# Correction exercice 12.1

- Commentez ce code
- Est-ce un algorithme glouton? Procède-ton par étape, avec une solution optimale localement, sans retour arrière?

```python
bag: List[int] = []
bag_weight = 0
for item_key in objets_par_valeur_massique.keys():
    bag_weight += item_weights[item_key]
    if bag_weight <= bag_capacity:
        bag += [item_key]
    else:
        break
```

- Quelle est la condition d'arrêt?

# Exercice 12.2- Algo Glouton

- Les règles par poids et valeur sont meilleurs avec ce jeu de données

```
Solution gloutonne
Item dans le sac A
Item dans le sac B
Valeur du sac 1100 et poids du sac 25, capacité résiduelle 5
```

- Ajoutez un item qui remplit le sac (de 30kgs) de plus grande valeur que n'importe quels autres items (701$)
- Quelle règle gloutonne est la meilleure?

# Exercice 12.3- Algo Glouton

```
solution pour rembourser la somme = 263
 La pièce de 1 centime est choisie 1 fois, il reste 262
 La pièce de 2 centimes est choisie 1 fois, il reste 260
 La pièce de 10 centimes est choisie 1 fois, il reste 250
 La pièce de 50 centimes est choisie 1 fois, il reste 200
 La pièce de 2 euro est choisie 1 fois, il reste 0
Au total on a utilisé 5 pièces
```

# Exercice 12.3- Algo Glouton

- Typez vos structures de données représentant les pièces et leur noms leur valeur
- Donnez un exemple de solution
- Identifier une solution intermédiaire, une solution optimale localement, une condition d’arrêt?

# Exercice 12.3- Algo Glouton

Comparez cet algorithme avec celui du pb du combriolage

```
def rendu_monnaie(somme_en_centimes: int, pieces_triees: Dict[int, int]) -> Dict[int, int]:
    ...
    quantite_de_pieces_choisies: Dict[int, int] = {
        un_centime_idx: 0, deux_centimes_idx: 0, cinq_centimes_idx: 0, dix_centimes_idx: 0,
        vingt_centimes_idx: 0, cinquante_centimes_idx: 0, un_euro_idx: 0, deux_euros_idx: 0}
    ...
    solde_courant: int = somme_en_centimes
    ...
    while solde_courant != 0:
        ...
        for key in pieces_triees.keys():
            if solde_courant >= pieces_triees[key]:
                quantite_de_pieces_choisies[key] += 1
                solde_courant -= pieces_triees[key]
                break
    return quantite_de_pieces_choisies
```

# Exercice 12.3- Algo Glouton

Supprimez la pièce de 1 centime, que se passe-t-il?

```python
def rendu_monnaie(somme_en_centimes: int, pieces_triees: Dict[int, int]) -> Dict[int, int]:
    ...
    quantite_de_pieces_choisies: Dict[int, int] = {
        un_centime_idx: 0, deux_centimes_idx: 0, cinq_centimes_idx: 0, dix_centimes_idx: 0,
        vingt_centimes_idx: 0, cinquante_centimes_idx: 0, un_euro_idx: 0, deux_euros_idx: 0}
    ...
    solde_courant: int = somme_en_centimes
    ...
    while solde_courant != 0:
        ...
        for key in pieces_triees.keys():
            if solde_courant >= pieces_triees[key]:
                quantite_de_pieces_choisies[key] += 1
                solde_courant -= pieces_triees[key]
                break
    return quantite_de_pieces_choisies
```

# Programmation dynamique

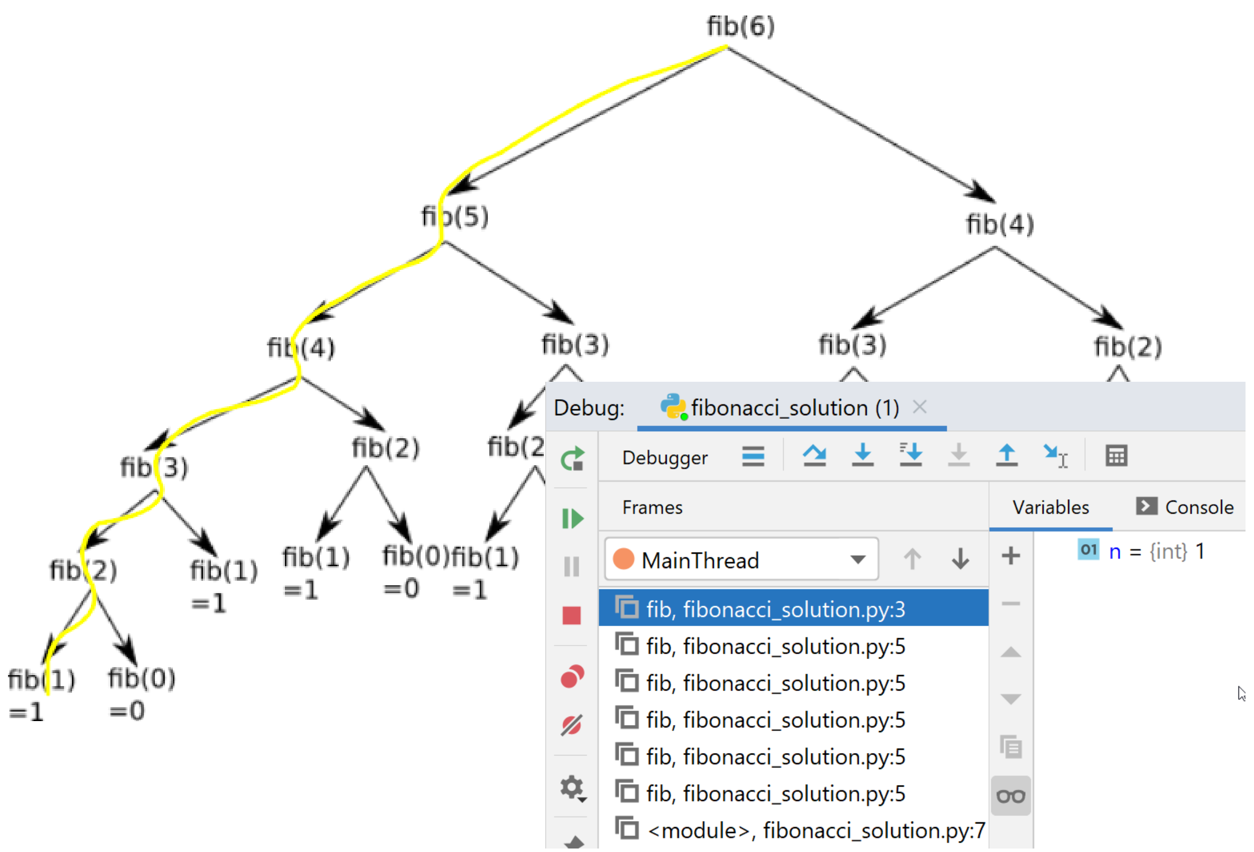
```python
def fib(n: int) -> int:
    if n < 2:
        return n
    else:
        return fib(n - 1) + fib(n - 2)

print(fib(6))
```

- Identifiez la condition d’arrêt et la relation de récurrence
- Représentez ce programme sous forme d’un arbre

# Programmation dynamique

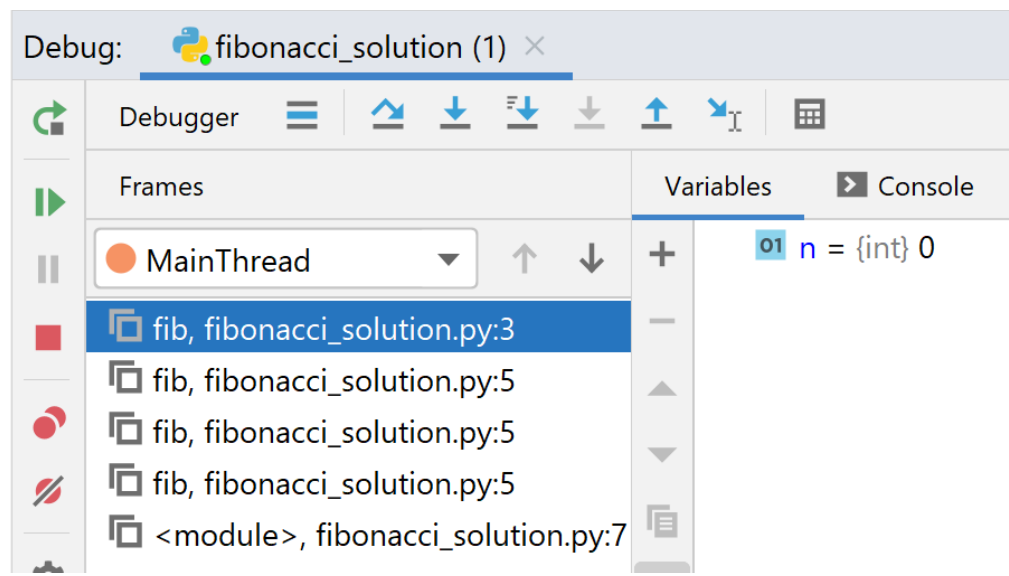
La pile d'exécution empile la fonction 6 fois avant d'arriver à une feuille (n=1)



# Programmation dynamique

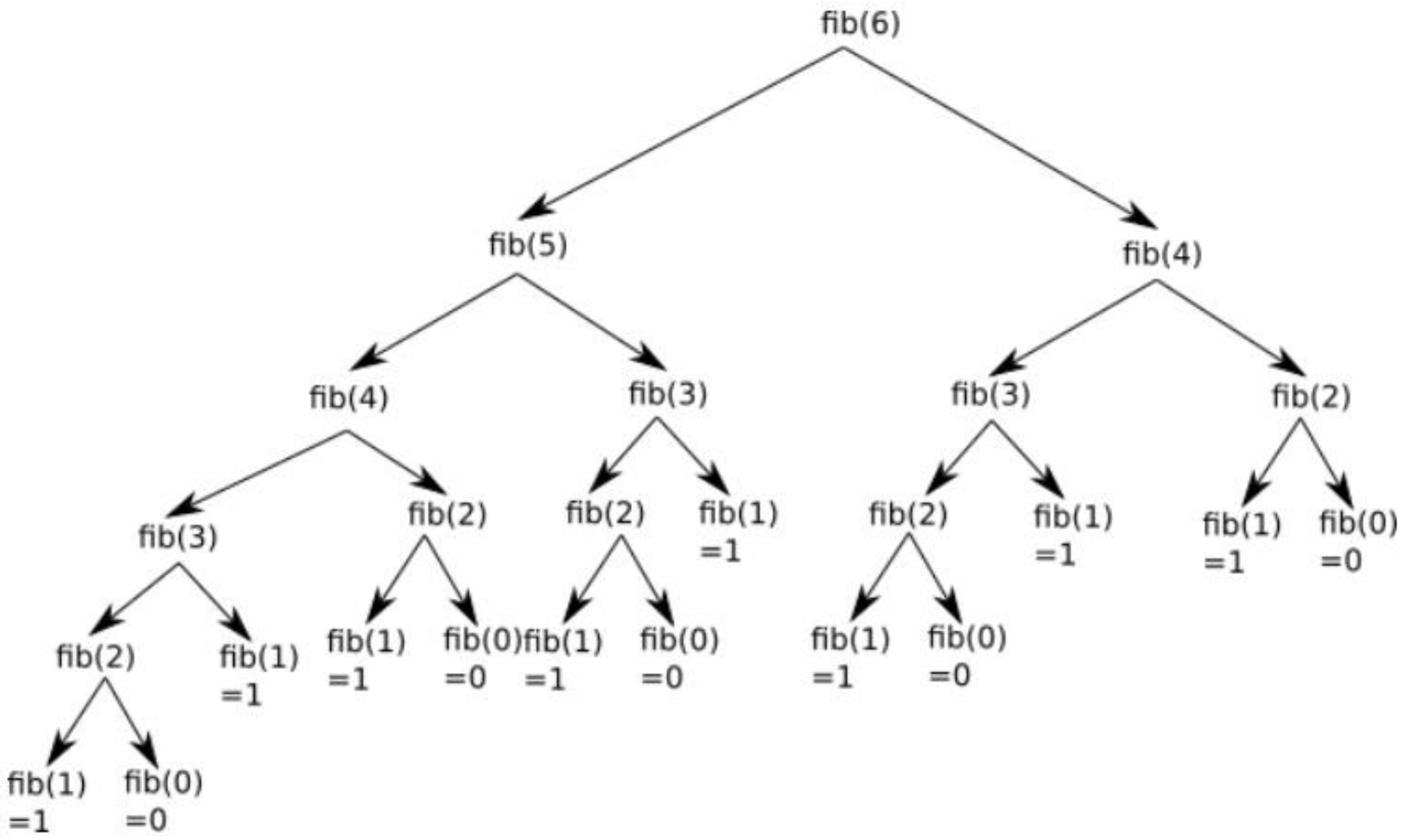
Obtient-on la même solution? Le même arbre?

```python
def fib(n: int)-> int:   n: 0
    if n < 2:
        return n
    else:
        #return fib(n - 1) + fib(n - 2)
        return fib(n - 2) + fib(n - 1)
```



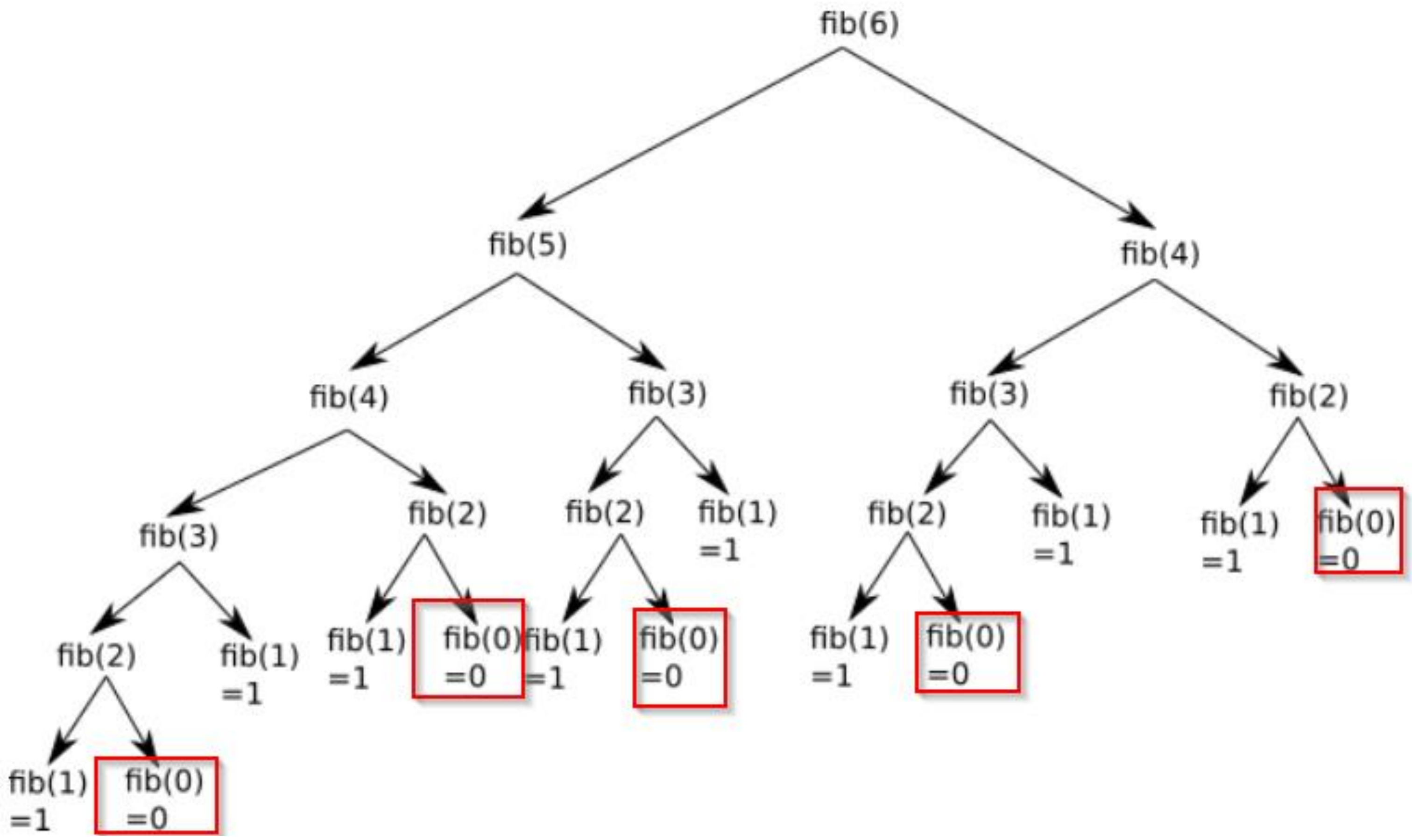
# Programmation dynamique

Combien de fois calcule-t-on fib(0)?



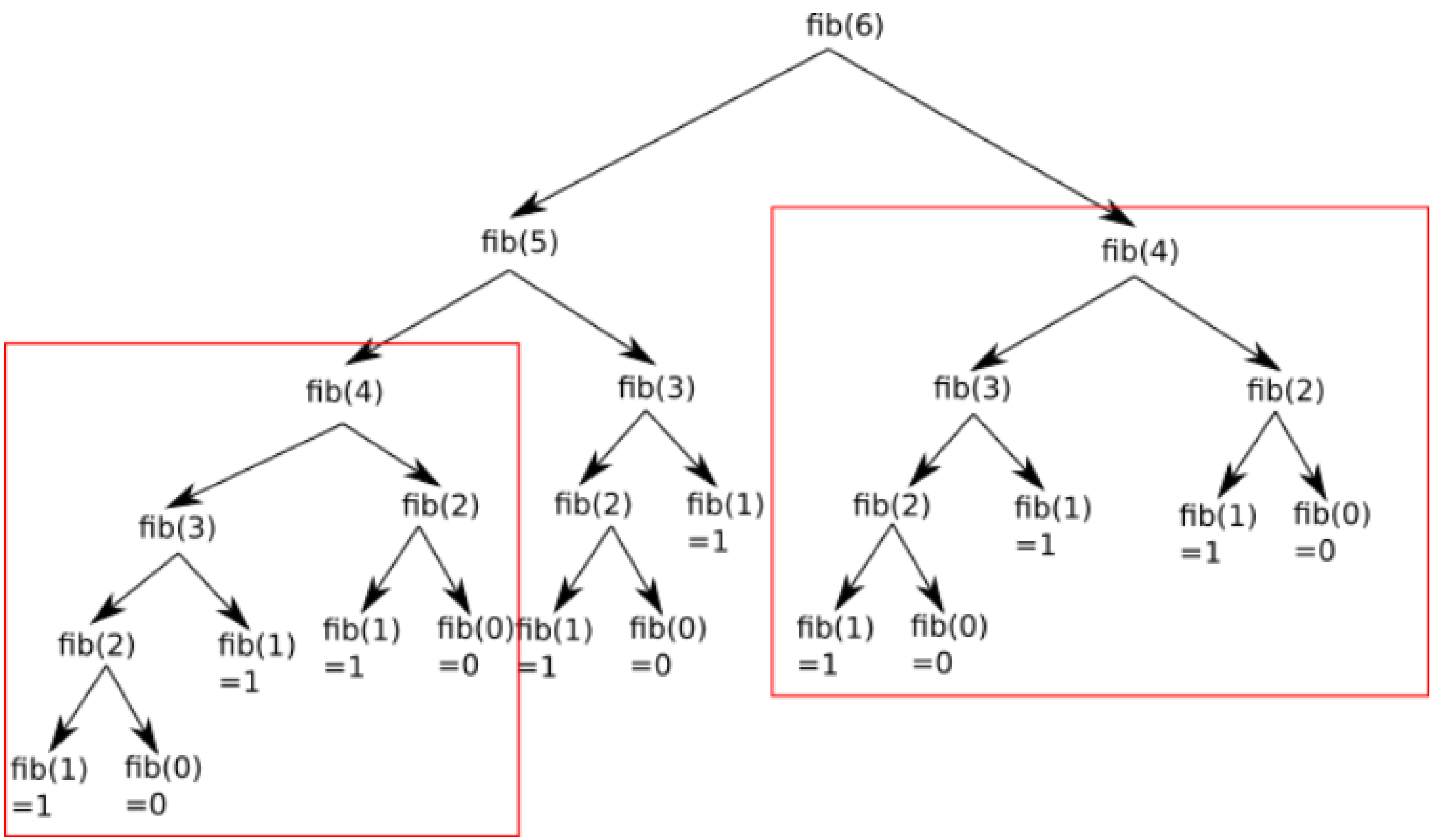
# Programmation dynamique

Combien de fois calcule-t-on fib(0)?



# Programmation dynamique

Combien de fois calcule-t-on fib(4)?



# Programmation dynamique

Comment compter le nombre d’appels redondants de n pour n>1?

```
{2: 5, 3: 3, 4: 2, 5: 1, 6: 1}
```

# Programmation dynamique

Comment mémoriser les calculs intermédiaires?

```
def fib_mem(n):
  mem = [0]*(n+1)  #permet de créer un tableau contenant n+1 zéro
  return fib_mem_c(n,mem)

def fib_mem_c(n,m):
  if n==0 or n==1:
    m[n]=n
    return n
  elif m[n]>0:
    return m[n]
  else:
    m[n]=fib_mem_c(n-1,m) + fib_mem_c(n-2,m)
    return m[n]
```

# Programmation dynamique

Améliorez ce code; typez-le, changer le nom des variables, améliorez les conditions d'arrêt, vérifier qu'on ne calcule pas plus d'1 fois fib(n)

```
8
{2: 1, 3: 1, 4: 1, 5: 1, 6: 1}
```

```python
def fib_mem(n):
  mem = [0]*(n+1)  #permet de créer un tableau contenant n+1 zéro
  return fib_mem_c(n,mem)

def fib_mem_c(n,m):
  if n==0 or n==1:
    m[n]=n
    return n
  elif m[n]>0:
    return m[n]
  else:
    m[n]=fib_mem_c(n-1,m) + fib_mem_c(n-2,m)
    return m[n]
```

# Programmation dynamique

[Richard Bellman, 1959] La programmation dynamique résout un problème en combinant des solutions de sous-problèmes (fonction récursive) résolus une seule fois (mémorisation de résultat intermédiaire)

# Rendu de monnaie - Relation de récurrence

- Si X = 0 alors Nb(X) = 0
- Si X > 0 alors
  Nb(X) = 1 + min(Nb(X-pi))
  avec $1 \leq i < n$ et $pi \leq X$

X est le montant à rembourser

Nb(X) est le nombre de pièces nécessaires pour rembourser X

pi est une pièce parmi n pièces disponibles

# Relation de récurrence

- P=[2, 5, 10, 50]
- Si X est remboursable il existe des entiers a, b, c, d et e tel que

$$X = 2a + 5b + 10c + 50d$$

- *Le nombre de pièces pour rembourser X est égal au nombre de pièces pour rembourser X -pi +1*

$$Nb(X) = a + b + c + d$$

$$Nb(X - 2) = a - 1 + b + c + d$$

$$Nb(X - 2) = Nb(X) - 1$$