Algorithme Glouton :

Il s'agit d'une stratégie pour résoudre un problème d'optimisation. Un algorithme glouton est un algorithme qui construit une solution à un problème d'optimisation en :

- ⋆ parcourant des éléments du problème sans revenir en arrière
- ⋆ en se basant sur des considérations et de choix locaux.

Le rendue de Monnaie :

On dispose de pièces de monnaie dont les montants sont des nombres entiers de l'unité de monnaie : $t_0 > t_1 > t_2 > ... > t_p$. Pour payer une somme S (entier naturel quelconque), on aimerait utiliser le nombre minimal de pièces possibles. Pour cela on commence par rendre la pièce ayant le plus grand montant, et on recommence jusqu'à ce qu'on doivent passer à la pièce avec le montant inférieur suivant. **etc...**

Exercice 1. On souhaite rendre la monnaie sur 50€ pour un montant de 14€ avec des pièces ou billets de 20€, 10€, 5€, 2€ et 1€.

1. Donner la liste de la monnaie rendue selon le procédé détaillé plus haut.
2. On ne dispose maintenant que de pièces/billets dont les montants sont 5€, 2€ et 1€. Préciser la monnaie rendue.
3. On ne dispose maintenant que de pièces/billets dont les montants sont 20€, 10€, 5€, 1€. Donner la liste des pièces rendues et comparer à la situation où on ne dispose maintenant que de pièces dont les montants sont 20€, 10€, 2€.
4. Que dire si on ne dispose que de pièces de 20€, 10€, 5€ ?
5. On souhaite créer un script **Python** pour répondre aux questions précédentes.
   (a) Compléter le script suivant pour qu'il réponde à la question 1.

```
somme=50-14
ListeMontants=[...,...,...,...,...]
ListeNbPieces=[0,0,0,0,0]
for k in range(...) :
    ListeNbPieces[k]=somme...ListeMontants[k]
    somme=...
print(ListeNbPieces)
```

   Rappel : // donne le quotient de la division euclidienne de deux entiers et % donne le reste.

   (b) Changer le script pour qui réponde à la question 2.

   (c) On considère maintenant le script suivant :

```
def money(somme,ListeMontants) :
    ListeNbPieces=[0 for x in ListeMontants]
    for k in range(len(ListeMontants)) :
        ListeNbPieces[k]=somme//ListeMontants[k]
    somme=somme % ListeMontants[k]
return somme,ListeNbPiece
```

   Tester le avec le différentes situations évoquées en 1.,2.,3.,4.

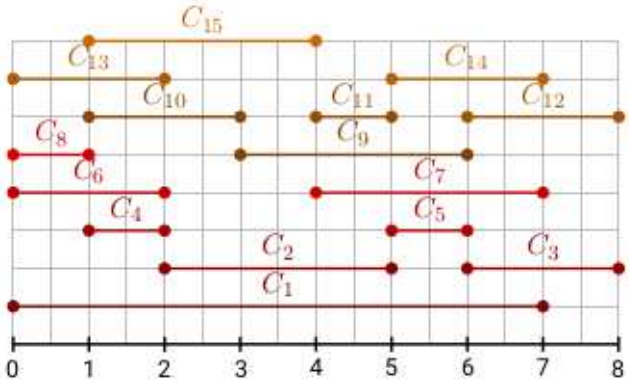
6. On peut se rendre compte que l'algorithme du rendu de monnaie proposé plus haut assure un nombre de pièce minimal avec une liste de montants T=[20,10,5,2,1]. Mais il n'en est pas toujours ainsi. En effet, si on considère la liste de montant $T = [18, 7, 1]$.
   - ⋆ Donner le détail de la monnaie rendue sur une somme de 21€.
   - ⋆ Est ce la solution optimale ?

Allocation de plages horaires :

Dans le problème d'allocation de plages horaires à des conférenciers, il est possible d'avoir recours à un algorithme glouton. Supposons que nous ayons a choisir parmi une liste de conférenciers dont ont donnerait les pages horaires sur lesquelles il peuvent intervenir auprès d'un même public. On adopte la stratégie suivante :

- ⋆ Classer les plages horaires de conférences par heures de fin croissantes.
- ⋆ Choisir le conférencier associé à la première plage horaire.
- ⋆ Choisir parmi les plages horaires suivantes celle du conférencier dont la plage horaire est compatible avec celle du premier conférencier.
- ⋆ Recommencer ainsi avec les plages horaires suivantes classées jusqu'à ce qu'il n'y en ait plus à traiter

Exercice 2. Considérons la situation illustrée par la figure suivante.



1. Classer les conférenciers par heures de fin croissantes
2. Puis en adoptant la stratégie décrite plus haut, proposer un planning.

Pour le codage, de plages horaires de $n \in \mathbb{N}^*$ conférenciers, nous choisissons de créer une liste de triplets $[d_i, f_i,' C_i']$ pour $i \in [[1; n]]$ avec d_i horaire de début et f_i horaire de fin de la conférence de C_i

1. Créer la variable *tableau_horaires* correspondant la situation de la figure F_1. Puis trier le suivant les horaires de fin des conférences croissantes avec la commande $sorted(tableau\_horaires, key = lambda fin : fin[1])$ Le script suivant permet de construire le tableau du planning des conférences une fois qu'il est trié :

```
def planning(tab_horaires):
    nb_intervalles=len(tab_horaires)
    tab_horaires(sorted(tab_horaires,key=lambda fin: fin[1]))
    tab_planning=[tab_horaires[0]]
            j=0
            for i in range(1,nb_intervalles):
        if tab_horaires[i][0]>=tab_horaires[j][1]:
            tab_planning.append(tab_horaires[i])
                    j=i
    return tab_planning
```

2. Utiliser le script précédent avec le tableau des horaires de la situation 1.
3. On souhaite maintenant tester cette stratégie de planning avec plusieurs situations. Pour cela on utilise le script suivant qui permet de créer une situation à partir d'un nombre de conférenciers choisi et une proposition aléatoire d'horaires.

```
import numpy as np
def tableau_horaires(nb_horaires):
    debut = 8
    fin = 17
    duree_max = 3
    tab_horaires = []
```

```
    for k in range(nb_horaires):
        duree = np.random.randint(1,duree_max)
        deb = np.random.randint(debut,fin-duree)
        tab_horaires.append([deb,deb+duree])
    return tab_intervalles
```

Commenter le script précédent en précisant les contraintes fixées pour l'établissement de la liste des conférences.

4. Tester les deux scripts précédents conjointement en faisant varier le nombre de conférences.