1 Problématique

L'approche gloutonne du rendu de monnaie permet de résoudre efficacement un problème qui a un temps de résolution long. Cependant la solution proposée peut dans certains cas de figure ne pas être optimale.

problème NP-complet relativement à la taille du système monétaire. greedy algorithm = solution approchée

Peut-on trouver une solution optimale en un temps raisonnable ?

2 Approche gloutonne

2.1 Algorithme

Un algorithme glouton fait un choix sur lequel il ne revient pas.

```python
def nb_pieces_glouton(somme: int, systeme: list) -> int:
    nb_piece = 0
    while not somme == 0:
        i = 0
        # l'agorithme choisit la plus grande pièce
        while systeme[i] > somme:
            i += 1
        somme -= systeme[i]
        nb_piece += 1
    return nb_piece
```

Code 1 – Approche gloutonne

2.2 Un exemple non optimal

Choisissons un autre système de monnaie (code 2).

```python
    systeme = [10, 4, 3, 1]
```

Code 2 – Système monétaire fictif

L'approche gloutonne ne donne pas la solution optimale.

Activité 1 : Dérouler à la main l'exécution de la fonction *nb_pieces_glouton* pour 6€ avec le système (simplifié) de monnaie européenne puis le système fictif.

Remarque

Le système monétaire européen est dit *canonique*.

Christophe Viroulaud

avant 1971, le système britannique n'était pas canonique : une livre sterling se divisait en 240 pence. Douze pence valaient un shilling et vingt shillings équivalaient à une livre.

3 Approche dynamique

3.1 Algorithme naïf

Pour être certain de trouver la solution optimale il faut énumérer toutes les possibilités.

```
1  def nb_pieces_naif(somme: int, systeme: list) -> int:
2      if somme == 0:
3          return 0
4      nb_mini = somme + 1 # somme = 1 + 1 + 1 ...
5      # Pour chaque pièce du système
6      for piece in systeme:
7          if piece <= somme:
8              nb_pieces = 1 + nb_pieces_naif(somme-piece, systeme)
9              if nb_pieces < nb_mini:
10                 nb_mini = nb_pieces
11     return nb_mini
```

Code 3 – Approche naïve

— ligne 4 on initialise *nb_mini* pour avoir une référence

— tester toutes les solutions pour toutes les pièces du système (ligne 6)

— Je prends la pièce possible (1+ de la ligne 8) et je cherche toutes les solutions pour le reste à rendre

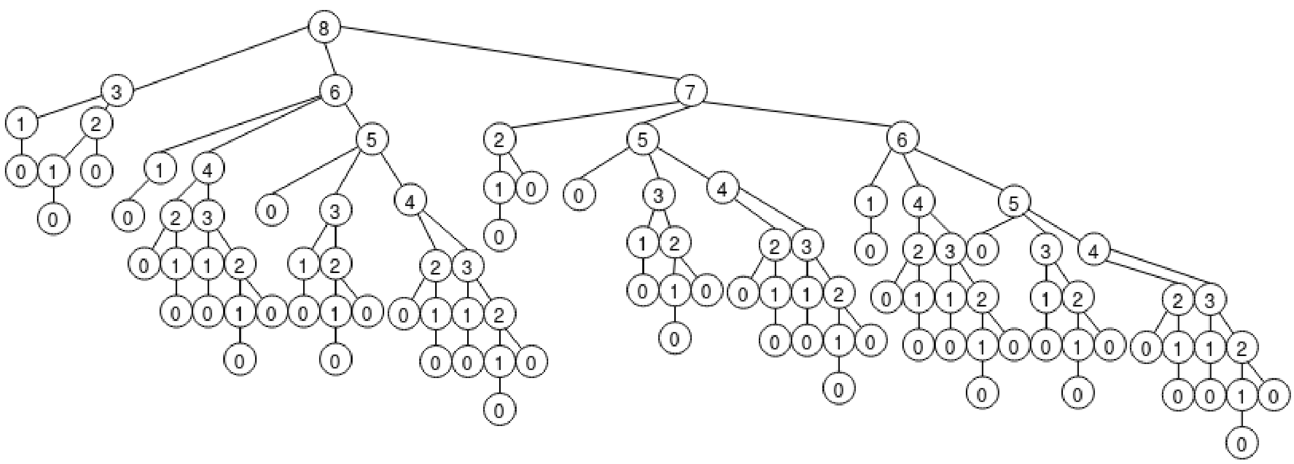

FIGURE 1 – Appels récursifs pour 8€

Activité 2 : En s'aidant de l'arbre, dérouler l'exécution de la fonction (pour les premiers cas) à la main afin d'en comprendre le fonctionnement.

Christophe Viroulaud

3.2 Top-down

L'approche naïve montre une redondance dans les calculs. L'utilisation d'un tableau *track* pour stocker les résultats intermédiaires permet d'éviter ce problème.

> **Activité 3 :**
>
> 1. Écrire la fonction **nb_pieces_TD(somme : int, systeme : list, tracl : list) → int** qui reprend l'algorithme naïf et utilise le tableau *track* de stockage intermédiaire.
> 2. Tester la fonction pour les deux systèmes monétaires.

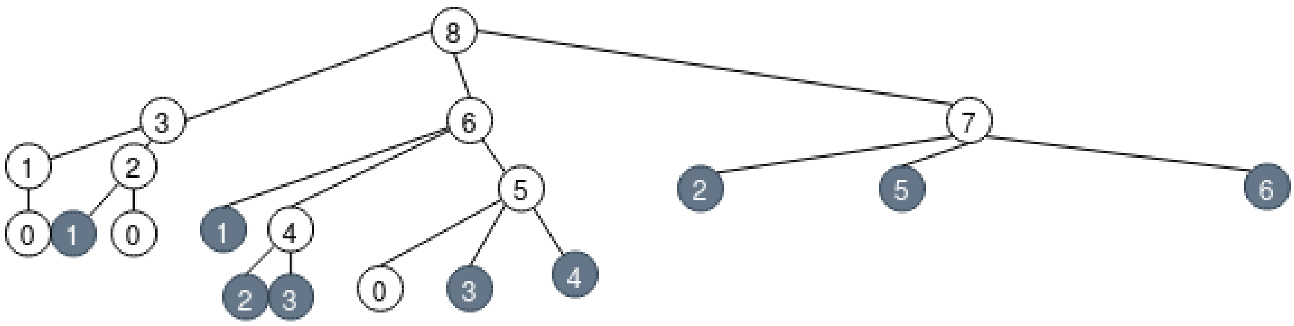

FIGURE 2 – Approche dynamique pour 8€

3.3 Bottom-up

L'approche itérative *bottom-up* trouve les solutions pour les petites sommes d'abord.

```python
def nb_pieces_BU(somme: int, systeme: list) -> int:
    track = [0 for _ in range(somme+1)]
    # pour chaque pièce de track on cherche le nombre minimum de pièces à rendre
    for x in range(1, somme+1):
        mini = somme+1
        for piece in systeme:
            if (piece <= x):
                nb_pieces = 1+track[x-piece]
                if nb_pieces < mini:
                    mini = nb_pieces
        track[x] = mini
    return track[somme]
```

Code 4 – Approche bottom-up

> **Activité 4 :** Écrire la fonction **nb_pieces_BU_sol(somme : int, systeme : list) → list** qui renvoie la liste des pièces à choisir pour rendre la monnaie. On utilisera un tableau *choix* de taille *somme+1* où chaque élément de rang x contiendra la valeur de la première pièce à rendre pour la somme x.

Christophe Viroulaud