Programmation dynamique
Rendu de monnaie

Christophe Viroulaud

Terminale - NSI

Algo 25

Pour répondre à un problème on peut utiliser plusieurs stratégies algorithmiques selon l'objectif à atteindre :

- trouver une solution exacte même si cela prend un temps long,
- trouver une solution approchée mais plus rapidement.

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

Sommaire

Le rendu de monnaie

Définition

Le problème du rendu de monnaie consiste à minimiser le nombre de pièces à rendre pour une somme donnée.
On dispose d'un système de monnaie (exemple : système européen).

Sommaire

Approche gloutonne - Principe

À retenir

La stratégie gloutonne est une méthode approchée de résolution. Elle consiste à faire un choix de résolution et ne pas revenir dessus.
Dans le problème du rendu de monnaie, le choix est fait de rendre d'abord **la plus grande pièce possible**.

Sommaire

Activité 1 : Le système monétaire est donné dans l'ordre décroissant :

```
1 systeme = [10, 5, 2, 1]
```

1. Écrire la fonction itérative **`rendu_monnaie(somme: int, systeme: list)`** $\rightarrow$ **`list`** qui renvoie la liste des pièces à rendre pour rembourser **`somme`**.
2. Écrire la fonction récursive équivalente.

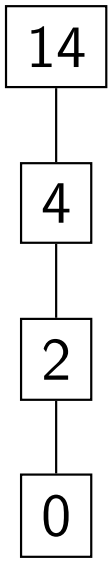
Correction

```python
def rendu_glouton(somme: int, systeme: list) -> list:
    res = []
    i_piece = 0
    while somme > 0:
        # si pièce est trop grande
        if systeme[i_piece] > somme:
            # on avance dans le système
            i_piece += 1
        else:
            res.append(systeme[i_piece])
            somme -= systeme[i_piece]

    return res
```

```
>>> rendu_glouton(14, [10, 5, 2, 1])
[10, 2, 2]
```

```python
def rendu_glouton_rec(somme: int, systeme: list, i_piece: int) -> list:
    if somme > 0:
        # si pièce est trop grande
        if systeme[i_piece] > somme:
            # on avance dans le système
            i_piece += 1
            return rendu_glouton_rec(somme, systeme, i_piece)
        else:
            # on rend la pièce
            somme -= systeme[i_piece]
            return [systeme[i_piece]] + rendu_glouton_rec(somme, systeme, i_piece)

    else:
        # cas limite
        return []
```

```
>>> rendu_glouton_rec(14, [10, 5, 2, 1], 0)
[10, 2, 2]
```



Sommaire

Optimalité

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

Code 1 – Système monétaire impérial britannique

> **Activité 2 :** Dérouler à la main l'exécution de la fonction **`rendu_monnaie`** pour 48€ avec le système impérial.

Correction

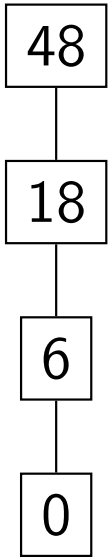


FIGURE 1 – Solution donnée par l'algorithme glouton.

Une meilleure solution

FIGURE 2 – Une meilleure solution.

Remarque

Le système monétaire européen garantit une solution optimale avec l'algorithme glouton. Ce système est dit **canonique**.

Sommaire

Approche dynamique - Algorithme naïf

À retenir

Pour être certain de trouver la solution optimale, il faut calculer toutes les possibilités.
Le principe d'optimalité de Bellman affirme qu'une solution optimale d'un problème s'obtient en combinant des solutions optimales à des sous-problèmes.

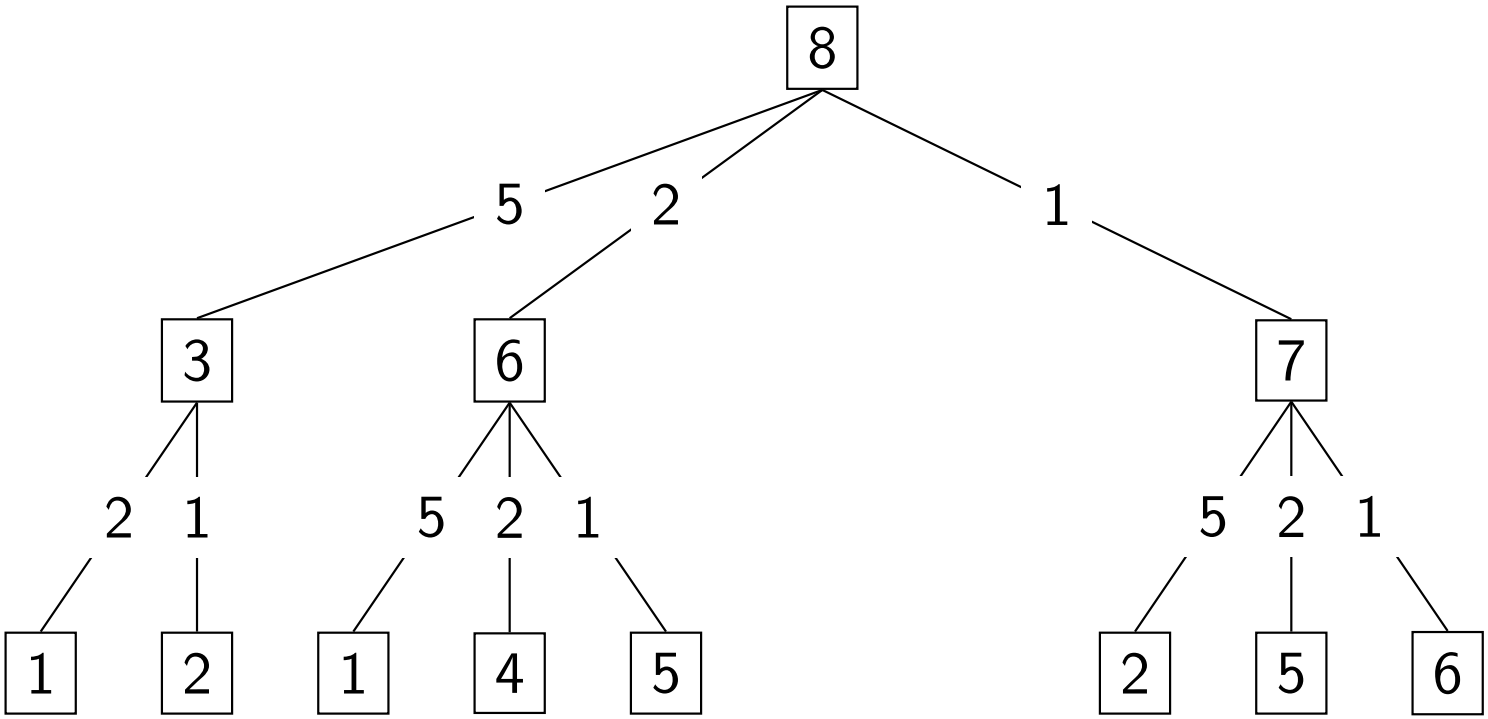


FIGURE 3 – Rendre 8€

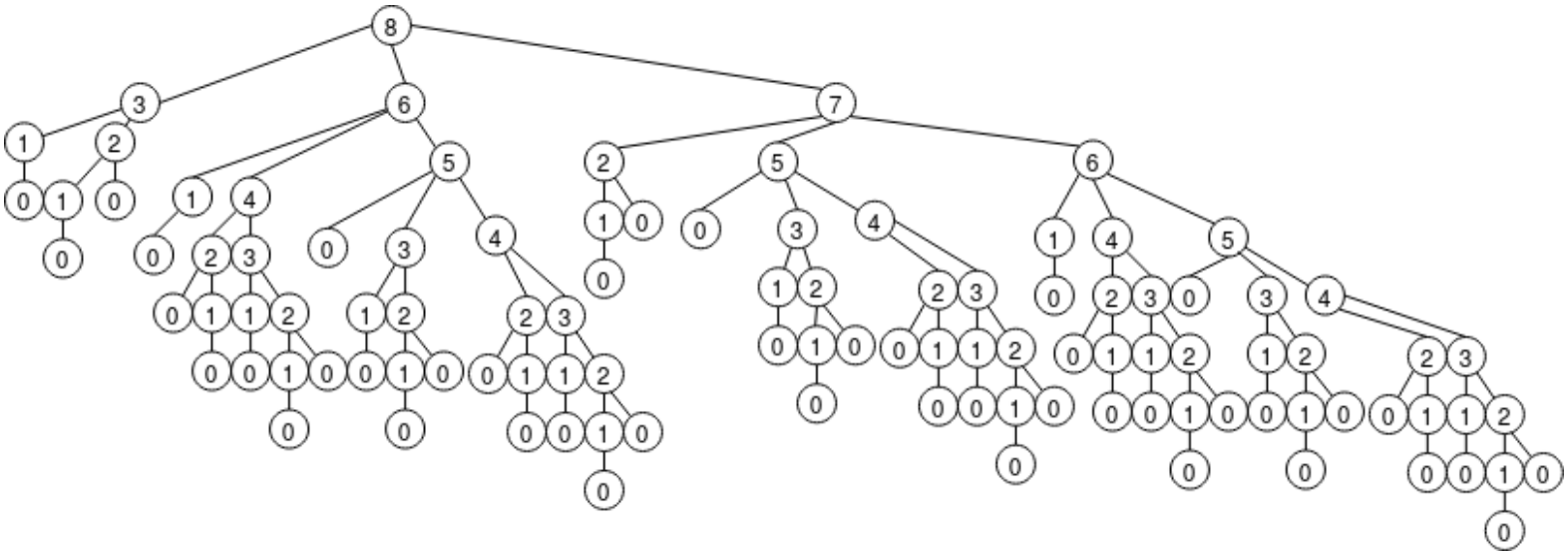


FIGURE 4 – Appels récursifs pour 8€

```python
def rendu_naif(somme: int, systeme: list) -> list:
    if somme > 0:
        # somme = 1 + 1 + 1 ...
        mini = [1 for _ in range(somme)]
        # Teste tous les possibles pour chaque pièce
        for piece in systeme:
            if piece <= somme:
                pieces = [piece] + \
                         rendu_naif(somme-piece, systeme)
                if len(pieces) < len(mini):
                    mini = pieces
        # garde le rendu minimum
        return mini
    else:
        return []
```

Activité 3 :

1. Dérouler le début de l'algorithme à la main pour en comprendre le fonctionnement.
2. Ajouter une variable globale `COMPTEUR` qui compte le nombre d'appels récursifs.

Correction


```
1  def rendu_naif(somme: int, systeme: list) -> list:
2      global C
3      if somme > 0:
4          mini = [1 for _ in range(somme)]
5          for piece in systeme:
6              if piece <= somme:
7                  C += 1
8                  pieces = [piece] + \
9                          rendu_naif(somme-piece, systeme)
10                 if len(pieces) < len(mini):
11                     mini = pieces
12         # garde le rendu minimum
13         return mini
14     else:
15         return []
```

```
1 >>> C = 0
2 >>> rendu_naif(14, [10, 5, 2, 1])
3 [10, 2, 2]
4 >>> C
5 2651
```

Sommaire

Approche descendante (top-down)

Activité 4 :

1. Écrire la fonction **`rendu_td(somme: int, systeme: list, track: list)`** $\rightarrow$ **`list`** qui s'inspire de la version naïve mais maintient un tableau **`track`** des solutions optimales pour chaque somme à rendre. Le tableau sera initialisé par :

```
1 track = [[] for _ in range(somme+1)]
```

2. Ajouter une variable globale **`COMPTEUR`** qui compte le nombre d'appels récursifs.

```python
def rendu_td(somme: int, systeme: list, track: list) -> list:
    # le choix de pièces a déjà été calculé
    if len(track[somme]) > 0:
        return track[somme]
    elif somme > 0:
        mini = [1 for _ in range(somme)]
        # Teste tous les possibles pour chaque pièce
        for piece in systeme:
            if piece <= somme:
                pieces = [piece] + \
                    rendu_td(somme-piece, systeme, track)
                if len(pieces) < len(mini):
                    mini = pieces
        # garde le rendu minimum
        track[somme] = mini
        return mini
    else:
        return []
```

```
1 >>> track = [[] for _ in range(s+1)]
2 >>> rendu_td(s, [10, 5, 2, 1], track)
3 [10, 2, 2]
```

Code 2 – Appel

```
1 >>> C = 0
2 >>> track = [[] for _ in range(s+1)]
3 >>> rendu_td(s, [10, 5, 2, 1], track)
4 [10, 2, 2]
5 >>> C
6 42
```

Code 3 – Nombre d'appels récursifs

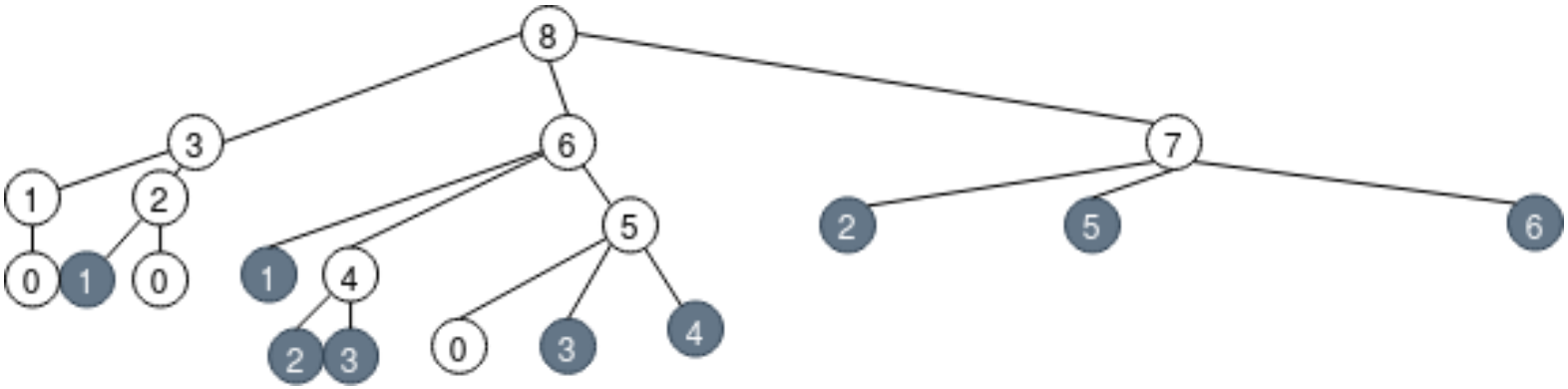


FIGURE 5 – Approche dynamique pour 8€

Sommaire

Approche ascendante (bottom-up)

Activité 5 :

1. Écrire la fonction **`rendu_bu(somme: int, systeme: list)`** $\rightarrow$ **`list`** qui complète un tableau **`track`** des solutions optimales, en le remplissant d'abord par les plus petites sommes à rendre. Le tableau sera initialisé par :

```
1 track = [[] for _ in range(somme+1)]
```

2. Ajouter une variable globale **`COMPTEUR`** qui compte le nombre d'appels récursifs.

Correction

```
1  def rendu_bu(somme: int, systeme: list) -> list:
2      track = [[] for _ in range(somme+1)]
3      # pour chaque pièce de track on cherche le
       nombre minimum de pièces à rendre
4      for x in range(1, somme+1):
5          # on ne rend que des pièces de 1
6          mini = [1 for _ in range(somme)]
7          for piece in systeme:
8              if piece <= x:
9                  # prend la solution optimale pour 'x
       -piece'
10                 pieces = [piece] + track[x-piece]
11                 if len(pieces) < len(mini):
12                     mini = pieces
13         track[x] = mini
14
15     return track[somme]
```

```
1 >>> rendu_bu(s, [10, 5, 2, 1])
2 [10, 2, 2]
```

Code 4 – Appel

```
track = [[], [1], [2], [2, 1], [2, 2], [5], [5, 1],
   [5, 2], [], [], [], [], [], [], []]
```

Code 5 – État de `track` au début de l'itération `x = 8`

```
for piece in systeme:
    if piece <= x:
        # prend la solution optimale pour 'x-piece'
        pieces = [piece] + track[x-piece]
        if len(pieces) < len(mini):
            mini = pieces
```

Code 6 – Pour `piece = 5`, la variable `pieces = [5] + [2, 1]`

```
>>> C = 0
>>> rendu_bu(s, [10, 5, 2, 1])
[10, 2, 2]
>>> C
42
```

Code 7 – Nombre d'appels récursifs