PROGRAMMATION DYNAMIQUE 2/3

RENDU DE MONNAIE

Le problème du rendu de monnaie est un problème classique d'algorithmique. Il s'énonce de la façon suivante : étant donné un système de monnaie (pièces et billets), comment rendre une somme donnée de façon optimale, c'est-à-dire avec le nombre minimal de pièces et billets ?

Par exemple, la meilleure façon de rendre 7 euros est de rendre un billet de cinq et une pièce de deux, même si d'autres façons existent (rendre 7 pièces de un euro, par exemple).

En première, ce problème nous a permis d'illustrer le concept d'algorithme glouton. Si ce type d'algorithme donne une solution au problème, elle n'est pas optimale en général.

Nous allons revisiter ce problème grace à la programmation dynamique qui nous fournit ici une solution optimale dans tous les cas avec une efficacité en temps remarquable.

1. Algorithme glouton (vu en première)

- La méthode « usuelle » pour rendre la monnaie est celle de l'algorithme glouton : tant qu'il reste quelque chose à rendre, choisir la plus grosse pièce qu'on peut rendre (sans rendre trop). C'est un algorithme très simple et rapide, et on appelle canonique un système de pièces pour lequel cet algorithme donne une solution optimale quelle que soit la valeur à rendre.
- Il se trouve que presque tous les systèmes de pièces réels de par le monde sont canoniques (dont celui de l'euro). On suppose ici que l'on peut choisir autant de pièces que l'on veut.

Exercice 1 :

1. Compléter la fonction `rendu_glouton(S, syst)` . Elle prend en paramètre `S` la somme à rendre et un tableau `syst` qui contient les valeurs des pièces d'un système de monnaie et renvoie le tableau `rendu` contenant les pièces rendues.
2. En considérant l'ancien système de monnaie britannique, trouver un exemple de solution non optimale renvoyée par l'algorithme

In [1]:
```
# 1.
def rendu_glouton(S,syst):
    i=0 #indice de la pièce à rendre
    rendu=[] # pièces rendues









    return rendu


#système européen
euros = [500, 200, 100, 50, 20, 10, 5, 2, 1]
print(rendu_glouton(314,euros))
```

```
[]
```

In [2]:
```
#2.
#ancien système anglais (avant 1971)
imperial = [240,120,60,30,24,12,1]
```

```
[]
```

2. Algorithme récursif

La réponse optimale est celle qui utilise le nombre minimal de pièces. On a vu que l'algorithme glouton échoue à la trouver en général (l'exemple de rendre 36 avec le système impérial suffit à le prouver).

Une approche récursive permet de résoudre le problème : soit p une valeur de l'une des pièces du système monétaire. Pour rendre une somme S de façon optimale, en utilisant au moins une fois la pièce de valeur p, il suffit de rendre p et la somme $S - p$ de façon optimale. Il n'y a plus qu'à choisir, parmi tous les choix possibles de p, celui qui permet d'utiliser le minimum de pièces.

Exercice 2:

1. Compléter la fonction récursive `rendu_recursif(S,syst)` qui prend en paramètres la somme `S` à rendre et `syst` le tableau des valeurs de pièces d'un système monétaire. Cette fonction renvoie le nombre minimal de pièces à rendre.

Aide :

Si la somme à rendre est 0:

- Aucune pièce n'est à rendre, on renvoie 0.

Sinon:

- On note r le nombre de pièces à rendre, initialisé à S (car au pire , on rend S pièces de 1).
- Pour chaque pièce p du système monétaire telle que $p <= S$:
  - On calcule le nombre de pièces optimal à rendre sur $S - p$ auquel on ajoute 1 (la pièce p).
  - Si ce résultat est inférieur à r, c'est la nouvelle valeur de r.
- On renvoie r.

In [3]:
```
#1.
def rendu_recursif(S, syst):
    if S==0:
        return 0
    r=S #nombre de pièces à rendre (au pire S=1+1+...+1)









    return r

print(rendu_recursif(36,imperial))
```

```
36
```

2. Tester cette fonction avec des sommes inférieures à 50.

In [4]:
```
#2.
print(rendu_recursif(30,euros))
```

```
30
```

Remarques :

- Cette verison récursive donne bien le nombre optimal de pièces à chaque fois.
- Malheureusement, elle est extrêmement lente, les mêmes calculs étant effectués de façon répétée. Une analyse du problème montrerait que la complexité est exponentielle.

3. Programmation dynamique

- Une idée classique consiste à mémoriser les résultats des appels pour être sûr qu'on n'aura pas besoin de les calculer plusieurs fois.
- Ici, le calcul pour rendre la somme S utilise le calcul pour rendre $S-p$ pour chaque valeur de p. Dans le pire des cas, le calcul pour rendre S utilise celui pour rendre $S-1$, puis celui pour $S-2$...pour 3, 2 et 1.
- On va donc créer un tableau conservant ces données, et calculer de façon systématique le rendu pour des valeurs croissantes de l'index de 0 à S.

Exercice 3 :

1. Compléter la fonction `rendu_dynamique(S,syst)` qui prend en paramètres une somme `S` et un système de monnaie `syst`. Cette fonction renvoie le nombre minimal de pièces pour rendre `S`.

Aide :

- Le tableau `np` (comme nombre de pièces) contient les solutions à la fin de l'éxécution , pour un système de pièces donné, pour toutes les sommes de 0 à S. Ainsi par exemples dans le système des euros, `np[0]=0` , `np[3]=2` , `np[100]=1` , etc.
- Au départ, on alloue ce tableau de longueur $S+1$ en l'initialisant avec des 0. On remarque que pour rendre 0 , il faut 0 pièces, la première case du tableau est donc déjà correctement initialisée. Reste à remplir le tableau pour toutes les sommes de 1 à S.
- Pour chaque somme de 1 à S, c'est à dire pous chaque indice du tableau:
  - On initialise `np[somme]` à `somme` , car on sait que dans le pire des cas, on fait la somme qu'avec des pièces de 1.
  - On cherche ensuite comment faire moins. Pour chaque valeur de pièce p dans le système monétaire utilisé :
    - On sait qu'il y a une solution en un nombre de pièces égal à `1+np[somme-p]` , c'est à dire 1 pièce de valeur p et la meilleure solution possible pour la somme $n-p$ déjà calculée.
      - Avec cette boucle sur toutes les pièces, on calcule donc le minimum de toutes les solutions possibles,ce sui donne bien la solution pour `somme` .
- Une fois sorti de ces deux boucles, on a calculé la solution pour toutes les sommes jusqu'à S.On renvoie `np[S]` .

In [5]:
```
def rendu_dynamique(S,syst):
    np=[0]*(S+1) # tableau des solutions de longueur S









    return np[S]

print(rendu_dynamique(314,euros))
```

```
0
```

2. A l'aide de tests, comparer l'efficacité de cet algorithme par rapport au précédent.

In [6]:
```
rendu_dynamique(30000,euros)
```

Out[6]:
```
0
```

Remarques :

- Cet algorithme a la même structure que celui de l'exercice précédent mais les appels récursifs ont été remplacés par des consultations du tableau `np` .
- Cette version qui utilise la programmation dynamique est bien plus efficace que la version récursive initiale !

Exercice 4 : Construction des solutions

- Le programme précédent renvoie le nombre minimal de pièces pour obtenir la somme S mais il ne donne pas la solution pour autant (ce que faisait par contre l'algorithme glouton), c'est à dire la liste des pièces utilisées.
- Nous allons modifier le programme précédent pour qu'il renvoie la liste des pièces utilisées dans la construction de la solution.

Recopier le code de la fonction de l'exercice précédent dans la nouvelle fonction ci-dessous, puis la compléter et la modifier pour qu'elle renvoie la liste des pièces de la solution optimale pour une somme `S` et un système monétaire `syst` donnés.

Aide : Voici un descriptif des modifications à apporter.

- A la fin de l'éxécution, le tableau `sol` contient pour chaque somme la liste des pièces qui la constituent de façon optimale. Ainsi, pour un système en euros, `sol[2]=[2]` , `sol[15]=[10,5]` , `sol[314]=[200,100,10,2,2]` , etc.
- Pour chaque somme entre 1 et S:
  - On initialise `sol[somme]` de telle sorte que sa longueur soit égale à `somme` (avec des pièces de 1).
  - Puis on examine toutes les pièces p du système monétaire(comme dans le programme précédent):
    - Lorsqu'une pièce p permet d'améliorer la solution(c'est à dire lorsque `p<=S` ET lorsque `1+ np[somme-p]<np[somme]` , la valeur des deux tableaux est mise à jour.
    - Pour mettre à jour la valeur de `sol[n]` , on récupère une copie de `sol[n-p]` (avec la méthode `.copy()`), puis on y ajoute la valeur p (avec `.append()` .

In [7]:
```
def rendu_dynamique_solution(S,syst):
    np=[0]*(S+1) # tableau des solutions de longueur S
    sol=[[]]*(S+1) #tableau contenant pour chaque solution, la liste des pièces utilisées









    sol[S].reverse()
    return sol[S]
```

In [8]:
```
#tests
rendu_dynamique_solution(36,imperial)
```

Out[8]: []

In [9]:
```
rendu_dynamique_solution(314,euros)
```

Out[9]: []