1. Problème du rendu de pièces

Lorsque vous passez à la caisse d'un magasin quelconque, il n'est pas rare que le caissier doive vous rendre de l'argent car le montant que vous lui avez donné est supérieur à celui que vous devez payer.

- Définissez sous une forme de liste croissante les différentes valeurs des pièces en centimes du système monétaire européen

```
# système monetaire
systeme_monetaire  =
```

- On appellera la fonction « rendu_glouton » qui prendra les attributs suivants :

```
def rendu_glouton(somme_a_rendre, sys_monetaire ):
```

- Cette fonction retourne une liste « rendu_liste » des valeurs des pièces rendues avec une logique gloutonne :

Tant que somme_a_rendre est supérieure à 0
    Si somme_a_rendre < valeur de la plus grande pièce
        Passer à la pièce de valeur immédiatement inférieure
    Sinon :
        Enlever à somme_a_rendre la valeur de la pièce
        Ajouter pièce à rendu_liste

- Implémenter cette fonction puis tester la pour les rendus de 58,57,55 centimes.

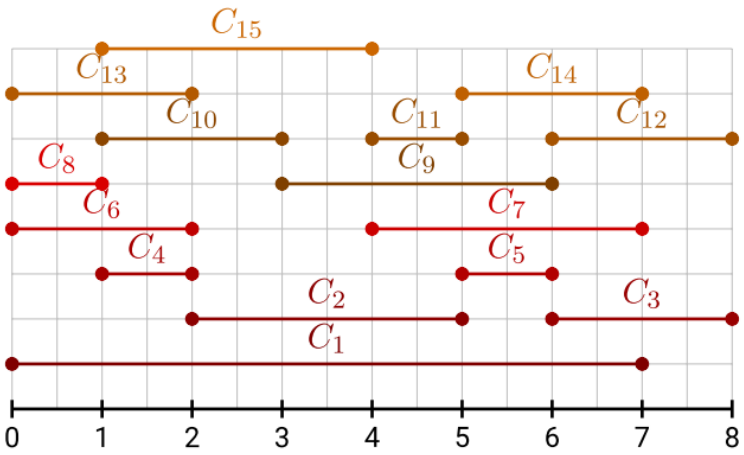
2. Problème d'optimisation d'occupation de temps

Une entreprise de logistique utilise des camions. Ceux-ci effectuent des voyages d'une certaine durée qui peuvent être assimilés à un intervalle de temps d'immobilisation : T[t_debut, t_fin [

Les demandes des clients sont transcrites en ce même type d'intervalle C1[t1, t2[etc …

L'objectif de l'entreprise étant de maximiser le nombre de clients possibles en sélectionnant les intervalles compatibles.

Voici un exemple de cas :



Une technique d'optimisation pour un camion consiste à :

- Classer les intervalles par heures de fin croissantes.
- Choisir le client associé au premier intervalle.
- Choisir parmi les intervalles suivants celui du client dont l'intervalles est compatible avec celui du premier client.
- Recommencer ainsi avec les intervalles classés suivants jusqu'à ce qu'il n'y en ait plus à traiter.

Le classement par heure de fin croissante donne :

C8 ≼ C4 ≼ C6 ≼ C13 ≼ C10 ≼ C15 ≼ C2 ≼ C11 ≼ C5 ≼ C9 ≼ C1 ≼ C7 ≼ C14 ≼ C3 ≼ C12

Puis on construit le planning d'un intervalle compatible à l'autre

On commence par C8 C8

Après C8 C4 et C10 sont compatibles. Mais C10 se termine plus tard que C4. Comme on classe les intervalles par fin croissantes on prendra C4 (c'est le glouton) C8 -> C4

Après C4 le seul intervalle compatible est C2 C8 -> C4 -> C2

Après C2 C5 et C14 sont compatibles mais là aussi on choisit C5 C8 -> C4 -> C2 -> C5

Après C5 C3 et C12 sont compatibles les deux se terminant à la même heure ils pourraient choisit également. Néanmoins comme les clients son examinés dans l'ordre de leur appellation c'est C3 qui sera sélectionné C8 -> C4 -> C2 -> C5- C3

2.1. Fonction "generer_intervalles"

Cahier des charges :

- Cette fonction prend en paramètre un nombre entier "n"
- Cette fonction retourne une liste d'intervalles : [temps_debut, temps_fin] de taille "n"
- L'intervalle total des temps est de 10 unités
- Chaque intervalle possède un temps de début compris entre 0 et 9 unités de temps
- Chaque intervalle possède une durée aléatoire au moins égale à 1 unité mais le temps de fin ne doit pas excéder 10

Ex : L = generer_intervalles(5) donne : [[4, 9], [5, 7], [9, 10], [5, 8], [0, 9]]

2.2. Fonction "nommer"

Cahier des charges :

- Cette fonction ajoute un nouvel élément à chaque intervalle de la liste précédente
- Cet élément est une chaine de caractère pour nommer l'intervalle du type :

Ex nommer(L) donne : [[4, 9, 'C_0'], [5, 7, 'C_1'], [9, 10, 'C_2'], [5, 8, 'C_3'], [0, 9, 'C_4']]

2.3. Fonction " tri_par_selection"

Cette fonction tri les intervalles dans le sens des temps de fin croissnt.

- Cette fonction est la même que celle du cours mais adaptée au type d'intervalle défini ci-dessus.
- Le tri est effectué dans le sens des "temps_fin" des intervalles croissant

Ex tri_par_sélection(L) donne : [[5, 7, 'C_1'], [5, 8, 'C_3'], [4, 9, 'C_0'], [0, 9, 'C_4'], [9, 10, 'C_2']]

2.4. Fonction " organiser"

Cette fonction retourne une liste des intervalles qui constitue la "meilleure" des successions des intervalles possibles pour organiser la location du camion d'un client à l'autre. On suppose que le camion a fini sa course pour un client il commence celle pour le client suivant.

C'est-à-dire que le temps_fin de l'intervalle précédent doit permettre de prendre un intervalle suivant avec le temps_debut possible et pas forcément identique.

Les intervalles étant triés par ordre de fin croissante le premier trouvé convient donc après il faut passer au suivant. C'est le glouton on prend l'intervalle le plus petit qui convienne après l'intervalle précédent.

Ex organiser(L) donne : [[5, 7, 'C_1'], [9, 10, 'C_2']]

2.5. Mise au point

Coder chacune de ces fonctions en faisant des tests unitaires pour les valider avant de passer à la suite. Commencer avec une liste d'intervalle de taille raisonnable 5 par exemple

Test final

Tester avec : test = [[0, 2, 'C_14'], [2, 3, 'C_7'], [0, 4, 'C_10'], [0, 4, 'C_12'], [4, 5, 'C_11'], \
[6, 7, 'C_1'], [2, 7, 'C_3'], [8, 9, 'C_9'], [4, 9, 'C_4'],[2, 9, 'C_13'],\
[1, 9, 'C_15'], [1, 9, 'C_16'],[1, 9, 'C_18'], [9, 10, 'C_5'], [9, 10, 'C_0'],\
[7, 10, 'C_2'], [1, 10, 'C_8'], [9, 10, 'C_17'], [9, 10, 'C_6'], [9, 10, 'C_19']]

Qui donne :

print(organiser(test))
[[0, 2, 'C_14'], [2, 3, 'C_7'], [4, 5, 'C_11'], [6, 7, 'C_1'], [8, 9, 'C_9'], [9, 10, 'C_5']]

Vérifier que les intervalles ne se chevauchent pas.

Vérifier que l'on progresse bien dans le temps

Vérifier que tous ces intervalles sont bien inscrits dans l'intervalle de temps total.

Que pouvez vous dire du planning trouvé ?

- Tous les trajets sont-ils bien pris en compte ?
- La solution est-elle optimale au sens du plus grands nombre de trajets à effectuer est pris pour ce planning ?