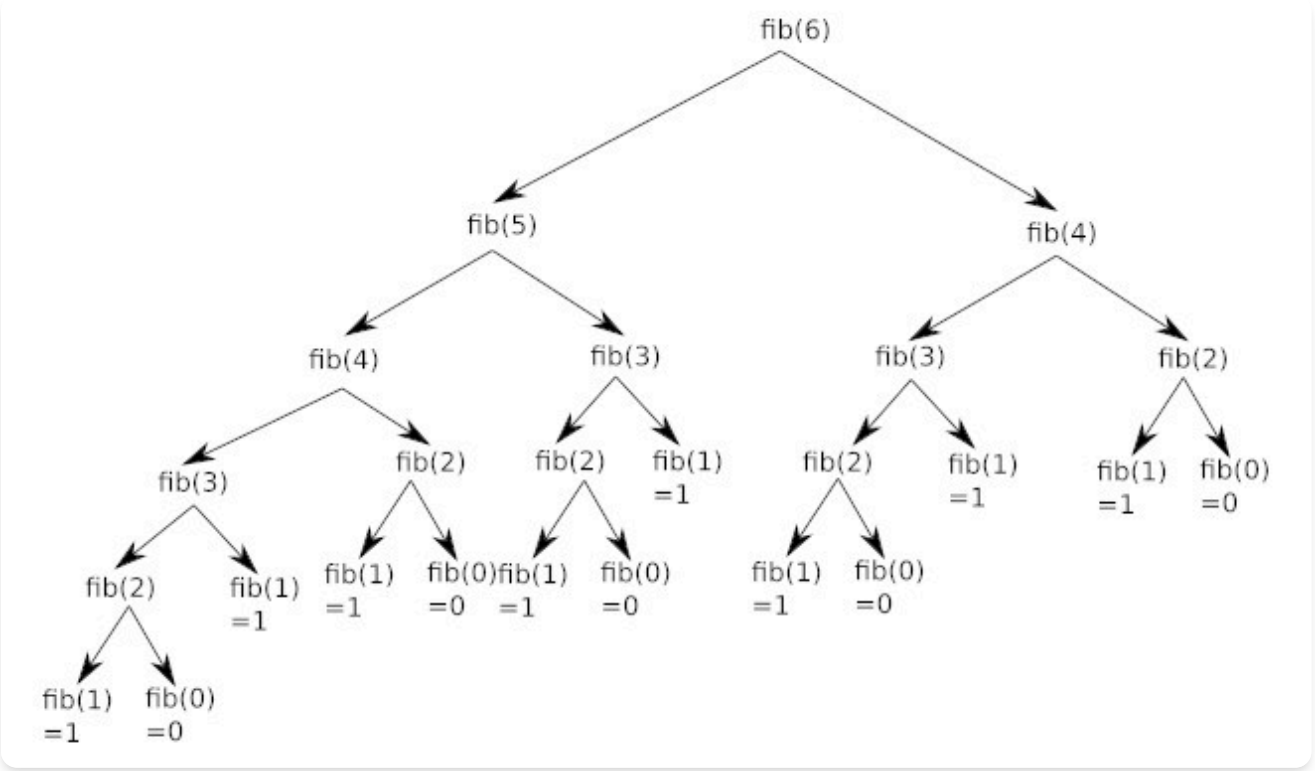
1) Suite de Fibonacci

Dans le chapitre sur la récursivité, nous avons étudié la programmation de la suite de Fibonacci :

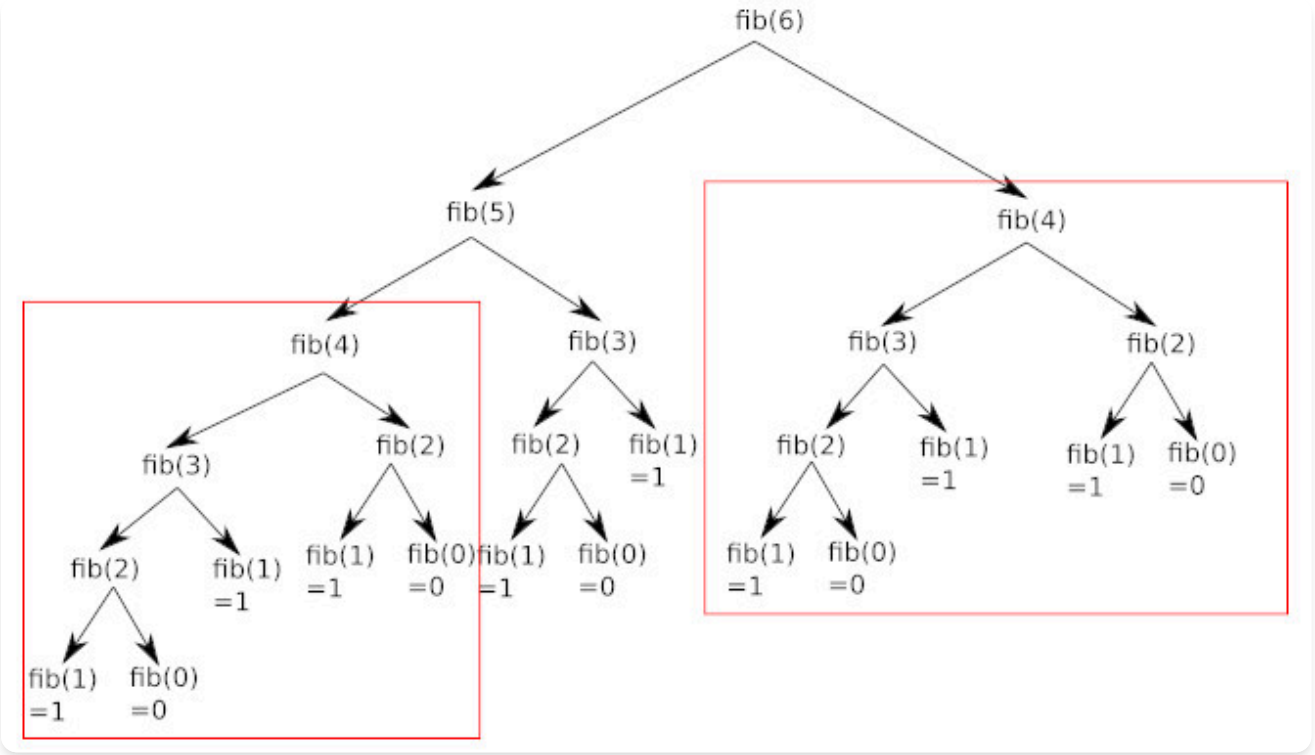
```
def fib(n) :
  if n < 2 :
    return n
  else :
    return fib(n-1) + fib(n-2)
```

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



Vous pouvez constater que l'on a une structure arborescente (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, vous avez remarqué que de nombreux calculs sont inutiles, car effectués 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 pourrait donc grandement simplifier le calcul en calculant une fois pour toutes fib(4), en "mémorisant" le résultat et en le réutilisant quand nécessaire :

```
def fib_mem(n):
    mem = [0]*(n+1)  #permet de créer un tableau contenant n+1 zéro
    def f_mem(n):
        if n == 0 or n == 1:
            mem[n] = n
            return n
        elif mem[n] > 0:
            return mem[n]
        else:
            mem[n]= f_mem(n-1) + f_mem(n-2)
        return mem[n]
    return f_mem(n)
```

Nous pouvons résumer ce programme comme suit :

- si la valeur de fib(a) n'a jamais été calculée (si la valeur de fib(a) n'est pas encore dans le tableau *mem*), elle est calculée, puis ensuite elle est stockée dans le tableau *mem*
- si la valeur de fib(a) a déjà été calculée (si la valeur de fib(a) est déjà dans le tableau *mem*), nous n'avons aucun calcul à faire, on utilise juste la valeur présente dans le tableau.

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.

En réfléchissant un peu sur le cas que nous venons de traiter, nous divisons un problème “complexe” (calcul de fib(6)) en une multitude de petits problèmes faciles à résoudre (fib(0) et fib(1)), puis nous utilisons les résultats obtenus pour les “petits problèmes” pour résoudre le problème “complexe”. Cela devrait vous rappeler la méthode “diviser pour régner” !

En fait, ce n’est pas tout à fait cela puisque dans le cas de la méthode “diviser pour régner”, la “mémorisation” des calculs n’est pas prévue. La méthode que nous venons d’utiliser se nomme “programmation dynamique”.

2) Programmation dynamique

a) introduction

Cette méthode a été introduite au début des années 1950 par Richard Bellman.

Il est important de bien comprendre que “programmation” dans “programmation dynamique”, ne doit pas s’entendre comme “utilisation d’un langage de programmation”, mais comme synonyme de planification et ordonnancement.

La programmation dynamique s’applique généralement aux problèmes d’optimisation. Nous avons déjà évoqué les problèmes d’optimisation lorsque nous avons étudié les algorithmes gloutons l’année dernière. N’hésitez pas, si nécessaire, à vous replonger dans ce cours.

La programmation dynamique s’applique quand les sous-problèmes se recoupent, c’est-à-dire lorsque les sous-problèmes ont des problèmes communs (dans le cas du calcul de fib(6) on doit calculer 2 fois fib(4). Pour calculer fib(4), on doit calculer 4 fois fib(2)…). Un algorithme de programmation dynamique résout chaque sous-problème une seule fois et mémorise sa réponse dans un tableau, évitant ainsi le recalcule de la solution chaque fois qu’il résout chaque sous-problème.

b) Approche descendante

i) Introduction

Il existe 2 facettes équivalentes de la programmation dynamique :

- la programmation descendante aussi appelée top-down (c’est la méthode que nous avons utilisée ci-dessus pour Fibonacci et que nous allons utiliser immédiatement ci-dessous)
- la programmation ascendante aussi appelée bottom-up (que nous étudierons un peu plus tard)

Poursuivons donc notre étude de l'approche descendante avec le problème du rendu de monnaie :

ii) Rendu de monnaie avec l'approche descendante

Nous allons maintenant travailler sur un problème d'optimisation déjà rencontré l'année dernière : le problème du rendu de monnaie.

Petit rappel : vous avez à votre disposition un nombre illimité de pièces de 2 cts, 5 cts, 10 cts, 50 cts et 1 euro (100 cts). Vous devez rendre une certaine somme (rendu de monnaie). Le problème est le suivant : "Quel est le nombre minimum de pièces qui doivent être utilisées pour rendre la monnaie"

La résolution "gloutonne" de ce problème peut être la suivante :

1. on prend la pièce qui a la plus grande valeur (il faut que la valeur de cette pièce soit inférieure ou égale à la somme restant à rendre)
2. on recommence l'opération ci-dessus jusqu'au moment où la somme à rendre est égale à zéro.

Prenons un exemple :

Partons du principe que nous avons 1 euro 77 cts à rendre :

- on utilise une pièce de 1 euro (plus grande valeur de pièce inférieure à 1,77 euro), il reste 77 cts à rendre
- on utilise une pièce de 50 cts (plus grande valeur de pièce inférieure à 0,77 euro), il reste 27 cts à rendre
- on utilise une pièce de 10 cts (plus grande valeur de pièce inférieure à 0,27 euro), il reste 17 cts à rendre
- on utilise une pièce de 10 cts (plus grande valeur de pièce inférieure à 0,17 euro), il reste 7 cts à rendre
- on utilise une pièce de 5 cts (plus grande valeur de pièce inférieure à 0,07 euro), il reste 2 cts à rendre
- on utilise une pièce de 2 cts (plus grande valeur de pièce inférieure à 0,02 euro), il reste 0 cts à rendre

L'algorithme se termine en renvoyant 6 (on a dû rendre 6 pièces)

Que se passe-t-il si nous avons à rendre 11 centimes ?

On utilise une pièce de 10 cts (plus grande valeur de pièce inférieure à 11 centimes), il reste 1 cts à rendre, il n'y a pas de pièce de 1 cts => l'algorithme est "bloqué"

Cet exemple marque une caractéristique importante des algorithmes gloutons : une fois qu'une "décision" a été prise, on ne revient pas "en arrière" (on a choisi la pièce de 10 cts, même si cela nous conduit dans une "impasse").

Rappel : dans certains cas, un algorithme glouton trouvera une solution, mais cette dernière ne sera pas "une des meilleures solutions possible" (une solution optimale).

Évidemment, le fait que notre algorithme glouton ne soit pas "capable" de trouver une solution ne signifie pas qu'il n'existe pas de solution… En effet, il suffit de prendre 1 pièce de 5 cts et 3 pièces de 2 cts pour arriver à 11 cts. Recherchons un algorithme qui nous permettrait de trouver une solution optimale, quelle que soit la situation.

Afin de mettre au point un algorithme, essayons de trouver une relation de récurrence :

Soit X la somme à rendre, on notera Nb(X) le nombre minimum de pièces à rendre. Nous allons nous poser la question suivante : Si je suis capable de rendre X avec Nb(X) pièces, quelle somme suis-je capable de rendre avec 1+Nb(X) pièces ?

Si j'ai à ma disposition la liste de pièces suivante : p_1, p_2, p_3, …, p_n et que je suis capable de rendre X cts, je suis donc aussi capable de rendre :

- $X-p_1$
- $X-p_2$
- $X-p_3$
- …
- $X-p_n$

(à condition que p_i (avec i compris entre 1 et n) soit inférieure ou égale à la somme restant à rendre)

Exemple : si je suis capable de rendre 72 cts et que j'ai à ma disposition des pièces de 2 cts, 5 cts, 10 cts, 50 cts et 1 euro, je peux aussi rendre :

- 72 - 2 = 70 cts
- 72 - 5 = 67 cts
- 72 - 10 = 62 cts
- 72 - 50 = 22 cts

Je ne peux pas utiliser de pièce de 1 euro.

Autrement dit, si $Nb(X-p_i)$ (avec i compris entre 1 et n) est le nombre minimal de pièces à rendre pour le montant $X-p_i$, alors $Nb(X)=1+Nb(X-p_i)$ est le nombre minimal de pièces à rendre pour un montant X. Nous avons donc la formule de récurrence suivante :

- si $X = 0$: $Nb(X) = 0$
- si $X > 0$: $Nb(X) = 1 + \min(Nb(X-p_i))$ avec $1 \leq i < n$ et $p_i \leq X$

Le "min" présent dans la formule de récurrence exprime le fait que le nombre de pièces à rendre pour une somme $X-p_i$ doit être le plus petit possible.

Étudions le programme suivant :

```
def rendu_monnaie_rec(P,X):
  if X == 0:
    return 0
  else:
    mini = 1000
  for i in range(len(P)) :
    if P[i] <= X :
      nb = 1 + rendu_monnaie_rec(P,X-P[i])
      if nb < mini:
        mini = nb
  return mini
```

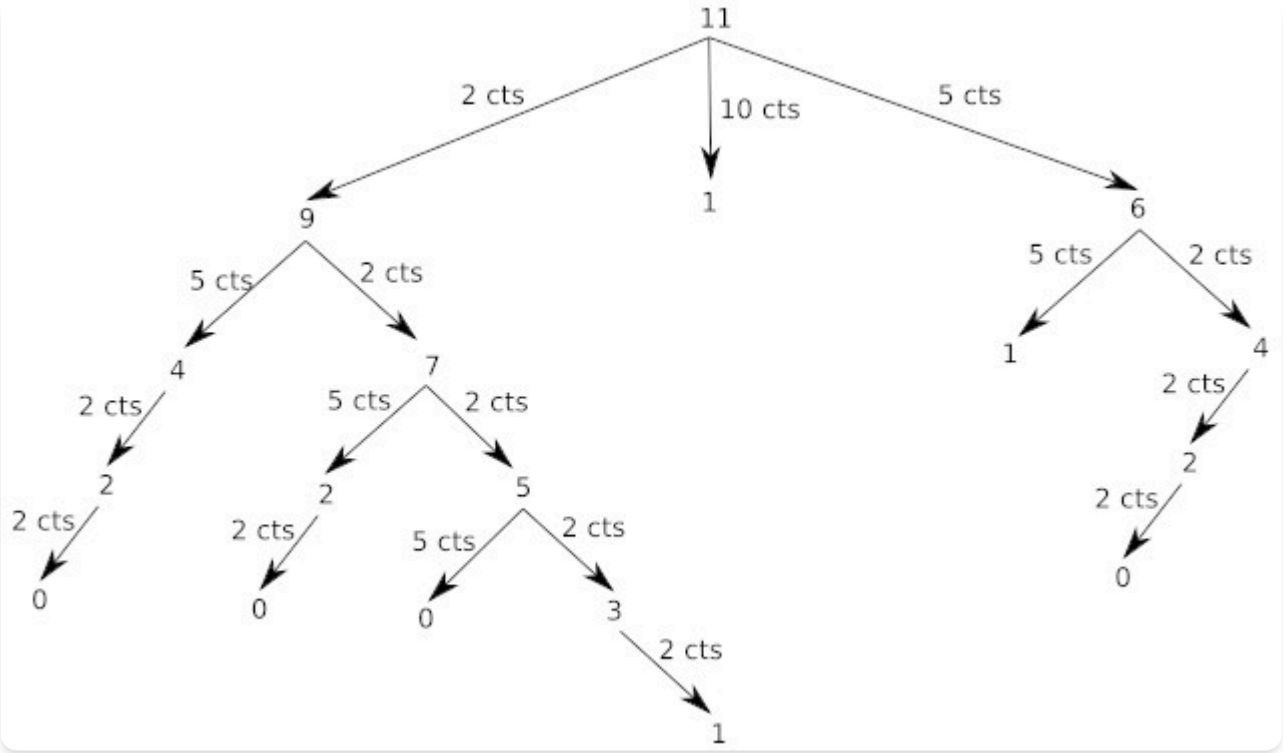
La fonction *rendu_monnaie_rec* prend en paramètre un tableau de pièces (P) et la somme à rendre (X). Elle renvoie le plus petit nombre de pièces possible. On retrouve la relation de récurrence définie juste au-dessus.

Pour être sûr de renvoyer le plus petit nombre de pièces, on attribue dans un premier temps la valeur 1000 à la variable mini (cette valeur 1000 est arbitraire, il faut juste une valeur suffisamment grande : on peut partir du principe que nous ne rencontrerons jamais de cas où il faudra rendre plus de 1000 pièces), ensuite, à chaque appel récursif, on "sauvegarde" le plus petit nombre de pièces dans cette variable mini.

Exemple : si on teste cette fonction en tapant *rendu_monnaie_rec((2,5,10,50,100),11)* dans la console Python, on obtient 4 (on doit utiliser au minimum 4 pièces pour rendre 11 cts)

Essayons de comprendre plus en détail comment le programme ci-dessous détermine ce résultat.

Étudions cet arbre :



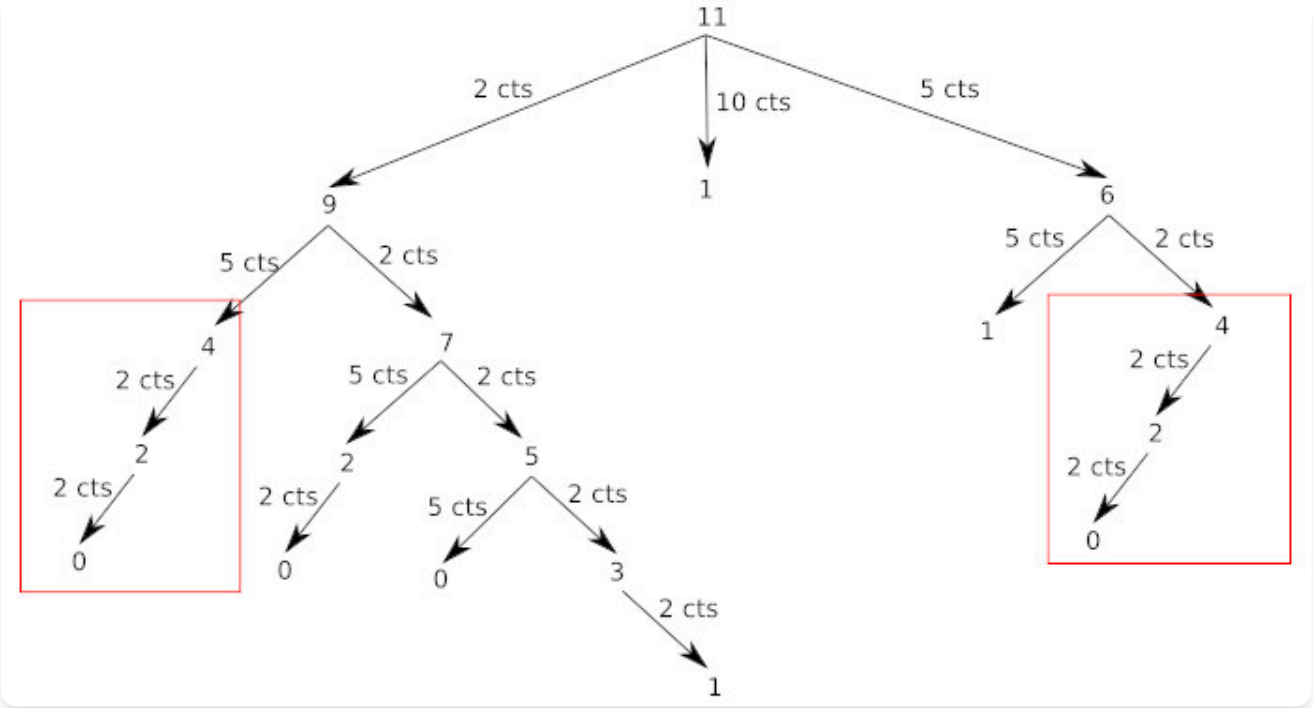
Plusieurs remarques s'imposent :

- comme vous pouvez le remarquer sur le schéma, tous les cas possibles sont "traités" (quand un algorithme "traite" tous les cas possibles, on parle souvent de méthode "brute force").
- pour certains cas, on se retrouve dans une "impasse" (cas où on termine par un "1"), dans cette situation, la fonction renvoie "1000" ce qui permet de s'assurer que cette "solution" (qui n'en est pas une) ne sera pas "retenue".
- la profondeur minimum de l'arbre (avec une feuille 0) est de 4, la solution au problème est donc 4 (il existe plusieurs parcours : (5,2,2,2), (2,5,2,2)... qui donne à chaque fois 4)

Si maintenant on tente d'exécuter la même fonction avec une valeur à rendre beaucoup plus grande (par exemple 171 cts), le programme plante ! Pourquoi ? Parce que les appels récursifs sont trop nombreux, on dépasse la capacité de la pile.

La programmation dynamique va nous permettre de résoudre ce problème.

Comme vous avez peut-être déjà dû le remarquer, même dans le cas simple évoqué ci-dessus (11 cts à rendre), nous faisons plusieurs fois exactement le même calcul. Par exemple on retrouve 2 fois la branche qui part de "4" :



Il va donc être possible d'appliquer la même méthode que pour Fibonacci.

À noter que dans des cas plus "difficiles à traiter" comme 171 cts, on va retrouver de nombreuses fois exactement les mêmes calculs, il est donc potentiellement intéressant d'utiliser la programmation dynamique.

Voici donc notre programme de calcul de rendu de monnaie modifiée :

```
def rendu_monnaie_mem(P,X):
    mem = [0]*(X+1)
    def r_monnaie_mem(P,X):
        if X == 0:
            return 0
        elif mem[X] > 0:
            return mem[X]
        else:
            mini = 1000
            for i in range(len(P)):
                if P[i] <= X:
                    nb = 1 + r_monnaie_mem(P,X-P[i])
                    if nb < mini:
                        mini = nb
                        mem[X] = mini
            return mini
    return r_monnaie_mem(P,X)
```

Ce programme ressemble beaucoup au programme utilisé pour la suite de Fibonacci, il ne devrait donc pas vous poser de problème.

Si maintenant nous testons ce programme en tapant dans la console Python *rendu_monnaie_mem((2,5,10,50,100),171)*, tout fonctionne parfaitement, il suffit d'une fraction de seconde pour obtenir le résultat qui est 7.

c) L'approche ascendante

i) Introduction

Comme nous l'avons déjà vu, l'approche descendante utilise une fonction récursive (en mémorisant les valeurs calculées afin d'éviter d'être obligé de les recalculer plus tard et permettant ainsi d'éviter un grand nombre d'appels récursifs). Dans le cas de l'approche ascendante, nous n'utiliserons pas de fonction récursive (on parle de méthode itérative : on utilisera des boucles à la place des appels récursifs).

ii) Fibonacci avec l'approche ascendante

L'idée de base de l'approche ascendante est relativement simple : la relation de récurrence nous montre que pour calculer, par exemple, fib(10), nous avons besoin de calculer fib(9) et fib(8), que pour calculer fib(9) nous avons besoin de calculer fib(8) et fib(7)…

Au lieu de recalculer plusieurs fois la même chose, commençons par calculer fib(2) (puisque par définition fib(0) = 0 et fib(1) = 1), puis calculons fib(3) (nous avons déjà fib(2) et fib(1)), puis calculons fib(4) (nous avons déjà fib(3) et fib(2)), puis calculons fib(5) (nous avons déjà fib(4) et fib(3))… afin, calculons fib(10) (nous avons déjà fib(9) et fib(8)).

Le programme Python permettant de déterminer fib(n) en utilisant la méthode ascendante est relativement simple :

```
def fib(n):
    tab = [0]*(n+1)
    tab[1] = 1
    for i in range(2, n+1):
        tab[i] = tab[i-1] + tab[i-2]
    return tab[n]
```

Vous pouvez constater que ce programme est simple à comprendre :

- pour "stocker" les différentes valeurs, on utilise une liste Python que l'on initialise avec n+1 zéro (afin d'avoir un tableau *tab* qui a des indices allant de 0 à n).
- avant le début de la boucle on a le tableau *tab* suivant : *[0,1,0,0,…]*
- la boucle permet de modifier le tableau pour qu'à l'indice *i* on trouve bien la valeur de *fib(i)*

On constate qu'avec cette approche ascendante, comme pour l'approche decendante, le calcul de *fib(i)* n'a été fait qu'une seule fois.

iii) Rendu de monnaie avec l'approche ascendante

L'idée est la même que pour Fibonacci, on remplit un tableau *tab*, avec cette fois l'indice *i* qui correspond à la somme à rendre et la valeur située à l'indice *i* qui correspond au nombre minimal de pièces à rendre :

- Pour rendre 0, il faut au minimum 0 pièce : nous aurons donc tab[0] = 0.
- Pour rendre 1, il faut au minimum 1 pièce (1 pièce de 1) : nous aurons donc tab[1] = 1.
- Pour rendre 2, il faut au minimum 1 pièce (1 pièce de 2) : nous aurons donc tab[2] = 1.
- Pour rendre 3, il faut au minimum 2 pièces (1 pièce de 1 et 1 pièce de 2) : nous aurons donc tab[3] = 2.
- ...

Si on désire rendre n, il faudra remplir le tableau jusqu'à l'indice n (tab[n])

Voici une fonction qui permet de remplir le tableau et donc de trouver une solution au rendu de monnaie :

```
def rendu_monnaie_asc(P,X):
    tab = [1000] * (X+1)
    tab[0] = 0
    for m in range(1, X+1):
        for piece in P:
            if m >= piece:
                tab[m] = min(tab[m], 1 + tab[m-piece])
    if tab[X] != 1000 :
        return tab[X]
```

Quelques remarques sur cette fonction :

- `tab = [1000] * (X+1)` on part du principe qu'il faudra toujours moins de 1000 pièces pour rendre une somme X.
- Nous avons 2 boucles imbriquées : une première boucle qui parcourt le tableau *tab* selon son indice (à chaque indice correspond une somme à rendre : indice 0, on veut rendre 0, indice 1 on veut rendre 1, ...). La deuxième boucle parcourt les pièces disponibles.
- Pour chaque montant, on parcourt toutes les pièces : si la valeur *piece* de la pièce courante est plus petite ou égale au montant courant *m*, *tab[m]* sera égale à *min(tab[m], 1 + tab[m-piece])*

Prenons un exemple : on désire rendre 6 et nous avons à notre disposition les pièces suivantes : [1, 2, 5, 10, 20, 50, 100].

Avant le début de la boucle, nous avons le tableau suivant : *tab = [0, 1000, 1000, 1000, 1000, 1000, 1000]*

Commençons la boucle :

- m = 1 et piece = 1 : *tab[1] = min(tab[1], 1 + tab[0]) = min(1000, 1+0) = 1*
- m = 1 et piece = 2 : on ne fait rien car piece > m, donc *tab[1] = 1*
- m = 2 et piece = 1 : *tab[2] = min(tab[2], 1 + tab[1]) = min(1000, 1+1) = 2*
- m = 2 et piece = 2 : *tab[2] = min(tab[2], 1 + tab[0]) = min(2, 1+0) = 1*
- m = 2 et piece = 5 : on ne fait rien car piece > m, donc *tab[2] = 1*
- m = 3 et piece = 1 : *tab[3] = min(tab[3], 1 + tab[2]) = min(1000, 1+3) = 3*
- m = 3 et piece = 2 : *tab[3] = min(tab[3], 1 + tab[1]) = min(3, 1+1) = 2*
- m = 3 et piece = 5 : on ne fait rien car piece > m, donc *tab[3] = 2*
- m = 4 et piece = 1 : *tab[4] = min(tab[4], 1 + tab[3]) = min(1000, 1+2) = 3*
- m = 4 et piece = 2 : *tab[4] = min(tab[4], 1 + tab[2]) = min(3, 1+1) = 2*
- m = 4 et piece = 5 : on ne fait rien car piece > m, donc *tab[4] = 2*
- m = 5 et piece = 1 : *tab[5] = min(tab[5], 1 + tab[4]) = min(1000, 1+2) = 3*
- m = 5 et piece = 2 : *tab[5] = min(tab[5], 1 + tab[3]) = min(3, 1+2) = 3*
- m = 5 et piece = 5 : *tab[5] = min(tab[5], 1 + tab[0]) = min(3, 1+0) = 1*
- m = 5 et piece = 10 : on ne fait rien car piece > m, donc *tab[5] = 1*
- m = 6 et piece = 1 : *tab[6] = min(tab[6], 1 + tab[5]) = min(1000, 1+1) = 2*
- m = 6 et piece = 2 : *tab[6] = min(tab[6], 1 + tab[4]) = min(2, 1+2) = 2*
- m = 6 et piece = 5 : *tab[6] = min(tab[6], 1 + tab[1]) = min(2, 1+1) = 2*
- m = 6 et piece = 10 : on ne fait rien car piece > m, donc *tab[6] = 2*

La boucle se termine, et la fonction *rendu_monnaie_asc([1, 2, 5, 10, 20, 50, 100],6)* renvoie bien 2

d) Programmation dynamique et mémoire

Comme nous venons de le voir, la programmation dynamique permet de résoudre des problèmes beaucoup plus rapidement, on améliore donc la complexité en temps d'un algorithme en utilisant la programmation dynamique. Cependant, il y a une contrepartie à cette amélioration : l'utilisation de plus de mémoire. En effet, il nous faudra utiliser plus de mémoire pour stocker les résultats des sous-problèmes. La programmation dynamique permet donc d'échanger de l'efficacité en temps contre de l'utilisation de la mémoire.

activité 16.1

Cette activité s'inspire librement du contenu du livre de Cormen, Leiserson, Rivest et Stein "Introduction to Algorithms" (édition The Mit Press)

L'entreprise sun & steel produit et vend des barres d'acier de différentes longueurs. Voici le prix de vente de ces barres d'acier en fonction de leurs longueurs :

longueur i en mètre	prix p en euro
1	1
2	5
3	8
4	9
5	10
6	17
7	17
8	20
9	24
10	30

Quand l'entreprise produit une barre de longueur n, elle a 2 possibilités :

- vendre la barre telle quelle
- découper la barre afin d'obtenir plusieurs barres plus petites

Par exemple pour une barre de 4 m de longueur l'entreprise peut :

- vendre la barre de 4 m ; gain = 9 euros
- découper la barre en 2 => 4 = 2 + 2 et vendre ces 2 barres ; gain = 5+5 = 10 euros
- découper la barre en 2 => 4 = 1 + 3 et vendre ces 2 barres ; gain = 8+1 = 9 euros
- découper la barre en 3 => 4 = 1+1+2 et vendre ces 3 barres ; gain = 1+1+5 = 7 euros
- ...

1- Continuez la liste ci-dessus afin d'obtenir toutes les possibilités pour une barre de 4 m. Quel est le revenu maximum possible pour la société ?

Partons maintenant du principe qu'au départ, la barre ne fait plus forcement 4 m mais qu'elle fait n mètres (n compris entre 1 et 10).

Pour chaque longueur n on peut déterminer le revenu maximum possible (que l'on notera r_n). Par exemple :

- pour n = 1 le revenu maximum sera de 1 euro, on aura donc $r_1 = 1$
- pour n = 2, il y a 2 possibilités :
  - on ne découpe pas la barre (revenu = 5)
  - on découpe la barre en 2 (revenu = 1+1 = 2)

  on obtiendra donc $r_2 = 5$
- pour n = 3, il y a 3 possibilités :
  - on ne découpe pas la barre (revenu = 8)
  - on découpe la barre en 2 (3 = 1+2 ; revenu = 1+5 = 6)
  - on découpe la barre en 3 (3 = 1+1+1 ; revenu = 1+1+1 = 3)

  on obtiendra donc $r_3 = 8$
- pour n = 4, il y a 5 possibilités :
  - on ne découpe pas la barre (revenu = 9)
  - on découpe la barre en 2 (4 = 2+2 ; revenu = 5+5 = 10)
  - on découpe la barre en 2 (4 = 1+3 ; revenu = 1+8 = 9)
  - on découpe la barre en 3 (4 = 1+1+2 ; revenu = 1+1+5 = 7)
  - on découpe la barre en 4 (4 = 1+1+1+1 ; revenu = 1+1+1+1 = 4)

  on obtiendra donc $r_4 = 10$
- pour n = 5, il y a 7 possibilités :
  - on ne découpe pas la barre (revenu = 10)
  - on découpe la barre en 2 (5 = 2+3 ; revenu = 5+8 = 13)
  - on découpe la barre en 2 (5 = 1+4 ; revenu = 1+9 = 10)
  - on découpe la barre en 3 (5 = 1+1+3 ; revenu = 1+1+8 = 10)
  - on découpe la barre en 3 (5 = 1+2+2 ; revenu = 1+5+5 = 11)

- on découpe la barre en 4 (5 = 1+1+1+2 ; revenu = 1+1+1+5 = 8)
- on découpe la barre en 5 (5 = 1+1+1+1+1 ; revenu = 1+1+1+1+1 = 5)

  on obtiendra donc $r_5 = 13$

- ...

L'idée de cette activité est d'écrire un programme Python qui permettra d'obtenir tous les r_n avec n compris entre 1 et 10.

On peut remarquer qu'il est aussi possible de calculer r_5 en appliquant la relation suivante :

$r_5 = \max(p_5, r_1 + r_4, r_2 + r_3, r_3 + r_2, r_4 + r_1)$

avec :

- p_5 le prix de la barre de 5 m
- r_1 le meilleur revenu possible pour 1 barre de 1 m (c'est-à-dire 1)
- r_2 le meilleur revenu possible pour 1 barre de 2 m (c'est-à-diree 5)
- r_3 le meilleur revenu possible pour 1 barre de 3 m (c'est-à-dire 8)
- r_4 le meilleur revenu possible pour 1 barre de 4 m (c'est-à-dire 10)

Nous avons donc :

$r_5 = \max(10, 1+10, 5+8, 8+5, 10+1) = \max(10,11,13,13,11) = 13$

Il est possible de généraliser cette relation en écrivant :

$r_n = \max(p_n, r_1+r_{n-1}, r_2+r_{n-2}, \ldots, r_{n-1}+r_1)$

2- Retrouvez la valeur de r_4 en utilisant la relation ci-dessus

On remarque que pour calculer r_n, il faut calculer r_{n-1}, r_{n-2}, ...

Cela devrait vous rappeler quelque chose : dans le chapitre consacré aux fonctions récursives nous avons vu que pour calculer la factorielle de n , il faut calculer la factorielle de n-1, la factorielle de n-2 ... Nous allons donc pouvoir écrire une fonction récursive afin de pouvoir résoudre notre problème (calculer tous les r_n pour n compris entre 1 et 10).

Avant d'écrire cette fonction récursive, analysons un peu la relation vue ci-dessus :

$r_n = \max(p_n, r_1+r_{n-1}, r_2+r_{n-2},\ldots, r_{n-1}+r_1)$

Dans cette relation, le premier élément (p_n) correspond au cas où la barre n'est pas découpée. Les autres éléments correspondent au cas où l'on découpe la barre en 2

morceaux de longueurs i et n-i avec i compris entre 1 et n (avec n inclus).

Il est donc possible de modifier la relation vue ci-dessus et d'écrire :

$r_n = \max(p_i+r_{n-i})$ avec i compris entre 1 et n (n inclus)

Par exemple, pour n=5, on retrouve :

$r_5 = \max(p_1+r_4, p_2+r_3, p_3+r_2, p_4+r_1, p_5+r_0)$ avec r_0 le revenu maximum pour une barre de longueur nulle (on a donc $r_0 = 0$)

Pour calculer r_5 il est donc nécessaire de calculer r_4 :

$r_4 = \max(p_1+r_3, p_2+r_2, p_3+r_1, p_4+r_0)$

mais il est aussi nécessaire de calculer r_3, r_2 et r_1 à la fois pour calculer r_5 mais aussi pour calculer r_4 … On retrouve bien notre structure récursive.

N.B : Si vous avez du mal à comprendre le pourquoi du comment de la relation vue ci-dessus ($r_n = \max(p_i+r_{n-i})$ avec i compris entre 1 et n (n inclus)), cela n'a pas une grande importance, contentez-vous de l'utiliser (le but de cette activité est ailleurs).

3- La fonction *revenu_barre* ci-dessous prend en paramètre la taille n de la barre (avant découpe) et renvoie le revenu maximum possible pour cette barre de longueur n :

```
def revenu_barre(n):
    prix = [0,1,5,8,9,10,17,17,20,24,30]
    if n == 0:
        return ...
    r = float('-inf')
    for i in range(1, n+1):
        r = max(r, prix[i] + ...)
    return r
```

Compétez la fonction *revenu_barre*

4- Utilisez la fonction *revenu_barre* pour calculer r_5, r_6, r_7, r_8, r_9 et r_{10}

5- L'entreprise a modifié ces tarifs, nous avons maintenant le tableau suivant :

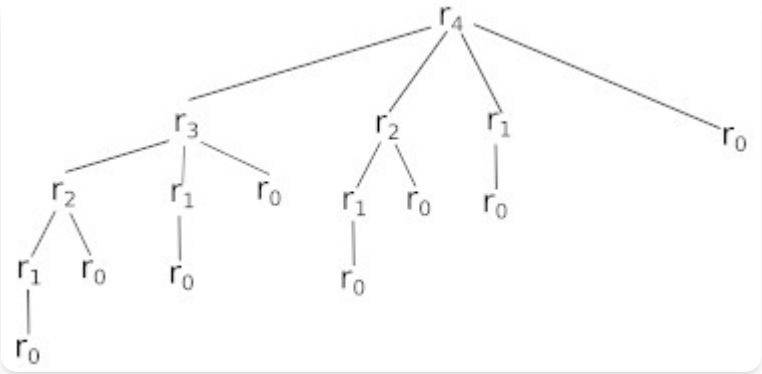
longueur i en mètre	**prix p en euro**
1	1
2	6

longueur i en mètre	prix p en euro
3	9
4	11
5	12
6	19
7	20
8	23
9	24
10	26

Modifiez la fonction *revenu_barre* afin de tenir compte de cette modification tarifaire.

Comme nous avons eu l'occasion de le voir ci-dessus, pour calculer r_n il est nécessaire de calculer r_{n-1}, r_{n-2}...mais pour calculer r_{n-1}, il sera aussi nécessaire de calculer r_{n-2} ! Nous allons donc calculer 2 fois r_{n-2}.

Afin de mieux visualiser la situation, voici un arbre qui permet de mieux visualiser les calculs nécessaires afin de déterminer r_4 :



Comme vous pouvez le constater, pour calculer r_4, il faut calculer :

- 1 fois r_3
- 2 fois r_2
- 4 fois r_1

Dans cet exemple, cela ne pose aucun problème, mais avec un n plus grand (par exemple r_{10}), nous aurions à effectuer de nombreuses fois exactement les mêmes calculs. On pourrait même imaginer si nous avions à notre disposition des barres de 100 m de long, un nombre

de calculs à effectuer qui arriverait aux limites des capacités de nos ordinateurs (calculs très longs à effectuer).

Nous sommes donc typiquement dans le cas où la programmation dynamique pourrait nous être utile afin d'éviter de refaire un grand nombre de fois exactement les mêmes calculs.

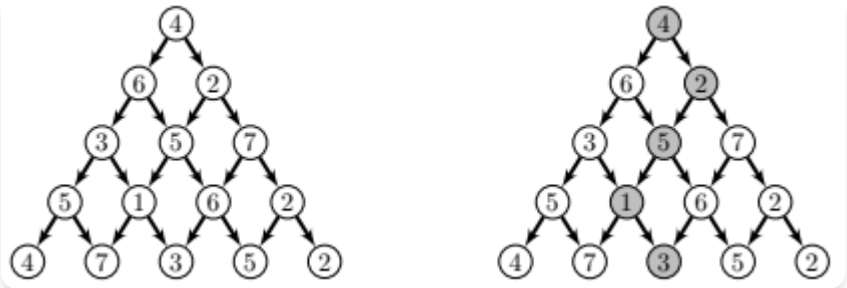
6- En vous inspirant de ce qui a été fait dans le cours (suite de Fibonacci et rendu de monnaie), écrivez une fonction *revenu_barre_dyn_desc(n)* permettant de calculer r_n. Cette fonction devra utiliser la programmation dynamique (approche descendante)

7- Toujours en utilisant la programmation dynamique, proposez une fonction *revenu_barre_dyn_asc(n)* permettant de calculer r_n mais cette fois-ci en utilisant une approche ascendante.

activité 16.2

Cette activité s'inspire très fortement de l'exercice 1 du sujet 0.B session 2024. Merci beaucoup à l'autrice ou à l'auteur de cet exercice.

Nolan vient de découvrir un petit jeu amusant dont le but est de trouver un chemin de score maximal dans une pyramide de nombres. Un exemple de pyramide de nombres est représenté ci-dessous. Un chemin dans cette pyramide doit partir du sommet de la pyramide et descendre jusqu'en bas en suivant des flèches, c'est-à-dire en choisissant à chaque niveau de descendre sur la droite ou sur la gauche. Le score d'un chemin est la somme des nombres rencontrés le long de ce chemin. Le chemin gris représenté sur la pyramide de droite a pour score 4 + 2 + 5 + 1 + 3 = 15.



Nolan souhaite utiliser son ordinateur pour chercher à résoudre ce genre de jeux efficacement. Pour cela, on représente chaque niveau par la liste des nombres de ce niveau et une pyramide est une liste de niveaux. La pyramide ci-dessus est donc représentée par la liste de listes `ex1 = [[4],[6,2],[3,5,7],[5,1,6,2],[4,7,3,5,2]]` .

1- Dessiner la pyramide représentée par la liste de listes `ex2 = [[3],[1,2],[4,5,9],[3,6,2,1]]` .

Un chemin dans une pyramide est représenté par la liste des indices des nombres qu'il parcourt à chaque niveau.

Le chemin gris est représenté par la liste `ch1 = [0,1,1,1,2]` . En effet, le 4 est à l'indice 0 du premier niveau, le
2, le 5 et le 1 sont tous à l'indice 1 de leurs niveaux respectifs et le 3 du dernier niveau est à l'indice 2.

2- Dessiner le chemin `ch2 = [0,1,2,2]` dans la pyramide représentée par `ex2` .

3- Calculer le score de `ch2` dans la pyramide représentée par ex2.

Dans un premier temps, Nolan souhaite écrire une fonction qui vérifie qu'une liste donnée représente bien un chemin
et une autre fonction qui calcule le score d'un chemin.

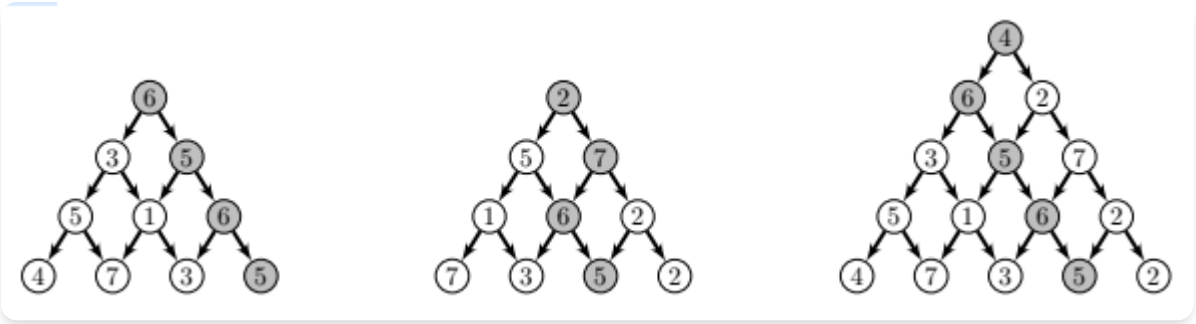
On remarque qu'une liste `ch` représente un chemin dans une pyramide p si elle commence par un 0, si lorsque a et
b sont deux éléments consécutifs dans `ch` alors b = a ou b = a + 1 et `ch` est de la même longueur que p.

4- Écrire une fonction `est_chemin(ch,p)` qui prend en paramètre une liste `ch` et une pyramide `p` et qui renvoie
un booléen indiquant si `ch` représente un chemin de la pyramide `p` . Vérifier que votre fonction est correct en passant à votre fonction le chemin `ch2` et la pyramide `ex2` (votre fonction doit renvoyer `True`). Tester aussi votre fonction à l'aide du chemin `ch3 = [0, 0, 0, 2]` et de la pyramide `ex2` (votre fonction doit renvoyer `False`).

5- Écrire une fonction `score(ch,p)` qui prend en paramètre un chemin `ch` et une pyramide `p` et qui renvoie le
score de `ch` dans `p` . Tester votre fonction à l'aide de la pyramide `ex1` et du chemin `ch1` (vous devez obtenir 15)

Maintenant que Nolan est capable de vérifier des chemins et de calculer leur score, il voudrait être capable de trouver un meilleur chemin, c'est-à-dire un chemin dont le score est maximal. Pour cela, Nolan analyse le problème en le décomposant.

Première observation : si on a des chemins maximaux `chm1` et `chm2` (représentés en gris ci-dessous) pour les
deux pyramides obtenues en enlevant le sommet de `ex1` , on obtient un chemin maximal en ajoutant le sommet 4
devant le chemin de plus grand score parmi `chm1` et `chm2` . Ici le score de `chm1` est 6 + 5 + 6 + 5 = 22 et le score de
`chm2` est 2 + 7 + 6 + 5 = 20 donc le chemin optimal dans `ex1` est celui obtenu à partir de `chm1` et dessinée à droite
dans `ex1` .



Deuxième observation : si la pyramide n'a qu'un seul niveau, il n'y a que le sommet, dans ce cas, il n'y a pas
de choix à faire, le seul chemin possible est celui qui contient le sommet et le nombre de ce sommet est le score
maximal que l'on peut obtenir.

Avec ces deux observations, on peut calculer le score maximal possible pour un chemin dans une pyramide p par
récurrence. Posons `score_max(i,j)` le score maximal possible depuis le nombre d'indice `j` du niveau `i` , c'est-à-dire
dans la petite pyramide issue de ce nombre. On a alors les relations suivantes :

- `score_max(len(p)-1 , j) = p[len(p)-1][j]`
- `score_max(i,j) = p[i][j] + max(score_max(i+1,j),score_max(i+1,j+1))`

Le score maximal possible pour p toute entière sera alors `score_max(0,0)` .

6- Écrire une fonction `calcule_score_max(p)` qui renvoie le score maximum pour une pyramide `p` .

Si on suit à la lettre la définition de `score_max` , on obtient une résolution dont le coût est exponentiel à cause de
la redondance des calculs. Par exemple `score_max(3,1)` va être calculer pour chaque appel à `score_max(2,0)` et
`score_max(2,1)` . Pour éviter cette redondance, Nolan décide de mettre en place une approche par programmation dynamique. Pour cela, il va construire une pyramide s dont le nombre à l'indice j du niveau i correspond à `score_max(i,j)` , c'est-à-dire au score maximal pour un chemin à partir du nombre correspondant dans `p` .

7- Écrire une fonction `pyramide_nulle(n)` qui prend en paramètre un entier et construit une pyramide remplie de 0. Cette pyramide aura `n` niveaux.

8- Écrire une fonction `prog_dyn(p)` qui prend en paramètre une pyramide `p` , et qui renvoie le score maximal pour un chemin dans `p` . Pour cela, elle construit une pyramide `s` pleine de 0 de la même taille et la remplit avec les valeurs de `score_max` en commençant

par le dernier niveau et en appliquant petit à petit les relations données ci-dessus. On pourra utiliser la programmation dynamique avec une approche ascendante.

9- Déterminer l'ordre de grandeur du coût de cette fonction pour une pyramide à n niveaux.

Ce qu’il faut savoir

- la programmation dynamique est une méthode algorithmique utilisée pour résoudre les problèmes d’optimisation (comme les méthodes gloutonnes vues en classe de première).
- 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**.

Ce qu’il faut savoir faire

vous devez être capable d’utiliser la programmation dynamique dans des cas simples (par exemple le problème du rendu de monnaie).