cour : mémorisation

1. Définition:

- La mémorisation (en anglais "memoization") est une technique d'optimisation en algorithmique qui consiste à stocker les résultats de calculs coûteux pour les réutiliser ultérieurement au lieu de recalculer ces résultats à chaque fois qu'ils sont nécessaires.
- Cela permet d'éviter les calculs répétitifs, d'améliorer les performances et de réduire la complexité temporelle de l'algorithme.
- La mémorisation est couramment utilisée pour résoudre des problèmes de manière récursive, en particulier dans le contexte de la programmation dynamique.

2. Principes Fondamentaux

Les principaux éléments de la mémorisation sont les suivants :

1. **Table de Mémorisation** :
    - Il s'agit d'une structure de données, généralement un tableau ou un dictionnaire, qui est utilisée pour stocker les résultats des calculs précédents.
    - Chaque élément de la table est associé à un ensemble de paramètres d'entrée et contient le résultat calculé pour ces paramètres.
2. **Vérification de la Table** :
    - Avant de calculer une valeur, on vérifie d'abord si elle existe dans la table de mémorisation :
        - Si elle existe, on la renvoie immédiatement, évitant ainsi un nouveau calcul.
3. **Calcul et Stockage** :
    - Si la valeur n'existe pas dans la table, on la calcule de manière récursive ou itérative, puis on la stocke dans la table de mémorisation pour une utilisation ultérieure.

3.Exemple : Suite de Fibonacci

- Prenons l'exemple classique de la suite de Fibonacci pour illustrer la mémorisation :
    - **La suite de Fibonacci :** est définie comme suit : F(0) = 0, F(1) = 1, et F(n) = F(n-1) + F(n-2) pour n > 1.
- **Algorithme Récursif sans Mémorisation :**

```
def fibonacci(n):
    if n <= 1:
        return n
```

```
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Cet algorithme récursif calcule la suite de Fibonacci en effectuant de nombreux calculs répétitifs.

Par exemple, pour calculer F(5), il recalculera F(3) et F(4), même si ces valeurs ont déjà été calculées. La complexité temporelle est exponentielle.

- **Algorithme avec Mémorisation :**

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

- Dans cet algorithme avec mémorisation, nous utilisons un dictionnaire `memo` pour stocker les résultats des calculs précédents.
- Avant de calculer F(n), nous vérifions d'abord si sa valeur existe dans le dictionnaire :
    - Si c'est le cas, nous la renvoyons immédiatement.
    - Sinon, nous calculons F(n) de manière récursive, stockons le résultat dans le dictionnaire, puis le renvoyons.
- Grâce à la mémorisation, les calculs répétitifs sont évités, et la complexité temporelle devient linéaire.

4. Avantages de la Mémorisation :

1. Réduction significative de la complexité temporelle, en particulier pour les problèmes récursifs.
2. Élimination des calculs redondants.
3. Amélioration des performances de l'algorithme.

RQ :

- **Limitations de la Mémorisation :**
    1. Utilisation de mémoire supplémentaire pour stocker les résultats intermédiaires.
    2. L'efficacité dépend de la capacité de prédire quels résultats seront réutilisés.