Les algorithmes gloutons

February 25, 2024

1 Les Algorithmes Gloutons - Définition

Définition 1 :

L'algorithme glouton choisit la solution optimale qui se présente à lui à chaque instant, sans se préoccuper, ni du passé ni de l'avenir. Il répète cette même stratégie à chaque étape jusqu'à avoir entièrement résolu le problème.

Définition 2 :

Un algorithme glouton est un algorithme qui effectue à chaque instant, le meilleur choix possible sur le moment, sans retour en arrière ni l'anticipation des étapes suivantes, dans l'objectif d'atteindre au final un résultat optimal.

2 Problème de rendu de monnaie

Nous cherchons à concevoir une fonction permettant de déterminer la méthode optimale pour rendre une somme d'argent donnée à un client, en minimisant à la fois le nombre de pièces et de billets utilisés. Nous sommes certains qu'il existe plusieurs façons de le faire, en combinant les billets et les pièces disponibles. Cette fonction prendra en paramètre la somme à rendre et une liste des valeurs des billets et des pièces d'argent disponibles. Elle retournera une liste représentant la combinaison optimale de billets et de pièces d'argent qui équivaut à la somme à rendre, tout en minimisant le nombre total de pièces et de billets utilisés.

Par exemple, si la somme à rendre est de 537 et que la liste des pièces disponibles est [200, 100, 50, 20, 10, 5, 2, 1], la fonction retournera [200, 200, 100, 20, 10, 5, 2].

Nous devons utiliser un algorithme de glouton.

Voici comment cela fonctionnerait pour l'exemple donné :

- Étape 1 : Le client doit 537. À cette étape, l'algorithme glouton nous oblige à choisir la meilleur solution à cette étape. Nous sélectionnons la valeur la plus grande possible parmi les pièces disponibles sans dépasser 537, qui est 200;
- Étape 2 : Le client doit maintenant 337. Nous choisissons à nouveau la pièce de 200 car elle est la plus grande parmi celles disponibles sans dépasser 337;
- Étape 3 : Maintenant, le client doit 137. Nous utilisons la pièce de 100 car elle est la plus grande sans dépasser 137;
- Étape 4 : Maintenant, le client doit 37. La plus grande pièce disponible est de 20, donc nous l'utilisons;

PR. EL KHAWLANI Ahmed

- Étape 5 : Le client doit maintenant 17. Nous utilisons la pièce de 10 car c'est la plus grande pièce disponible sans dépasser 17;
- Étape 6 : Maintenant, le client doit 7. Nous utilisons la pièce de 5 car elle est la plus grande sans dépasser 7;
- Étape 7 : Enfin, le client doit 2. Nous utilisons la pièce de 2 car c'est la plus grande sans dépasser 2.

Cet algorithme glouton nous permet de rendre la monnaie avec le moins de billets et de pièces possibles, en utilisant à chaque étape le billet de plus grande valeur disponible sans dépasser le montant dû par le client.

2.1 Méthode 1

[25]:
```
def rendu_monnaie1(montant, pieces):  # pieces est une liste qui contient les␣
↪valeurs des feuilles et des pièces disponibles
 pieces.sort(reverse = True)          # Trier les valeurs des feuilles et␣
↪pièces dans l'ordre décroissant
 i = 0
 rendu = []                           # rendu est une liste qui contiendra les␣
↪valeurs des pièces et feuilles rendues
 while montant > 0:
   piece = pieces[i]
   if piece <= montant:
     rendu.append(piece)
     montant = montant - piece
   else:
     i = i + 1
 return rendu
```

[26]:
```
R = rendu_monnaie1(537, [200, 50, 100, 20, 10, 1, 2, 5])
print(R)
```

```
[200, 200, 100, 20, 10, 5, 2]
```

2.2 Méthode 2

[27]:
```
def rendu_monnaie2(montant, pieces):  # pieces est une liste qui contient les␣
↪valeurs des feuilles et des pièces disponibles
 pieces.sort(reverse = True)          # Trier les valeurs des feuilles et␣
↪pièces dans l'ordre décroissant
 rendu = []                           # rendu est une liste qui contiendra les␣
↪valeurs des pièces et feuilles rendues
 for i in range(len(pieces)):
   piece = pieces[i]
   while piece <= montant:
     rendu.append(piece)
     montant = montant - piece
```

PR. EL KHAWLANI Ahmed

```
    return rendu
```

[28]:
```
R = rendu_monnaie2(537, [200, 50, 100, 20, 10, 1, 2, 5])
print(R)
```

```
[200, 200, 100, 20, 10, 5, 2]
```

3 Problème du sac à dos

Étant donné plusieurs objets possédants chacun un poids et une valeur et étant donné un poids maximum pour le sac, quels objets faut-il mettre dans le sac de manière à maximiser la valeur totale sans dépasser le poids maximal autorisé pour le sac ?

Supposons que le poids max supporté par le sac est 40 Kg

Soient les objets suivants

Objet	Poids (Kg)	Valeur (Prix)
A	15	500
B	24	400
C	9	350
D	25	750
E	5	400
F	12	800
G	2	1400
H	18	550

:

L'application de l'algorithme glouton implique la prise systématique du choix optimal à chaque étape. L'objectif est de maximiser la valeur tout en respectant la contrainte de poids maximal. Pour ce faire, à chaque étape, nous optons pour l'ajout de l'objet ayant le rapport valeur/poids le plus élevé. Cela commence par la création d'une liste contenant ces rapports pour chaque objet. À chaque itération, l'objet avec le plus grand rapport valeur/poids est sélectionné et ajouté au sac.

Avec une capacité de sac à dos de 40, nous procédons comme suit :

1. Calcul des rapports valeur/poids pour chaque objet :

- A : 500/15 33.33 ;
- B : 400/24 16.67 ;
- C : 350/9 38.89 ;
- D : 750/25 = 30 ;

PR. EL KHAWLANI Ahmed

- E : 400/5 = 80 ;
- F : 800/12 66.67 ;
- G : 1400/2 = 700 ;
- H : 550/18 30.56 ;

2. Étapes de sélection des objets :

- Étape 1 : Capacité restante : 40. Meilleur rapport valeur/poids : G (700). Poids de G est 2 inférieur à 40 donc : Ajout de G au sac ;
- Étape 2 : Capacité restante : 38. Meilleur rapport valeur/poids : E (80). Poids de E est 5 inférieur à 38 donc : Ajout de E au sac ;
- Étape 3 : Capacité restante : 33. Meilleur rapport valeur/poids : F (66.76). Poids de F est 12 inférieur à 33 donc : Ajout de F au sac ;
- Étape 4 : Capacité restante : 21. Meilleur rapport valeur/poids : C (38.89). Poids de C est 9 inférieur à 21 donc : Ajout de C au sac ;
- Étape 5 : Capacité restante : 12. Meilleur rapport valeur/poids : A (33.33). Poids de A est 15 supérieur à 12 donc : On n'ajoute pas A au sac ;
- Étape 6 : Capacité restante : 12. Meilleur rapport valeur/poids : H (30.56). Poids de H est 18 supérieur à 12 donc : On n'ajoute pas H au sac ;
- Étape 7 : Capacité restante : 12. Meilleur rapport valeur/poids : D (30). Poids de D est 25 supérieur à 12 donc : On n'ajoute pas D au sac ;
- Étape 8 : Capacité restante : 12. Meilleur rapport valeur/poids : B (16.67). Poids de B est 24 supérieur à 12 donc : On n,ajoute pas B au sac.

Sur python, il existe beaucoup de manières pour implémenter nos objets avec leurs poids et valeurs.

3.1 Méthode 1

[29]:
```
# Méthode 1 d'implémentation :
Objets1 = ["A", "B", "C", "D", "E", "F", "G", "H"]  # Liste des noms des objets
Poids = [15, 24, 9, 25, 5, 12, 2, 18]               # Liste poids des objets
Valeur = [500, 400, 350, 750, 400, 800, 1400, 550]  # Liste des valeurs des␣
↪objets
```

[30]:
```
# Méthode 1
def remplir_sac1(Objets, Poids, Valeur, Poids_Max):
  Sac = []
  Poids_Sac = 0
  rapport = [Valeur[i] / Poids[i] for i in range(len(Poids))]   # Liste des␣
↪rapports valeur/poids
  while Objets != [] and Poids_Sac < Poids_Max:
    rapport_obj = max(rapport)                        # On choisit le maximum␣
↪rapport
    i = rapport.index(rapport_obj)                    # On cherche l'indice de ce␣
↪rapport pour connaître l'objet concerné
    obj = Objets.pop(i)                               # On supprime l'objet de la␣
↪liste des objets et on le stock dans la variable obj
    p = Poids.pop(i)                                  # On supprime le poids␣
↪correspondant de la liste des poids et on le stock dans la variable p
```

PR. EL KHAWLANI Ahmed

```
    val = Valeur.pop(i)                                  # On supprime la valeur␣
↪correspondante de la liste des valeurs et on le stock dans la variable val
    rap = rapport.pop(i)                                 # On supprime le rapport␣
↪correspondant de la liste des rapports et on le stock dans la variable rap
    if p <= Poids_Max - Poids_Sac:                       # Si le poids est inférieur␣
↪à la capacité restante du sac
        Sac.append(obj)                                  # Ajouter l'élément dans le␣
↪sac
        Poids_Sac = Poids_Sac + p                        # Actualiser le poids du sac
  return Sac
```

[31]:
```
# Test
Objets1 = ["A", "B", "C", "D", "E", "F", "G", "H"]  # Liste des noms des objets
Poids = [15, 24, 9, 25, 5, 12, 2, 18]               # Liste poids des objets
Valeur = [500, 400, 350, 750, 400, 800, 1400, 550]  # Liste des valeurs des␣
↪objets
SAC = remplir_sac1(Objets1, Poids, Valeur, 40)
print(SAC)
```

```
['G', 'E', 'F', 'C']
```

3.2 Méthode 2

[32]:
```
# Méthode 2 d'implémentation
# Chaque objet est représenté par une liste qui contient son nom, son poids et␣
↪sa valeur
Objets2 = [['A', 15, 500], ['B', 24, 400], ['C', 9, 350], ['D', 25, 750], ['E',␣
↪5, 400], ['F', 12, 800], ['G', 2, 1400], ['H', 18, 550]]
```

Méthode 2 - 1 - Sans trier les objets

[33]:
```
# Méthode 2 - 1
def remplir_sac21(Objets, Poids_Max):
  Sac = []
  Poids_Sac = 0
  rapport = [Objets[i][2] / Objets[i][1] for i in range(len(Objets))]   # Liste␣
↪des rapports valeur/poids
  while Objets != [] and Poids_Sac < Poids_Max:
    rapport_obj = max(rapport)                           # On choisit le maximum␣
↪rapport
    i = rapport.index(rapport_obj)                       # On cherche l'indice de ce␣
↪rapport pour connaître l'objet concerné
    obj = Objets.pop(i)                                  # On supprime l'objet de la␣
↪liste des objets et on le stock dans la variable obj (type liste)
    nobj = obj[0]                                        # Nom de l'objet
    p = obj[1]                                           # Poids de l'objet
    val = obj[2]                                         # Valeur de l'objet
```

PR. EL KHAWLANI Ahmed

```
    rap = rapport.pop(i)                                # On supprime le rapport␣
 ↪correspondant de la liste des rapports et on le stock dans la variable rap
    if p <= Poids_Max - Poids_Sac:                      # Si le poids est inférieur␣
 ↪à la capacité restante du sac
      Sac.append(nobj)                                  # Ajouter l'élément dans le␣
 ↪sac
      Poids_Sac = Poids_Sac + p                         # Actualiser le poids du sac
  return Sac
```

[34]:
```
# Test
Objets2 = [['A', 15, 500], ['B', 24, 400], ['C', 9, 350], ['D', 25, 750], ['E',␣
 ↪5, 400], ['F', 12, 800], ['G', 2, 1400], ['H', 18, 550]]
SAC = remplir_sac21(Objets2, 40)
print(SAC)
```

```
['G', 'E', 'F', 'C']
```

Méthode 2 - 2 - En triant les objets - Sans utiliser .sort()

[35]:
```
# Méthode 2 - 2

# On crée une fonction qui trie une liste des objets dans l'ordre décroissant␣
 ↪selon les rapports valeur/poids
# On modifiera le tri par selection classique
def tri_selection(L):
  for i in range(len(L) - 1):
    k = i
    for j in range(i + 1, len(L)):
      if L[j][2] / L[j][1] > L[k][2] / L[k][1]:    # Ligne modifiée
        k = j
    L[i], L[k] = L[k], L[i]

def remplir_sac22(Objets, Poids_Max):
  Sac = []
  Poids_Sac = 0
  tri_selection(Objets)                                 # Trier les objets dans␣
 ↪l'ordre décroissant selon les rapports valeur/poids
  i = 0
  while i < len(Objets) and Poids_Sac < Poids_Max:
    obj = Objets[i]
    if obj[1] <= Poids_Max - Poids_Sac:
      Sac.append(obj[0])
      Poids_Sac = Poids_Sac + obj[1]
    i = i + 1
  return Sac
```

PR. EL KHAWLANI Ahmed

[36]:
```
# Test
Objets2 = [['A', 15, 500], ['B', 24, 400], ['C', 9, 350], ['D', 25, 750], ['E',␣
 ↪5, 400], ['F', 12, 800], ['G', 2, 1400], ['H', 18, 550]]
SAC = remplir_sac22(Objets2, 40)
print(SAC)
```

```
['G', 'E', 'F', 'C']
```

Méthode 2 - 3 - En triant les objets - En utilisant .sort()

[37]:
```
# Méthode 2 - 3
def rapport(Objet):
  return Objet[2] / Objet[1]

def remplir_sac23(Objets, Poids_Max):
  Sac = []
  Poids_Sac = 0
  Objets.sort(reverse = True, key = rapport)      # Trier les objets dans␣
 ↪l'ordre décroissant selon les rapports valeur/poids
  i = 0
  while i < len(Objets) and Poids_Sac < Poids_Max:
    obj = Objets[i]
    if obj[1] <= Poids_Max - Poids_Sac:
      Sac.append(obj[0])
      Poids_Sac = Poids_Sac + obj[1]
    i = i + 1
  return Sac
```

[38]:
```
# Test
Objets2 = [['A', 15, 500], ['B', 24, 400], ['C', 9, 350], ['D', 25, 750], ['E',␣
 ↪5, 400], ['F', 12, 800], ['G', 2, 1400], ['H', 18, 550]]
SAC = remplir_sac23(Objets2, 40)
print(SAC)
```

```
['G', 'E', 'F', 'C']
```

3.3 Méthode 3

[39]:
```
# Méthode 3 d'implémentation :
# Chaque objet est représenté par un dictionnaire qui contient son nom, son␣
 ↪poids et sa valeur
Objets3 = [{'Objet': 'A', 'Poids': 15, 'Valeur': 500},
 {'Objet': 'B', 'Poids': 24, 'Valeur': 400},
 {'Objet': 'C', 'Poids': 9, 'Valeur': 350},
 {'Objet': 'D', 'Poids': 25, 'Valeur': 750},
 {'Objet': 'E', 'Poids': 5, 'Valeur': 400},
 {'Objet': 'F', 'Poids': 12, 'Valeur': 800},
 {'Objet': 'G', 'Poids': 2, 'Valeur': 1400},
```

PR EL KHAWLANI Ahmed

```
 {'Objet': 'H', 'Poids': 18, 'Valeur': 550}]
```

Méthode 3 - 1 - Sans trier les objets

[40]:
```
# Méthode 3 - 1
def remplir_sac31(Objets, Poids_Max):
  Sac = []
  Poids_Sac = 0
  rapport = [Objets[i]['Valeur'] / Objets[i]['Poids'] for i in␣
 ↪range(len(Objets))]   # Liste des rapports valeur/poids
  while Objets != [] and Poids_Sac < Poids_Max:
    rapport_obj = max(rapport)                          # On choisit le maximum␣
 ↪rapport
    i = rapport.index(rapport_obj)                      # On cherche l'indice de ce␣
 ↪rapport pour connaître l'objet concerné
    obj = Objets.pop(i)                                 # On supprime l'objet de la␣
 ↪liste des objets et on le stock dans la variable obj (type liste)
    nobj = obj['Objet']                                 # Nom de l'objet
    p = obj['Poids']                                    # Poids de l'objet
    val = obj['Valeur']                                 # Valeur de l'objet
    rap = rapport.pop(i)                                # On supprime le rapport␣
 ↪correspondant de la liste des rapports et on le stock dans la variable rap
    if p <= Poids_Max - Poids_Sac:                      # Si le poids est inférieur␣
 ↪à la capacité restante du sac
      Sac.append(nobj)                                  # Ajouter l'élément dans le␣
 ↪sac
      Poids_Sac = Poids_Sac + p                         # Actualiser le poids du sac
  return Sac
```

[41]:
```
# Test
Objets3 = [{'Objet': 'A', 'Poids': 15, 'Valeur': 500},
 {'Objet': 'B', 'Poids': 24, 'Valeur': 400},
 {'Objet': 'C', 'Poids': 9, 'Valeur': 350},
 {'Objet': 'D', 'Poids': 25, 'Valeur': 750},
 {'Objet': 'E', 'Poids': 5, 'Valeur': 400},
 {'Objet': 'F', 'Poids': 12, 'Valeur': 800},
 {'Objet': 'G', 'Poids': 2, 'Valeur': 1400},
 {'Objet': 'H', 'Poids': 18, 'Valeur': 550}]
SAC = remplir_sac31(Objets3, 40)
print(SAC)
```

```
['G', 'E', 'F', 'C']
```

Méthode 3 - 2 - En triant les objets - Sans utiliser .sort()

[42]:
```
# Méthode 3 - 2
```

PR. EL KHAWLANI Ahmed

```
# On crée une fonction qui trie une liste des objets dans l'ordre décroissant␣
 ↪selon les rapports valeur/poids
# On modifiera le tri par selection classique
def tri_selection(L):
  for i in range(len(L) - 1):
    k = i
    for j in range(i + 1, len(L)):
      if L[j]['Valeur'] / L[j]['Poids'] > L[k]['Valeur'] / L[k]['Poids']:    #␣
 ↪Ligne modifiée
        k = j
    L[i], L[k] = L[k], L[i]

def remplir_sac32(Objets, Poids_Max):
  Sac = []
  Poids_Sac = 0
  tri_selection(Objets)                    # Trier les objets dans l'ordre␣
 ↪décroissant selon les rapports valeur/poids
  i = 0
  while i < len(Objets) and Poids_Sac < Poids_Max:
    obj = Objets[i]
    if obj['Poids'] <= Poids_Max - Poids_Sac:
      Sac.append(obj['Objet'])
      Poids_Sac = Poids_Sac + obj['Poids']
    i = i + 1
  return Sac
```

[43]:
```
# Test
Objets3 = [{'Objet': 'A', 'Poids': 15, 'Valeur': 500},
 {'Objet': 'B', 'Poids': 24, 'Valeur': 400},
 {'Objet': 'C', 'Poids': 9, 'Valeur': 350},
 {'Objet': 'D', 'Poids': 25, 'Valeur': 750},
 {'Objet': 'E', 'Poids': 5, 'Valeur': 400},
 {'Objet': 'F', 'Poids': 12, 'Valeur': 800},
 {'Objet': 'G', 'Poids': 2, 'Valeur': 1400},
 {'Objet': 'H', 'Poids': 18, 'Valeur': 550}]
SAC = remplir_sac32(Objets3, 40)
print(SAC)
```

```
['G', 'E', 'F', 'C']
```

Méthode 3 - 3 - En triant les objets - En utilisant .sort()

[44]:
```
# Méthode 3 - 3
def rapport(Objet):
  return Objet['Valeur'] / Objet['Poids']

def remplir_sac33(Objets, Poids_Max):
  Sac = []
```

PR. EL KHAWLANI Ahmed

```
  Poids_Sac = 0
  Objets.sort(reverse = True, key = rapport)    # Trier les objets dans l'ordre␣
 ↪décroissant selon les rapports valeur/poids
  i = 0
  while i < len(Objets) and Poids_Sac < Poids_Max:
    obj = Objets[i]
    if obj['Poids'] <= Poids_Max - Poids_Sac:
      Sac.append(obj['Objet'])
      Poids_Sac = Poids_Sac + obj['Poids']
    i = i + 1
  return Sac
```

[45]:
```
# Test
Objets3 = [{'Objet': 'A', 'Poids': 15, 'Valeur': 500},
 {'Objet': 'B', 'Poids': 24, 'Valeur': 400},
 {'Objet': 'C', 'Poids': 9, 'Valeur': 350},
 {'Objet': 'D', 'Poids': 25, 'Valeur': 750},
 {'Objet': 'E', 'Poids': 5, 'Valeur': 400},
 {'Objet': 'F', 'Poids': 12, 'Valeur': 800},
 {'Objet': 'G', 'Poids': 2, 'Valeur': 1400},
 {'Objet': 'H', 'Poids': 18, 'Valeur': 550}]
SAC = remplir_sac33(Objets3, 40)
print(SAC)
```

```
['G', 'E', 'F', 'C']
```

4 Problème d'organisation

Des professeurs sont invités à présenter leurs cours dans une salle. Mais leurs disponibilités ne leur permettent d'intervenir qu'à des horaires bien définis. Le problème est de construire un planning d'occupation de la salle avec le plus grand nombre de professeurs.

Pour résoudre ce problème, nous avons une liste de cours où chaque cours est représenté par une liste contenant le nom du professeur et les heures de début et de fin du cours. Voici la liste des cours :

Cours = [[“P1”, 9, 10], [“P2”, 8, 13], [“P3”, 9, 11], [“P4”, 10, 11], [“P5”, 11, 13], [“P6”, 10, 12]]

Chaque élément de la liste “Cours” est une liste contenant trois éléments :

- Le premier élément est le nom du professeur ;
- Le deuxième élément est l'heure de début du cours ;
- Le troisième élément est l'heure de fin du cours.

Maintenant, nous devons créer un planning pour occuper la salle avec le plus grand nombre possible de professeurs tout en respectant les contraintes d'horaires (Deux cours de deux différents professeurs ne doivent pas chevaucher).

Dans ce cas, nous pouvons adopter une approche gloutonne où nous choisissons d'abord le professeur qui libérera la salle le plus tôt possible. Voici comment nous pouvons procéder :

PR EL KHAWLANI Ahmed

1. Trier la liste des cours selon l'heure de fin croissante ;
2. Sélectionner le premier cours dans la liste triée, car il libère la salle le plus tôt possible ;
3. Si un cours se termine à la même heure que le précédent, nous pouvons choisir de manière arbitraire, tant qu'il n'empêche pas la sélection d'un cours ultérieur ;
4. Répéter les étapes 2 et 3 jusqu'à ce que tous les cours soient planifiés ou que la salle soit occupée pendant toute la journée.

[46]:
```
# Résolution
def hf(P):
  return P[2]  # Fonction qui retourne l'heure de fin d'un cours

def organisation(cours):
  cours.sort(key = hf)        # Trier la liste cours selon l'ordre croissant des␣
 ↪heures de fin de cours
  salle = [cours[0]]          # Ajouter le premier cours à la salle, c'est le␣
 ↪premier cours qui libérera la salle le plus tôt possible
  for c in cours[1 :]:             # Parcourir les cours selon l'ordre croissant␣
 ↪des heures de fin
    if c[1] >= hf(salle[-1]):      # Si le cours de chevauche pas avec le␣
 ↪dernier cours ajouté à la salle
      salle.append(c)              # Ajouter le cours à la salle
  return salle
```

[47]:
```
# test
Cours = [["P1", 9, 10], ["P2", 8, 13], ["P3", 9, 11], ["P4", 10, 11], ["P5",␣
 ↪11, 13], ["P6", 10, 12]]
Salle = organisation(Cours)
print(Salle)
```

```
[['P1', 9, 10], ['P4', 10, 11], ['P5', 11, 13]]
```

PR. EL KHAWLANI Ahmed