Programmation dynamique

1.Définition

La programmation dynamique est une technique algorithmique utilisée pour résoudre des problèmes complexes en décomposant ces derniers en sous-problèmes plus simples, puis en résolvant chaque sous-problème une seule fois et en stockant sa solution. Cette approche permet d'éviter de recalculer les mêmes solutions à plusieurs reprises, réduisant ainsi le temps de calcul et améliorant l'efficacité de l'algorithme. La programmation dynamique est largement utilisée dans divers domaines tels que l'optimisation, les jeux, l'intelligence artificielle et d'autres problèmes algorithmiques.

Exemples d'algorithmes qui se prêtent à la programmation dynamique :

- Fibonacci : Calcul de la suite de Fibonacci en utilisant la méthode de la programmation dynamique pour éviter les recalculs inutiles.

- Le problème du sac à dos (Knapsack problem) : Trouver la combinaison optimale d'articles à placer dans un sac à dos limité par une capacité maximale, en maximisant la valeur totale des articles.

- Le problème du plus court chemin dans un graphe (Shortest Path) : Trouver le chemin le plus court entre deux sommets dans un graphe pondéré.

- Le problème du voyageur de commerce (Traveling Salesman Problem) : Trouver le plus court chemin qui passe une seule fois par chaque ville et retourne à la ville de départ.

- Le problème du rendu de monnaie (Coin Change Problem) : Trouver le nombre minimum de pièces de monnaie nécessaires pour rendre une somme donnée.

2.Suite de Fibonacci

Rappel :

En mathématiques, la suite de Fibonacci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. En bourse, la suite de Fibonacci sert principalement à repérer une correction, ou une continuation de tendance.

La suite de Fibonacci est définie par :

- $u_0 = 0$, $u_1 = 1$
- et par la relation de récurrence suivante avec n entier et n > 1 : $u_n = u_n\text{-}1 + u_n\text{-}2$

Ce qui nous donne pour les 6 premiers termes de la suite de Fibonacci :

- $u_0 = 0$
- $u_1 = 1$
- $u_2 = u_1 + u_0 = 1 + 0 = 1$
- $u_3 = u_2 + u_1 = 1 + 1 = 2$
- $u_4 = u_3 + u_2 = 2 + 1 = 3$
- $u_5 = u_4 + u_3 = 3 + 2 = 5$

Exemple :

Code python	Résultat dans la console
`print(fibo(10))`	`55`

Q1 Compléter et traduire l'algorithme de la fonction `def fib(n):` en python, puis valider le programme.

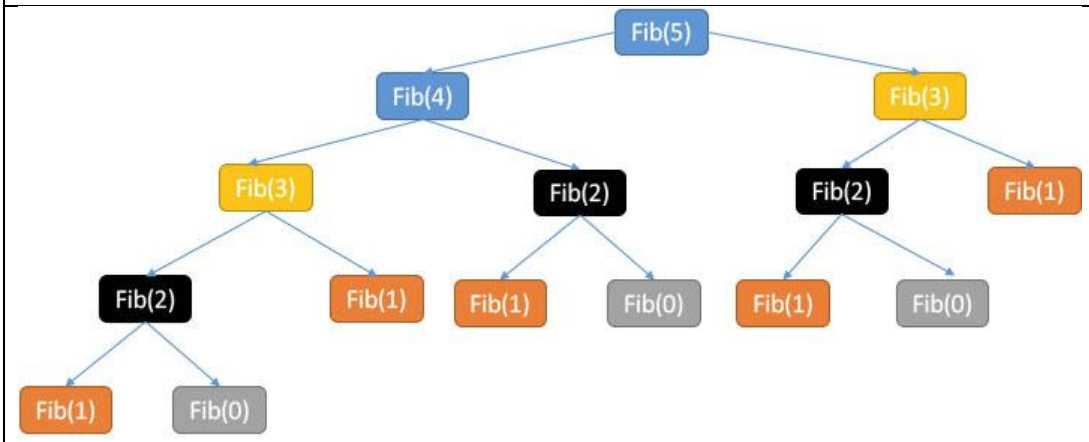
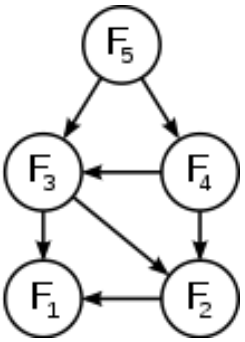
Algorithme à compléter	Code python à compléter

`fonction fib(n :entier) : entier`
`  si n< ?? alors`
`  retourner #a compléter`
`  sinon`
`  retourner #a compléter`
`  fin si`
 	
`affiche(fib (10))`
	`def fib(n):`
	`  if n<2:`
	`  return n`
	`  return fib(n-1)+fib(n-2)`
	
	`print(fib(10))`

Cette solution élégante présente cependant un gros défaut :

On effectue plusieurs fois le même calcul ce qui va induire une complexité en temps considérable.

Traitement du calcul de fib(5) :

L'arbre des appels	Graphe de dépendance des sous problème
	

Actuellement pour calculer fib(5) on doit calculer :

- 1 fois fib(4)
- 2 fois fib(3)
- 3 fois fib(2)
- 5 fois fib(1)
- 3 fois fib(0)

On a donc en tout 15 appels si on compte l'appels initial de F5.

On pourrait donc grandement simplifier le calcul en calculant par exemple une fois pour toutes fibo(2), en "mémorisant" le résultat et en le réutilisant quand nécessaire.

3.Mémoriser pour gagner

On se propose donc de mémoriser les résultats des sous problèmes et de les réutiliser si nécessaire.

3.1 Version itérative

Q2 Compléter et traduire l'algorithme de la fonction `def fib(n):` en python, puis valider le programme. Exécuter le code dans python tutor.

Algorithme à compléter	Code python à compléter

`fonction fib(n :entier) : entier`
 	
`  tab ← [0,1]`
`  Pour i de 2 à n`
`  tab.append(tab[i-1] + tab[i-2])`
`  retourner tab[n]`
`affiche(fib (10))`
	`def fib(n):`
	`  tab = [0, 1]`
	
	`  for i in range(2, n+1):`
	`  tab.append(tab[i-1] + tab[i-2])`
	
	`  return tab[n]`
	
	`print(fib(10))`

3.2 Version récursive

Q3 Tester la fonction `def fib(n):` en python.

On utilise un dictionnaire pour mémoriser les calculs intermédiaires. Exécuter le code dans python tutor.

Code python à analyser

```
def fib(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    else:
        memo[n] = fib(n-1, memo) + fib(n-2, memo)
        print(memo)
        return memo[n]

# Tester la fonction fibonacci pour n = 10
n = 10
print(fib(10))
```

```
{2: 1}
{2: 1, 3: 2}
{2: 1, 3: 2, 4: 3}
{2: 1, 3: 2, 4: 3, 5: 5}
{2: 1, 3: 2, 4: 3, 5: 5, 6: 8}
{2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13}
{2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21}
{2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21, 9: 34}
{2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21, 9: 34, 10: 55}
55
```

4.Retour sur le rendu de monnaie

Rappel :

4.1 Algorithme glouton

La situation : 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 la somme de 1,77€.

Q4 Compléter le tableau suivant et indique le nombre de pièces rendues.

Pièces	100cts	50 cts	10 cts	5 cts	2cts	Nombre de pièces rendues
Choisies	1	1	2	1	1	6

Vous devez maintenant rendre la somme de 11 cts.

Q5 Compléter le tableau suivant et indique le nombre de pièces rendues. Identifier le problème

Pièces	100cts	50 cts	10 cts	5 cts	2cts	Nombre de pièces rendues
Choisies	0	0	1	0	0	1

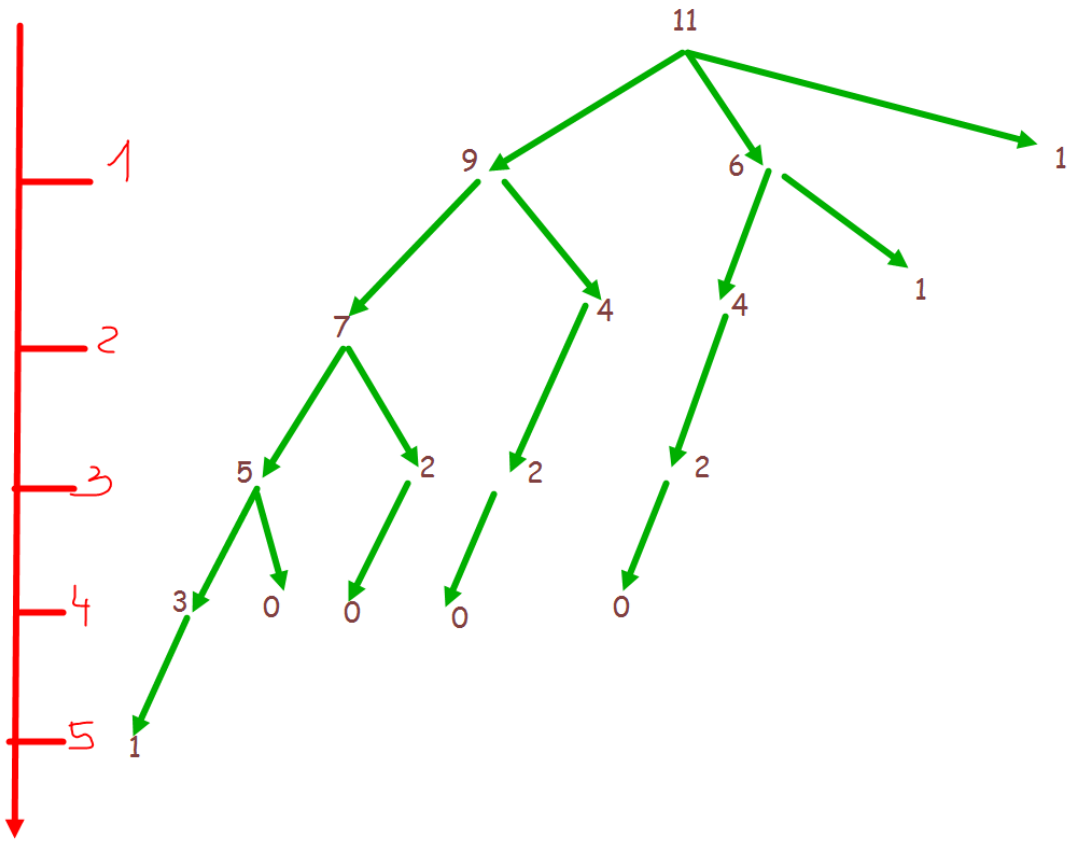
Il n'y a pas de pièce de 1 cts.

Q6 Trouvez une solution sans utiliser d'algorithme glouton.

Pièces	100cts	50 cts	10 cts	5 cts	2cts	Nombre de pièces rendues
Choisies	0	0	0	1	3	4

4.2 Résolution du problème

Utilisation de la méthode force brute descendante en testant toutes les solutions



Q7 Compléter les questions suivantes :

Combien dénombrez-vous de chemins ? 7

Combien l'arbre contient-il de nœuds ? 18

Parmi ces chemins, comment identifier ceux qui rendent correctement la monnaie : ils terminent par 0

Combien identifiez-vous de tels chemins ? 4 chemins

Quelle est la profondeur dans l'arbre de ces chemins ? 4

Correspondent-ils à des solutions différentes ? Non car cela revient à faire : 2 + 2 + 2 + 5

Quelle est la solution au problème ?

4, car la profondeur mini d'un chemin terminant par 0 est de 4

Q8 Tester la fonction `def rendu(pieces, s, memo={}) :` en python.

On utilise un dictionnaire pour mémoriser les calculs intermédiaires. Exécuter le code dans python tutor.

Code python à analyser

```
def rendu(pieces, s, memo={}):
    if s == 0:
        return 0, []
    elif s < 0:
        return float('inf'), []
    elif s in memo:
        return memo[s]

    min_pieces = float('inf')
    meilleures_pieces = []

    for p in pieces:
        nombre_pieces, pieces_rendues = rendu(pieces, s - p, memo)
        if nombre_pieces + 1 < min_pieces:
            min_pieces = nombre_pieces + 1
            meilleures_pieces = [p] + pieces_rendues

    memo[s] = min_pieces, meilleures_pieces
    print(memo)
    return min_pieces, meilleures_pieces

# Test de la fonction rendu
pieces = [2, 5, 10, 50, 100]
somme_a_rendre = 11

nombre_min_pieces, pieces_rendues = rendu(pieces, somme_a_rendre)
print("Nombre minimal de pièces nécessaires pour rendre la somme :",
nombre_min_pieces)
print("Pièces rendues :", pieces_rendues)
```

```
{1: (inf, [])}
{1: (inf, []), 3: (inf, [])}
{1: (inf, []), 3: (inf, []), 5: (1, [5])}
{1: (inf, []), 2: (1, [2]), 3: (inf, []), 5: (1, [5])}
{1: (inf, []), 2: (1, [2]), 3: (inf, []), 5: (1, [5]), 7: (2, [2, 5])}
{1: (inf, []), 2: (1, [2]), 3: (inf, []), 4: (2, [2, 2]), 5: (1, [5]), 7: (2, [2, 5])}
{1: (inf, []), 2: (1, [2]), 3: (inf, []), 4: (2, [2, 2]), 5: (1, [5]), 7: (2, [2, 5]), 9: (3, [2, 2, 5])}
{1: (inf, []), 2: (1, [2]), 3: (inf, []), 4: (2, [2, 2]), 5: (1, [5]), 6: (3, [2, 2, 2]), 7: (2, [2, 5]), 9: (3, [2, 2, 5])}
{1: (inf, []), 2: (1, [2]), 3: (inf, []), 4: (2, [2, 2]), 5: (1, [5]), 6: (3, [2, 2, 2]), 7: (2, [2, 5]), 9: (3, [2, 2, 5]), 11: (4, [2, 2, 2,
5])}
Nombre minimal de pièces nécessaires pour rendre la somme : 4
Pièces rendues : [2, 2, 2, 5]55
```

5.Conclusion

1. **Approche générale** :
   - La programmation dynamique implique la résolution de problèmes en divisant ces derniers en sous-problèmes plus simples, en résolvant chaque sous-problème une seule fois, puis en stockant sa solution pour éviter de recalculer.

2. **Stockage des résultats** :
   - La programmation dynamique utilise la mémorisation (memoization) pour stocker les résultats des sous-problèmes déjà résolus, ce qui permet d'éviter les recalculs inutiles.

3. **Complexité temporelle** :
   - La programmation dynamique peut réduire la complexité temporelle d'un algorithme en évitant les recalculs, ce qui peut souvent conduire à des solutions plus efficaces.

4. **Stratégie d'implémentation** :
   - La programmation dynamique est souvent utilisée lorsque les sous-problèmes se chevauchent, c'est-à-dire qu'ils partagent des sous-problèmes communs.

En résumé, la programmation dynamique est une technique algorithmique spécifique qui vise à résoudre des problèmes en évitant les recalculs inutiles grâce à la mémorisation des résultats des sous-problèmes