Algos_gloutons

December 13, 2022

#

NUMERIQUE ET SCIENCES INFORMATIQUES 1ère NSI

0.1 Séquence N° 6 : Les algorithmes gloutons

Les objectifs de cette séquence sont :

- Définir et résoudre un problème grâce à un algorithme glouton ;
- Écrire un algorithme de tri ;
- Décrire un invariant de boucle qui prouve la correction des tris par insertion et par sélection ;
- Comparer le coût de différents algorithmes de tri.

Problématique 1 :

Nous disposons d'un distributeur automatique de sandwichs. L'appareil doit rendre la monnaie à l'usager.

Si vous lisez ce document, vous devez avoir un répertoire nommé "rendu_monnaie. Celui-ci qui contient les fichiers html, css et javascript (non finalisé), bien rangés dans des répertoires correspondants.

Travail demandé :

- Analysez le fonctionnement et le code fourni;
- Tracez l'algorithme de la fonction "renduMonnaie" pour que l'appareil rendre la monnaie avec un **nombre minimal de pièces** (voir méthode ci-dessous). Celle-ci :
  - Utilise les variables globales ci-dessous ;
  - Calcule le montant à rendre ;
  - Détermine le nombre et la valeur des pièces rendues ;
  - Met à jour la page HTML ;
  - Désactive les boutons "Valider" et "Commander", ainsi que l'action sur les pièces.
- Complétez le script "script_distributeur.js" avec votre fonction. Vous pouvez tester le coeur de votre fonction dans une cellule ci-dessous.

En complément de la formation sur le langage javascript, voici un lien utile : https://www.w3schools.com/js/

Nous avons à résoudre un problème d'**optimisation**.

1. **Solution naïve**

La solution a laquelle on pense immédiatement est d'énumérer toutes les combinaisons de possibles, de sélectionner celles qui impliquent un minimum de pièces et de choisir la meilleure.

Cette solution, dite de **force brute**, fonctionnera toujours mais est très loin d'être efficace. En effet, si elle est simple dans certains cas, elle implique en général un nombre très important de combinaisons différentes, ce qui nuit grandement à l'efficacité de notre solution.

2. **Solution gloutonne**

La méthode gloutonne vise donc à optimiser la résolution d'un problème en partant du principe suivant : des choix locaux optimaux, étape après étape, devraient produire un résultat global optimal.

Les algorithmes gloutons présentent l'avantage d'une conception relativement aisée à mettre en oeuvre. Cependant, le prix à payer est qu'ils ne fourniront **pas toujours la solution optimale** au problème donné : on parle alors d'**heuristique gloutonne**.

Dans le **système de pièces européen** (en centimes : 1, 2, 5, 10, 20, 50, 100, 200), on peut montrer que l'algorithme glouton donne toujours une **solution optimale**. Pour i allant de 1 à n a<-montantRecu-montantAttendu Tant que a >= coins[i] 1) a <- a - coins[i] 2) chosen[i] <- chosen[i] + 1 Retourner le tableau chosen et la variablemontantRendu ==

[ ]:
```
tant que montantrendu>0
    si montant rendu >= piece[i]
        alors montantrendu=montantrendu-piece[i]
            mettre pièce dans liste piècerendues
        sinon incrémenter index
    fin si
fin tant que


let pieces=[200,100,50,20,10,5,2,1]; // liste des pièces acceptées par la␣
 ↪machine en centimes
let piecesRendues=[]; // liste des pièces rendues à l'utilisateur
let montantRecu=400; // montant reçu de l'utilisateur
let montantAttendu=150; // prix du sandwich
let montantRendu=0; // montant à rendre
lett index=0
montantrendu=montantrecu-montantattendu
while(montantrendu>0){
    if (montantrendu>=pieces[i]){
        montantrendu-=piceces[i]
        piecesrendues.push((pieces[i]/100)+"€")
    }
    else{
        indexex+=150
    }
}
```

[18]:
```
%%javascript //Déclare le type de script du notebook
let pieces=[200,100,50,20,10,5,2,1]; // liste des pièces acceptées par la␣
 ↪machine en centimes
let piecesRendues=[]; // liste des pièces rendues à l'utilisateur
let montantRecu=400; // montant reçu de l'utilisateur
let montantAttendu=150; // prix du sandwich
let montantRendu=0; // montant à rendre
function moneyback(){
    let x=montantRecu-montantAttendu;
    let montantrendu=x;
    let n=pieces.length;
    let chosen = [];
    let i=0;
    while(x>0){
        if(pieces[i] > x){
            i = i+1;
        }
        else{
            chosen.push(pieces[i]);
            x -= pieces[i];
        }
    }
    return[chosen, montantrendu]
    }
let a,b=moneyback()
console.log(a,b)
```

```
<IPython.core.display.Javascript object>
```

soit morceaux = [200,100,50,20,10,5,2,1]; // liste des pièces acceptées par la machine en centimes laissez piecesRendues = []; // liste des pièces rendues à l'utilisateur laissez montantRecu = 400; // montant reçu de l'utilisateur laissez montantAttendu = 150; // prix du sandwich laissez montantRendu = 0; // montant à rendre function moneyback () let x = montantRecu-montantAttendu; laissez montantrendu = x; soit n = morceaux.longueur; laissez choisi = []; soit i = 0; tandis que (x> 0) if (morceaux [i]> x) i = i + 1; autre selected.push (morceaux [i]); x - = pièces [i]; retour [choisi, montantrendu] soit a, b = moneyback () console.log (a, b) **Problématique 2** :

Envoyer des objets dans l'espace est une action très coûteuse. Bien que ces objets échapperont à la pesanteur lorsqu'ils seront dans l'espace, il est nécessaire de les y envoyer en minimisant le coût de l'opération.

Vous travaillez pour l'entreprise chargée du prochain ravitaillement de la Station Spatiale Internationale, votre rôle va être de préparer le contenu du prochain lanceur. Votre équipe a réuni "n" objets (cela peut être de la nourriture, du carburant, du nouveau matériel, des expériences scientifiques à mener, etc.), et possède à disposition un espace dans le lanceur pouvant contenir au maximum M kilos. Chaque objet sera décrit selon deux caractéristiques :

- Sa masse m en kilo (avec 0<m<=M)
- Sa valeur v pour la mission (avec 0<v<100)

Afin d'optimiser au mieux les coûts du lancement, vous devez trouver un arrangement d'objets tel que la valeur cumulée de ces derniers (notée V) soit maximale, tout en faisant attention à ne pas dépasser les M kilos supportés par votre espace dans le lanceur.

Toutes les données fournies en entrée (n, M, m, v) sont des nombres entiers.

Le problème est complexe, aussi pour commencer, on vous demande de réfléchir à la solution sur un exemple plus simple.

Objet	Masse	Valeur
objet1	140	65
objet2	60	25
objet3	65	25
objet4	20	30

Avec n=4 et M=200 kg.

Proposez des stratégies pour résoudre le problème d'optimisation (la solution de force brute sera ignorée).

Quelque soit la solution retenue, il y a nécessité de réaliser un tri. les algorithmes de tri

1- Première stratégie On choisit les objets dont la valeur est la plus grande.

[3]:
```
# -*- coding: utf-8 -*-
"""
Programme d'aide au chargement du lanceur
"""
def affiche_liste(liste:list,masse_max:int):
    valeur=0
    masse=0
    objet=""
    for i in range (len(liste)):
        valeur+=liste[i][2]
        masse+=liste[i][1]
        objet+=liste[i][0]+' '
    print('Bilan : ')
    print(f'masse maximale autorisée : {masse_max} kg')
    print(f'objets embarqués : {objet}')
    print(f'masse embarquée : {masse} kg')
    print(f'valeur embarquée : {valeur}')

def choisir(masse_maxi:int, objets:list)->list:
    masse_container = 0
    container = []
    index_valeur=2
    index_masse=1

    for i in range(len(objets)):
```

```
        min=i
        for j in range(i+1,len(objets)):
            if objets[min][index_valeur]<objets[j][index_valeur]:
                min=j
        tmp=objets[i]
        objets[i]=objets[min]
        objets[min]=tmp

    for objet in range(len(objets)):
         if masse_container + objets[objet][index_masse] <= masse_maxi:
            masse_container += objets[objet][index_masse]
            container.append(objets[objet])
    return container

masse_maxi=180
## liste des objets possibles à embarquer
mes_objets=[['Objet1',140,25],
            ['Objet2',60,25],
            ['Objet3',65,20],
            ['Objet4',20,15]]

mon_container=(choisir(masse_maxi,mes_objets))
affiche_liste(mon_container,masse_maxi)
```

```
Bilan :
masse maximale autorisée : 180 kg
objets embarqués : Objet1 Objet4
masse embarquée : 160 kg
valeur embarquée : 40
```

1. Décrire la postcondition sur les résultats.
   La valeur retournée par la fonction choisir() est une liste.
2. Décrire la précondition sur les arguments.
   Les valeurs réclamées la fonction affiche_liste() sont : une liste, et un entier.
   Les valeurs réclamées la fonction choisir() sont : deux entiers.
3. Une méthode de tri est développée. De laquelle s'agit-il ?
   tri par séléction
4. Justifier que la complexité temporelle de cet algorithme est quadratique.
   on note la présence de 2 boucles imbriquées : donc O(n^2) -> Quadratique
5. La valeur embarquée est-elle optimale ?
   La valeur est optimale : Obj1 = 60; Obj2 = 35 –> 95

2- Deuxième stratégie On choisit les objets dont le rapport valeur/masse est le plus grand

[4]:
```
#%%time
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
```

```
Programme d'aide au chargement du lanceur
"""

def affiche_bilan(dico:dict,masse_max:int):
    valeur=0
    masse=0
    objet=""
    for cle, data in dico.items():
        valeur+=data[1]
        masse+=data[0]
        objet+=cle+' '
    print('Bilan : ')
    print(f'masse maximale autorisée : {masse_max} kg')
    print(f'objets embarqués : {objet}')
    print(f'masse embarquée : {masse} kg')
    print(f'valeur embarquée : {valeur}')

def calcul_ratio(mes_objets:list)->list:
    #Calcul du ratio Valeur/Poids et l'ajoute aux listes des objets
    for i in range(len(mes_objets)):
        mes_objets[i].append(mes_objets[i][2]/mes_objets[i][1])
    return mes_objets

def selection_objets(mes_objets:list,masse_maxi:int)->dict:
    #Boucle de sélection des objets
    i=0
    nb_objets=len(mes_objets)
    mon_container={}
    while nb_objets>0:
        critere=mes_objets[i][1]
        if critere <= masse_maxi:
            mon_container[mes_objets[i][0]]=(mes_objets[i][1],mes_objets[i][2])
            masse_maxi-=critere
        i+=1
        nb_objets-=1
    return mon_container

masse_maxi=200
## liste des objets possibles à embarquer
mes_objets=[['Objet1',140,25],
            ['Objet2',60,5],
            ['Objet3',65,15],
            ['Objet4',20,10]]


calcul_ratio(mes_objets)
#Tri de ratio dans l'ordre croissant (ratio valeur/masse)
```

```
mes_objets.sort(key=lambda L:L[3], reverse=True)
container=selection_objets(mes_objets,masse_maxi)
affiche_bilan(container,masse_maxi)
```

```
Bilan :
masse maximale autorisée : 200 kg
objets embarqués : Objet4 Objet3 Objet2
masse embarquée : 145 kg
valeur embarquée : 30
```

1. Dans la boucle de sélection des objets, identifier en justifiant le variant de boucle. masse_maxi-=critere
2. Dans la boucle de sélection des objets, quel est l'invariant de boucle ? nb_objet car décrémentation
3. La valeur embarquée est-elle optimale ? Oui

On se propose maintenant de réexécuter les programmes précédents avec des valeurs différentes.

1- Première stratégie valeurs : 25,25,15,10 avec une masse_maxi = 180 Optimal

2- Deuxième stratégie valeurs : 25,5,15,10 avec une masse_maxi = 200 Pas opti

[ ]: