

# CHAPITRE 10 : Algorithmique 2

---

- 1 Histoire ..... 2
- 2 Algorithme des  $k$  plus proches voisins ..... 3
- 3 Recherche dichotomique dans un tableau trié ..... 5
- 4 Algorithmes gloutons ..... 10
  - 4.1 Premier exemple d'algorithme glouton : le sac à dos ..... 10
  - 4.2 Deuxième exemple d'algorithme glouton : le rendu de monnaie ..... 15
  - 4.3 D'autres exemples qui se ramènent au problème du sac à dos..... 17

# CHAPITRE 10 : Algorithmique 2

## 1 Histoire

**-220** *Apparition des tableaux triés*

Le recours au tri préalable d'un tableau pour trouver un élément plus facilement apparait chez les babyloniens en 220 avant Jésus-Christ.

**1000** *Tri d'après plusieurs critères*

Le scientifique arabe **Alhazen** évoque la possibilité de trouver un individu répondant à *plusieurs critères* simultanément.

**1928** *Toutes les questions sont-elles décidables ?*

David Hilbert, mathématicien allemand pose la question : Est-ce que tous les problèmes mathématiques qui n'ont pas encore de solution pourraient être résolus **par un procédé algorithmique** ?

**1936** *La machine (théorique) de Turing*

Une *machine de Turing* ne désigne pas une seule machine mais toute **une famille de machines**.

**C'est un modèle abstrait** très simple du fonctionnement des appareils mécaniques de calcul, tel un ordinateur. Cette "machine" a été décrite par l'informaticien Alan Turing. Elle contient un ruban infini avec des caractères qu'elle peut lire puis effacer et écrire. Elle contient dans un registre son 'état'. Enfin elle possède une table de transition qui est une liste d'instructions.

### Exemple

1. La machine lit sur le ruban **B**.
2. La machine est dans l'état **1**.
3. La machine lit donc en '**B1**' dans la table de transition la prochaine instruction qui est 'C2<' .

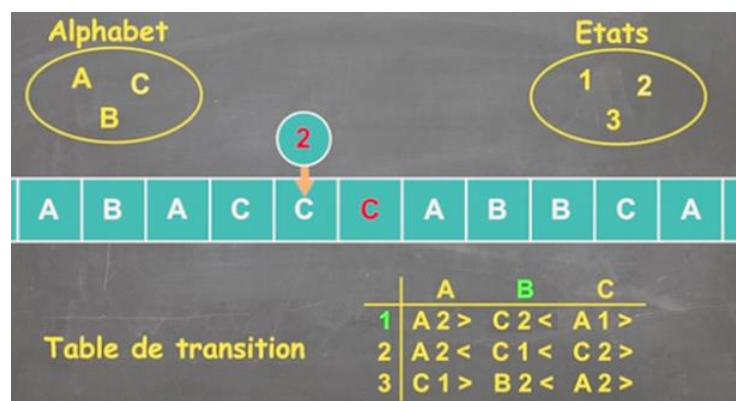
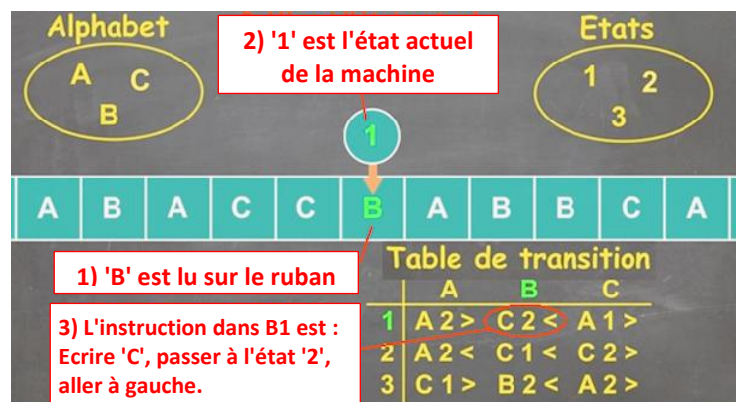
Cela signifie :

- Sur le ruban, à la place de 'B' écrire 'C'.
- Passer dans l'état '2'.
- Aller à gauche.

La machine exécute l'instruction : *elle efface le B et écrit un C à la place, elle passe à l'état 2 et elle déplace sa tête de lecture-écriture d'une case à gauche sur le ruban.*

La machine recommence :

1. La machine lit sur le ruban '**C**'.
2. La machine est dans l'état '**2**'.
3. La machine lit donc en '**C2**' dans la table de transition la prochaine instruction qui est 'C2>'.



## 2 Algorithme des $k$ plus proches voisins

L'algorithme des  $k$  plus proches voisins ( $k \in \mathbb{N}$ ) permet **d'attribuer une catégorie** à un nouvel individu.

L'algorithme se base sur des **exemples déjà classés dans des catégories**.

Par exemple si  $k = 3$ , et que l'algorithme trouve que les trois exemples **les plus proches selon des critères**<sup>1</sup> sont dans les catégories  $C_1, C_2, C_1$ , alors comme la classe  $C_1$  est majoritaire, l'algorithme dira que le nouvel individu appartient à la catégorie  $C_1$ .












**En anglais c'est l'algorithme kNN** ( $k$  Nearest Neighbors).

### Exemple

Nous connaissons des individus "exemples" :  $A, B, C, D, E, F, G, H, I, J$  pour lesquels on connaît :

- **Leurs critères** : le couple (Age ; Nombre d'amis sur Facebook).
- **Leur catégorie**<sup>2</sup> : "aime le dernier film" ou "n'aime pas le dernier film".

Pour plus de facilité, on a attribué la couleur bleue à ceux qui aiment le film et la couleur rouge à ceux qui ne l'aiment pas. Ci-contre le **jeu de données** :

	A = (15, 50)
	B = (20, 10)
	C = (25, 15)
	D = (30, 18)
	E = (40, 40)
	F = (42, 4)
	G = (55, 20)
	H = (60, 5)
	I = (65, 13)
	J = (70, 39)
	P = (52, 7)

**Le point  $P$**  après le jeu de données **est le point à classer**.

### Exemple

L'individu  $A$  a 15 ans et 50 amis. Il n'aime pas le film. Donc il est rouge.

**On ne connaît que les critères** de l'individu  $P$  :

- il a 52 ans et 7 amis sur Facebook.

On veut prédire, à partir de ses critères (52 ; 7), en se basant sur l'apprentissage qui a été fait avant (on a saisi les dix points de  $A$  à  $J$ ), la catégorie à laquelle appartiendra  $P$  (soit on le classe parmi ceux qui aiment, soit parmi ceux qui n'aiment pas le film).

On cherche, dans l'ensemble dix individus (de  $A$  à  $J$ ), lesquels sont les  $k$  plus proches voisins de  $P$ . Disons pour notre exemple, que nous travaillons avec  $k = 3$ .

<sup>1</sup> **Critères** : on dit aussi caractéristiques (*features* en anglais).

<sup>2</sup> **Catégories** : on dit aussi **classes** ou encore **étiquettes** (*labels* en anglais).

Donc on fera dans l'ordre :

1. Extraire par exemple d'un fichier csv (ou écrire à la main) les listes de critères  $x = [15, 20, \dots]$  et  $y = [50, 10, \dots]$  et la liste de catégories couleurs = ['rouge', 'rouge', ...].
2. Créer la liste des sous-listes [critères, catégorie] (c'est le jeu de données) :

```
liste = [[15, 50, 'rouge'], [20, 10, 'rouge'], [25, 15, 'rouge'], [30, 18, 'bleu'],\
         [40, 40, 'rouge'], [42, 4, 'bleu'], [55, 20, 'bleu'], [60, 5, 'rouge'],\
         [65, 13, 'rouge'], [70, 39, 'rouge']]
```

3. Écrire la fonction distance( $x_A, y_A, x_B, y_B$ ) qui renvoie la distance  $AB$ . Rappel : la formule de la distance<sup>3</sup> entre deux points  $P$  et  $Q$  est :

$$PQ = \sqrt{(x_Q - x_P)^2 + (y_Q - y_P)^2}$$

puis écrire la fonction calcule\_distances( $x, y, liste$ ) qui prend en argument les coordonnées du point à classer  $P$  et la liste du jeu de données. Elle renvoie la liste des doublets de la forme :

```
((distance entre P et 1er élément de la liste, catégorie du 1er élément),
 (distance entre P et 2e élément de la liste, catégorie du 2e élément),
 (distance entre P et 3e élément de la liste, catégorie du 3e élément)...
```

#### **Exemple**

On suppose que liste vaut [[15, 50, 'rouge'], [20, 10, 'rouge'],...

```
liste_distances = calcule_distances(52, 7, liste)
```

liste\_distances vaut alors [(56.72741841473134, 'rouge'), (32.14031735997639, 'rouge'),...

4. Écrire la fonction tri(tableau) qui trie la liste liste\_distances par distances croissantes.

#### **Exemple**

```
liste_distances vaut [(56.72741841473134, 'rouge'), (32.14031735997639, 'rouge'),..
```

```
liste_distances_triee = tri(liste_distances)
```

```
liste_distances_triee vaut [(8.246211251235321, 'rouge'), (10.44030650891055, 'bleu'),...
```

5. Écrire la fonction classe( $k, tableauTrie$ ) qui prend en argument la valeur  $k$  du nombre de plus proches voisins qu'on considère et le tableau de doublets trié liste\_distances\_triee.

Cette fonction doit renvoyer un dictionnaire.

#### **Exemple**

```
liste_distances_triee vaut [(8.246211251235321, 'rouge'), (10.44030650891055, 'bleu'),...
```

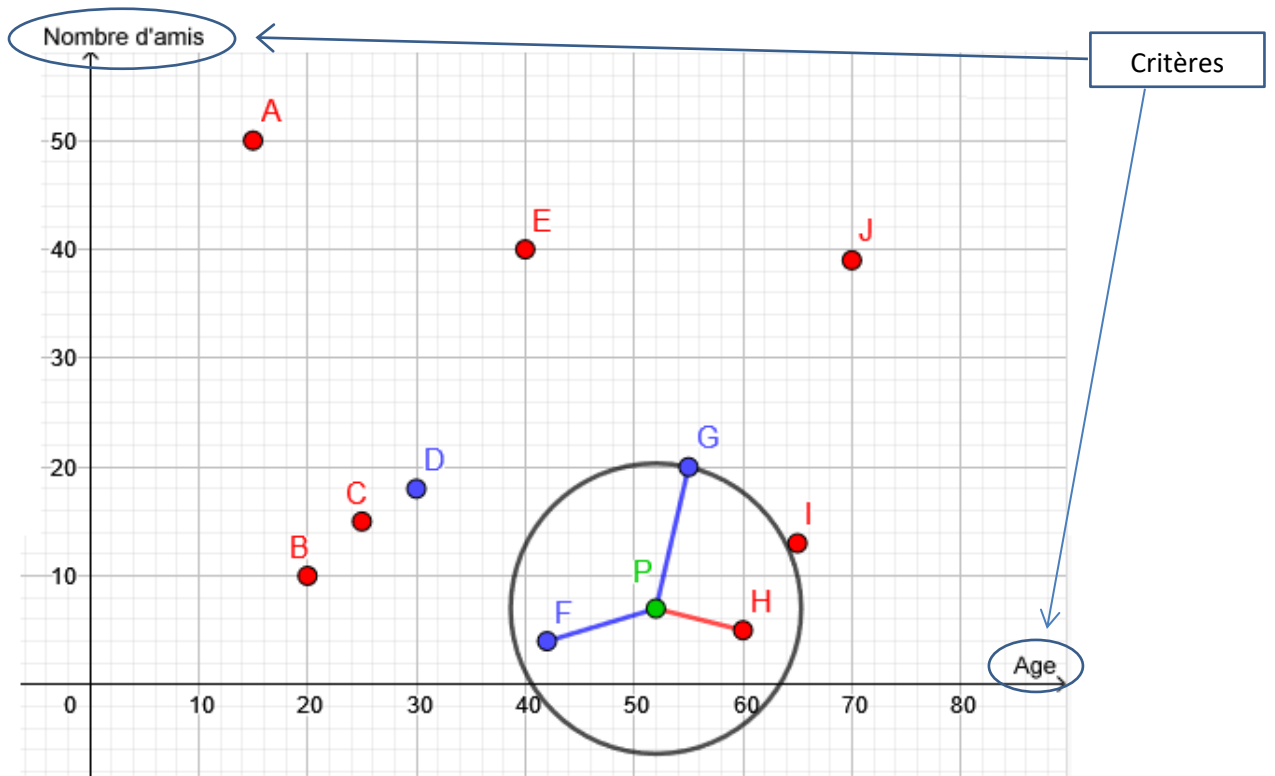
```
mon_dico = classe(3, liste_distances_triee)
```

```
mon_dico vaut alors {'rouge': 1, 'bleu': 2}
```

- Dans cet exemple, on voit que la classe majoritaire est 'bleu'. Donc le point  $P(52; 7)$  est à classer dans la classe 'bleu'. C'est à dire que, selon cet algorithme et selon le jeu de données fourni, selon la valeur  $k = 3$  choisie comme nombre de plus proches voisins de  $P$ , on prédit qu'un individu ayant 52 ans et 7 amis sur Facebook aimera le film.

---

<sup>3</sup> Cette formule est celle de la **distance euclidienne**, du nom du mathématicien Euclide de l'antiquité grecque.



*P est l'entrée à classer à l'aide de ses  $k = 3$  plus proches voisins. Les classes possibles sont rouge et bleu.*

- Les trois plus proches voisins sont de couleurs rouge, bleu, bleu donc la sortie prendra la valeur "classe bleue". Ici, la sortie ne peut prendre qu'un nombre fini de valeurs (deux valeurs : bleu ou rouge). Il s'agit d'un **problème de classification**.
- Si la sortie peut prendre une infinité de valeurs (comme la taille d'un individu dans un intervalle des nombres réels), il s'agit d'un **problème de régression**.

**Remarque :** Dans l'algorithme kNN, une bonne valeur de  $k$  est l'entier le plus proche de  $\sqrt{n}$ .

### 3 Recherche dichotomique dans un tableau trié

Dichotomie signifie **couper en deux**.

- **Méthode**

La recherche dichotomique de l'index d'un élément  $x$  présent<sup>4</sup> dans un tableau **trié** consiste à :

- Comparer  $x$  avec la valeur de l'élément du milieu du tableau.
- Si  $x$  est égal à l'élément du milieu, alors on a trouvé la position de  $x$ .
- Si  $x$  est inférieur à la valeur du milieu, alors on recommence dans la première moitié sinon dans la deuxième moitié.
- On continue tant que la position de  $x$  n'a pas été trouvée.

<sup>4</sup> On supposera toujours que l'élément  $x$  est bien dans le tableau.

### Exemple

Soit un tableau trié de 100 éléments. On cherche l'index de 208 qui est présent dans le tableau.

```
liste_triee = [11, 30, 42, 43, 66, 68, 85, 93, 98, 111, 127, 128, 138, 155, 160, 160, 161,
 174, 178, 208, 231, 238, 250, 254, 259, 262, 265, 270, 272, 285, 292, 302, 309, 351, 352,
 358, 368, 381, 382, 397, 410, 416, 447, 447, 448, 449, 471, 471, 487, 495, 513, 516, 540, 5
44, 557, 557, 572, 590, 601, 602, 613, 615, 627, 633, 633, 635, 658, 662, 682, 686, 698, 70
2, 711, 725, 727, 747, 771, 777, 780, 800, 804, 806, 810, 852, 852, 856, 861, 873, 876, 887
, 893, 927, 931, 935, 966, 971, 975, 984, 996, 999]
```

Pour comprendre plus facilement les explications, on peut faire afficher<sup>5</sup> la liste des doublets (index, valeur) :

```
liste_triee_avec_index = [(0, 11), (1, 30), (2, 42), (3, 43), (4, 66), (5, 68), (6, 85), (
7, 93), (8, 98), (9, 111), (10, 127), (11, 128), (12, 138), (13, 155), (14, 160), (15, 160)
, (16, 161), (17, 174), (18, 178), (19, 208), (20, 231), (21, 238), (22, 250), (23, 254), (
24, 259), (25, 262), (26, 265), (27, 270), (28, 272), (29, 285), (30, 292), (31, 302), (32,
309), (33, 351), (34, 352), (35, 358), (36, 368), (37, 381), (38, 382), (39, 397), (40, 41
0), (41, 416), (42, 447), (43, 447), (44, 448), (45, 449), (46, 471), (47, 471), (48, 487),
(49, 495), (50, 513), (51, 516), (52, 540), (53, 544), (54, 557), (55, 557), (56, 572), (5
7, 590), (58, 601), (59, 602), (60, 613), (61, 615), (62, 627), (63, 633), (64, 633), (65,
635), (66, 658), (67, 662), (68, 682), (69, 686), (70, 698), (71, 702), (72, 711), (73, 725
), (74, 727), (75, 747), (76, 771), (77, 777), (78, 780), (79, 800), (80, 804), (81, 806),
(82, 810), (83, 852), (84, 852), (85, 856), (86, 861), (87, 873), (88, 876), (89, 887), (90
, 893), (91, 927), (92, 931), (93, 935), (94, 966), (95, 971), (96, 975), (97, 984), (98, 9
96), (99, 999)]
```

- La taille du tableau est `n = 100`. `debut = 0` et `fin = 100`.  
`valeur_trouve = False`.

Le tableau va de `liste_triee[0]` qui vaut 11 à `liste_triee[99]` qui vaut 999.

On veut trouver la position de l'élément `x = 208` qui est dans le tableau.

```
m = (0 + 100) // 2 = 50
```

On compare `x` avec la valeur de l'élément du milieu `liste_triee[50]` qui vaut 513.

`208 < 513` donc on conserve la première moitié donc `fin = 50`. On recommence dans la moitié qui contient `x` :

```
[(0, 11), (1, 30), (2, 42), (3, 43), (4, 66), (5, 68), (6, 85), (7, 93), (8, 98), (9, 111),
(10, 127), (11, 128), (12, 138), (13, 155), (14, 160), (15, 160), (16, 161), (17, 174), (1
8, 178), (19, 208), (20, 231), (21, 238), (22, 250), (23, 254), (24, 259), (25, 262), (26,
265), (27, 270), (28, 272), (29, 285), (30, 292), (31, 302), (32, 309), (33, 351), (34, 352
), (35, 358), (36, 368), (37, 381), (38, 382), (39, 397), (40, 410), (41, 416), (42, 447),
(43, 447), (44, 448), (45, 449), (46, 471), (47, 471), (48, 487), (49, 495), (50, 513)]
```

---

<sup>5</sup> L'instruction est : `liste_triee_avec_index = [i for i in enumerate(liste_triee)]`

- La taille de cette moitié est  $n = 51$ .  $\text{debut} = 0$  et  $\text{fin} = 50$ .  
 $\text{valeur\_trouve} = \text{False}$ .

La moitié du tableau va de `liste_triee[0]` qui vaut 11 à `liste_triee[50]` qui vaut 513.

On veut trouver la position de l'élément  $x = 208$  qui est dans le tableau.

$$m = (0 + 50) // 2 = 25$$

On compare  $x$  avec la valeur de l'élément du milieu `liste_triee[25]` qui vaut 262.

$208 < 262$  donc on conserve la première moitié donc  $\text{fin} = 25$ . On recommence dans la moitié qui contient  $x$  :

`[(0, 11), (1, 30), (2, 42), (3, 43), (4, 66), (5, 68), (6, 85), (7, 93), (8, 98), (9, 111), (10, 127), (11, 128), (12, 138), (13, 155), (14, 160), (15, 160), (16, 161), (17, 174), (18, 178), (19, 208), (20, 231), (21, 238), (22, 250), (23, 254), (24, 259), (25, 262)]`

- La taille de cette moitié est  $n = 26$ .  $\text{debut} = 0$  et  $\text{fin} = 25$ .  
 $\text{valeur\_trouve} = \text{False}$ .

La moitié du tableau va de `liste_triee[0]` qui vaut 11 à `liste_triee[25]` qui vaut 262.

On veut trouver la position de l'élément  $x = 208$  qui est dans le tableau.

$$m = (0 + 25) // 2 = 12$$

On compare  $x$  avec la valeur de l'élément du milieu `liste_triee[12]` qui vaut 138.

$208 \geq 138$  donc on conserve la deuxième moitié donc  $\text{debut} = 12$ . On recommence dans la moitié qui contient  $x$  :

`[(12, 138), (13, 155), (14, 160), (15, 160), (16, 161), (17, 174), (18, 178), (19, 208), (20, 231), (21, 238), (22, 250), (23, 254), (24, 259), (25, 262)]`

- La taille de cette moitié est  $n = 14$   $\text{debut} = 12$  et  $\text{fin} = 25$ .  
 $\text{valeur\_trouve} = \text{False}$ .

La moitié du tableau va de `liste_triee[12]` qui vaut 138 à `liste_triee[25]` qui vaut 262.

On veut trouver la position de l'élément  $x = 208$  qui est dans le tableau.

$$m = (12 + 25) // 2 = 18$$

On compare  $x$  avec la valeur de l'élément du milieu `liste_triee[18]` qui vaut 178.

$208 \geq 178$  donc on conserve la deuxième moitié donc  $\text{debut} = 18$ . On recommence dans la moitié qui contient  $x$  :

`[(18, 178), (19, 208), (20, 231), (21, 238), (22, 250), (23, 254), (24, 259), (25, 262)]`

- La taille de cette moitié est  $n = 8$       $\text{debut} = 18$  et  $\text{fin} = 25$ .  
    $\text{valeur\_trouve} = \text{False}$ .

La moitié du tableau va de `liste_triee[18]` qui vaut **178** à `liste_triee[25]` qui vaut **262**.

On veut trouver la position de l'élément  $x = 208$  qui est dans le tableau.

$$m = (18 + 25) // 2 = 21$$

On compare  $x$  avec la valeur de l'élément du milieu `liste_triee[21]` qui vaut 238.

$208 < 238$  donc on conserve la première moitié donc  $\text{fin} = 21$ . On recommence dans la moitié qui contient  $x$  :

`[(18, 178), (19, 208), (20, 231), (21, 238)]`

- La taille de cette moitié est  $n = 4$

La moitié du tableau va de `liste_triee[18]` qui vaut **178** à `liste_triee[21]` qui vaut **238**.

On veut trouver la position de l'élément  $x = 208$  qui est dans le tableau.

$$m = (18 + 21) // 2 = 19$$

On compare  $x$  avec la valeur de l'élément du milieu `liste_triee[19]` **qui vaut 208**.

$208 = 208$  donc  $\text{valeur\_trouve} = \text{True}$ . On a trouvé la position de  $x$  dans la liste triée du départ. C'est l'index 19

### Remarque

A chaque tour de boucle la taille du tableau, dans lequel on cherche  $x$ , est à peu près divisée par 2. Voici une version de l'algorithme de dichotomie implémentée en Python :

```
# Algorithme de dichotomie
```

```
def cherche(tableau, x):
    """
    Cherche l'index dans le tableau de la valeur x
    entrée en paramètre.
    Cette valeur est obligatoirement dans le tableau.

    Paramètres
    -----
    tableau : de type liste d'entiers
    x : de type entier

    Retourne
    -----
    m : de type entier
    C'est l'index de la valeur.

    """
```



```

assert x in tableau, "x doit être dans le tableau"

debut = 0
fin = len(tableau)

valeur_trouve = False # Passera à True lorsque
                       # valeur sera trouvée.
while valeur_trouve == False and debut < fin:
    m = (debut + fin) // 2
    if tableau[m] == x:
        valeur_trouve = True
    else:
        if x < tableau[m]:
            fin = m
        else:
            debut = m
return m

```

- **Coût** : Si on compte uniquement le nombre de comparaisons et si on se place dans le pire cas (l'index de x n'est trouvé qu'au dernier tour de boucle) l'algorithme fait **un seul tour de boucle de plus lorsque la longueur du tableau** passé en paramètre **double**.

**On dit que sa complexité est en logarithme base 2 de  $n$ . On note cela  $O(\log_2(n))$ .** Si on se réfère au graphique du classement des algorithmes (chapitre 7) on voit que cette complexité est excellente. Autrement dit, l'algorithme est très rapide même sur des tableaux de longueur  $n$  très grande.

- **Preuves de correction**

Correction totale	
Terminaison	Correction partielle
<p>Il y a une boucle while, donc on doit prouver qu'elle se termine grâce à <b>un variant de boucle</b>. Ce variant est la quantité <math>fin - debut</math>.</p> <p>A chaque tour de boucle, soit la variable <i>debut</i>, soit la variable <i>fin</i>, prend la valeur de <math>m</math> qui est situé entre <i>debut</i> et <i>fin</i>. Donc la quantité positive <math>fin - debut</math> diminue strictement à chaque tour. Donc en un temps fini cette quantité devient nulle. <math>fin - debut = 0</math> <math>fin = debut</math></p> <p>Donc la condition de maintien dans la boucle <math>debut &lt; fin</math> devient fausse. <i>L'algorithme se termine.</i></p>	<p>Soit la propriété " x est dans le tableau allant de <math>tableau[debut]</math> à <math>tableau[fin]</math> inclus ".</p> <ul style="list-style-type: none"> <li>• Cela est vrai sur le tableau initial (par hypothèse)</li> <li>• La valeur de l'index <math>m</math> est calculée de façon que si x est dans le tableau précédent alors x est dans la moitié de tableau suivante.</li> <li>• Cette propriété est donc vraie à n'importe quel tour de boucle.</li> </ul> <p>En particulier elle est vraie lorsque <math>debut=fin</math>. L'algorithme se termine et la dernière moitié ne contient plus qu'un seul élément qui est x et dont <math>m</math> est l'index.</p> <p><b>Remarque</b> : Dans le cas où x est trouvé avant que <math>debut=fin</math>, cela signifie que la boucle a été arrêtée et que x a comme index <math>m</math>.</p>

## 4 Algorithmes gloutons

Un problème d'optimisation est un problème pour lequel on cherche la meilleure solution (selon un critère défini) dans un ensemble de solutions possibles.

La **solution optimale** est la solution la meilleure selon le critère défini.

Les algorithmes gloutons<sup>6</sup> sont souvent utilisés pour résoudre ces problèmes d'optimisation. On cherche une solution optimale en effectuant le meilleur choix possible à chaque étape de l'algorithme. Lorsqu'un choix est fait, il n'est pas modifié par la suite. On se retrouve donc à chaque étape, avec un problème de plus en plus petit à résoudre. Cette méthode ne fournit pas toujours la solution optimale.

### 4.1 Premier exemple d'algorithme glouton : le sac à dos

Un voleur est en train de cambrioler une maison : il repère 6 objets, ayant tous une certaine valeur et une certaine masse. Comment va-t-il choisir les objets à emporter sachant qu'il ne peut pas mettre dans son sac à dos plus de 15 Kg ?

Objet	Valeur	Masse	Valeur/Masse
Objet 1	126	14	9
Objet 2	32	2	16
Objet 3	20	5	4
Objet 4	5	1	5
Objet 5	18	6	3
Objet 6	80	8	10

Il y a plusieurs types de choix possibles.

- Le voleur préfère emporter d'abord les objets qui ont la plus grande valeur (en €).
- Le voleur préfère emporter d'abord les objets qui ont la plus petite masse (en Kg)
- Le voleur préfère emporter d'abord les objets qui ont le plus grand rapport Valeur/Masse (en €/Kg).

On appelle cela des **heuristiques**.

"**Une heuristique est un raisonnement** formalisé de résolution de problème dont on tient pour plausible, mais non pour certain, qu'il conduira à la détermination d'une solution satisfaisante du problème."

- Pour savoir quelle est la meilleure heuristique, on programme un algorithme glouton.

---

<sup>6</sup> Glouton car à chaque étape il fait le meilleur choix possible. En anglais on dit "greedy algorithm".

- Si la première heuristique est suivie, l'algorithme glouton va trier le tableau par valeurs décroissantes puis il prendra (ou non) dans cet ordre les objets tant que le cumul des masses reste inférieur ou égal à 15. La valeur du sac à dos est alors de **131 €**.
- Si la deuxième heuristique est suivie, l'algorithme glouton va trier le tableau par masses croissantes puis il prendra (ou non) dans cet ordre les objets tant que le cumul des masses reste inférieur ou égal à 15. La valeur du sac à dos est alors de **75 €**.
- Si la troisième heuristique est suivie, l'algorithme glouton va trier le tableau par rapports Valeurs/Masses décroissants puis il prendra (ou non) dans cet ordre les objets tant que le cumul des masses reste inférieur ou égal à 15. La valeur du sac à dos est alors de **117 €**.

L'objet examiné est pris (ou non) si le cumul des masses avec ce nouvel objet reste inférieur ou égal à 15 Kg (ou non).

**Exemple détaillé avec la troisième heuristique "prendre d'abord les objets qui ont le plus grand rapport Valeur/Masse (en €/Kg)."**

La fonction glouton fonctionne à l'aide de listes séparées extraites du tableau des objets :

- **tableau des objets** : Il contient la ligne des descripteurs et six lignes de données.

identifiant	valeur	masse
1	126	14
2	32	2
3	20	5
4	5	1
5	18	6
6	80	8

- **liste des identifiants** : elle contient n = 6 lignes.

1
2
3
4
5
6

liste\_id ← Première colonne du tableau des objets privée de son en-tête.

- **liste des valeurs** : elle contient n = 6 lignes.

126
32
20
5
18
80

liste\_va ← Deuxième colonne du tableau des objets privée de son en-tête.

- **liste des masses** : elle contient n = 6 lignes.

14
2
5
1
6
8

liste\_ma ← Troisième colonne du tableau des objets privée de son en-tête.

Voici le pseudo code :

```

Algorithmme : glouton(liste_id, liste_va, liste_ma):
  n ← longueur(liste_id)

  # Fabrication de la liste des valeurs massiques.
  liste_va_ma ← liste vide
  pour i allant de 0 à n-1
    liste_va_ma ← liste_va[i] / liste_ma[i]

  # Fabrication du tableau d'objets avec seulement les données utiles.
  tab_ob ← liste vide
  pour i allant de 0 à n-1
    tab_ob ← liste des trois valeurs liste_id[i], liste_ma[i], liste_va_ma[i]

  # Fabrication du tableau trié par valeurs massiques décroissantes.
  tab_ob_trie ← tri de tab_ob, selon les valeurs massiques décroissantes.

  # Fabrication du sac.
  sac ← liste vide
  masse_sac ← 0
  i ← 0
  tant que masse_sac + tab_ob_trie[i][1] < 15:
    Ajouter dans sac (tab_ob_trie[i][0])
    masse_sac = masse_sac + tab_ob_trie[i][1]
    i ← i + 1
  renvoyer sac

```

Avec :

```
# Fabrication de la liste des valeurs massiques.  
liste_va_ma ← liste vide  
pour i allant de 0 à n-1  
    liste_va_ma ← liste_va[i] / liste_ma[i]
```

On obtient

**liste\_va\_ma** : elle contient n = 6 lignes.

9.0
16.0
4.0
5.0
3.0
10.0

Avec :

```
# Fabrication du tableau d'objets avec seulement les données utiles.  
tab_ob ← liste vide  
pour i allant de 0 à n-1  
    tab_ob ← liste des trois valeurs liste_id[i], liste_ma[i], liste_va_ma[i]
```

On obtient

**tab\_ob** : il contient n = 6 lignes.

1	14	9.0
2	2	16.0
3	5	4.0
4	1	5.0
5	6	3.0
6	8	10.0

Avec :

```
# Fabrication du tableau trié par valeurs massiques décroissantes.  
tab_ob_trie ← tri de tab_ob, selon les valeurs massiques décroissantes.
```

On obtient

**tab\_ob\_trie** : il contient n = 6 lignes.

2	2	16.0
6	8	10.0
1	14	9.0
4	1	5.0
3	5	4.0
5	6	3.0

Avec

```
# Fabrication du sac.
sac ← liste vide
masse_sac ← 0
i ← 0
tant que masse_sac + tab_ob_trie[i][1] < 15:
    Ajouter dans sac (tab_ob_trie[i][0])
    masse_sac = masse_sac + tab_ob_trie[i][1]
    i = i + 1
renvoyer sac
```

On obtient (avec une masse maximale totale : 15 kg)

A prendre	Objet	Valeur	Masse	Valeur/Masse	Masse cumul.
✓	Objet 2	32	2	16.0	2.0
✓	Objet 6	80	8	10.0	10.0
X	Objet 1	126	14	9.0	
✓	Objet 4	5	1	5.0	11.0
X	Objet 3	20	5	4.0	
X	Objet 5	18	6	3.0	

- L'algorithme passe en revue les enregistrements dans le tableau trié à partir de la première ligne, et prend dans le sac à dos les objets à condition que la masse cumulée reste inférieure ou égale à masse max = 15 kg :
  - La masse de l'objet en première ligne est inférieure ou égale à 15 Kg donc son identifiant est ajouté à la liste sac.
    - La masse cumulée = 2 Kg.
  - La masse cumulée + masse de l'objet en deuxième ligne est inférieure ou égale à 15 Kg donc il est ajouté à la liste sac.
  - La masse cumulée + masse de l'objet en troisième ligne est supérieure à 15 Kg donc il n'est pas ajouté à la liste sac.
  - La masse cumulée + masse de l'objet en quatrième ligne est inférieure à 15 Kg donc il est ajouté à la liste sac.
  - La masse cumulée + masse de l'objet en cinquième ligne est supérieure à 15 Kg donc il n'est pas ajouté à la liste sac.
  - La masse cumulée + masse de l'objet en sixième ligne est supérieure à 15 Kg donc il n'est pas ajouté à la liste sac.
    - La masse cumulée = 11 Kg.
- La valeur des objets du sac à dos est  $32 + 80 + 5 = 117$ .

117 € est une bonne solution, mais on n'est pas certain qu'elle soit optimale.

Par une méthode exhaustive (c'est à dire une étude complète de toutes les façons de choisir des objets parmi les 6 objets tout en respectant la contrainte sur le poids cumulé), on montre que le choix optimal est :

A prendre	Objet	Valeur	Masse	Valeur/Masse
X	Objet 1	126	14	9
✓	Objet 2	32	2	16
✓	Objet 3	20	5	4
X	Objet 4	5	1	5
X	Objet 5	18	6	3
✓	Objet 6	80	8	10

La masse cumulée est  $2 + 5 + 8 = 15$  donc la contrainte de la masse cumulée inférieure ou égale à 15 Kg est respectée. La valeur des objets du sac à dos est  $32 + 20 + 80 = 132$  € qui est la solution optimale.

Avec peu d'objets comme ici, la méthode exhaustive est encore possible. Mais avec un grand nombre d'objets elle est beaucoup trop coûteuse en temps de calcul.

On prendra donc la méthode gloutonne. Elle ne donne pas toujours le résultat optimal mais elle est rapide et son résultat est intéressant.

## 4.2 Deuxième exemple d'algorithme glouton : le rendu de monnaie

On cherche à rendre la monnaie avec un nombre minimal de pièces et billets.

Voici notre système de monnaie (exprimé en euros) :

Pièces : 0,01 ; 0,02 ; 0,05 ; 0,1 ; 0,2 ; 1 ; 2 .

Billets : 5 ; 10 ; 20 ; 50 ; 100 ; 200 ; 500.

On cherche par exemple à rendre 53 euros.

- Principe de l'algorithme glouton de rendu de monnaie :

Il rend la monnaie **en commençant par la pièce ou le billet avec la plus grande valeur possible** (en restant inférieur à la somme à rendre). Il recommence ainsi jusqu'à obtenir une valeur nulle. On note

- système : p-uplet de valeurs de billets et de pièces classé par valeurs décroissantes.
- somme\_a\_rendre : montant à obtenir.
- somme\_restante : montant qui reste à rendre à la fin de chaque étape.
- rendu : liste renvoyée qui contient le nombre de chaque valeur de billets et de pièces.

### Exemple

système = (200, 100, 50, 20, 10, 5, 2, 1)

somme\_a\_rendre = 53

Algorithme : rendu\_de\_monnaie(somme\_a\_rendre, systeme):

```

# Initialisation de somme_restante.
somme_restante ← somme_a_rendre

# Initialisation de rendu (la liste des nombres de billets ou pièces par valeurs).
rendu ← liste de la même longueur que le p-uplet systeme mais remplie de 0.

# i est l'index de la valeur de systeme qui est examinée dans la boucle interne.
i ← 0

tant que somme_restante est non nulle
  # Calcul de l'index de la valeur de billet ou pièce à rendre dans ce tour de boucle.
  tant que systeme[i] est strictement supérieure à somme_restante
    i ← i + 1 # Au dernier tour de cette boucle interne, i a comme valeur l'index
              # de la valeur de billet ou pièce à rendre maintenant.

  # Calcul du nombre de billets ou pièces à rendre dans cette valeur.
  rendu[i] ← quotient de somme_restante divisé par systeme[i]
  # Calcul de la somme_restante au début du tour de boucle externe suivant.
  somme_restante ← reste de somme_restante divisé par systeme[i]

renvoyer rendu

```

**Etape 1** : premier tour de la boucle *tant que* externe

- Evolution des variables dans la boucle *tant que* interne :

i	0	1	2		
systeme[i]	systeme[0] vaut 200	systeme[1] vaut 100	systeme[1] vaut 50		
systeme[i]>53	Vrai	Vrai	Faux		

Sortie de la boucle *tant que* interne avec i qui vaut 2

rendu[2] ← quotient de 53 divisé par 50 donc la liste rendu vaut [0, 0, 1, 0, 0, 0, 0, 0].  
 somme\_restante ← reste de 53 divisé par 50 donc somme\_restante vaut 3.

**Etape 2** : deuxième tour de la boucle *tant que* externe

- Evolution des variables dans la boucle *tant que* interne :

i	2	3	4	5	6
systeme[i]	systeme[2] vaut 50	systeme[3] vaut 20	systeme[4] vaut 10	systeme[5] vaut 5	systeme[6] vaut 2
systeme[i]>53	Vrai	Vrai	Vrai	Vrai	Faux

Sortie de la boucle *tant que* interne avec i qui vaut 6

rendu[6] ← quotient de 3 divisé par 2 donc la liste rendu vaut [0, 0, 1, 0, 0, 0, 1, 0].  
 somme\_restante ← reste de 3 divisé par 2 donc somme\_restante vaut 1.



### Etape 3 : troisième tour de la boucle *tant que* externe

- Evolution des variables dans la boucle *tant que* interne :

i	6	7			
systeme[i]	systeme[6] vaut 2	systeme[7] vaut 1			
systeme[i]>1	Vrai	Faux			

Sortie de la boucle *tant que* interne avec i qui vaut 7

rendu[7] ← quotient de 1 divisé par 1 donc la liste rendu vaut [0, 0, 1, 0, 0, 0, 1, 1].

somme\_restante ← reste de 1 divisé par 1 donc somme\_restante vaut 0.

La condition somme\_restante est non nulle de maintien dans la boucle tant que externe est devenue fausse. Il n'y a plus de nouvelle étape. L'algorithme renvoie la liste rendu qui vaut :

[0, 0, 1, 0, 0, 0, 1, 1], dans le système (200, 100, 50, 20, 10, 5, 2, 1)

#### Résumé des étapes

Etapes	On rend	rendu	somme_restante
Initialisation		[0, 0, 0, 0, 0, 0, 0, 0].	53
1	1 billet de 50 €	[0, 0, 1, 0, 0, 0, 0, 0].	3
2	1 pièce de 2 €	[0, 0, 1, 0, 0, 0, 1, 0].	1
3	1 pièce de 1 €	[0, 0, 1, 0, 0, 0, 1, 1].	0

### 4.3 D'autres exemples qui se ramènent au problème du sac à dos

- Découper de matériaux, afin de minimiser les chutes.
- Choisir les projets les plus rentables dans le monde de la finance.
- Réaliser le meilleur chargement possible d'un ensemble de camions.

Le problème du sac à dos fait partie des problèmes dits "NP<sup>7</sup>-complets". Ils sont réputés pour être les problèmes les plus difficiles à résoudre en optimisation combinatoire.

---

<sup>7</sup> NP : Non déterministe Polynomial.