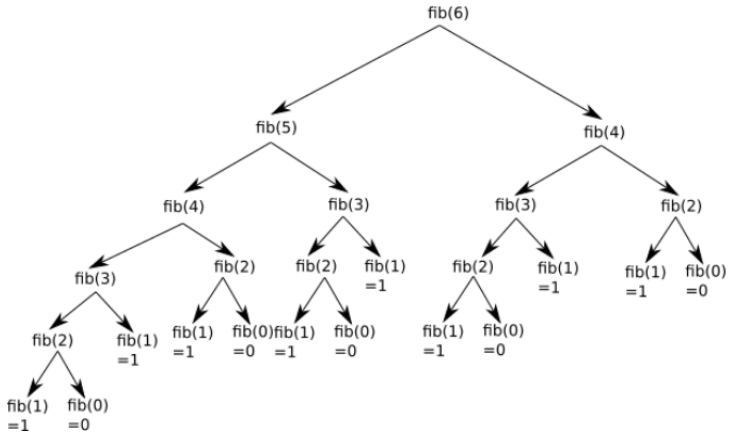
Algorithmique: programmation dynamique:

1. **Suite de Fibonacci:**
   Revenons sur ce qui a été vu dans le cours consacré à la récursivité (`https://pixees.fr/informatiquelycee/n_site/nsi_term_fctRec.html`). Dans l'exercice 5 de ce cours, on vous demande d'écrire une fonction récursive qui permet de calculer le n ième terme de la suite de Fibonacci. Voici normalement ce que vous avez dû obtenir:

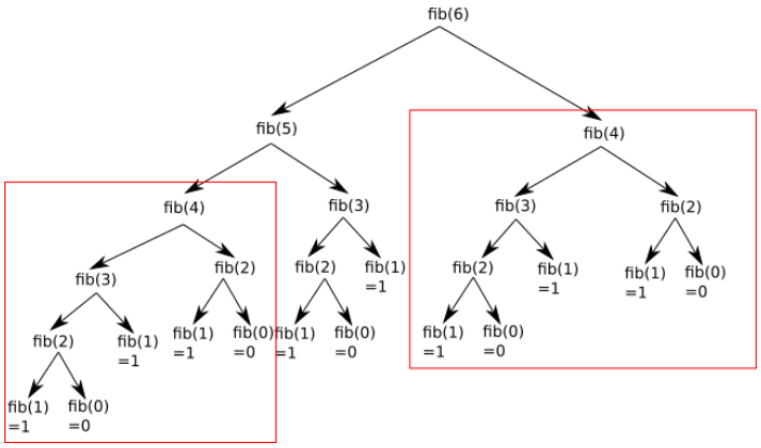
```
def fib(n) :
  if n < 2 :
    return n
  else :
    return fib(n-1)+fib(n-2)
```

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

Exercice 1: Après avoir étudié attentivement le programme ci-dessous, testez-le:

```
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]
```

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 faite, 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:**
Comme nous venons de le voir, 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. 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 (`https://pixees.fr/informatiquelycee/n_site/nsi_prem_glouton_algo.html`). N'hésitez pas, si nécessaire à vous replonger dans ce cours.

   Comme déjà évoqué plus haut, à la différence de la méthode diviser pour régner, 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-sous-problème une seule fois et mémorise sa réponse dans un tableau, évitant ainsi le recalcul de la solution chaque fois qu'il résout chaque sous-sous-problème.

3. **Programmation dynamique et rendu de monnaie:**
Nous allons maintenant travailler sur un problème d'optimisation déjà rencontré l'année dernière (`https://pixees.fr/informatiquelycee/n_site/nsi_prem_glouton_algo.html`): 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:

   - 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)
   - on recommence l'opération ci-dessus jusqu'au moment où la somme à rendre est égale à zéro.

   Prenons un exemple: 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)

Exercice 2: Appliquez l'algorithme glouton vu ci-dessus avec la somme à rendre égale à 11 centimes.

Comme vous l'avez sans doute remarqué, si on applique l'algorithme glouton à cet exemple, nous ne trouvons pas de réponse. En effet:

- on utilise une pièce de 10 cts (plus grande valeur de pièce inférieure à 11 euros), il reste 1 cts à rendre
- il n'y a pas de pièce de 1 cts $\longrightarrow$ l'algorithme est "bloqué"

Cet exemple marque une caractéristique importante des algorithmes glouton : 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. Si j'ai à ma disposition la liste de pièces suivante: $p_1, p_2, p_3, ..., p_n$, j'ai pu rendre auparavant $X - p_1$ ou $X - p_2$ ou $X - p_3$, ... , $X - p_n$ ((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, j'ai pu rendre auparavant rendre 72 - 2 = 70 cts ou 72 - 5 = 67 cts ou 72 - 10 = 62 cts ou 72 - 50 = 22 cts. Je ne peux pas utiliser de pièce de 1 euro.

Nous avons donc la formule de récurrence suivante:

$$\begin{cases} si \quad X = 0 : Nb(X) = 0 \\ si \quad X > 0 : Nb(X) = 1 + min(Nb(X - p_i)) \quad avec \quad 1 \leq i < n \quad et \quad p_i \leq X \end{cases}$$

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.

Exercice 3: Étudiez attentivement le programme Python suivant:

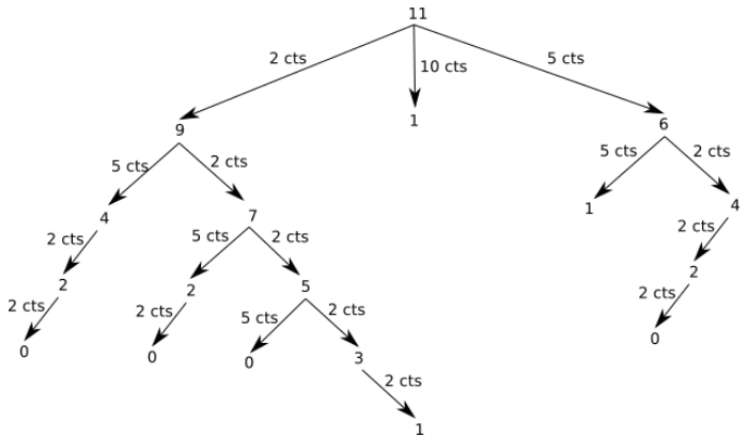
```
def rendu_monnaie_rec(P,X):
  if X==0:
    return 0
  else:
    mini = 1000
  for i in range(len(P)):
    if P[i]<=X:
      nb = 1 + rendu_monnaie_rec(P,X-P[i])
      if nb<mini:
        mini = nb
  return mini

pieces = (2,5,10,50,100)
```

Comme vous l'avez sans doute remarqué, 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.

Exercice 4:
Testez le programme de l'exercice 3 en saisissant dans la console Python "rendu_monnaie_rec(pieces,11)" (on recherche le nombre minimum de pièces à rendre pour une somme de 11 cts. Les pièces disponibles sont: 2 cts, 5 cts, 10 cts, 50 cts, 100 cts (1 euro))

Le programme de l'exercice 3 n'est pas des plus simple à comprendre (même si on retrouve bien la formule de récurrence définit un peu au-dessus), voici un schéma (avec une somme à rendre de 11 centimes) qui vous permettra de mieux comprendre le principe de cet algorithme:

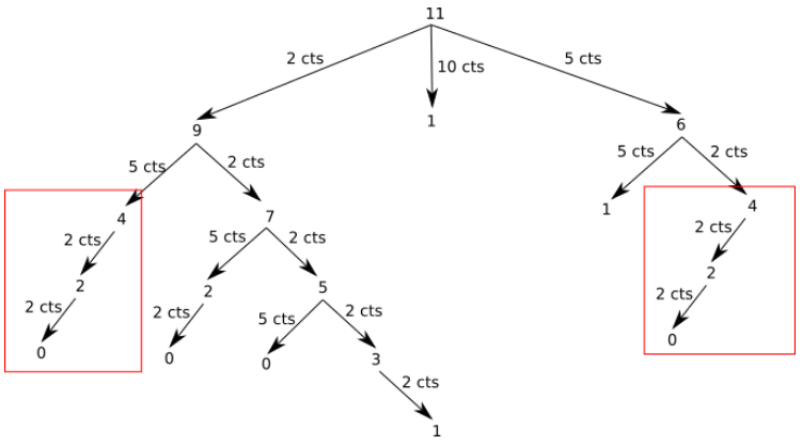


Plusieurs remarques s'imposent:

- comme vous pouvez le remarquer sur le schéma, tous les cas 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)

Exercice 5:
Testez le programme de l'exercice 3 en saisissant dans la console Python "rendu_monnaie_rec(pieces,171)" (on recherche le nombre minimum de pièces à rendre pour une somme de 1,71 euro. Les pièces disponibles sont : 2 cts, 5 cts, 10 cts, 50 cts, 100 cts (1 euro)).

Comme vous pouvez le constater le programme ne permet pas d'obtenir une solution, pourquoi? Parce que les appels récursifs sont trop nombreux, on dépasse la capacité de la pile. 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: la programmation dynamique. À noter que dans des cas plus "difficiles à traiter" comme 1,71 euro, on va retrouver de nombreuses fois exactement les mêmes calculs, il est donc potentiellement intéressant d'utiliser la programmation dynamique.

Exercice 6: Étudiez attentivement le programme Python suivant:

```
def rendu_monnaie_mem(P,X):
  mem = [0]*(X+1)
  return rendu_monnaie_mem_c(P,X,mem)

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

pieces = (2,5,10,50,100)
```

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

Exercice 7: Testez le programme de l'exercice 6 en saisissant dans la console Python "rendu_monnaie_mem(p (on recherche le nombre minimum de pièces à rendre pour une somme de 1,71 euro. Les pièces disponibles sont : 2 cts, 5 cts, 10 cts, 50 cts, 100 cts (1 euro)).

Comme vous pouvez le constater, au contraire du cas précédent (programme de l'exercice 3), il suffit d'une fraction de seconde pour que le programme basé sur la programmation dynamique donne une réponse correcte.