. Plus il y a d'opérations à effectuer, plus la complexité est grande et donc plus le temps de calcul est grand. Autrement dit notre objectif est d'obtenir des algorithmes avec la complexité la plus base possible.

ii. Mise en situation

Pour établir la complexité de l'algorithme du tri par sélection, nous allons comptabiliser le nombre de comparaisons entre 2 entiers.

Prenons par exemple le tableau

t	12	8	23	10	15

Si nous nous intéressons à l'étape qui nous permet de passer de

12	8	23	10	15	à	8	12	23	10	15

($i = 0$) nous avons **4 comparaisons** : 12 avec 8, puis 8 avec 23, puis 8 avec 10 et enfin 8 avec 15.

Si nous nous intéressons à l'étape qui nous permet de passer de

8	12	23	10	15	à	8	10	23	12	15

($i = 1$) nous avons **3 comparaisons** : 12 avec 23, puis 12 avec 10, et enfin 10 avec 15.

Si nous nous intéressons à l'étape qui nous permet de passer de

8	10	23	12	15	à	8	10	12	23	15

($i = 2$) nous avons **2 comparaisons** : 23 avec 12 et 12 avec 15

Si nous nous intéressons à l'étape qui nous permet de passer de

8	10	12	23	15	à	8	10	12	15	23

($i = 3$) nous avons **1 comparaison** : 23 avec 15

Pour trier un tableau comportant 5 éléments nous avons :

4 + 3 + 2 + 1 = 10 comparaisons

Dans le cas où nous avons un tableau à trier qui contient n éléments, nous aurons :

$$(n-1)+(n-2)+(n-3)+\ldots+2+1=\frac{n(n-1)}{2}=\frac{1}{2}n^2-\frac{1}{2}n \text{ comparaisons.}$$

Preuve :

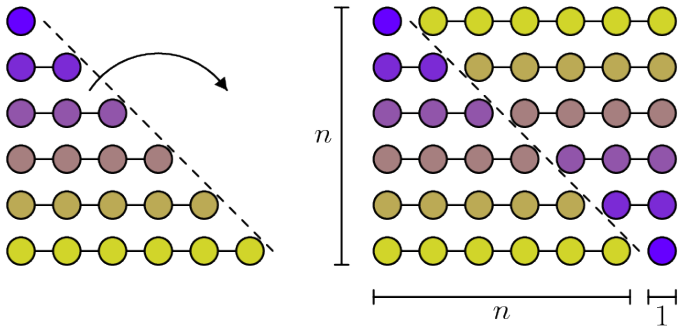
(1) Par le calcul :

On ajoute deux fois cette somme en inversant l'ordre :

$$\begin{array}{rcccccc} & 1 & + & 2 & +\ldots & +n-2 & +n-1 \\ + & n-1 & +n-2 & & +\ldots & +2 & +1 \\ \hline & \underbrace{n+n+\ldots+n+n}_{n\times(n-1)} & & & & & \end{array}$$

c'est-à-dire $2(1+2+\ldots+n-1)=n(n-1)$, ce qu'il fallait démontrer à une division près.

(2) Avec un simple schéma :



On néglige les coefficients ainsi que la partie en n, qui progresse bien moins vite que la partie en n^2 et est négligeable pour les grandes valeurs de n. On note la complexité $O\left(n^2\right)$, on dit que la complexité est **quadratique**.
On verra ci-après un algorithme un peu plus efficace.

e. Preuve

Un algorithme se prouve tout comme un théorème : une bonne intuition ne suffit pas à montrer que l'algorithme fonctionne.
La preuve se fait en deux temps :

- D'abord on montre que l'algorithme se finit (preuve de terminaison/d'arrêt) ;
- Puis on montre que l'algorithme donne le résultat attendu (preuve de correction/validité).

La preuve de terminaison est simple : l'algorithme étant basé sur deux boucles **pour**, la boucle s'arrête forcément. La variable d'itération de la boucle **pour** est appelée **variant** de boucle.

Remarque : la recherche d'un variant est indispensable pour prouver la terminaison d'un programme. C'est souvent un entier positif qui décroît strictement, le programme s'arrêtant quand cette quantité vaut 0 (ou un autre nombre positif). Dans le cas que l'on vient d'étudier, le variant augmente, mais la boucle s'arrête quand le variant atteint une valeur maximale prédéfinie (la longueur du tableau).

La preuve de correction fait appel à un **invariant** de boucle, c'est-à-dire à une propriété qui reste toujours vraie lors de l'exécution du programme, avant et après chaque itération, et qui en assure la validité. L'invariant que l'on utilise ici est : « la partie gauche du tableau est triée ».
Une fois trouvée un invariant, on procède ensuite en trois étapes : initialisation, conservation (ou transmission), terminaison. L'initialisation est la vérification de l'invariant à la première étape, la conservation permet de conserver la propriété invariant lors d'une itération : si l'invariant est vrai avant la boucle, alors il est vrai après la boucle. Enfin la terminaison assure que le programme renvoie le résultat demandé lors de la dernière itération.
Initialisation : lors de la première étape de la boucle, le tableau de gauche ne contient pas d 'élément, il est donc trié car vide.
Conservation : on suppose que le tableau a cette structure :

	Avant la 2ème boucle de variable d'itération j	
Indices 0 à i	Indice $i+1$	Indices $i+2$ à (longueur du tableau – 1)
déjà trié	Pas encore trié	pas encore trié

La 2ème boucle du programme parcourt tous les éléments d'indice $i+1$ à $n-1$, où n est la longueur du tableau. Cette boucle recherche le plus petit élément présent dans la partie droite du tableau, et l'échange avec l'élément d'indice $i+1$. Après exécution de la boucle le tableau est :

	Après 2ème boucle de variable d'itération j	
Indices 0 à i	Indice $i+1$	Indices $i+2$ à (longueur du tableau – 1)
déjà trié	trié	pas encore trié

L'invariant est bien vérifié après l'itération, l'étape de transmission est juste.
Terminaison : à la dernière étape, la partie de droite « pas encore triée » est vide vu que la boucle de variable d'itération i s'arrêt au dernier élément du tableau. Le tableau est entièrement trié : l'algorithme est correct.

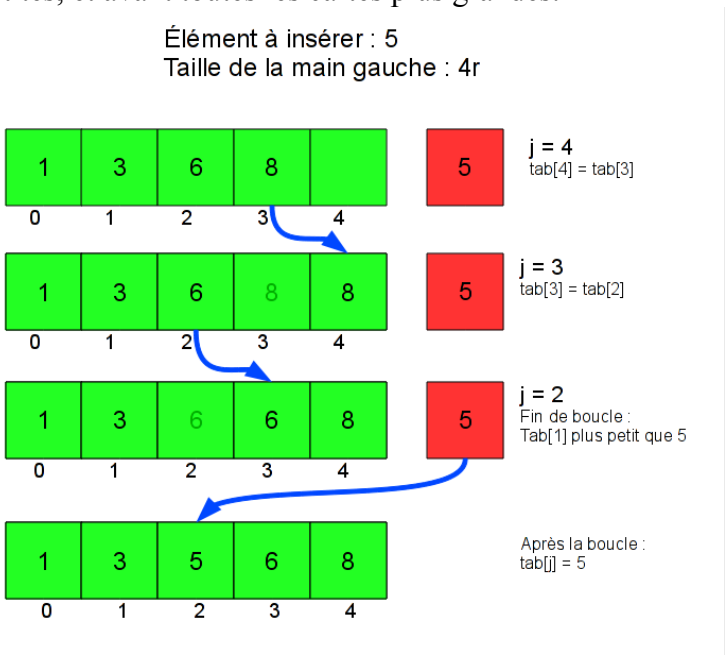
3. Tri par insertion

a. Principe

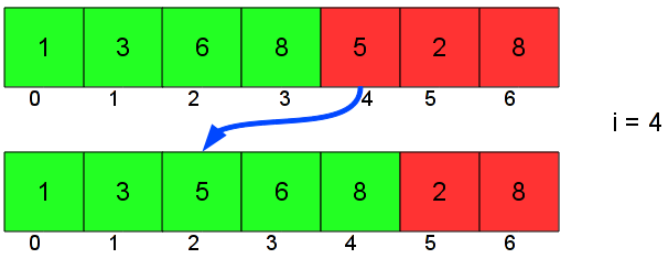
C'est le tri du joueur de cartes. On fait comme si les éléments à trier étaient donnés un par un, le premier élément constituant, à lui tout seul, une liste triée de longueur 1. On range ensuite le second élément pour constituer une liste triée de longueur 2, puis on range le troisième élément pour avoir une liste triée de longueur 3 et ainsi de suite...
Le principe du tri par insertion est donc d'insérer à la nième itération le nième élément à la bonne place.

b. Explication

Pour mieux visualiser, on va considérer que les cartes non triés sont dans notre main gauche tandis que et les cartes triés sont dans notre main droite. Commençons tout d'abord par l'opération d'insertion d'une carte dans la main droite depuis la main gauche. On veut placer la carte après toutes les cartes plus petites, et avant toutes les cartes plus grandes.



Pour faire cela avec un tableau, on a dû décaler certaines cartes : 6 était en position 2 avant l'insertion, elle est en position 3 après. De même, la carte 8 a été décalée. Plus généralement, il faut décaler d'une case vers la droite toutes les cartes plus grandes que la carte à insérer.
Pour faire cela, une bonne méthode est de commencer par la droite : on décale la carte la plus à droite (8), puis celle juste à gauche (6), jusqu'au moment où on tombe sur une carte plus petite que celle qu'on veut insérer, qu'il ne faut pas décaler. Une fois qu'on a fait ces décalages, on peut insérer la carte, à la position à laquelle on s'est arrêté de décaler.
Donc si on considère que la main gauche correspond au début du tableau, qui est déjà trié, et que la main droite correspond au reste du tableau, on obtient :



On décale donc d'une case vers la droite tous les éléments déjà triés qui sont plus grand que la valeur à insérer (5), puis on dépose cette dernière dans la case ainsi libéré.

Il est assez simple à visualiser que si la valeur à insérer *v* est plus grande que toutes les valeurs de la partie triée, *v* ne changera pas de place et on passera à la valeur suivante. De même, si *v* est plus petite que toutes les valeurs de la partie triée, on devra déplacer l'ensemble des valeurs triées vers la droite.

c. *Algorithme*

```
def tri_insertion(tab):
    """
    Donnée : prend un tableau tab de longeur n
    Résultat : renvoie le tableau trié par ordre croissant
    """
    for i in range(1,len(tab)):
        valeur = tab[i]
        j = i
        while j>0 and tab[j-1]>valeur:
            tab[j]=tab[j-1]
            j = j-1
        tab[j]=valeur
    return tab
#Test
print(tri_insertion([0,8,3,4,9,50,2]))
[0, 2, 3, 4, 8, 9, 50]
```

d. *Complexité*

i. Le "pire des cas"

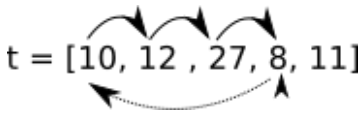
Nous allons nous intéresser à la complexité en temps dans "le pire des cas". On parle de "complexité dans le pire des cas" quand on s'intéresse uniquement au cas où le nombre d'opérations élémentaires est le plus grand.

À quoi correspond le pire des cas pour un algorithme de tri ? Tout simplement quand le tableau initial est "trié à l'envers" (les entiers sont classés du plus grand au plus petit), comme dans cet exemple :

t	5	4	3	2	1

ii. Principe

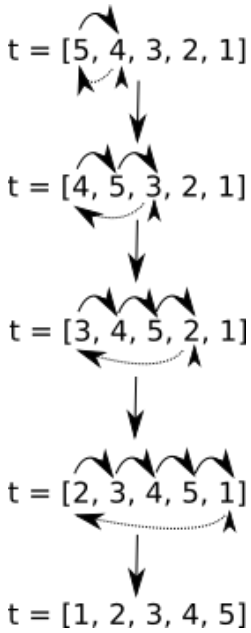
Pour déterminer la complexité de l'algorithme de tri par insertion nous n'allons pas rechercher le nombre d'opérations élémentaires, mais dans un soucis de simplicité, directement nous intéresser au "nombre de décalages effectués" pour trier entièrement un tableau. On compte comme décalage, le nombre d'éléments du tableau à déplacer pour insérer une valeur.



Pour l'étape ci-dessus nous avons 3 décalages (décalages du 10, du 12 et du 27). Nous ne tiendrons pas compte du "placement" du nombre en cours de traitement (8 dans notre exemple) symbolisé par la flèche du bas.

iii. Exemple et généralisation

Évaluons le nombre de décalages nécessaires pour trier le tableau t = [5, 4, 3, 2, 1]



1 + 2 + 3 + 4 = 10 décalages

Dans le cas où nous avons un tableau à trier qui contient n éléments, nous aurons :

$$(n-1)+(n-2)+(n-3)+\ldots+2+1=\frac{n(n-1)}{2}=\frac{1}{2}n^2-\frac{1}{2}n \text{ comparaisons.}$$

(puisque pour 5 éléments nous avons 1 + 2 + 3 + 4).

Vous avez sans doute déjà remarqué que nous avons un résultat similaire au tri par sélection (sauf que nous nous intéressons ici aux décalages, alors que pour le tri par insertion nous nous intéressions aux comparaisons, mais cela ne change rien au problème).

Nous allons donc trouver exactement le même résultat que pour le tri par sélection : une complexité en $O(n^2)$, quadratique.

En revanche, il existe des situations où le tri par insertion est meilleur que le tri par sélection.

iv. Le meilleur des cas, conclusion.

Supposons par exemple que le tableau soit déjà trié et examinons ce qui se passe. Pour chaque tour de boucle de *i*, la valeur à insérer est donc déjà bien positionné. Par conséquent, l'algorithme ne fait pas de tour de boucle "while".

Ainsi donc, le tri par insertion se limite à *n* appels. Dans le meilleur des cas, la complexité est **linéaire**, en $O(n)$.

Plus généralement, le tri par insertion se comporte favorablement si le tableau est "presque trié", ce qui arrive plus souvent qu'on ne le pense avec des données réalistes.

e. Preuve

A faire en exercice sur le modèle du tri par sélection.

f. A retenir

Le tri par sélection et le tri par insertion sont deux algorithmes de tri élémentaires, qui peuvent être utilisés pour trier des tableaux. le tri par insertion a un meilleur comportement que le tri par sélection lorsque le tableau est presque trié. Les deux restent peu efficaces dès que les tableaux contiennent plusieurs milliers d'éléments. Il existe de meilleurs algorithmes de tri, plus complexes, dont celui offert par Python avec la méthode sort() ou la fonction sorted().

Algorithmes gloutons

Les algorithmes gloutons sont utilsiés pour résoudre des problèmes d'optimisation, où l'on cherche une « bonne » solution, si possible la meilleure, à un problème complexe. La qualité des solutions est mesurée par une fonction mathématique.
Un algorithme glouton va effectuer une succession de choix, sans jamais les remettre en question (pas de retour en arrière), pour construire étape par étape une solution au problème donné.

1. Exemple 1 : problème du sac à dos

a. Énoncé :« Étant donné plusieurs objets possédant chacun un poids et une valeur, et étant donné une contenance poids maximum pour le sac, quels objets faut-il mettre dans le sac de manière à optimiser la valeur totale ? »

b. Données :

Objets	A	B	C	D	E
Valeur	10	7	1	3	2
Poids	9	12	2	7	5

Poids maximal dans le sac : 15 kg.

c. Solution optimale : la solution optimale est de prendre les objets A et E où on obtient 12 pour un poids de 14 kg.

d. Méthode brute : tout tester y compris ce qui est idiot du point de vue humain.
Complexité de la méthode brute, exemple :

- On peut mettre 0 objet dans le sac : 1 possibilité
- On peut mettre 1 objet dans le sac : 5 possibilités (A, B, C, D ou E)
- On peut mettre 2 objets dans le sac : 10 possibilités
  - AB, AC, AD, AE (comme il n'y pas d'ordre, cela donne aussi BA, CA, DA, EA)
  - BC, BD, BE
  - CD, CE
  - DE
- On peut mettre 3 objets dans le sac : 10 possibilités.
  - Plutôt que de faire la liste comme on l'a fait avec deux objets, on peut remarquer que mettre 3 objets dans le sac c'est en laisser 2 en dehors du sac. On se retrouve ramené au cas de 2 objets.
- On peut mettre 4 objets dans le sac : 5 possibilités (on laisse 1 objet en dehors du sac, même raisonnement que 3 objets par rapport à 2)
- Enfin, on peut mettre les 5 objets dans le sac : 1 possibilité.
- On a au total $1+5+10+10+5+1=32$ possibilités
- Avec 4 objets $1+4+6+4+1=16$ (pour 2 objets on a AB, AC, AD, BC, BD, CD)
- Avec 6 objets $1+6+15+20+15+6+1=64$ (à vérifier

Conclusion : on montre que la complexité est en $O\left(2^n\right)$. On peut le conjecturer à l'aide des résultats précédents ; en effet $2^4=16$, $2^5=32$ et $2^6=64$. La complexité est **exponentielle**. Les temps de calcul augmentent très rapidement. Pour $n=250$, l'ordre de grandeur du temps de calcul est de 625 µs en complexité $O\left(n^2\right)$ (tri par sélection) , alors il est de… 10^{59} années en complexité $O\left(2^n\right)$ (on rappelle que l'âge de l'univers est à peu près $14{,}5\times10^9$ années). Avec $n=300$, $2^{300}\simeq2\cdot10^{90}\gg10^{84}$ qui est le nombre d'atomes dans l'univers à quelques-uns près.

e. Principe de l'algorithme glouton

Le glouton prend le meilleur plat, puis celui qui lui plait le plus, etc. jusqu'à satiété. L'algorithme fonctionne suivant le même principe : il effectue une suite de choix en prenant le meilleur choix localement, sans jamais revenir en arrière.

Retour sur l'exemple : qu'est-ce que c'est que le « meilleur choix » ?
Il est nécessaire d'avoir des données triées, avec un critère de tri à déterminer. Dans le problème de sac à dos, un « bon » critère est le rapport *valeur*/*poids*.
Les données sont complétées par la ligne V/P :

Objets	A	B	C	D	E
Valeur	10	7	1	3	2
Poids	9	12	2	7	5
V/P	1,1	0,58	0,5	0,43	0,4

Les données doivent être triées pour que le « glouton » les ingurgite dans le bon ordre.

Qualité de la solution : l'algorithme glouton va sélectionner l'objet A puis l'objet C, pour une valeur de 11. La solution n'est pas optimale, mais s'en rapproche.
Complexité : Le parcours de la liste triée est en $O(n)$, négligeable devant le coût du tri en $O(n^2)$ –si on utilise un des algorithmes de tris vu auparavant–, ou en $O(n \log n)$ –si on utilise les tris intégrés à Python–. La complexité finale est donc quadratique, ou encore moins (linéarithmique), bien plus rapide que la complexité exponentielle.

Conclusion : un algorithme glouton offre un bon compromis rapidité/qualité de la solution obtenue.

- Dans certains cas, on obtient la meilleure solution : rendu de monnaie

2. Exemple 2 : Rendu de monnaie

Énoncé du problème : « Étant donné un système de monnaie (pièces et billets), comment rendre une somme de façon optimale, c'est-à-dire combien de pièces/billets doit on rendre au minimum ? »

a. Système européen

Le système européen est conçu de telle manière que le nombre de pièces (ou billets) donnés pour rendre la monnaie est optimal si on utilise un algorithme glouton.
Sans compter les centimes, les valeurs des pièces/billets sont 1, 2, 5, 10, 20, 50, 100, 200.
Pour rendre 49 €, on rend deux billets de 20 €, un billet de 5 € et deux pièces de 2 €.
L'algorithme glouton est basé sur le tri par valeur décroissante des pièces/billets.

b. Exercice

Trouver un système de pièces et de billets où l'algorithme glouton ne donne pas la solution optimale (on cherche toujours à rendre 49 €)

Exercice avec un algorithme glouton

On a à notre disposition, une liste d'élèves qui doivent présenter leur exposé dans une salle. Seulement, leurs disponibilités ne leur permettent d'intervenir qu'à des horaires bien définis.
On dit que deux exposés d'élèves sont compatibles si leurs créneaux ne se chevauchent pas. Le problème est de construire un planning avec le plus grand nombre d'élèves.
Soit la liste d'horaires des exposés suivante : 8h-13h, 12h-17h, 9h-11h, 14h-16h, 11h-12h.
En réfléchissant sur papier :

1. Combien d'élèves pourront faire leur exposé sur une seule journée ?
2. Choisir une stratégie gloutonne pour sélectionner des exposés.

En programmant en Python :
Les exposés, qui commencent ^tot le matin et finissent tard le soir, sont donnés par la liste ci-après :

- `Exposes =[ [1,4] , [0,6] , [3,5] , [12,13] , [8,11] , [8,12] , [3,13] , [6,10] , [5,9] , [4,8] , [5,7] , [13,16] , [15,17] , [16,19] ]`
- `debut = [1,0,3,12,8,8,3,6,5,4,5,13,15,16]`
- `fin = [4,6,5,13,11,12,13,10,9,8,7,16,17,19]`

On reprend ce problème de l'exercice précédent mais cette fois on suppose avoir n exposés et deux tableaux `début[]` et `fin[]` de taille *n*, tels que `début[i]` et `fin[i]` contiennent les horaires de l'exposé *i*.
Ce long exercice peut se faire partiellement, en faisant uniquement les questions 3b et 7.

3. Fonctions préliminaires : la fonction du a sert pour les méthodes 2 et 3, la fonction du b sert pour toutes les méthodes.
   a. Créer une fonction qui vérifie la compatibilité d'une activité avec la liste existante, déjà triée par ordre de début. Spécifications :

```
def compatible(liste,element):
    """
    Vérifie si un élément [c,d] est compatible avec les éléments d'une
liste
    @param liste : composée d'éléments du type [a_i, b_i] avec
           a_i < b_i entiers
    @param element : [c,d] avec c < d entier
    @return indice : 4 cas :
        indice de la liste tel que b_indice <= c < d <= a_indice + 1
           s'il existe ;
        -1 si element s'insère en début de liste ;
        indice = len(liste) si element s'insère en fin de liste ;
        None sinon.
    """
```

   b. Créer une fonction de tri d'une liste en fonction des éléments d'une autre liste de même longueur. On pourra reprendre un des algorithmes de tri déjà vus et l'adapter.
4. Méthode 1 : écrire un algorithme gourmand qui trie les exposés par ordre croissant de leur date de début, et donne un ordre de passage compatible en fonction de ce tri. La méthode est-elle optimale ? Justifier (on donnera un contre exemple, on peut s'aider d'un dessin). *On peut s'inspirer de la progression proposée dans le 7*
5. Méthode 2 : écrire un algorithme gourmand qui trie les exposés par ordre croissant de leur durée, et donne un ordre de passage compatible en fonction de ce tri. La méthode est-elle optimale ? Justifier (on donnera un contre exemple, on peut s'aider d'un dessin).
6. Méthode 3 : écrire un algorithme gourmand qui trie les exposés par ordre croissant de leur nombre d'incompatibilités, et donne un ordre de passage compatible en fonction de ce tri. *Un schéma n'est pas inutile pour trouver les conditions d'incompatibilité de deux exposés.* La méthode est-elle optimale ? Justifier (on donnera un contre exemple, on peut s'aider d'un dessin).

Méthode 4, qui est la plus facile à programmer (avec la méthode 1 équivalente en difficulté) : tri par date de fin. La stratégie est la suivante, en commençant par le début de journée : choisir l'exposé dont l'heure de fin arrive le plus tôt [parmi les exposés dont l'heure de début se trouve après les exposés déjà choisis. Cette méthode est optimale, la démonstration est acceptée.

7.
   a. Écrire une fonction `prochaine[debut, fin, hd]` qui sélectionne l'exposé dont l'heure de début n'est pas inférieure à hd, et s'arrêtant le plus tôt. La fonction devra renvoyer None si aucun horaire n'est compatible.
   b. En déduire une fonction `selection[debut, fin]` qui, en supposant que toutes les heures soient positives, sélectionne autant d'exposés que possible en suivant la stratégie gloutonne.
   La fonction affichera les numéros d'exposés sélectionnés.

Recherche dichotomique

1. Introduction.

a. *« Devine un nombre »*

En début d'année, vous avez programmé le jeu « devine un nombre ». Dans ce jeu, l'ordinateur choisit un nombre au hasard entre 1 et 100, le joueur humain doit le deviner en proposant des valeurs. L'ordinateur répond « trouvé », « trop petit » ou « trop grand ».

La stratégie que vous avez adoptée, en tant que joueur, pour trouver le plus vite possible, est de proposer :

- 50 en premier ;
- Puis soit 25 soit 75 ;
- Puis la moitié de l'intervalle possible (arrondi bien sûr)
- Etc… ceci jusqu'à trouver la bonne réponse.

Vous avez fait une recherche dichotomique (du grec ancien διχοτομια, *dikhotomia* « division en deux parties »).

b. *Pourquoi cette méthode est efficace ?*

Supposons que vous ayez eu choisi[1] de répondre à partir de 1 et en donnant tous les entiers successivement jusqu'à la bonne réponse. Combien aurait-il fallu faire d'essais :

- Dans le meilleur des cas ?
- Dans le pire des cas ?
- En moyenne (intuitivement) ?

Si on généralise au cas où l'ordinateur choisit un nombre entre 1 et n, on peut donner la complexité :

- Dans le meilleur des cas ?
- Dans le pire des cas ?
- En moyenne (intuitivement) ?

Revenons maintenant à la stratégie de dichotomie.

Combien faut-il d'essais dans le meilleur des cas ?

Combien faut-il d'essais, dans le pire des cas, si le nombre à deviner est :

- Entre 1 et 100 ?
- Entre 1 et 200 ?
- Entre 1 et 400 ?
- Entre 1 et 800 ?

Qu'y a-t-il de remarquable dans les résultats précédents ?

On remarque que quand on multiplie la taille par 2, on ajoute juste une opération. On peut ainsi montrer que si on prend des nombres entre 1 et 10000 on aura maximum 14 essais. On cherche en fait la puissance de 2 immédiatement supérieure à 10000, l'exposant donne le nombre d'essais.

On dit que la progression est **logarithmique** (plus précisément $\log_2(n)$).

2. Principe de la recherche dichotomique

La recherche dichotomique est l'expression la plus simple du principe *diviser pour régner*, qui est un principe fondamental en algorithmique, d'une efficacité redoutable.

La recherche dichotomique se fait uniquement sur un tableau trié.

Une recherche dichotomique procède ainsi:

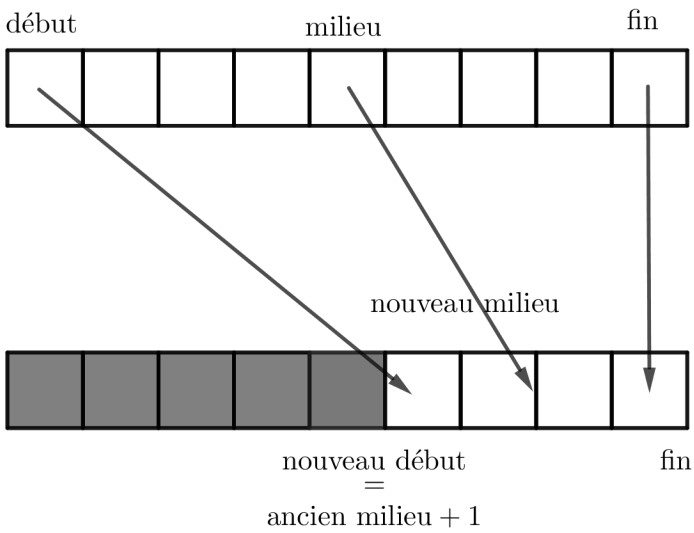
① Trouver la position la plus centrale du tableau (si le tableau est vide, sortir).

② Comparer la valeur de cette case à l'élément recherché.

[1] Un petit passé simple du subjonctif de temps en temps fait du bien. Avec les liaisons à l'oral c'est encore mieux ☺

③ Si la valeur est égale à l'élément, alors retourner la position, sinon répéter les étapes ① et ② dans la moitié de tableau pertinente.

On peut toujours se ramener à une moitié de tableau sur un tableau trié en ordre croissant. Si la valeur de la case est plus petite que l'élément, on continuera sur la moitié droite, c'est-à-dire sur la partie du tableau qui contient des nombres plus grands que la valeur de la case. Sinon, on continuera sur la moitié gauche.
En supposant que `valeur_cherchée > milieu`, on a le schéma ci-contre lors du passage d'une étape à une autre.



Algorithme (en Python).

```
def dichotomie(liste, valeur):
    '''
    recherche d'une valeur par dichotomie dans un tableau trie
    stratégie : on réduit la recherche avec un indice de début et un
                indice de fin, sur le tableau [début, fin]
    @param liste :    une liste d'entiers, triée
    @param valeur :   entier
    @return trouve :  booléen vrai si valeur est dans le tableau, faux
                sinon
    '''
    trouve = False
    debut = 0
    fin = len(liste)-1
    while not(trouve) and debut <= fin :
        milieu = (debut + fin)//2
        if liste[milieu] == valeur :
            trouve = True
        elif liste[milieu] > valeur:
            fin = milieu - 1
        else:
            debut = milieu + 1
    return(trouve)
```

Remarque : on peut facilement modifier l'algorithme pour qu'il renvoie également l'indice de l'élément recherché dans le tableau.

3. Complexité (pire des cas)

Comme pour les tris, il s'agit de répondre à la question : « étant donné un tableau de taille n, combien d'itérations dans la boucle effectue l'algorithme dans le pire des cas ? ».
Le pire des cas se produit quand l'élément n'est pas dans le tableau.
Comme vu en introduction, chaque passage dans la boucle divise la taille du tableau par 2. Si le tableau est de taille n, alors :

- Après la première itération la recherche se fait sur un tableau de taille $n/2$ (division entière)
- Après la deuxième itération la recherche se fait sur un tableau de taille $n/4$
- …

- Après la *p*-ième itération la recherche se fait sur un tableau de taille $n/2^n$

La recherche se termine quand le tableau est de taille 1

On résout donc l'équation : $\frac{n}{2^p}=1 \Leftrightarrow 2^P=n \Leftrightarrow p=\log_2(n)$; on introduit la ainsi fonction logarithme de base 2, notée, $\log_2$ ou parfois lg.

La complexité est en $O(\log(n))$ (le 2 est inutile dans la complexité, les logarithmes de différentes bases ont proportionnels entre eux)

Calcul de $\log_2(x)$.

Définition : le logarithme entier (en base 2) d'un entier *n* strictement positif est le nombre des bits de son écriture binaire diminué d'une unité.

Exemples : les trois premiers exemples donnent la « vraie » valeur du logarithme, le dernier donne la valeur entière.

$\log_2 1=0 \qquad \log_2 2=1 \qquad \log_2 4=\log_2 \overline{100}_2=2 \quad \text{Ent}\left[\log_2 623\right]=\text{Ent}\left[\log_2 \overline{1001101111}_2\right]=9$

Calcul du logarithme en base 2 non entier d'un nombre quelconque.

Définissons le logarithme de base 2 par la propriété suivante :

« Les puissances de 2 ont un logarithme en base 2 qui vaut exactement $\log_2 2^n=n$ »

On en déduit que $\log_2\left(2^n\times 2^p\right)=\log_2 2^{n+p}=n+p=\log_2 2^n+\log_2 2^p$

Généralisons, on admet qu'il existe des fonctions vérifiant pour tous les entiers (et même les réels) strictement positifs : $\log_2(a\times b)=\log_2 a+\log_2 b$

Ceci donne immédiatement la propriété : $\log_2 a^2=2\log_2 a \Leftrightarrow \log_2 a=\frac{1}{2}\log_2 a^2$ propriété (1)

Comment peut-on alors calculer $\log_2 3$?

Comme $2<3<4$, alors $1<\log_2 3<2$.

On écrit $\log_2 3=\log_2\left(2\times\frac{3}{2}\right)=\log_2 2+\log_2\frac{3}{2}=1+\log_2\frac{3}{2}$

(2) On va à chaque fois chercher à obtenir un nombre dans $[1\,;2[$ pour le dernier $\log_2$ *truc*

$$\log_2 3\overset{(1)}{=}1+\frac{1}{2}\log_2\left(\frac{3}{2}\right)^2=1+\frac{1}{2}\log_2 2{,}25=1+\frac{1}{2}\log_2(2\times 1{,}125)=1+\frac{1}{2}(\log_2 2+\log_2 1{,}125)$$

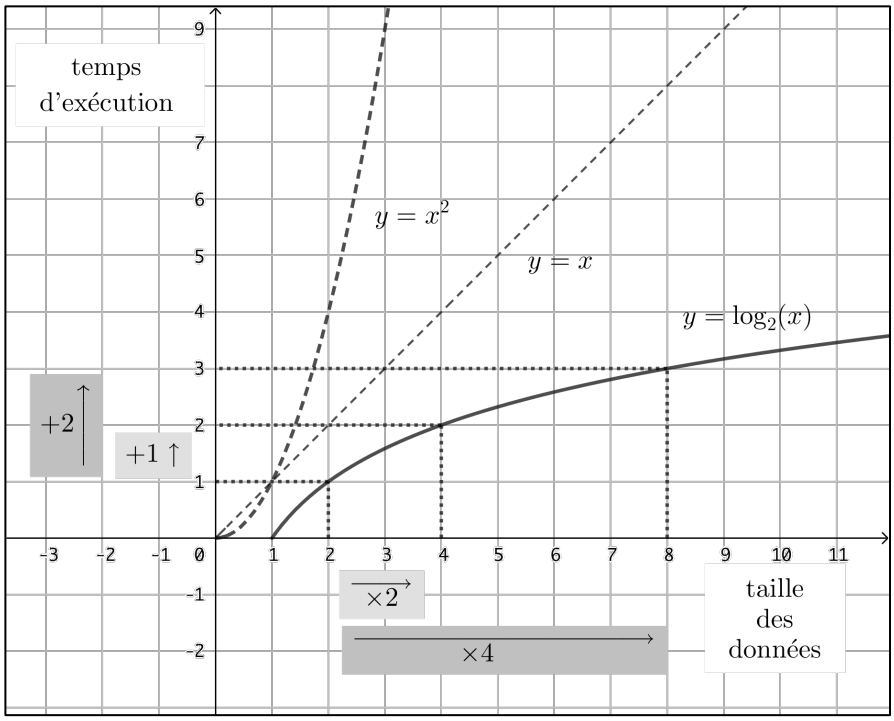
$$\log_2 3=1+\frac{1}{2}(1+\log_2 1{,}125)=1+\frac{1}{2}+\frac{1}{2}\times\frac{1}{2}\log_2(1{,}125^2)=1+\frac{1}{2}+\frac{1}{4}\log_2 1{,}265625 \quad \text{cf. (2)}$$

$$\log_2 3=1+\frac{1}{2}+\frac{1}{8}\log_2 1{,}601806640625 \qquad \text{cf. (1) et (2)}$$

$$\log_2 3=1+\frac{1}{2}+\frac{1}{16}\log_2 2{,}25657845139...=1+\frac{1}{2}+\frac{1}{16}+\frac{1}{32}\log_2 1{,}2828922569...$$

D'où $\log_2 3\simeq 1+\frac{1}{2}+\frac{1}{16}\simeq\overline{1{,}1001}_2$: on calcule « facilement » une valeur approchée du logarithme d'un entier en base 2.

La rapidité de l'algorithme est montrée par les courbes suivantes :



4. Preuve.

On doit prouver à la fois que l'algorithme donne bien le résultat recherché (preuve de correction ou de validité), et qu'il se termine (preuve de terminaison ou d'arrêt).
On reprend les notations de l'algorithme :

Preuve de l'arrêt (ou de terminaison) :

Le **variant** de boucle $longueur = fin - début$ décroît strictement à chaque itération.

En effet à chaque étape on divise par deux, *longueur* devient $\left(nouvelle\right) longueur = \dfrac{fin - début}{2}$

Le variant *longueur* diminue strictement lors de chaque étape. Par ailleurs c'est un entier, donc il finira par atteindre 0 (si *valeur* n'est pas trouvée avant).

Par ailleurs la condition d'arrêt de l'algorithme est :

tant que $\boxed{\text{non}(trouvé) \text{ et } début \leq fin}$ **faire**

Donc

- soit l'algorithme trouve lors d'une étape et il s'arrête ;
- soit le variant *longueur* atteint 0, dans ce cas $longueur = 0 \Leftrightarrow fin - début = 0 \Leftrightarrow fin = début$. A cette étape, on repasse une dernière fois dans la boucle. En effet l'algorithme déroulera les instructions suivantes :

```
si liste[milieu] = valeur alors trouvé =Vrai
sinon
        si liste[milieu] < valeur alors
                début ← milieu + 1
        sinon
                fin ← milieu – 1
```

Comme $fin = début$, on a $milieu = \dfrac{début + fin}{2} = début = fin$ (1).

Si $liste[milieu] = valeur$ l'algorithme s'arrête

Si $liste[milieu] < valeur$ alors $fin \leftarrow milieu - 1$, c'est à dire $fin \leftarrow début - 1$ d'après (1). La

condition *début* ≤ *fin* n'est plus vérifiée.
De même, si *milieu* > *valeur* alors *début* ← *milieu* + *1* , c'est à dire *début* ← *fin* + *1* d'après (1). La condition *début* ≤ *fin* n'est plus vérifiée.
Donc l'algorithme s'arrête dans tous les cas.

Preuve de validité (ou de correction) :

Remarque : on suppose que *valeur* est entre le minimum et le maximum du tableau, pour simplifier (sinon il faut écrire un morceau de preuve supplémentaire).
L'**invariant** de boucle est : *valeur* est compris entre *liste*[*début*] et *liste*[*fin*] pour toute étape *n*.
Initialisation de l'algorithme :

Lors de la première itération, d'après la remarque précédente, on a :
$liste\left[\,début\,\right] \leq valeur \leq liste\left[\,fin\,\right]$.

Conservation : passage d'une étape à un autre :

On se place à une étape *n*, et on suppose que $liste\left[\,début_n\,\right] \leq valeur \leq liste\left[\,fin_n\,\right]$, où $début_n$ et fin_n désignent le début du tableau et la fin du tableau à l'étape *n*.
On déroule l'algorithme :

- on calcule $milieu = \dfrac{début_n + fin_n}{2}$
- Si *liste*[*milieu*] = *valeur*, alors on sort de la boucle et on ne passe pas à l'étape *n* + 1, l'algorithme est juste puisqu'il a trouvé le résultat attendu
- Si *liste*[*milieu*] < *valeur*, alors $début_{n+1} = milieu + 1$ et $fin_{n+1} = fin_n$
  Comme le tableau est trié, on a
  $valeurs\,du\,tableau\,jusqu'à\,l'indice\,milieu < liste[milieu+1] \leq valeur$
  Ou avec une notation équivalente plus compacte :
  $liste[0 : milieu] < liste[milieu+1] \leq valeur$
  Comme $liste\left[\,milieu+1\,\right] = liste\left[\,début_{n+1}\,\right]$, puisque $début_{n+1} = milieu + 1$, alors
  $liste\left[\,début_{n+1}\,\right] \leq valeur$
  Par ailleurs, *fin* n'a pas changé : $fin_{n+1} = fin_n$.
  Donc $valeur \leq liste\left[\,fin_{n+1}\,\right]$ puisqu'on avait déjà $valeur \leq liste\left[\,fin_n\,\right]$
  On a bien $liste\left[\,début_{n+1}\,\right] \leq valeur \leq liste\left[\,fin_{n+1}\,\right]$ à l'étape *n* + 1
- Si *liste*[*milieu*] > *valeur*, alors $fin_{n+1} = milieu - 1$ et $début_{n+1} = début_n$
  Comme le tableau est trié, on a
  $valeur \leq liste[milieu-1] < valeurs\,du\,tableau\,jusqu'à\,l'indice\,milieu$
  Ou avec une notation équivalente plus compacte :
  $valeur \leq liste[milieu-1] < liste[milieu : fin]$
  Comme $liste\left[\,milieu-1\,\right] = liste\left[\,fin_{n+1}\,\right]$, puisque $fin_{n+1} = milieu - 1$, alors
  $liste\left[\,début_{n+1}\,\right] \leq valeur$
  Par ailleurs, *début* n'a pas changé : $début_{n+1} = début_n$.
  Donc $liste\left[\,début_{n+1}\,\right] \leq valeur$ puisqu'on avait déjà $liste\left[\,début_n\,\right] \leq valeur$
  On a bien $liste\left[\,début_{n+1}\,\right] \leq valeur \leq liste\left[\,fin_{n+1}\,\right]$ à l'étape *n* + 1
- Dans tous les cas, on a le résultat recherché $liste\left[\,début_{n+1}\,\right] \leq valeur \leq liste\left[\,fin_{n+1}\,\right]$

On a bien prouvé que l'invariant reste … invariant d'une étape à l'autre, donc si *valeur* est dans le tableau, l'algorithme finira par la trouver (la taille du tableau diminuant, cf. preuve de terminaison).

Terminaison, et aussi que se passe-t-il si valeur n'est pas dans le tableau ?

En toute dernière étape, soit l'algorithme trouve la valeur demandée, et renvoie Vrai comme il doit le faire. Soit la liste restante est vide puisque *début* ≤ *fin* n'est plus vérifiée (on a soit *fin* = *début* – 1, soit *début* = *fin* + 1). Alors l'algorithme renvoie Faux, c'est bien le résultat attendu

Un exercice sur la recherche dichotomique

Reprendre les données sur les films. Les trier par titre, et programmer la recherche dichotomique pour trouver les données sur les films « Courage Under Fire », « Army Of Darkness », « The Square », « My Fair Lady ».

Algorithme des *k* plus proches voisins (*k*-nn)

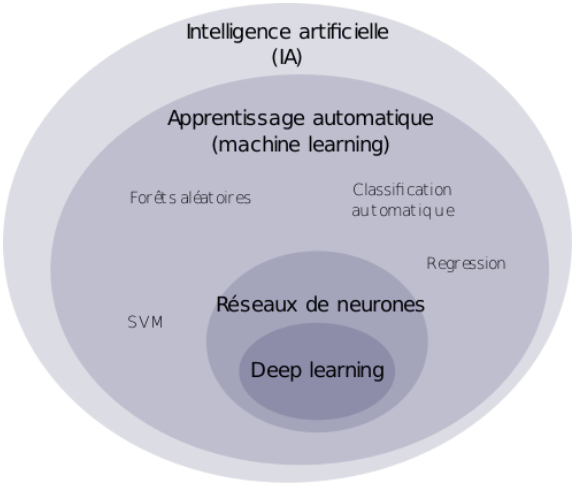
La méthode *k*-nn (*k* nearest neighbors) est une méthode d'apprentissage supervisé utilisée en apprentissage automatique (en intelligence artificielle ou en exploration de données, pour simplifier, cf. ci-dessous pour plus de précisions). Précisons quelques termes et notions avant de nous pencher sur l'algorithme proprement dit.

1. Qu'est-ce que l'IA ?

L'intelligence artificielle est « l'ensemble des théories et des techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence ». Beaucoup de spécialistes du sujet utilisent souvent « stupidité artificielle », en effet les résultats réels sont bien éloignés des fantasmes du grand public. Les IA actuelles sont très efficaces sur des problèmes spécifiques, qui demandent une grosse capacité de calcul. Par contre, elles sont incapables par exemple de traduire correctement un texte, et encore moins de comprendre le sens. On oppose ici IA faible – voire très faible, même si cette notion n'existe pas ! – et IA forte, cette dernière étant une machine capable d'éprouver une impression de réelle conscience.

Les IA actuelles fonctionnent de manière très différente du cerveau humain. Un bébé ne pourra jamais gagner au jeu de go contre un expert, ce qu'une IA fait, mais à l'inverse, il lui suffit de voir quelques fois un chat pour en reconnaître par la suite. Alors qu'il faut des millions d'images pour l'apprentissage d'une machine, et il subsistera encore des bugs surprenant (changer un pixel invisible à l'œil nu suffit à changer le chat en.. avocat pour une IA).

Comme le montre le schéma ci-contre (merci wikipédia), les méthodes d'IA, d'apprentissage automatique, de réseaux de neurones et de deep learning sont imbriquées les unes dans les autres.



Les méthodes d'IA comprennent d'autres types de briques logicielles que l'apprentissage automatique, comme les « moteurs de règles », utilisés en aide au diagnostic médical. Mais aussi des méthodes basées sur la logique, d'autres sur les probabilités, ou encore des algorithmes simulant des procédés naturels ou artificiels :

- algorithmes génétiques où une « population de solutions » évolue vers la meilleure solution (en temps fini, vers une « très bonne » solution seulement). On simule des croisements aléatoires entre les « solutions » et on leur associe un score, qui détermine leur chance de survie et de reproduction. On rajoute aussi des mutations aléatoires.

- algorithmes d'essaims, dont ceux de colonies de fourmis, où les « fourmis » construisent petit à petit le meilleur chemin (ici aussi, en temps fini, on n'aura qu'un « très bon » chemin). La méthode consiste à simuler le dépôt de phéromones sur les chemins possibles, phéromones qui disparaissent au cours du temps sur les chemins les plus longs et s'accumulent sur les plus courts. Le petit nom de ce type de méthodes est « algorithmes stigmergiques » (à replacer dans une conversation entre amis ☺)
- le recuit simulé : cette méthode est basée sur le fait que pour renforcer un métal, comme l'épée de Conan le Barbare, on alterne des cycles de refroidissement lent et de réchauffage (le recuit). Ceci renforce le métal, qui, en un temps infini toujours, atteint sa solidité maximale.

La méthode *k*-nn fait partie de la couche « apprentissage automatique » (mais pas de « réseaux de neurones »). On l'utilisera ici comme une méthode de classification (on cherche à ranger des objets dans des catégories) supervisée (les catégories sont connues à l'avance). On peut mélanger des méthodes de différentes couches ; votre honoré professeur a produit un travail mélangeant algorithmes génétiques, classification non supervisée (sans connaître les groupes à l'avance), et analyses discriminantes (une méthode utilisée notamment en deep learning). Les méthodes de régression cherchent à prévoir une valeur (future par exemple) à partir de l'observation de l'existant. Les méthodes de classification et de régression sont souvent des méthodes statistiques. Enfin, les algorithmes de forêts aléatoires servent par exemple à juger de la qualité des articles de wikipédia, tandis que les méthodes SVM permettent également la classification.

Les « réseaux de neurones » datent d'avant les recherches en IA (premières propositions en 1873 !). Il s'agissait de simuler le fonctionnement cérébral. Ils ne sont utilisés en IA que depuis que les ordinateurs ont une puissance suffisante, et sont identifiés à tort par le grand public avec l'intelligence artificielle. Ces réseaux nécessitent une phase d'apprentissage, où, plus le problème est complexe, plus il faut d'exemples. Par ailleurs, ils ont plusieurs défauts dont la connaissance est indispensable lorsque l'on travaille avec (ce qui n'est souvent pas le cas pour des personnes utilisant les exemples ci-dessous) :

- base d'apprentissage incomplète : la reconnaissance faciale fonctionne bien mieux pour les visages de type « caucasien », que « noirs » ou « asiatiques » (du moins en occident, il n'est pas sûr qu'en Chine cela ne soit pas l'inverse)
- reproduction des biais donnés par les exemples/règles : le système utilisé pour prévoir la récidive utilisé par dans certaines juridictions aux USA est raciste (alors que les statistiques donnent les mêmes risquent pour noirs et blancs). Ce n'est pas dû à la programmation, mais aux règles choisies. Une de ces règles est que le risque augmente si la personne habite dans un quartier défavorisé, une autre que le risque augmente si la personne connaît un délinquant/criminel. Ces règles se multiplient les unes les autres et stigmatisent les habitants des quartiers populaires.
- Conservatisme : une personne ayant eue un emprunt refusé par une IA aura d'autant plus de mal à en obtenir un autre, si le fait est connu.
- Opacité : il est actuellement impossible de savoir comment un réseau de neurones est parvenu à une conclusion. Un champ de recherches est en développement pour expliciter ces décisions, afin que le public puisse avoir confiance dans ces décisions.

Enfin le « deep learning », apprentissage profond, est très complexe. Ces méthodes ont été utilisées par le programme auto-apprenant qui a battu le champion du monde du jeu de go. Certaines promesses du deep learning sont mensongères ; comme la traduction automatique, ou la reconnaissance des émotions, qui fonctionnent nettement moins bien que leurs pendants humains. D'autres sont au contraire efficaces, comme l'aide au diagnostic médical. Notons également que sur la reconnaissance de formes/d'images, comme les panneaux de signalisation, il est très facile de tromper ces algorithmes pour l'instant.

Parmi les langages phares pour l'intelligence artificielle, citons le Python, ainsi que R (spécialisé en statistiques et analyse de données) et dans une moindre mesure le C/C++, toujours apprécié pour sa rapidité. Le Lisp et le Prolog sont des langages « historiques » de l'IA.

2. Présentation de l'algorithme *k*-nn

L'algorithme *k*-nn se base sur un jeu de données comportant plusieurs paramètres en général quantitatifs (qui se mesurent à l'aide d'un nombre), et un paramètre qualitatif : la classe de l'objet.
L'objectif de l'algorithme est de classer un nouvel individu (au sens statistique) dans une des classes de la population existante.
Déroulement de l'algorithme :

- Calcul de toutes les distances euclidiennes entre le jeu de données connues et le nouvel individu.
- Tri des éléments du jeu de données par les distances dans l'ordre croissant
- Sélection des *k* voisins les plus proches : ce sont les *k* premiers éléments triés à l'étape précédente
- Le nouvel individu appartient à la classe majoritaire dans les *k* plus proches voisins

Remarques :

- *k* est un paramètre fixé dans le programme. Plus *k* est grand et plus la classification est juste, mais les frontières entre les classes deviennent floues. Attention le résultat peut varier suivant les valeurs de *k*.
- S'il n'y a que deux classes, il est préférable de choisir *k* impair pour éviter les cas d'égalité.
- Pour des paramètres quantitatifs, on utilise souvent la distance euclidienne :
  - $AB=\sqrt{\left(x_B-x_A\right)^2+\left(y_B-y_A\right)^2}$ avec deux paramètres (deux dimensions)
  - $AB=\sqrt{\left(x_{1B}-x_{1A}\right)^2+\left(x_{2B}-x_{2A}\right)^2+\ldots+\left(x_{nB}-x_{nA}\right)^2}$ avec *n* paramètres (*n* dimensions)
- On peut utiliser des distances différentes, par exemple :
  - Distance « Manhattan » : somme des valeurs absolues, correspondant à la distance parcourue dans une ville où toutes les rues sont à angle droit. $AB=\left|x_B-x_A\right|+\left|y_B-y_A\right|$. Parfois utilisée quand les données ne sont pas du même type.
  - Distance de Hamming si l'on souhaite comparer des chaînes de caractères (cas d'un correcteur orthographique simple). Cette distance compte le nombre de caractères différents entre deux chaînes. Il y a d'autres distances plus efficaces pour comparer des mots.
  - Entre des paramètres qualitatifs, comme « aimer la pizza », « couleur », on peut utiliser une distance du type « 0 si la valeur est la même » et « 1 si les valeurs sont différentes ». C'est à nouveau la distance de Hamming.
- S'il y a trop de paramètres (10 ou plus), le phénomène de « malédiction de la dimension » entre en jeu, il y a trop de dimensions par rapport au nombre de données. Il faut alors identifier les paramètres pertinents. Des méthodes algorithmiques existent, mais dans des problèmes de sciences expérimentales ; l'expertise du scientifique est souvent aussi pertinente.
- Si la répartition des données en classes est déséquilibrée, la qualité de la prédiction sera moins bonne.
- L'algorithme est coûteux si les données sont nombreuses.
- Voir le notebook pour le programme Python

3. Compléments

On peut améliorer la prédiction en rajoutant un poids sur chaque voisin. Le coefficient $1/d$ est souvent utilisé, où *d* est la distance calculée précédemment.
La méthode *k*-nn peut être aussi utilisée pour faire de la régression. Dans ce cas, le dernier paramètre n'est plus qualitatif mais quantitatif. On cherche alors à prédire la valeur de ce dernier paramètre à partir de la moyenne des valeurs observées sur les voisins. Par exemple, connaissant les moyennes des élèves dans toutes les matières, on pourrait essayer de prédire la moyenne en NSI d'un élève à partir des moyennes des autres matières.