Programmation dynamique

1. Principe de la programmation dynamique

La programmation dynamique, introduite au début des années 1950 par Richard Bellman, est une méthode pour résoudre des problèmes en combinant des solutions de sous-problèmes, tout comme les méthodes de type diviser pour régner.
Un algorithme de programmation dynamique résout chaque sous-sous-problème une seule fois et mémorise sa réponse dans un tableau, évitant ainsi le re-calcul de la solution chaque fois qu'il résout chaque sous-sous-problème.
La programmation dynamique s'applique généralement aux problèmes d'optimisation, comme ceux des algorithmes gloutons par exemples.

Le terme « programmation » désigne la « planification », et n'a pas de rapport avec les langages de programmation.

2. Le problème du rendu de monnaie

Le problème du rendu de monnaie est de déterminer le nombre minimum de pièces qui doivent être utilisées pour rendre la monnaie.

2.1. Résolution par la méthode gloutonne

Vous avez à votre disposition un nombre illimité de pièces de **1 euro (100 centimes), 50 centimes, 10 centimes, 5 centimes et 2 centimes**. Vous devez rendre une certaine somme (rendu de monnaie). Le problème est le suivant : "Quel est le nombre minimum de pièces qui doivent être utilisées pour rendre la monnaie"

La résolution gloutonne de ce problème peut être la suivante :

- On prend la pièce qui a la plus grande valeur (il faut que la valeur de cette pièce soit inférieure ou égale à la somme restant à rendre)
- On recommence l'opération ci-dessus jusqu'au moment où la somme à rendre est égale à zéro.

✍ Appliquer cette méthode pour une somme de 1€77 (177 centimes) à rendre. Combien faut-il rendre de pièces ?

✍ Appliquer cette méthode pour une somme de 11 centimes à rendre. Conclure.

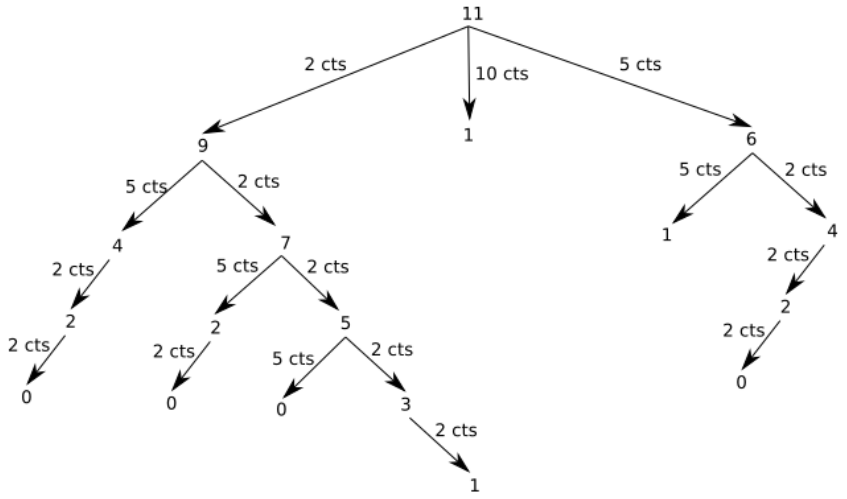
Le fait que la méthode gloutonne ne fonctionne pas, ne veut pas dire qu'il n'existe pas de solution.
✍ Donner une solution pour la somme de 11 centimes à rendre.

✍ Vérifier les résultats des deux premières questions avec la fonction ci-dessous.

```
def rendu_monnaie_rec(pieces_dispo, somme):
    """ renvoie de manière gloutonne et récursive, le nombre minimal de pièces
    pour rendre la somme avec les pièces disponibles """
    if somme == 0:
        return 0
    else:
        piece = pieces_dispo[0]
        if piece <= somme:
            return 1 + rendu_monnaie_rec(pieces_dispo, somme - piece)
        else:
            return rendu_monnaie_rec(pieces_dispo[1:], somme)
```

2.2. Résolution par la méthode de force brute

La force brute consiste à tester toutes les solutions possibles.
Examinons en détails le processus pour la somme de 11cts à l'aide d'un arbre.



Plusieurs remarques :

- Tous les cas sont "traités" (quand un algorithme "traite" tous les cas possibles, on parle souvent de méthode de "force brute".
- Pour certains cas, on se retrouve dans une "impasse" (cas où on termine par un "1").
- La profondeur minimum de l'arbre (avec une feuille 0) est de 4, la solution au problème est donc 4 (il existe plusieurs parcours : (5,2,2,2), (2,5,2,2)... qui donne à chaque fois 4)/

Afin de mettre au point un algorithme, essayons de trouver une relation de récurrence :
Soit X la somme à rendre, on notera Nb(X) le nombre minimum de pièces à rendre. Nous allons nous poser la question suivante : Si je suis capable de rendre X avec Nb(X) pièces, quelle somme suis-je capable de rendre avec 1+Nb(X) pièces ?
Si j'ai à ma disposition la liste de pièces suivante : p_1, p_2, p_3, ..., p_n et que je suis capable de rendre X cts, je suis donc aussi capable de rendre :
$X-p_1$
$X-p_2$
$X-p_3$
...
$X-p_n$
(à condition que p_i (avec i compris entre 1 et n) soit inférieure ou égale à la somme restant à rendre)
Exemple : si je suis capable de rendre 72 cts et que j'ai à ma disposition des pièces de 2 cts, 5 cts, 10 cts, 50 cts et 1 euro, je peux aussi rendre :
72 - 2 = 70 cts
72 - 5 = 67 cts
72 - 10 = 62 cts
72 - 50 = 22 cts
Je ne peux pas utiliser de pièce de 1 euro.
Autrement dit, si $Nb(X-pp_i)$ (avec i compris entre 1 et n) est le nombre minimal de pièces à rendre pour le montant $X-p_i$, alors $Nb(X)=1+Nb(X-p_i)$ est le nombre minimal de pièces à rendre pour un montant X. Nous avons donc la formule de récurrence suivante :
si X=0 : Nb(X)=0
si X>0 : $Nb(X)=1+\min(Nb(X-p_i))$ avec $1\leq i<n$ et $pi\leq X$

Le "min" présent dans la formule de récurrence exprime le fait que le nombre de pièces à rendre pour une somme $X-p_i$ doit être le plus petit possible.

Le programme ci-dessous réalise ce processus.

```
def rendu_monnaie_rec(P,X):
  if X==0:
    return 0
  else:
    mini = 1000
  for i in range(len(P)):
    if P[i]<=X:
      nb = 1 + rendu_monnaie_rec(P,X-P[i])
      if nb<mini:
        mini = nb
  return mini
```

La fonction *rendu_monnaie_rec* prend en paramètre un tableau de pièces (P) et la somme à rendre (X). Elle renvoie le plus petit nombre de pièces possible. On retrouve la relation de récurrence définie juste au-dessus.

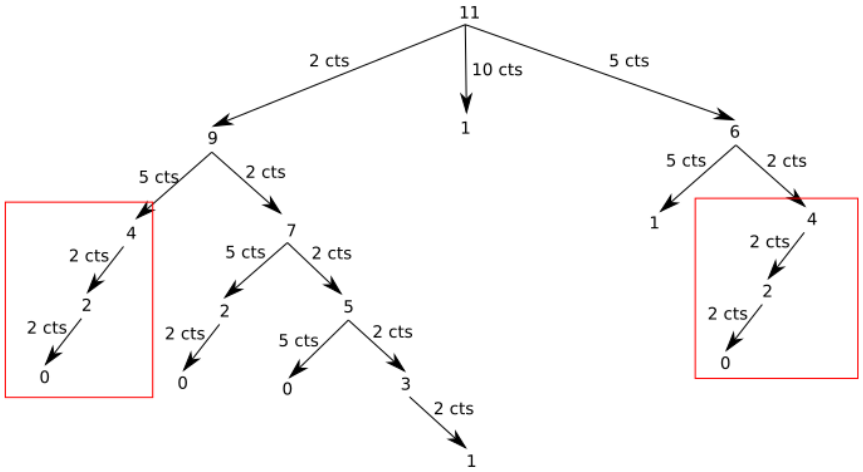
Pour être sûr de renvoyer le plus petit nombre de pièces, on attribue dans un premier temps la valeur 1000 à la variable mini (cette valeur 1000 est arbitraire, il faut juste une valeur suffisamment grande : on peut partir du principe que nous ne rencontrerons jamais de cas où il faudra rendre plus de 1000 pièces), ensuite, à chaque appel récursif, on "sauvegarde" le plus petit nombre de pièces dans cette variable mini.

✍ Écrire ce programme et le faire fonctionner pour la somme de 11cts.
✍ Faire évoluer la somme à rendre 12 cts, 15 cts, ... À partir de quelle somme le programme est-il visiblement lent ?

Comme vous l'avez sans doute constaté le programme ne permet pas toujours d'obtenir une solution, car les appels récursifs sont trop nombreux, on dépasse la capacité de la pile.

2.3. Résolution par la programmation dynamique

En y regardant de plus près, on s'aperçoit que certains calculs se font plusieurs fois (le rendu de 4 cts par exemple dans le schéma ci-dessous).
On pourrait donc grandement simplifier le calcul en calculant une fois pour toutes rendu de monnaie pour 4, en "mémorisant" le résultat et en le réutilisant quand nécessaire :



Cette méthode est la programmation dynamique.

```
def rendu_monnaie_mem(P,X):
  mem = [0]*(X+1)
  return rendu_monnaie_mem_c(P,X,mem)

def rendu_monnaie_mem_c(P,X,m):
  if X==0:
    return 0
  elif m[X]>0:
    return m[X]
  else:
    mini = 1000
    for i in range(len(P)):
      if P[i]<=X:
        nb=1+rendu_monnaie_mem_c(P,X-P[i],m)
        if nb<mini:
          mini = nb
          m[X] = mini
  return mini
```

Source : https://info-mounier.fr/terminale_nsi/algorithmique/prog_dynamique.php
https://eduscol.education.fr/document/30079/download
https://pixees.fr/informatiquelycee/term/c16c.html