Christophe Viroulaud

Terminale NSI

L'approche gloutonne du rendu de monnaie permet de résoudre efficacement un problème mais ne donne pas toujours une solution optimale.

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

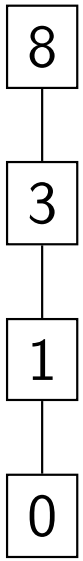
L'algorithme glouton fait un choix définitif

```
1  def nb_pieces_glouton(somme: int,
       systeme: list) -> int:
2      nb_piece = 0
3      while not somme == 0:
4          i = 0
5          # l'agorithme choisit la plus
               grande pièce
6          while systeme[i] > somme:
7              i += 1
8          somme -= systeme[i]
9          nb_piece += 1
10     return nb_piece
```

Code 1 – Approche gloutonne

Exemple d'exécution

```
1 systeme = [50, 20, 10, 5, 2, 1]
2 somme = 8
3 nb_pieces_glouton(somme, systeme)
```



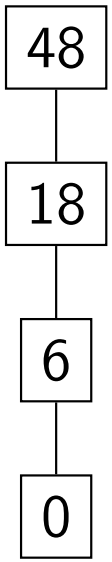
Un système non optimal (non canonique)

```
1 systeme = [30, 24, 12, 6, 3, 1]
```

Code 2 – Système monétaire impérial britannique

Activité 1 : Dérouler à la main l'exécution de la fonction *nb_pieces_glouton* pour 48€ avec le système (simplifié) de monnaie européenne puis le système impérial.

La solution proposée par l'algorithme glouton



Une meilleure solution

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

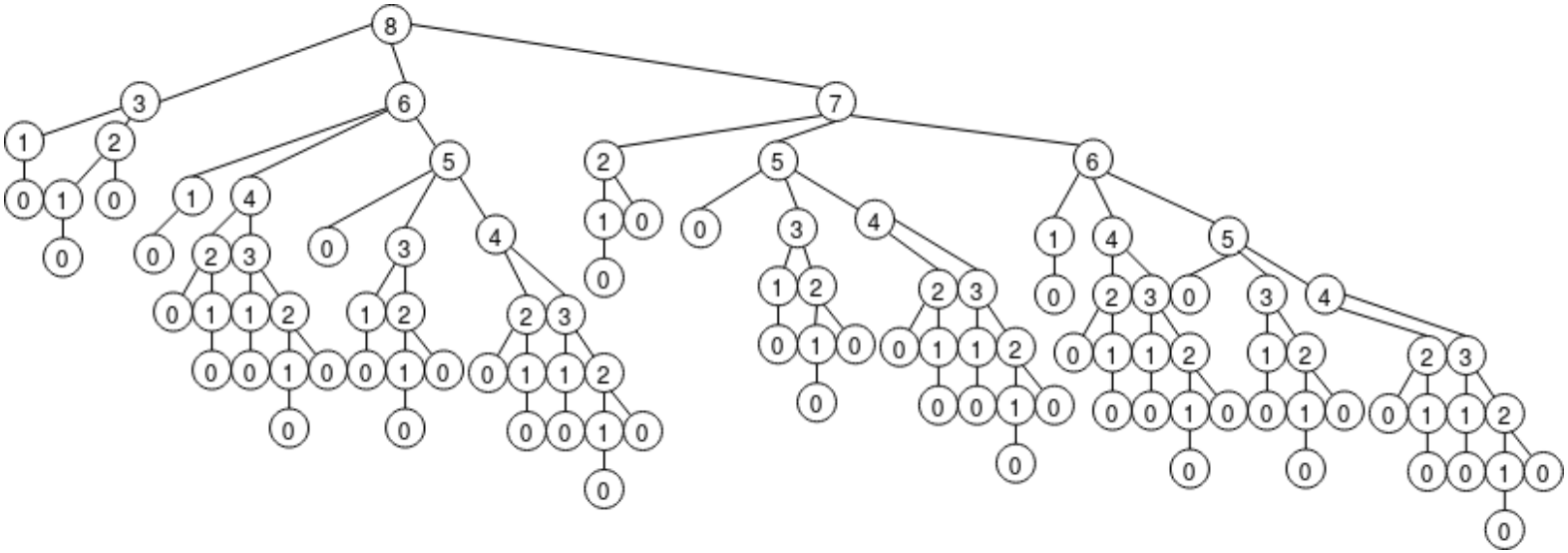


FIGURE – Appels récursifs pour 8€

On teste récursivement toutes les solutions pour chaque pièce du système (ligne 6).

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.

- Choisir 5, reste 3
  - Choisir 2, reste 1
    - Choisir 1, reste 0 $\rightarrow$
      remontée d'appel, nb_pieces $= 1$, nb_mini $= 1$
  - Choisir 1, reste 2
    - Choisir 2, reste 0
    - Choisir 1, reste 1
      Choisir 1, reste 0 $\rightarrow$
      remontée d'appel, nb_pieces $= 1$, nb_mini $= 1$
- Choisir 2, reste 6
- Choisir 1, reste 7

L'utilisation d'un tableau *track* pour éviter la redondance des calculs.

Activité 3 :

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

```
1  def nb_pieces_TD(somme: int, systeme: list, track:
       list) -> int:
2      if somme == 0 :
3          track[0] = 0
4          return track[0]
5      # le nombre de pièces a déjà été calculé
6      if track[somme] > 0 :
7          return track[somme]
8      nb_mini = somme + 1 # somme = 1 + 1 + 1 ...
9      for piece in systeme:
10         if piece <= somme:
11             nb_pieces = 1 + nb_pieces_TD(somme-
                   piece, systeme, track)
12             if nb_pieces < nb_mini:
13                 nb_mini = nb_pieces
14     # Pour cette somme on a trouvé le nb de pièces
           mini
15     track[somme] = nb_mini
16     return track[somme]
```

```
1 systeme = [10, 5, 2, 1]
2 somme = 8
3 # Le tableau de stockage est
     initialisé
4 track = [-1 for _ in range(somme+1)]
5 nb_pieces_TD(somme, systeme, track)
```

Code 4 – Appel de la fonction

Englober le cas limite dans track

```
1  def nb_pieces_TD2(somme: int, systeme: list, track:
       list) -> int:
2      if track[somme] >= 0:
3          return track[somme]
4      nb_mini = somme + 1 # somme = 1 + 1 + 1 ...
5      for piece in systeme:
6          if piece <= somme:
7              nb_pieces = 1 + nb_pieces_TD2(somme-
                   piece, systeme, track)
8              if nb_pieces < nb_mini:
9                  nb_mini = nb_pieces
10     track[somme] = nb_mini
11     return track[somme]
```

```
1 systeme = [10, 5, 2, 1]
2 somme = 8
3 # Le tableau de stockage est
     initialisé
4 track = [-1 for _ in range(somme+1)]
5 track[0] = 0
6 nb_pieces_TD2(somme, systeme, track)
```

Code 5 – Appel de la fonction

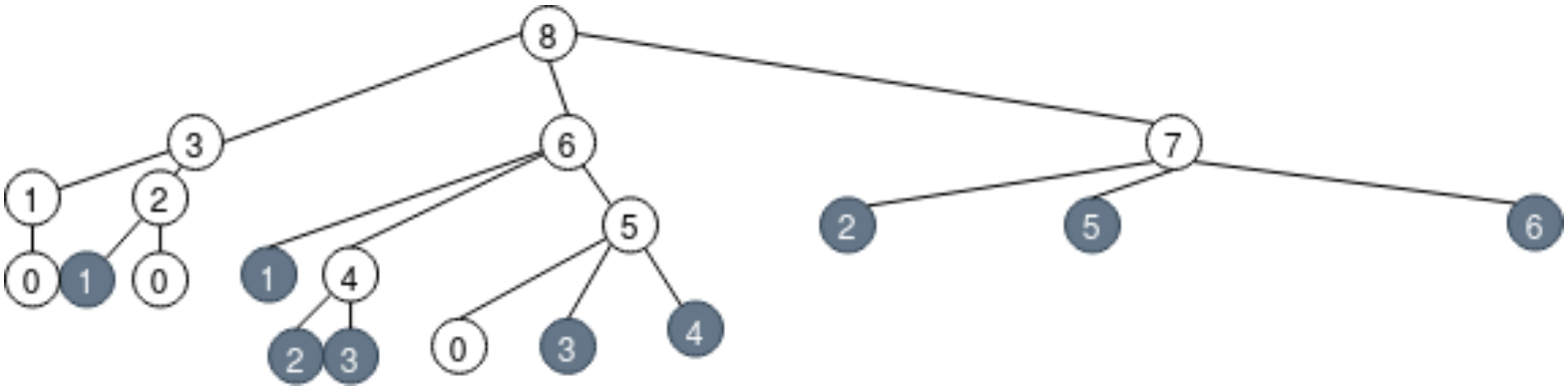


FIGURE – Approche dynamique pour 8€

Approche itérative

```
1  def nb_pieces_BU(somme: int, systeme: list) -> int:
2      track = [0 for _ in range(somme+1)]
3      # pour chaque pièce de track on cherche le nombre minimum de pièces àrendre
4      for x in range(1, somme+1):
5          mini = somme+1
6          for piece in systeme:
7              if (piece <= x):
8                  nb_pieces = 1+track[x-piece]
9                  if nb_pieces < mini:
10                     mini = nb_pieces
11         track[x] = mini
12     return track[somme]
```

Code 6 – Approche bottom-up

Le tableau se remplit d'abord par les petites valeurs

▶

```
1 track = [0, 0, 0, 0, 0, 0, 0, 0]
```

Le tableau se remplit d'abord par les petites valeurs

-
```
track = [0, 0, 0, 0, 0, 0, 0, 0]
```

-
```
track = [0, 1, 0, 0, 0, 0, 0, 0]
```

Le tableau se remplit d'abord par les petites valeurs

- ```
  track = [0, 0, 0, 0, 0, 0, 0, 0]
  ```

- ```
  track = [0, 1, 0, 0, 0, 0, 0, 0]
  ```

- ```
  track = [0, 1, 1, 0, 0, 0, 0, 0]
  ```
  ```

Le tableau se remplit d'abord par les petites valeurs

- ```
  track = [0, 0, 0, 0, 0, 0, 0, 0]
  ```

- ```
  track = [0, 1, 0, 0, 0, 0, 0, 0]
  ```

- ```
  track = [0, 1, 1, 0, 0, 0, 0, 0]
  ```

- ```
  track = [0, 1, 1, 2, 0, 0, 0, 0]
  ```

Le tableau se remplit d'abord par les petites valeurs

- ```
  track = [0, 0, 0, 0, 0, 0, 0, 0]
  ```

- ```
  track = [0, 1, 0, 0, 0, 0, 0, 0]
  ```

- ```
  track = [0, 1, 1, 0, 0, 0, 0, 0]
  ```

- ```
  track = [0, 1, 1, 2, 0, 0, 0, 0]
  ```

- ```
  track = [0, 1, 1, 2, 2, 0, 0, 0]
  ```
  ```

Activité 4 : Écrire la fonction
nb_pieces_BU_sol(somme : int, systeme : list)
$\rightarrow$ **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.

Création des stockages

```
1  def nb_pieces_BU_sol(somme: int, systeme: list) ->
       list:
2      track = [0 for _ in range(somme+1)]
3      # choix[x] contient la première pièce à rendre
           pour la somme x
4      choix = [0 for _ in range(somme+1)]
```

Remarque

On aurait également pu utiliser un dictionnaire pour stocker les choix de pièces effectués : la clé est la somme à rendre et la valeur correspondante la première pièce à rendre.

Remplissage des stockages

```
1     # pour chaque pièce de track on cherche le
          nombre minimum de pièces àrendre
2     for x in range(1, somme+1):
3         mini = somme+1
4         for piece in systeme:
5             if (piece <= x):
6                 nb_pieces = 1+track[x-piece]
7                 if nb_pieces < mini:
8                     mini = nb_pieces
9                     choix_piece = piece
10        track[x] = mini
11        choix[x] = choix_piece
```

Reconstruction de la solution

```
1  # reconstitution des pièces à rendre
2  rendre = somme
3  resultat = []
4  while rendre > 0 :
5      resultat.append(choix[rendre])
6      rendre -= choix[rendre]
7  return resultat
```