Chapitre 3

Programmation dynamique

Sommaire

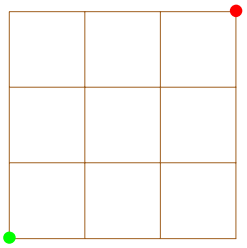
3.1 Introduction

NSI Tle - JB Duthoit

3.1.1 Un exemple pour comprendre !

Approche

On désire relier le point rouge à partir du point vert. On ne peut se déplacer que sur les traits horizontaux vers la droite et le long des traits verticaux vers le haut.



☛ Combien existe-t-il de chemins différents ?

Exercice 14.17
combien y a-t-il de chemins différents sur une grille de 10×10 ?
***

☛ De façon générale, la programmation dynamique est une technique qui évite de ne pas calculer la même chose plusieurs fois. Dans l'exemple ci-dessus, on utilise un tableau à deux dimensions pour stocker les nombres de chemins déjà calculés...

3.1.2 Un autre exemple avec la suite de fibonnaci

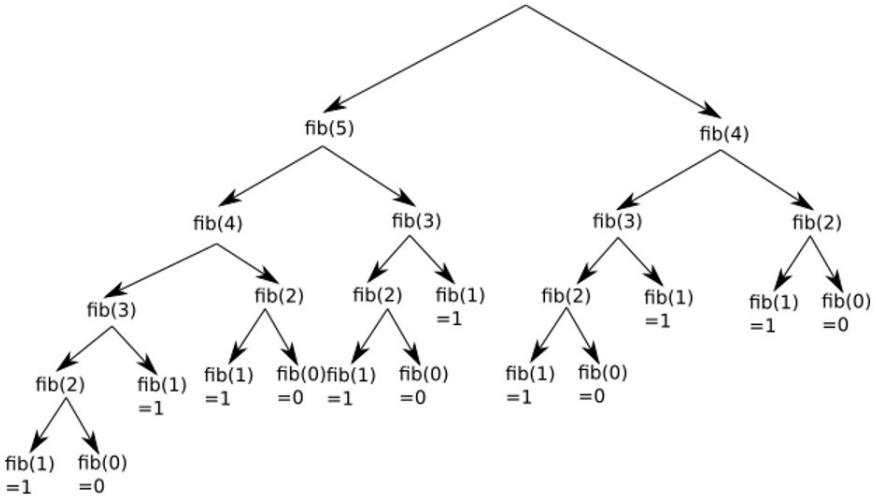
Il s'agit de la suite de Fibonacci. Rappelons la formule de récurrence définissant cette suite :

$$u_n = \begin{cases} 0 \; si \; n = 0 \\ 1 \; si \; n = 1 \\ u_{n-1} + u_{n-2} \; si \; n \geq 2 \end{cases}$$

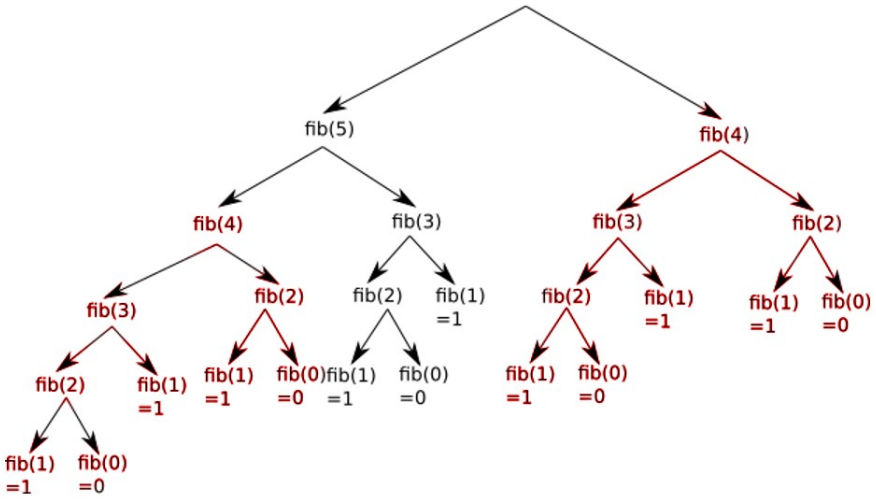
Ses premiers termes sont donc 0, 1, 1, 2, 3, 5, 8, 13, 21, etc.

Exercice 14.18
construire l'algorithme récursif d'une fonction calculant le terme d'indice n de la suite. Le traduire en programme python.
***

Pour n=6, il est possible d'illustrer le fonctionnement de ce programme avec le schéma ci-dessous :



On peut alors que constater que plusieurs appels sont réalisés avec une même valeur du paramètre (typique dans les algorithmes récursifs), si on additionne toutes les feuilles de cette structure arborescente (fib(1)=1 et fib(0)=0), on retrouve bien 8.



En observant attentivement le schéma ci-dessus, on remarque que de nombreux calculs sont inutiles, car effectué 2 fois : par exemple on retrouve le calcul de fib(4) à 2 endroits (en haut à droite et un peu plus bas à gauche). On réalise ainsi un grand nombre d'opérations inutiles car redondantes. Celles-ci sont très coûteuses et expliquent la complexité exponentielle de cet algorithme.

On pourrait donc grandement simplifier le calcul en calculant une fois pour toutes fib(4), en "mémorisant" le résultat et en le réutilisant quand nécessaire.

Cela n'est bien sûr possible que si les sous-problèmes ne sont pas indépendants. Cela signifie donc que ces sous-problèmes ont des sous-sous-problèmes communs.
Dans le cas qui nous intéresse, on peut légitimement s'interroger sur le bénéfice de cette opération de "mémorisation", mais pour des valeurs de n beaucoup plus élevées, la question ne se pose même pas, le gain en termes de performance (temps de calcul) est évident. Pour des

valeurs n très élevées, dans le cas du programme récursif "classique" (n'utilisant pas la "mémorisation"), on peut même se retrouver avec un programme qui "plante" à cause du trop grand nombre d'appels récursifs. La méthode que nous venons d'utiliser se nomme "programmation dynamique".

La programmation dynamique, comme la méthode diviser pour régner, résout des problèmes en combinant des solutions de sous-problèmes. Cette méthode a été introduite au début des années 1950 par Richard Bellman.

3.2 La programmation dynamique

NSI Tle - JB Duthoit

La programmation dynamique s'applique généralement aux problèmes d'optimisation.
Sa mise en pratique peut prendre deux formes :

3.2.1 Forme "Top down", dite de mémoïsation

On utilise directement la formule de récurrence.
Lors d'un appel récursif, avant d'effectuer un calcul on regarde dans le tableau de mémoire cache si ce travail n'a pas déjà été effectué.

3.2.2 forme "Bottom Up"

On résout d'abord les sous problèmes de la plus "petite taille", puis ceux de la taille "d'au dessus", etc
Au fur et à mesure on stocke les résultats obtenus dans le tableau de mémoire cache.
On continue jusqu'à la taille voulue.

3.2.3 Application sur la suite de Fibonacci

Nous allons maintenant reprendre l'exemple de la suite de Fibonacci et lui appliquer les deux méthodes précédentes.

Un algorithme "Top down" pour la suite de Fibonacci

Exercice 14.19
Notre mémoire cache sera ici une liste.

Rappelons que son rôle va être de mémoriser les résultats des sous-problèmes de tailles inférieures à celui du problème à résoudre.

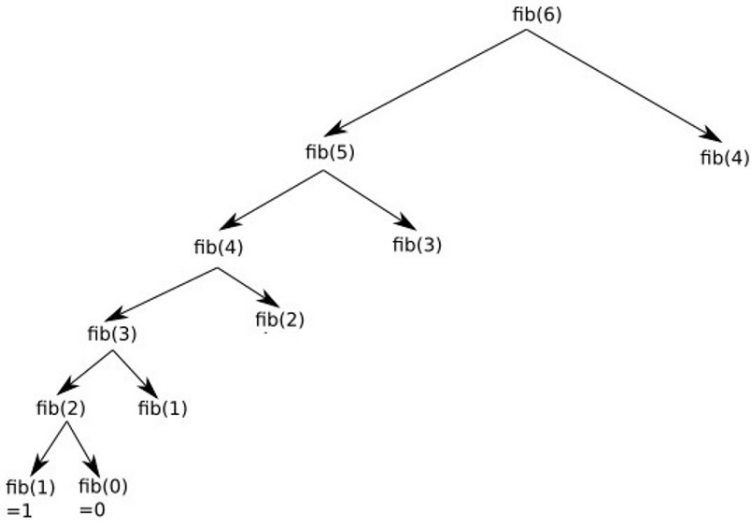
Pour la suite de Fibonacci, si l'on veut calculer le terme de rang n, il nous faudra ainsi mémoriser les termes d'indices 1,2,..., n+1. Cette liste aura donc n + 1 éléments.

☛ Construire l'algorithme "Top down" pour la suite de Fibonacci. L'implémenter ensuite en langage python.
☛ Comparer les vitesses d'exécution des deux programmes. ***

Il faut bien comprendre qu'il s'agit quasiment de la fonction récursive vue en introduction. La seule différence, mais bien sûr majeure au niveau de l'efficacité, réside dans la condition "si table[n+1] = None". Elle permet de vérifier dans la mémoire cache si le terme en question de la suite à déjà été calculé ou non. Si oui on le retourne et la fonction prend fin, sinon on le calcule récursivement, on stocke sa valeur dans la mémoire cache et on la retourne.

Il est assez facile de voir que la complexité de cette fonction n'est plus exponentielle comme dans sa première version mais linéaire. Moralement il faut en effet remplir chacune des n + 1

cases de la mémoire cache, et ce à coût constant.



La différence avec le premier arbre est flagrante, et heureusement d'ailleurs puisque l'on a tout fait pour. Dès qu'un appel récursif se fait avec une valeur déjà calculée, les appels suivants n'ont pas lieu.

Un algorithme Bottom Up pour la suite de Fibonacci

Puisqu'elle a le même rôle, il est logique que la mémoire cache soit comme dans le cas Top Down une liste à n + 1 éléments.
La différence est que cette liste ne vas plus se remplir récursivement, en partant du rang n et en décrémentant jusqu'à 0 ou 1, mais itérativement, en partant cette fois de 0 et 1 et en incrémentant jusqu'à n.

Voici l'algorithme correspondant :

```
1  VARIABLE
2  n : entier
3  table : tableau d'entiers DEBUT
4  Function fib_ bottom _ up (n)
5  |   table ← [0] * (n+1)
6  |   table[1] ← 1
7  |   for i allant de 2 à n do
8  |   |   table[i] ← table[i-1] + table [i-2]
9  |   end
10 |   Retourner table[n]
11 end
```

Là aussi il est facile de voir que la complexité est linéaire.

On verra sur des exemples plus délicats qu'une des différences entre les deux approches réside dans le fait que dans un algorithme Bottom Up on résout tous les sous-problèmes de taille inférieure, alors que dans un algorithme Top Down on ne résout que ceux nécessaires.

Savoir-Faire 3.1

Implémenter ces programmes, et comparer les durées d'exécution.

3.3 Le problème du rendu de monnaie

NSI Tle - JB Duthoit

Le problème relativement simple du rendu de monnaie va nous permettre dans cette partie de bien appréhender tous les concepts de la programmation dynamique présentés auparavant.

On considère un système de pièces de monnaie.

La question est la suivante : quel est le nombre minimal de pièces à utiliser pour rendre une somme donnée ? De plus, quelle est la répartition des pièces correspondante ?

Savoir-Faire 3.2

Dans la zone euro, le système actuellement en circulation est S =(1,2,5,10,20,50,100,200,500). Quelle est la solution optimale pour rendre 6 euros (recherche déconnectée) ?

Demandons-nous ce qu'un humain ferait dans une telle situation. Il commencerait sans doute par rendre la plus grande pièce "possible", puis ferait de même avec le reste jusqu'à ce que la somme soit rendue. C'est d'ailleurs ce que font des millions de commerçants quotidiennement.

D'un point de vue algorithmique cela donne :

- Choisir la plus grande pièce du système de monnaie inférieure ou égale à la somme à rendre.
- Déduire cette pièce de la somme.
- Si la somme n'est pas nulle recommencer à l'étape 1.

Savoir-Faire 3.3

- Écrire cet algorithme. On donne le prototype :
  - fonction rendu_ glouton(liste : tableau d'entiers (liste des pièces ordonnées) , somme : entier (somme à rendre))
  - Retourne un entier (le nombre de pièce à rendre)
- L'implémenter en python
- Valider les tests unitaires suivants :
  - rendu_ glouton([1,2,5,10,50,100], 177) == 6
  - rendu_ glouton([1,3,4], 6) == 3

Remarque

Ce type d'algorithme s'appelle **algorithme glouton**, il a été étudié en première !
Mais cet algorithme n'est pas toujours optimal...

Savoir-Faire 3.4

On doit rendre 6 euros avec le système de billet suivant : Par contre, pour rendre cette même somme avec le système (1,3,4).

- Proposer la solution obtenue par un algorithme glouton
- Est-ce la solution optimale ?

3.3.1 Approche naïve

Dans un premier temps, on va s'intéresser uniquement au nombre minimal de pièces à rendre. On reconstituera la répartition correspondante plus tard.

Formalisons un peu ce problème avec quelques notations mathématiques :

- Le système de pièces de monnaie peut être modélisé par un n-uplet d'entiers naturels $S = (p_1, p_2, ..., p_n)$, où p_i représente la valeur de la pièce i.
- On suppose que $p_1 = 1$ et que $p_1 < p_2 < ... < p_n$.
- Une somme à rendre est un entier naturel X.
- Une répartition de pièces est un n-uplet d'entiers naturels $(x_1, x_2, ..., x_n)$, où x_1 représente le nombre de pièces p_1, x_2 le nombre de pièces p_2, etc.
- Le nombre total de pièces d'une telle répartition est donc $\sum_{i=0}^{n} x_i$

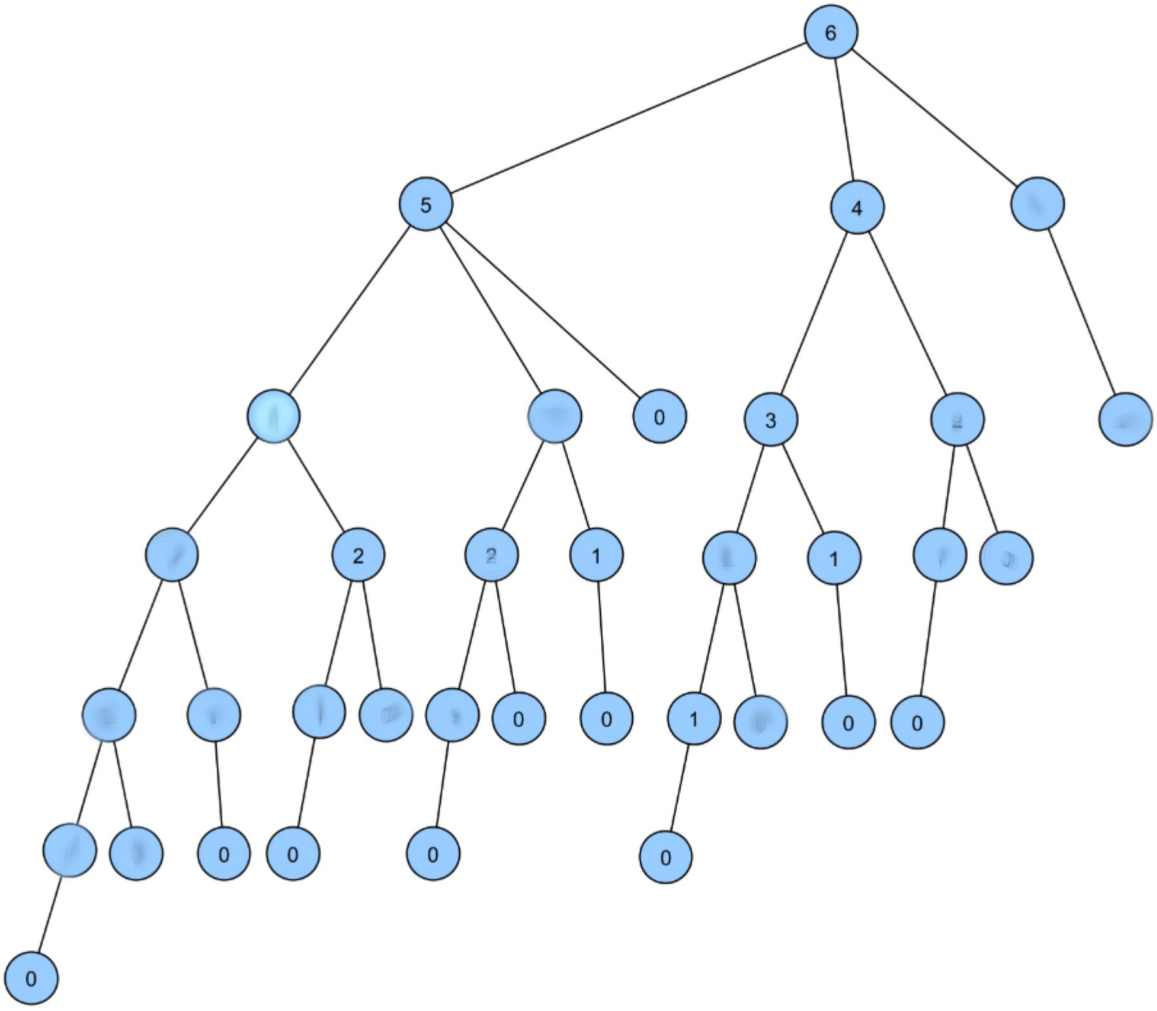
Pour une somme X, on va noter C[X] le nombre minimal de pièces.

On peut se demander quelles sont les sommes inférieures à X obtenables à partir de X.

On peut choisir de rendre d'abord p1, ou p2, ou p3, etc. Ces sommes sont donc X – p1, X – p2, ..., X – pn.

Savoir-Faire 3.5

Considérons l'arbre suivant qui matérialise ce rendu de monnaie. Les valeurs indiquées dans les noeuds correspondent à la somme restant à rendre.
Compléter cet arbre :



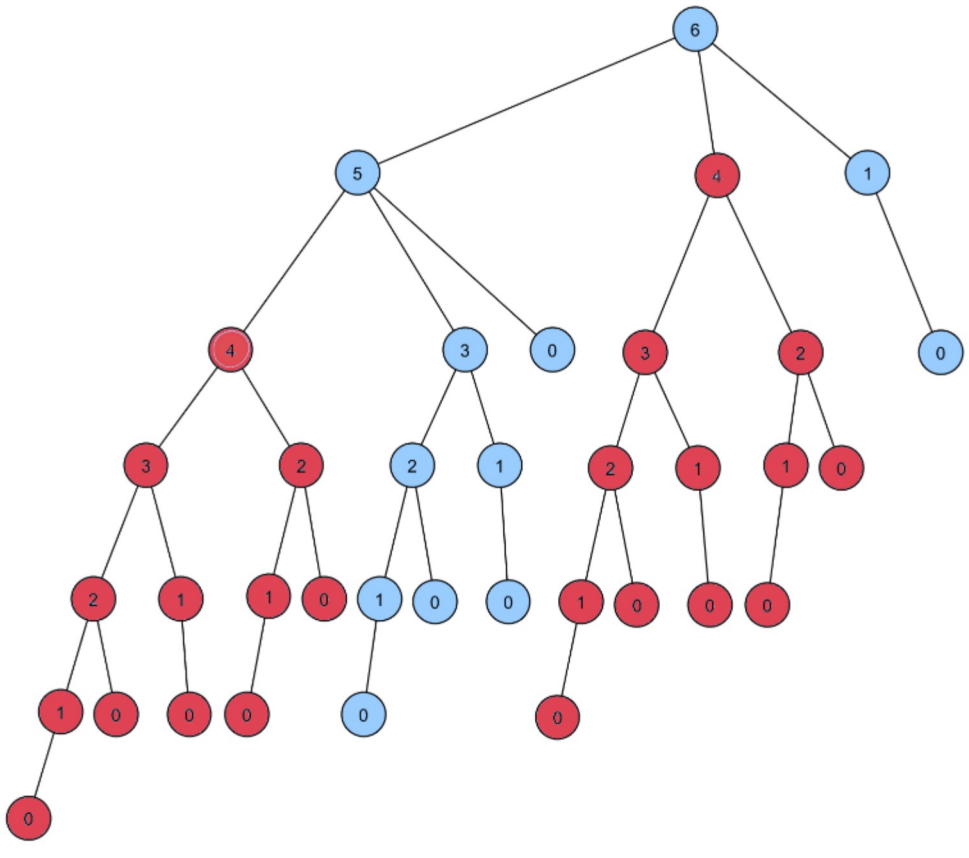
Si l'on sait comment rendre chacune de ces sommes de façon optimale, on saura le faire également pour X. Il suffira de prendre la meilleure de ces possibilités, i.e. celle correspondant à un plus petit nombre de pièces, et de rajouter 1.
Ce +1 correspondant au choix de la première pièce.
La condition d'arrêt à la récursivité sera bien sûr l'obtention d'une somme nulle.
Grâce aux éléments précédents, nous pouvons maintenant présenter cette formule de récurrence :

$$C[X] = \begin{cases} 0 \; si \; X = 0 \\ 1 + \min\limits_{\substack{1 \leq i \leq n \\ p_i < X}} C[X - p_i] \end{cases}$$

Voici l'algorithme :

```
1  VARIABLE
2  X : entier : la somme à rendre
3  S : tableau d'entiers de longueur l : liste des pièces dispos
4  DEBUT
5  Function rendu_ monnaie_ recursif (S,X)
6      if X == 0 then
7          retourner 0
8      else
9          mini ← X + 1
10         for i de 0 à l-1) do
11             if S[i]>X then
12                 nb ← 1 + rendu_ monnaie _ recursif(S,X - S[i])
13             end
14             if nb < mini then
15                 mini ← nb
16             end
17         end
18         Retourner mini
19     end
20 end
```

On remarque donc de multiples appels redondants (un appel redondant en rouge sur l'image suivante), et ce même si notre paramètre initial était petit.



3.3.2 Approche dynamique Top Down

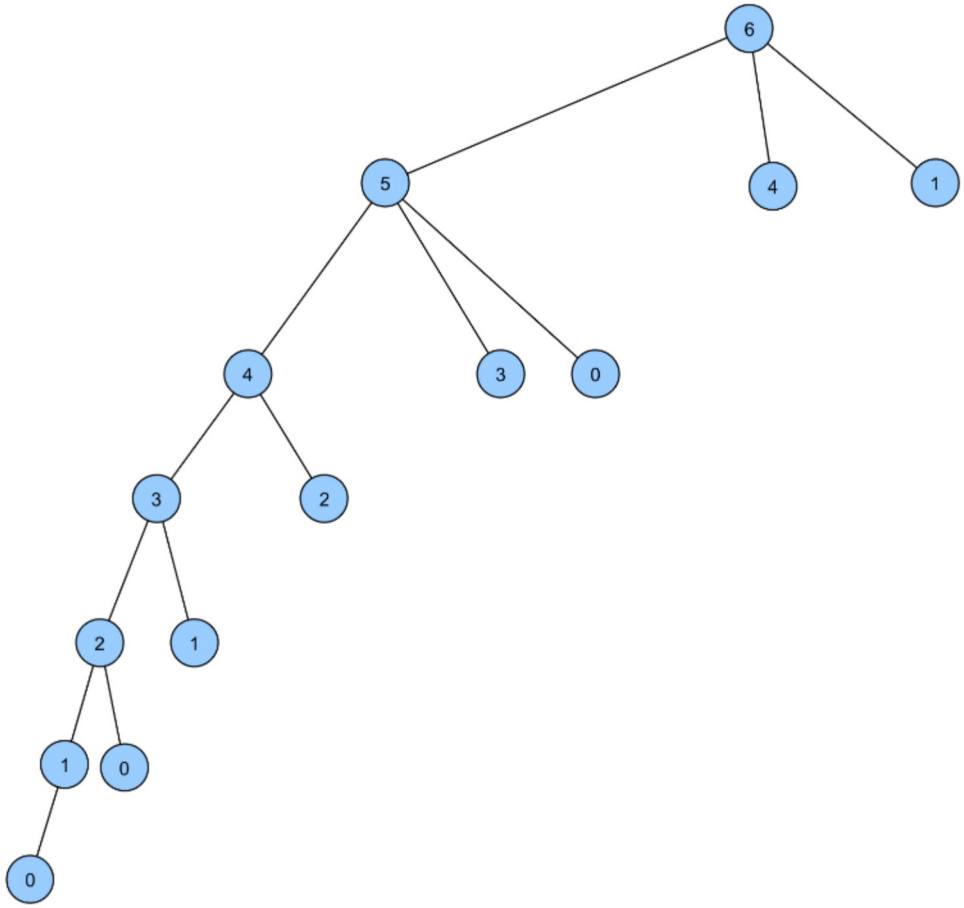
On va adopter dans cette sous-partie une technique de mémoïsation.

Rappelons-en brièvement le principe : au lieu de recalculer plusieurs fois les solutions des mêmes sous-problèmes, on va les mémoriser dans une mémoire cache

Pour une somme X, il va ainsi falloir enregistrer les résultats pour les sommes 0, 1, ..., X - 1. La mémoire cache, que l'on notera table, sera donc une liste unidimensionnelle à X + 1 éléments. Pour $0 \leq x \leq X$, table[x+1] sera donc égal au nombre de pièces minimal que l'on doit utiliser pour rendre une somme x. La solution à notre problème initial étant alors table[X+1]. Avec une approche Top Down, on va construire cette liste table de façon récursive en partant de notre somme initiale X. Cette fonction sera donc très proche de la version naïve et peu efficace présentée dans la sous-partie précédente. La seule différence est que lors d'un appel récursif qui n'est pas terminal, on va se demander si la valeur en question n'a pas déjà été calculée en regardant dans notre mémoire cache. Si oui on la retourne, sinon on la calcule par récursivité et on met à jour notre mémoire cache pour ne pas avoir à effectuer de nouveau ce calcul lors d'un appel postérieur.

```
VARIABLE
X : entier : la somme à rendre
S : tableau d'entiers de longueur l : liste des pièces dispos
table : tableau
DEBUT
Function rendu_ monnaie_ top_ down (S,X)
    table ← [0] * (n+1)
    Retourner (S,X,table)
end
Function rendu_ monnaie_ table (S,X,table)
    if X=0 then
        Retourner 0
    else
        if table[X]>0 then
            Retourner table[X]
        else
            mini ← x+1
            for i de 0 à l-1 do
                if S[i]<X then
                    nb = 1 + rendu_ monnaie_ table(S,X-S[i],table)
                    if nb < mini then
                        mini ← nb
                        table[X] ← mini
                    end
                end
            end
        end
    end
end
```

Voici l'arbre construit avec l'algorithme précédent. On voit bien que les redondances ont disparu !



Savoir-Faire 3.6

Compléter ce tableau, qui correspond à notre table, pour S = 11.

0	1	2	3	4	5	6	7	8	9	10	11

3.3.3 Approche dynamique Bottom Up

Pour une somme de X, notre mémoire cache sera comme dans le cas Top Down une liste à X+1 éléments.

La différence est qu'avec une approche Bottom Up, on va remplir cette fois notre liste de façon itérative en partant de la plus petite valeur possible à rendre

Le calcul des différents éléments de table provenant lui toujours de la formule de récurrence.

Comme précédemment, la solution à notre problème initial sera table[X].

Voici la fonction adoptant cette approche Bottom Up :

```
1  VARIABLE
2  X : entier : la somme à rendre
3  S : tableau d'entiers de longueur l : liste des pièces dispos
4  table : tableau
5  DEBUT
6  Function rendu_ monnaie_ bottom_ up (S,X)
7      table ← [0] * (n+1)
8      for i de 1 à X do
9          mini ← X
10         for i de 1 à l-1 do
11             if S[i] <= X and (1+table[X-S[I])<mini then
12                 mini = 1 + table[X - S[i]]
13             end
14         end
15         table[X] ← mini
16     end
17     Retourner table[X]
18 end
```