Chapitre n° 12 : quelques algorithmes classiques

Compétences attendues

- Savoir écrire un algorithme de tri.
- Décrire un invariant de boucle qui prouve la correction des tris par insertion, par sélection.
- Justifier la terminaison de ces algorithmes de tri et montrer que leur coût est quadratique au pire cas.
- Écrire un algorithme qui prédit la classe d'un élément en fonction de la classe majoritaire de ses k plus proches voisins.
- Résoudre un problème grâce à un algorithme glouton.

1 Algorithmes de tris

1.1 Tri par sélection

1.1.1 Principe de l'algorithme

On dispose de n données. On cherche la plus petite donnée et on la place en première position, puis on cherche la plus petite donnée parmi les données restantes et on la place en deuxième position, et ainsi de suite.

Si les données sont les éléments d'une liste *liste*, l'algorithme consiste donc à faire varier un indice i de 0 à $n-2$ (donc jusqu'à l'avant-dernier élément). Pour chaque valeur de i on cherche dans la tranche $[i : n]$ (éléments d'indices allant de i à $n-1$ inclus) le plus petit élément et on l'échange avec $liste[i]$.

1.1.2 Programmation en Python

```
1  def tri_selection(liste):
2      for i in range(len(liste) - 1):
3          i_mini = i  # indice initial du minimum
4          mini = liste[i]  # valeur initiale du minimum
5          for j in range(i + 1, len(liste)):
6              if liste[j] < mini:
7                  i_mini = j  # nouvel indice temporaire du minimum
8                  mini = liste[j]  # nouvelle valeur temporaire du
       minimum
9          liste[i], liste[i_mini] = liste[i_mini], liste[i]
```

Remarque : cette fonction modifie la liste passée en argument. Il faut faire une copie de la liste d'origine si on ne veut pas la modifier, et renvoyer ensuite la copie de la liste une fois triée.

1.1.3 Terminaison de l'algorithme

Cet algorithme comporte deux boucles *for* imbriquées donc le nombre de passages dans ces deux boucles est parfaitement déterminé et il est évidemment fini.

1.1.4 Correction de l'algorithme

Pour prouver la correction de l'algorithme, nous utiliserons l'invariant : « pour chaque i, la liste est une permutation de la liste initiale, la liste $liste[0 : i+1]$ (éléments d'indices de 0 à i inclus) est triée et tous les éléments de la liste $liste[i+1 : n]$ (éléments d'indices de $i+1$ à $n-1$ inclus) sont supérieurs à tous les éléments de la liste $liste[0 : i+1]$ ».

Initialisation : après le premier passage dans la boucle pour $i = 0$, la liste $liste[0 : 1]$ ne contient que l'élément $liste[0]$ qui est le minimum de toute la liste, donc qui est bien inférieur à tous les éléments de $liste[1 : n]$. *La propriété est donc vraie pour* $i = 0$.

Hérédité : si après un passage pour $i = k$ quelconque, la liste $liste[0 : k+1]$ est triée et tous les éléments de la liste $liste[k+1 : n]$ sont supérieurs à tous les éléments de la liste $liste[0 : k+1]$, alors au passage suivant le minimum de la liste $liste[k+1 : n]$ est placé en position d'indice $k+1$.
Ce minimum est est supérieur à tous les éléments de la liste $liste[0 : k+1]$ et inférieur à tous les éléments de la liste $liste[k+2 : n]$. *La propriété est donc vraie pour* $i = k+1$.

Conclusion : la propriété est donc encore vraie après le dernier passage pour $i = n-2$. À ce moment la liste $liste[0 : n-1]$ est triée et le dernier élément, d'indice $n-1$, est supérieur à tous les éléments de la liste $liste[0 : n-1]$. *Donc la liste* $liste[0 : n]$*, c'est à dire toute la liste, triée.*

1.1.5 Coût de l'algorithme

Nous avons deux boucles *for* imbriquées. Pour une liste de longueur n, quels que soient les éléments, le nombre de comparaisons est le même.

Pour chaque valeur de i, j prend les valeurs de $i+1$ à $n-1$, soit $n-i-1$ valeurs. Et pour chaque valeur de j, une unique comparaison est effectuée. Donc, pour chaque valeur de i nous avons exactement $n-i-1$ comparaisons.

Au total, cela fait $(n-1) + (n-2) + \cdots + 2 + 1$ comparaisons. Un calcul mathématique nous donne $\frac{n \cdot (n-1)}{2}$ comparaisons.

Nous avons un coût quadratique, i.e. de l'ordre de n^2 comparaisons, quelle que soit la liste de longueur n, même si elle est déjà triée !

L'algorithme de tri par sélection n'est pas très efficace, mais il est simple à programmer, et peut être utilisé dans le cas de listes n'étant pas trop longues.

Remarque : au coût des comparaisons nous pourrions rajouter le coût des affectations (2 pour chaque valeur de i et jusqu'à 2 pour chaque valeur de j) et des permutations (0 ou 1 pour chaque valeur de i).
Ca ne changerait pas le résultat : le coût de l'algorithme de tri par sélection est quadratique.

1.2 Tri par insertion

1.2.1 Principe de l'algorithme

On dispose de n données et on procède par étapes. À chaque étape, on suppose que les k premières données sont triées et on insère une donnée supplémentaire à la bonne place parmi ces k données.

Si les données sont les éléments d'une liste *liste*, l'algorithme consiste donc à faire varier un indice i de 1 à $n-1$ (donc du deuxième au dernier élément). Pour chaque valeur de i on cherche dans la tranche $[0 : i]$ (éléments d'indices allant de 0 à $i-1$ inclus) à quelle place doit être inséré l'élément $liste[i]$, qu'on appelle *la clef*. Pour cela on compare la clef successivement aux données précédentes, en commençant par l'élément d'indice $i-1$, puis en remontant dans la liste jusqu'à trouver la bonne place, c'est à dire entre deux données successives, l'une étant plus petite et l'autre plus grande que la clef. Si la clef est plus petite que toutes les données précédentes, elle se place en premier. Pour ce faire, on décale d'une place vers la droite les données plus grandes que la clef après comparaison.

1.2.2 Programmation en Python

```
1  def tri_insertion(liste):
2      for i in range(1, len(liste)):  # on parcourt la liste du deuxi
       ème élément jusqu'au dernier
3          k = i  # indice initial de la clef
4          clef = liste[i]  # valeur à insérer
5          while k > 0 and clef < liste[k - 1]:
6              liste[k] = liste[k - 1]  # on décale la valeur vers la
       droite
7              k -= 1  # on diminue de 1 l'indice temporaire de la
       clef i.e. on la décale d'un cran vers la gauche
8          liste[k] = clef  # on sort de la boucle while lorsqu'on a
       trouvé la bonne position pour la clef
```

Remarque : cette fonction modifie la liste passée en argument. Il faut faire une copie de la liste d'origine si on ne veut pas la modifier, et renvoyer ensuite la copie de la liste une fois triée.

1.2.3 Terminaison de l'algorithme

Pour la terminaison, il étudier les deux boucles. La boucle externe est une boucle *for*, donc le nombre de passages est déterminé et fini.

La boucle interne est une boucle *while*. Les valeurs prises par le variant k constituent une suite d'entiers décroissante incluse dans la suite des entiers de i à 1. Il y a donc au plus i passages dans la boucle *while*.

1.2.4 Correction de l'algorithme

Pour prouver la correction de l'algorithme, nous utiliserons l'invariant : « pour chaque i, la liste est une permutation de la liste initiale et la liste $liste[0 : i + 1]$ est triée ».

Initialisation : après le premier passage dans la boucle pour $i = 1$, le premier élément $liste[0]$ et la clef d'indice 1 sont rangés dans l'ordre. Donc la liste $liste[0 : 2]$ est triée. *La propriété est donc vraie pour* $i = 1$.

Hérédité : si après un passage pour $i = k$ quelconque, la liste $liste[0 : k+1]$ est triée, alors au passage suivant l'élément $liste[k + 1]$ est inséré à la bonne place parmi les éléments de la liste $liste[0 : k + 1]$ ou reste à sa place. Donc la liste $liste[0 : k + 2]$ est triée. *La propriété est donc vraie pour* $i = k + 1$.

Conclusion : la propriété est donc encore vraie après le dernier passage pour $i = n - 1$. *À ce moment la liste liste*$[0 : n]$*, c'est à dire toute la liste, est triée.*

1.2.5 Coût de l'algorithme

Nous avons deux boucles imbriquées. Pour une liste de longueur n, le nombre de comparaisons peut être différent suivant la liste.

Si la liste est déjà triée, pour chaque valeur de i, k prend la valeur de i et il y a une seule comparaison, le test $clef < liste[k - 1]$. La variable i prenant $n - 1$ valeurs, cela fait un total de $n - 1$ comparaisons. Le coût de l'algorithme est alors linéaire, de l'ordre de n.

Si par contre les éléments de la liste sont rangés par ordre décroissant (ce qui correspond au pire cas), alors pour chaque valeur de i, k prend les valeurs de i à 1, soit i valeurs et donc i comparaisons.

Au total, cela fait $1 + 2 + \cdots + (n - 2) + (n - 1)$ comparaisons. Un calcul mathématique nous donne $\frac{n\cdot(n-1)}{2}$ comparaisons.

Au pire cas (et en moyenne), nous avons un coût quadratique, i.e. de l'ordre de n^2 *comparaisons.*

L'algorithme de tri par sélection n'est pas très efficace, sauf dans le cas d'une liste presque déjà triée.

Remarque : au coût des comparaisons nous pourrions rajouter le coût des affectations (3 pour chaque valeur de i et 0 ou 2 pour chaque valeur de k).
Ça ne changerait pas le résultat : le coût de l'algorithme de tri par insertion est quadratique, au pire cas et en moyenne.

2 Algorithme des plus proches voisins

2.1 Présentation des algorithmes d'apprentissage

Les algorithmes traditionnels vus jusqu'ici visent à apporter efficacement des réponses correctes à des problèmes précisément définis.

Les algorithmes par apprentissage visent à apporter une réponse plausible, mais pas nécessairement exacte, à un problème auquel il est difficile d'appliquer un algorithme traditionnel. Le problème visé peut typiquement :

- être d'une complexité telle que le calcul d'une réponse exacte prendrait beaucoup trop de temps (par exemple le choix du meilleur coup à jouer dans une partie de go) ;
- ne pas avoir de définition suffisamment précise (par exemple le choix de la meilleure traduction en français d'une phrase en langue étrangère) ;
- être basé sur des données incomplètes ou imprécises (par exemple le choix de la meilleure publicité à montrer à un utilisateur d'internet en fonction de ses goûts et de son humeur du moment) ;
- etc.

Les algorithmes d'apprentissage sont utilisés notamment dans le domaine de l'intelligence artificielle. Pour répondre à de tels problèmes, ils ont besoin d'une grande quantité d'exemples associant des données d'entrée et les réponses attendues, dont ils se servent pour essayer de deviner une réponse convenable lorsqu'on leur propose de nouvelles données d'entrées.

2.2 Principe de l'algorithme des plus proches voisins

L'algorithme des k *plus proches voisins est un algorithme d'apprentissage.*

Le but de cet algorithme est d'émettre une prévision sur une caractéristique d'un élément (on parle de *classe* de cet élément) en fonction de l'étude de ses k plus proches voisins, où k est à préciser.

Suivant les cas, la caractéristique de l'élément pourra alors être celle de la majorité de ses k plus proches voisins, ou une moyenne, ou une moyenne pondérée en fonction de la distance des voisins, etc.

2.3 Notion de distance

Pour définir la notion de « plus proches » voisins, il faut définir la notion de distance. Si les éléments sont des points répartis dans un plan, la distance peut être directement la distance euclidienne, calculable à partir des coordonnées.

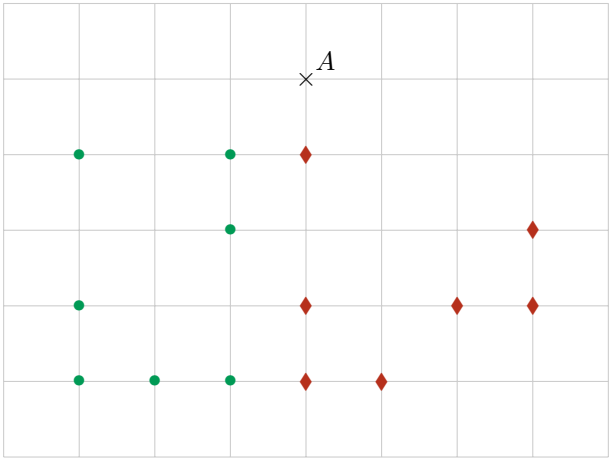
Par exemple, pour deux points A et B de coordonnées (x_A, y_A) et (x_B, y_B), la distance euclidienne vaut $\sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$.

Néanmoins, la distance peut être calculée différemment, par exemple en suivant un quadrillage. De plus, les éléments ne sont pas nécessairement des points. Par exemple, ils peuvent être des caractères, des chaînes de caractères, des pixels colorés, etc.

Il est par exemple possible de regarder la distance entre les coordonnées colorimétriques d'un pixel et de ses voisins pour délimiter le contour d'une image.

2.4 Exemple

Imaginons le problème suivant : un caméléon, positionné au point A, doit choisir la classe qu'il doit prendre pour mieux se fondre dans son environnement, soit *rond vert*, soit *losange rouge*.



Exercice : la distance considérée est la distance géométrique usuelle. Quelle est la classe prise par le caméléon si on applique l'algorithme du plus proche voisin ? Même question pour l'algorithme des 3, puis des 5 plus proches voisins ?

Réponse : voici les valeurs obtenues si on applique l'algorithme des k plus proches voisins.

— Si $k = 1$, le caméléon prend la classe *losange rouge* car le plus proche voisin est de cette classe.

— Si $k = 3$, le caméléon prend la classe *rond vert* car les trois plus proches voisins sont, dans l'ordre, ♦ ● ●. Il y a une majorité de *ronds verts*.

— Si $k = 5$, le caméléon prend la classe *rond vert* car les cinq plus proches voisins sont, dans l'ordre, ♦ ● ● ♦ ●. Il y a une majorité de *ronds verts*.

3 Algorithmes gloutons

3.1 Présentation des algorithmes d'optimisation

Un problème d'optimisation a deux caractéristiques : une fonction que l'on doit maximiser ou minimiser et une série de contraintes auxquelles il faut satisfaire. On peut essayer de résoudre ce problème en écrivant un algorithme qui énumère les possibilités de manière exhaustive afin de trouver la meilleure. C'est un algorithme en général simple, mais souvent inutilisable en machine à cause de son coût.

Exemples : pour avoir quelques ordres de grandeur, voici des valeurs de longueur n de tableaux donnant *un nombre d'opérations proche d'un milliard*, en fonction de la complexité de l'algorithme.

- Complexité logarithmique : $log(n)$. Dans ce cas, $n = 10^{300\,000\,000}$.
- Complexité racinaire : $\sqrt{n}$. Dans ce cas, $n = 10^{18}$.
- Complexité linéaire : n. Dans ce cas, $n = 1\,000\,000\,000$.
- Complexité quadratique : n^2. Dans ce cas, $n = 31\,600$.
- Complexité cubique : n^3. Dans ce cas, $n = 1\,000$.
- Complexité exponentielle : 2^n. Dans ce cas, $n = 30$.
- Complexité factorielle : $n!$. Dans ce cas, $n = 12$.

Il n'est donc pas toujours possible de trouver le choix globalement optimal (sur l'ensemble du problème).

En revanche, il est possible de trouver des choix localement optimaux : ce sont les meilleurs parmi un ensemble restreint de choix. Le principe d'un algorithme glouton est de sélectionner à chaque étape de l'algorithme le meilleur choix parmi les propositions.

Suivant les problèmes, l'algorithme glouton fourni parfois la solution globalement optimale, mais ça n'est pas systématique.

Cette présentation pouvant paraître abstraite, nous allons appliquer des algorithmes gloutons dans deux exemples classiques.

3.2 Problème du sac à dos

Le premier exemple classique est le « problème du sac à dos ». Nous sommes devant un ensemble de n objets. A chaque objet o_i, nous associons une valeur v_i et une masse m_i. Le but est d'emporter dans le sac à dos l'ensemble d'objets qui a la plus grande valeur totale, la contrainte étant que le sac ne supporte qu'une masse maximale M.

L'algorithme glouton sélectionnera à chaque étape l'objet suivant à ajouter au sac à dos.
Il y a ici trois manière de définir ce meilleur choix d'objet :

— soit on choisit l'objet qui a la valeur maximale ;

— soit on choisit l'objet qui a la masse minimale ;

— soit on choisit l'objet qui a le rapport valeur/masse maximal.

Nous ne considérons à la première étape que les objets ayant une masse $m_i \leq M$. L'algorithme glouton choisit le meilleur objet, que nous noterons O_1, de valeur V_1 et de masse M_1.
Ensuite, nous recommençons parmi les objets de masse $m_i \leq M - M_1$, et ainsi de suite, jusqu'à ce qu'il n'y ait plus d'objet disponible satisfaisant la contrainte.

Exemple : le sac à dos peut contenir 15 kg. La liste des 6 objets possibles est donnée dans le tableau suivant.

Objet	Valeur (€)	Masse (kg)	Valeur/Masse ($€\cdot kg^{-1}$)
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

Dans le programme, les objets seront représentés par une liste du type $['objet1', 126, 14]$.

Nous définissons trois fonctions qui prennent en paramètre un objet et renvoie respectivement la valeur, l'inverse de la masse et le rapport valeur/masse de l'objet.

```
1  def valeur(obj):
2      return obj[1]
3
4  def inv_masse(obj):
5      return 1 / obj[2]
6
7  def rapport(obj):
8      return obj[1] / obj[2]
```

Nous définissons ensuite une fonction *glouton* qui prend en paramètres une liste d'objets, une masse maximale (celle que peut supporter le sac à dos) et le type de choix utilisé (par valeur, par masse ou par rapport valeur/poids).

La première chose à faire est de trier la liste suivant le type de choix utilisé par ordre décroissant. Nous utilisons la fonction *sorted* de Python pour obtenir la liste des objets triée et l'affecter à la variable *copie*. Le paramètre $key = choix$ permet d'appeler la fonction passée par l'argument *choix* comme critère de tri et le paramètre $reverse = True$ permet d'obtenir la liste triée par ordre décroissant.

Ensuite, nous parcourons la liste triée et ajoutons les noms des objets un par un tant que la masse totale ne dépasse pas la masse maximale. La valeur totale et la masse totale sont stockées dans les variable *valeur* et *masse*.

```
def glouton(liste, masse_max, choix):
    copie = sorted(liste, key=choix, reverse=True)
    reponse = []
    valeur = 0
    masse = 0
    i = 0
    while i < len(liste) and masse <= masse_max:
        nom, val, mas = copie[i]
        if masse + mas <= masse_max:
            reponse.append(nom)
            masse += mas
            valeur += val
        i += 1
    return reponse, valeur
```

Pour tester ce programme, il reste à définir la liste d'objets, puis à exécuter la fonction glouton pour chacun des choix possibles.

```
objets = [['objet 1', 126, 14], ['objet 2', 32, 2],
          ['objet 3', 20, 5], ['objet 4', 5, 1],
          ['objet 5', 18, 6], ['objet 6', 80, 8]]

print(glouton(objets, 15, valeur))
print(glouton(objets, 15, inv_masse))
print(glouton(objets, 15, rapport))
```

L'exécution du programme donne les résultats suivants.

— Par valeur : (['objet 1', 'objet 4'], 131).

— Par masse : (['objet 4', 'objet 2', 'objet 3', 'objet 5'], 75).

— Par rapport valeur/masse : (['objet 2', 'objet 6', 'objet 4'], 117).

Le choix par valeur est le plus intéressant ici. Néanmoins, il ne fournit pas la solution optimale qui est :
$(['objet 2', 'objet 3', 'objet 6'], 132)$.

Remarque : dans une version fractionnaire du problème où on pourrait choisir une fraction de chaque objet (par exemple s'ils étaient liquides ou en poudre), le meilleur choix serait le rapport valeur/masse et l'algorithme glouton fournirait systématiquement la solution optimale.

3.3 Problème du rendu de monnaie

Le second exemple classique est le « problème du rendu de monnaie ». Le but est d'utiliser le minimum de pièces et de billets pour rendre une somme

donnée.

La somme à rendre sera notée r. Le système de pièces et billets utilisé sera un n-uplet $S = (p_0, p_1, \ldots, p_{n-1})$, où p_i représente la valeur de la pièce (ou du billet) d'indice i. Ces valeurs constituent une suite d'entiers strictement croissante, avec $p_0 = 1$. En effet, si $p_0 > 1$, certaines sommes pourraient ne pas être rendues.

Le problème consiste à trouver une liste d'entiers positifs $[x_0, x_1, \ldots, x_{n-1}]$ qui vérifie la contrainte $x_0p_0 + x_1p_1 + \cdots + x_{n-1}p_{n-1} = r$, en minimisant le nombre de pièces et billets rendus $x_0 + x_1 + \cdots + x_{n-1}$.

Exemple : on peut considérer le système suivant $S = (1, 2, 5, 10, 20, 50, 100, 200, 500)$, qui correspond notamment soit aux pièces (en centime d'euros), soit au pièces et billets (en euros).

Imaginons que la somme à rendre soit $r = 8$€. Il existe alors sept combinaisons possibles pour rendre cette somme avec des pièces de 1 € et 2 € et des billets de 5 €, les triplets sont : $[8, 0, 0]$; $[6, 1, 0]$; $[4, 2, 0]$; $[3, 0, 1]$; $[2, 3, 0]$; $[1, 1, 1]$ et $[0, 4, 0]$.
Le triplet optimal est donc $[1, 1, 1]$ avec 3 pièces.

Le programme suivant permet de retrouver tous ces triplets possibles :

```
S = (1, 2, 5)
for i in range(9):
    for j in range(5):
        for k in range(2):
            somme = i * S[0] + j * S[1] + k * S[2]
            if somme == 8:
                print([i, j, k])
```

Nous allons maintenant appliquer un algorithme glouton à cet exemple. A chaque étape, l'algorithme glouton choisira la pièce de plus forte valeur, ne dépassant pas la somme restant à rendre. La somme restant encore à rendre est alors mise à jour et on répète cette étape jusqu'à arriver à une somme à rendre nulle.

Programme utilisant l'algorithme glouton :

```
def glouton (S, r):
    n = len(S)
    i = n - 1
    solution = n * [0]
    while r > 0:
        while S[i] > r:
            i -= 1
        solution[i] += 1
        r -= S[i]
    return solution

S = (1, 2, 5)
print(glouton(S, 8))
```

Si on applique cette fonction *glouton* sur notre exemple, on obtient le triplet optimal $[1, 1, 1]$.

Remarque : on peut démontrer qu'avec le système de pièces et billets en vigueur dans la zone euro, l'algorithme glouton renvoie toujours la solution optimale au problème du rendu de monnaie.

En revanche, avec d'autres systèmes de pièces, il peut ne pas fournir la solution optimale. Par exemple, en 1967 en Angleterre on trouvait des pièces de 1 penny, 3 pence, 4 pence, 6 pence et 12 pence valant un shilling (« penny » est le singulier de « pence »).

Avec ce système, pour rendre 8 pence, l'algorithme glouton fournit la solution $[2, 0, 0, 1]$, soit deux pièces d'un penny et une pièce de 6 pence, tandis que la solution optimale serait $[0, 0, 2, 0]$, soit deux pièces de 4 pence.