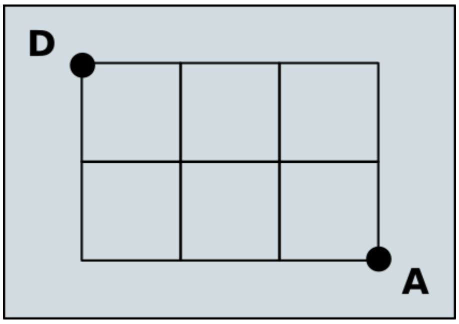
Principes de la programmation dynamique

1. Un premier exemple débranché

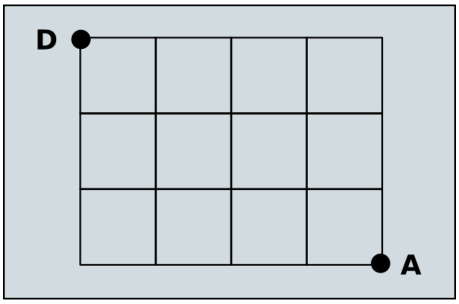
Parcours sur une grille

Énoncé

1. Combien y-a-t'il de chemin menant du point au point sur le graphique suivant, en ne se déplaçant à chaque pas que vers la droite ou vers le bas ?



2. Combien y-a-t'il de chemin menant du point au point sur le graphique suivant, en ne se déplaçant à chaque pas que vers la droite ou vers le bas ?



Solution

A venir !

2. La suite de Fibonacci

La **suite de Fibonacci** est une suite définie par une récurence d'ordre 2 de la manière suivante, :

Calculer

Énoncé

Calculer les 10 premiers termes de la suite de Fibonacci.

Solution

Les 10 premiers termes sont : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Notation

On notera le nombre de la suite de Fibonacci de rang . Par exemple et .

Algorithmiquement parlant, la suite de Fibonacci étant une suite définie par récurrence, nous serions tentés de créer une fonction récursive pour calculer les termes de la suite. Pour ce faire, nous pourrions utiliser la fonction suivante :

```python
def fibo(n : int) -> int :
    if n == 0 :
        return  0
    elif n == 1 :
        return 1
    else :
        return fibo(n-1) + fibo(n-2)
```

La question que nous devons nous poser est : est-ce un choix judicieux ?

Tester et voir les limites

Énoncé

1. Tester la fonction `fibo` avec le code suivant :

   ```python
   import time
   for n in range(40) :
       start = time.perf_counter()
       print(f"fibo({n}) = {fibo(n)}", end="")
       end = time.perf_counter()
       print(f" Temps : {end - start}")
   ```

   Que constate-t'on ?

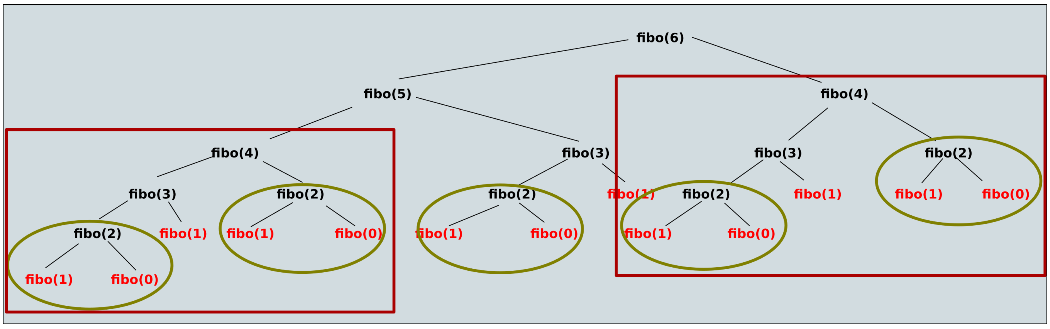
2. Réaliser un schéma de la pile d'appels récursif effectués lors de l'exécution de `fibo(6)` .

Solutions

1. Le temps d'exécution croît de manière exponentielle.
2. On a la construction suivante :

Multiples appels

Dans l'exemple précédent de calcul de `fibo(6)`, on peut constater que les appels récursifs sont nombreux, et souvent pour calculer plusieurs fois la même chose :



Ainsi :

- `fibo(2)` est calculé à 5 reprises ;
- `fibo(3)` est calculé à 3 reprises ;
- `fibo(4)` est calculé à 2 reprises.

Le nombre d'appels augmente exponentiellement en fonction de `n`. Par exemple le calcul récursif de `fibo(20)` nécessite appels au calcul `fibo(2)`, celui de `fibo(30)` le nécessite fois, celui de `fibo(40)` le nécessite fois....

Si la limite de récursion (qui est de 1000 par défaut pour Python) n'est pas atteinte pour `fibo(40)`, le temps de calcul, lui, croît aussi exponentiellement...

3. Programmation dynamique

3.1. Premiers exemples sur la suite de Fibonacci

En considérant l'algorithme précédant, on comprend bien qu'il est particulièrement inefficace de calculer plusieurs fois le même sous-calcul. Afin d'améliorer le temps de calcul de l'algorithme, nous décidons donc de **mémoriser les calculs déjà effectués** dans un tableau. Il existe deux méthodes différentes :

Programmation dynamique de la suite de Fibonacci

Méthode ascendante

On va calculer les nombres de la suite de Fibonacci jusqu'à en partant de et . On appelle ce type de méthode une méthode **Bottom-Up**. Ce n'est pas une méthode récursive.

```python
def fiboAsc(n : int) -> int :
    F = [0]*(n+1)
    F[1] = 1
    for i in range(2,n+1) :
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

Méthode descendante

On va calculer les nombres de Fibonacci récursivement, mais en sauvegardant les calculs déjà effectués dans une liste Python, en profitant de sa *mutabilité*. On appelle ce type de méthode une approche **Top-Down** :

```python
def fiboDesc(n : int) -> int :

    memo = [0, 1]+[None]*(n-1)

    def compute(n, memo) :
        if memo[n] is  None :
            memo[n] = compute(n-1, memo) + compute(n-2, memo)
        return memo[n]

    return compute(n, memo)
```

L'explication la plus simple du fonctionnement est visible dans Thonny, en utilisant le debugger, ou bien ici, pour un exemple sur `fiboDesc(6)`.

3.2. Principes de la programmation dynamique

La **programmation dynamique**, introduite au début des années 1950 par Richard Bellman, est une méthode pour résoudre des problèmes en combinant des solutions de sous-problèmes, tout comme les méthodes de type *diviser pour régner*.

Un algorithme de programmation dynamique résout chaque sous-sous-problème une seule fois et mémorise sa réponse dans un tableau, évitant ainsi le re-calcul de la solution chaque fois qu'il résout chaque sous-sous-problème.

La programmation dynamique s'applique généralement aux **problèmes d'optimisation**, comme ceux que nous avons vu l'an passé lorsque nous avons étudié les algorithmes gloutons.

3.3. Le problème du rendu de monnaie

Largement inspiré de https://isn-icn-ljm.pagesperso-orange.fr/NSI-TLE/res/res_rendu_de_monnaie.pdf.

Le problème : introduction et traitement débranché

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.

Questions

Énoncé

1. Appliquer cette méthode pour une somme de 1€77 (177cts) à rendre.
2. Appliquer cette méthode à la somme de 11 centimes.
    a. Quel est le problème ?
    b. Proposer une solution permettant de rendre 11 centimes. Est-elle unique ?

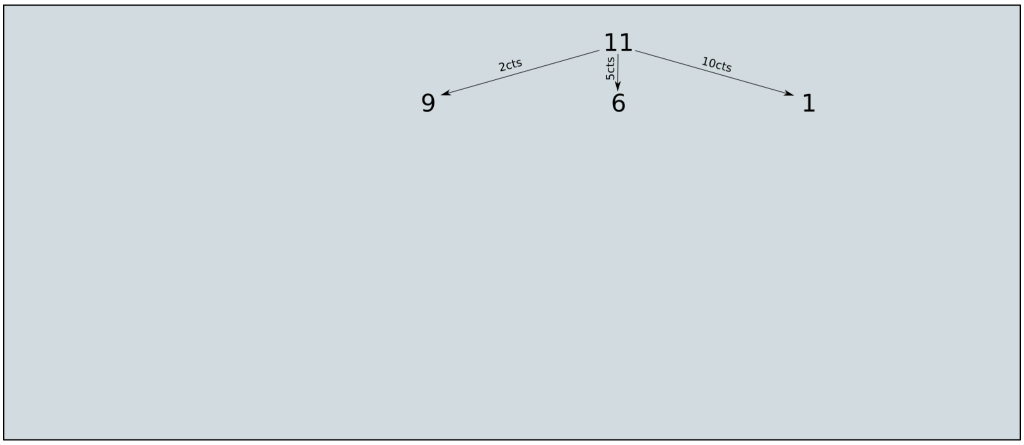
Mise au point d'un algorithme récursif

Nous allons essayer de mettre au point un algorithme récursif donnant une solution au problème de rendu de monnaie en utilisant le **nombre minimal de pièces**.

Questions

Énoncé

1. Compléter l'arbre suivant donnant l'ensemble des possibilités de répartition des pièces :



2. Combien de chemins sont des impasses (on termine avec 1 cts restant) ? Combien de solutions existent ? Quelle est la solution utilisant le nombre minimal de pièces ?

Force Brute

Quand une méthode traite tous les cas possibles, on parle souvent de méthode en **force brute**.

3. Compléter la fonction suivante pour qu'elle donne le nombre minimal de pièces utilisées pour une somme `s` donnée :

```
def rendu_monnaie_rec(P : list, s : int) -> int:
    """ renvoie le nombre minimal de pièces pour rendre la somme s
    en utilisant le jeu de pièces P"""

    if s==0:
        return 0
    else:
        mini = float('inf') # On fixe le nombre de piècé à l'infini
    for i in range(len(P)):
        if ... <= s:
            nb = 1 + ...
            if nb < mini:
                mini = nb
    return mini
```

4. Testez la fonction avec le jeu de pièces `(2, 5, 10, 100)`, et pour des sommes augmentant à partir de 11 cts. A partir de quelle v somme le programme devient-il visiblement plus lent ?

Passage en programmation dynamique

On constate dans la partie précédente que la méthode précédente fait de trop nombreux appels récursifs, qui ralentissent considérablement le temps de calcul, voire plante le programme dès que la taille maximale de la pile est dépassée.

On va donc utiliser la *programmation dynamique* pour accélérer la vitesse de traitement du problème :

Questions

Énoncé

On considère la fonction suivante :

```
def renduMonnaie1(P : list, s : int) -> int | None :
    nb = [0]+[None] * (s)
    for n in range(1, s+1) :
        for p in P :
            if p <= ... and nb[...] is not None :
                if nb[n] is ... or ... > 1 + nb[n-p]:
                    nb[n] = 1 + nb[n-p]
    return ...
```

1. Compléter la fonction afin qu'elle renvoie le nombre minimal de pièce pour rendre la monnaie, ou `None` s'il est impossible de rendre la monnaie.

1. Est-ce une méthode ascendante ou descendante ?
2. Créer une fonction `renduMonnaie2(P : list, s : int) -> int | None` utilisant l'autre méthode.

Pour aller plus loin