6

La Programmation Dynamique

Extrait du programme

Thème : Algorithmique

Contenus :

Programmation dynamique.

Capacités attendus :

Utiliser la programmation dynamique pour écrire un algorithme.

Commentaires :

Les exemples de l'alignement de séquences ou du rendu de monnaie peuvent être présentés.

La discussion sur le coût en mémoire peut être développée.

I. La suite de Fibonacci

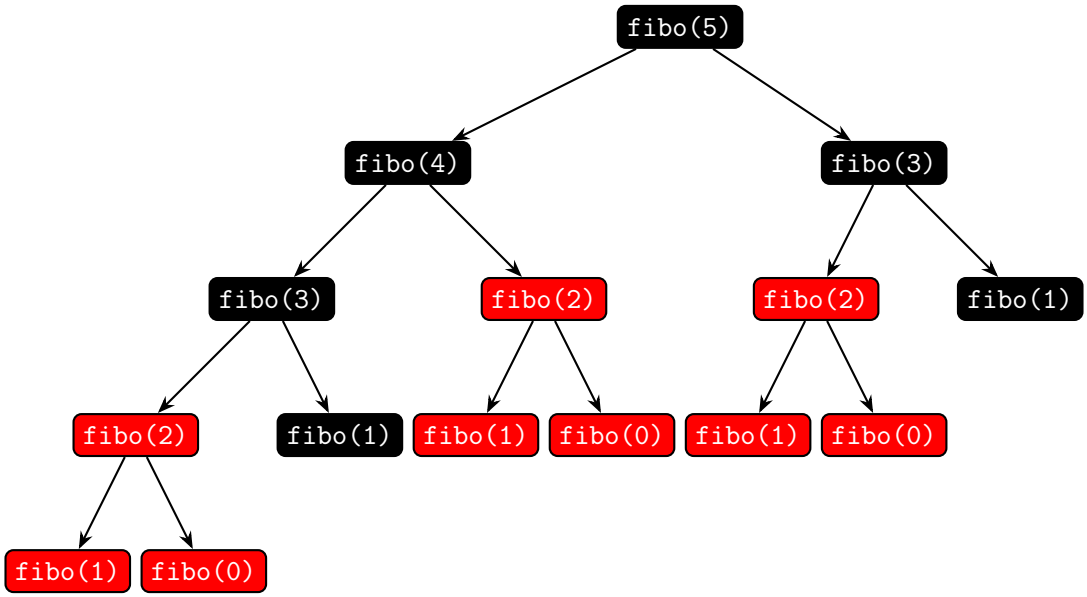
La suite de Fibonacci est définie par la récurrence suivante :

$$\begin{cases} \mathcal{F}_0 = 0 \\ \mathcal{F}_1 = 1 \\ \mathcal{F}_{n+2} = \mathcal{F}_{n+1} + \mathcal{F}_n \end{cases}$$

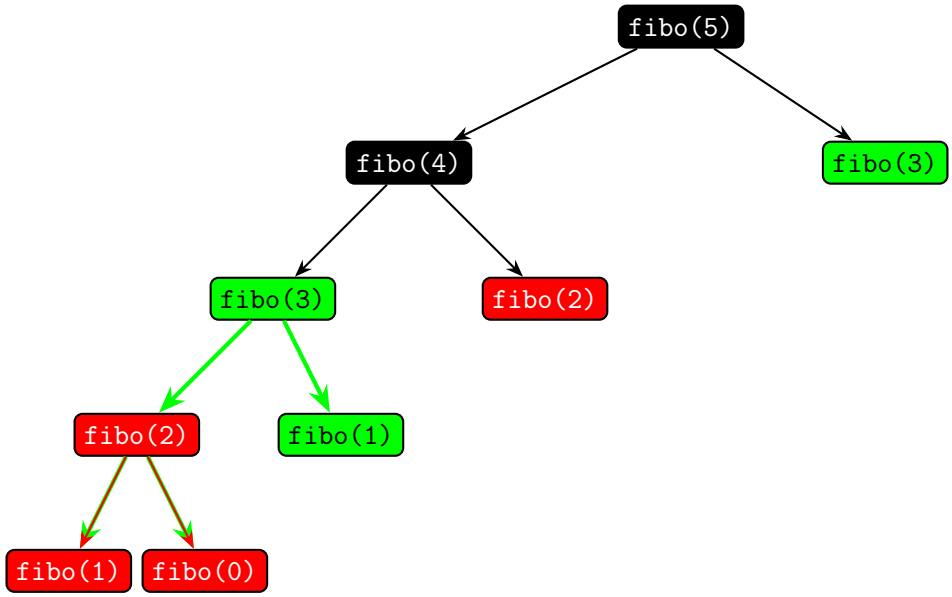
Cette définition par récurrence nous amène naturellement à programmer une fonction récursive pour calculer les termes de la suite :

```
1 def fibo(n):
2     if n < 2 :
3         return n
4     else :
5         return fibo(n-1) + fibo(n-2)
```

Voici l'arbre d'appels correspondant à `fibo(5)` :



On constate rapidement qu'il y a un problème car on effectue de multiples appels redondants. Ainsi `fibo(2)` est appelé 3 fois (en rouge), `fibo(3)` est quant à lui appelé 2 fois. Ce serait plus intéressant si on pouvait enregistrer les résultats intermédiaires. L'arbre d'appel serait alors le suivant :



Ainsi , dans le premier cas nous avons effectue en tout 14 appels pour calculer **fibo(5)**. Dans le deuxième cas, nous n'avons plus que 8 appels.
Voici deux versions, une itérative et une autre récursive de deux fonctions qui calculent le n-ième terme de la suite de Fibonacci en gardant au fur et à mesure la mémoire des calculs effectuées précédemment.

Fibonacci version dynamique iterative (méthode ascendante)

```
def fibo_dyn(n):
    #tableau qui va stocker les résultats
    f = [0]*(n+1)
    if n > 0:
        f[1] = 1
        for i in range(2,n+1):
            f[i] = f[i-2] + f[i-1]
    return f[n]
```

Fibonacci version dynamique récursive (méthode descendante)

```
def fibo(n):
    #tableau qui va stocker les résultats
    f = [None]*(n+1)
    return fibo_rec_dyn(n,f)

def fibo_rec_dyn(n,f):
    if n<=1 :
        f[n] = n
    else :
        f[n] = fibo_rec_dyn(n-1,f) + fibo_rec_dyn(n-2,f)
    return f[n]
```

II. Programmation dynamique

D'après mon ami Wikipédia, nous avons la définition suivante :

En informatique, la programmation dynamique est une méthode algorithmique pour résoudre des problèmes d'optimisation. Le concept a été introduit au début des années 1950 par Richard Bellman.
La programmation dynamique consiste **à résoudre un problème** en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands **en stockant les résultats intermédiaires**.

Dans le cas de la suite de Fibonacci, nous avons bien réduit le problème en le réduisant à des sous-problèmes beaucoup plus simples (additionner deux termes) tout en stockant les résultats intermédiaires.

III. Le problème du rendu de monnaie

Nous avions étudié ce problème en première. Il s'agit de rendre la monnaie avec le moins de pièce possible.
Nous avions vu qu'il existait un algorithme qui donne toujours la solution mais qui est peu efficace : la résolution par **la force brute**. Cette méthode consiste tout simplement à tester toutes les solutions les unes après les autres. Évidemment cette solution fonctionne, mais ... elle peut prendre beaucoup (trop !) de temps.

Nous avons alors chercher une solution par un algorithme glouton.

- Cet algorithme peut retrouver la bonne solution.
  Par exemple si on veut rendre 58€ avec des pièces de 10, 5, 2 et 1, la solution est bien la solution optimale : 5 pièces de 10€, 1 pièce de 5€, 1 pièce de 2€ et une pièce de 1€.
- Cet algorithme peut trouver une solution, mais elle n'est pas optimale.
  Par exemple si on veut rendre 12€ avec des pièces de 10, 6 et 1, la solution gloutonne est : 1 pièces de 10€ et 2 pièces de 1€ soit 3 pièces au total.
  Alors que la solution optimale est 2 pièces de 6€.
- Cet algorithme peut se tromper.
  Par exemple si on veut rendre 8€ avec des pièces de 5 et 2, la solution gloutonne est : 1 pièces de 5€ et 1 pièce de 2€ soit un total de 7€.
  Dans ce cas, il n'y a pas de solution mais l'algorithme glouton en propose néanmoins une [1].

```
def rendu_monnaie_glouton(pieces, s) :
    """
    Entrées :
        pieces est un tableau d'entiers ordonnées (décroissant)
        s est la somme à rendre
    Sortie :
        M est le tableau des multiplicités pour chaque pièce, dans le même ordre
    """

    # création du tableau des mutliplicités
    M = [0]*len(pieces)
    reste = s
    i = len(pieces)-1
    while reste > 0 and i >= 0 :
        if reste - pieces[i] >= 0:
            # on peut rendre la pièce i
            reste = reste - pieces[i]
            M[i] = M[i] + 1
        else :
            i = i - 1
    return M
```

1. Il serait facile de rajouter au programme une ligne pour vérifier que la solution proposée n'est pas fausse. Malgré tout, même fausse, la solution gloutonne peut servir de première approximation à notre recherche.

Une solution récursive :

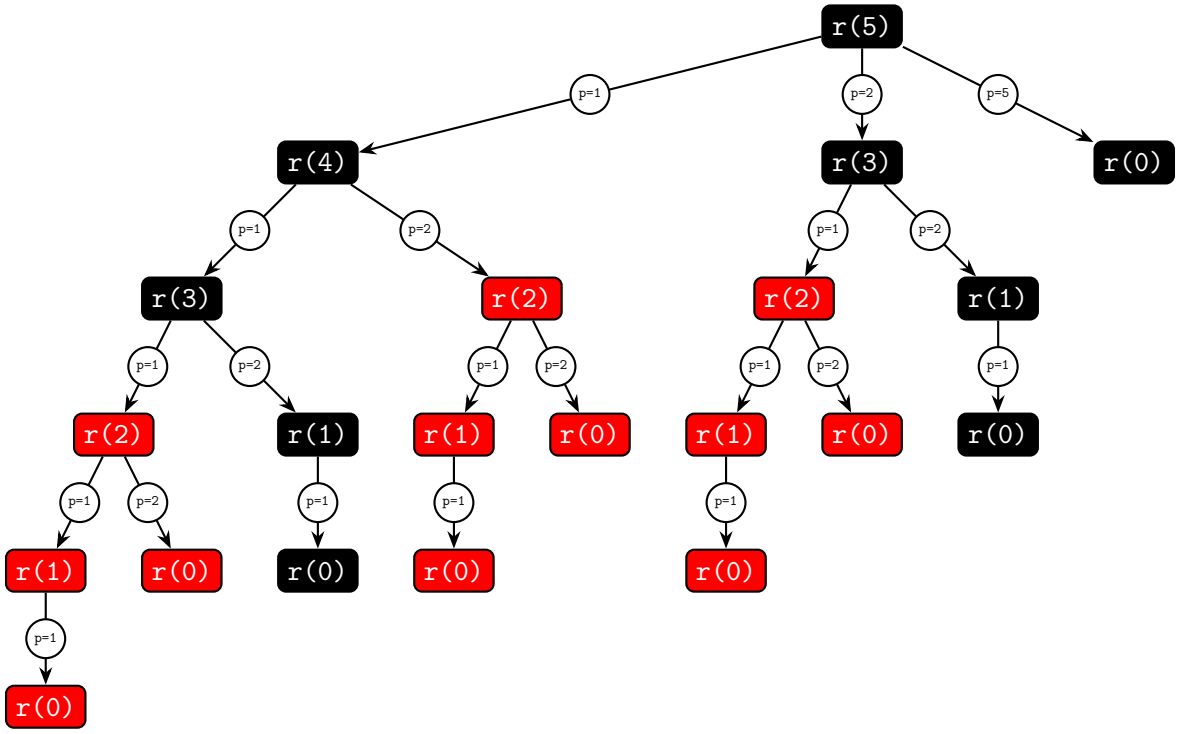
Tout d'abord, pour être certain d'avoir une solution, nous aurons toujours des pièces de valeur 1€. En effet, au pire, si on doit rendre la somme `n`, il nous suffit d'avoir `n` pièces de 1€. Mais ce n'est pas forcément optimale.

Ensuite pour résoudre le problème de recherche d'un nombre de pièces optimal, nous allons procéder par récursivité et dans un premier temps, nous allons uniquement rechercher le nombre de pièces. Pour le détail des pièces à rendre on verra plus tard.

- Le cas de base est la somme `s` à rendre est nulle. Donc on ne rend rien.
- Sinon, par défaut, on prend le pire cas, autrement dit, `s` fois la pièce de 1 (ligne 12).
  Ensuite pour chaque pièce `p`, si elle est bien inférieure à la somme à rendre `s`, on compare entre le nombre de pièces `nbr` et celui que l'on pourrait obtenir (par récursivité) en ne rendant plus que la somme `s-p` (ligne 15).

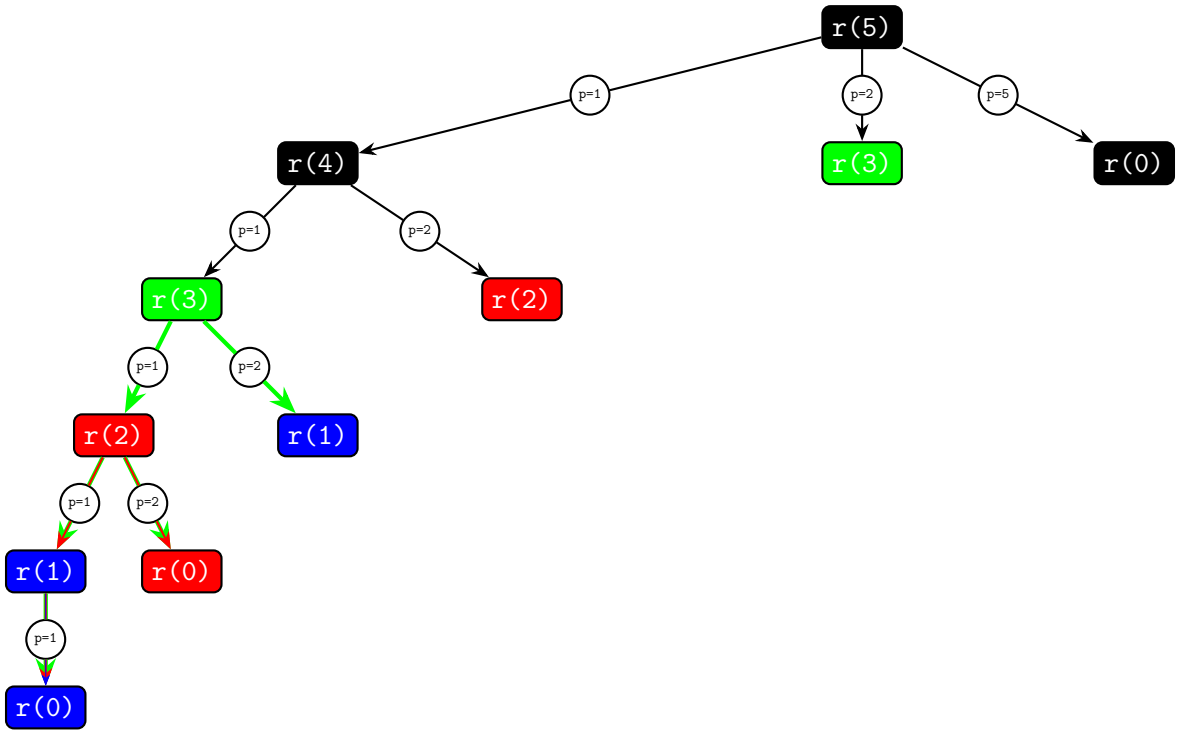
```
1  def rendu_monnaie_recursif(pieces, s) :
2      """
3      Entrées :   pieces est un tableau d'entiers ordonnées (croissant),
4                          s la somme à rendre
5          Sortie : nbr est le nombre de pièces à rendre
6      """
7
8      if s == 0:
9          return 0
10     else :
11         # pire cas s = 1 + 1 + 1 + ... + 1
12         nbr = s
13         for p in pieces :
14             if p <= s :
15                 nbr = min(nbr, 1+rendu_monnaie_recursif(pieces, s-p))
16         return nbr
```

Examinons l'arbre des appels pour une somme `s` de 5 avec les pièces `[1, 2, 5, 10]` (pour simplifier, on écrira `r(5)` au lieu de `rendu_monnaie_recursif([1, 2, 5, 10], 5)`).



On remarque par exemple que `r(2)` qui renvoie le nombre minimum de pièces pour rendre la somme de 2 est calculé 3 fois (en rouge). De même `r(1)` est calculé 5 fois ou encore `r(3)` est calculé 2 fois.

Un autre exemple très parlant, `rendu_monnaie_recursif([1, 2], 47)` demanderait plus de 12 milliards d'appels[2] ! Avec une version dynamique, autrement dit, en mémorisant les résultats intermédiaires, nous aurions le graphe des appels suivants :



Une version dynamique :
Voici une version dynamique du problème du rendu de monnaie.

```
1  def rendu_monnaie_dynamique(pieces, s) :
2      """
3      Entrées :
4          pieces est un tableau d'entiers ordonnées (croissant)
5          nbr est le tableau contenant le nombre de pièces à rendre pour chaque
       nombre inférieur ou égal s
6      """
7
8      nbr = [0]*(s+1)
9      for n in range(1,s+1):
10         #pire cas nbr = 1 + 1 + 1 + ... + 1
11         nbr[n] = n
12         for p in pieces :
13             if p <= n :
14                 nbr[n] = min(nbr[n], 1 + nbr[n-p])
15     return nbr[s]
```

Il ne « reste » plus qu'à rajouter la création d'un tableau qui enregistre également de manière dynamique, les propositions de rendu de monnaie dans les cas déjà résolus.
Ainsi le tableau `sol` sera composé de tableaux contenant toutes les solutions optimales intermédiaires. Par exemple pour un rendu de monnaie de 8€ nous obtenons le tableau suivant :
`[[], [1], [2], [2, 1], [2, 2], [5], [5, 1], [5, 2], [5, 2, 1]]`.
Comment l'interpréter ?
`sol[8]` donne la solution optimale `[5, 2, 1]` autrement dit pour rendre 8€ on rend une pièce de 5, une pièce de 2 et une pièce de 1.
Mais on a aussi toutes les solutions optimales pour les rendus inférieurs à 8.

2. Pour ceux qui sont intéressés par la démonstration, on y fait référence à la suite de Fibonacci. Ainsi `rendu_monnaie_recursif([1, 2], n)` se fait avec $\mathcal{F}_{n+3} - 1$ appels et $\mathcal{F}_{50} = 12\ 586\ 269\ 025$.

`sol[4]` donne la solution optimale `[2, 2]` autrement dit pour rendre 4€ on rend une pièce de 2 et une autre pièce de 2.
`sol[0]` donne la solution optimale `[]` autrement dit pour rendre 0€ on ne rend aucune pièce.
...

```
def rendu_monnaie_dynamique(pieces, s) :
    """
    Entrées :
        pieces est un tableau d'entiers ordonnées (croissant)
        nbr est le tableau contenant le nombre de pièces à rendre pour chaque nombre inférieur ou égal s
        sol est la solution
    """

    nbr = [0]*(s+1)
    sol = [[]]*(s+1)
    for n in range(1,s+1):
        #pire cas nbr = 1 + 1 + 1 + ... + 1
        nbr[n] = n
        sol[n] = [1]*n
        for p in pieces :
            if p <= n and 1 + nbr[n-p] < nbr[n] :
                nbr[n] = 1 + nbr[n-p]
                sol[n] = sol[n-p].copy()
                sol[n].append(p)
                print(sol)
    return sol[s]
```