Complexité des algorithmes

EXERCICES :

2-d Tree :

1.

1. **Finding the median point p in P_y:**
   - The points are already sorted along their y-coordinates in the sequence P_y.
   - Finding the median (the $n/2$-th smallest element) in a sorted array takes constant time:

$$O(1)$$

2. **Dividing P_y into two sorted subsequences $P_{y,1}$ and $P_{y,2}$:**
   - This involves iterating through P_y and checking whether each point belongs to the lower half-plane or the upper half-plane based on the x-coordinate of p (determined during the previous step).
   - This step requires one linear scan of P_y, which takes:

$$O(n)$$

2.

Steps

1. **Iterate through P_x:**
   - P_x is already sorted by x-coordinates.
   - For each point in P_x, check whether its x-coordinate is less than, equal to, or greater than the x-coordinate of p.
   - This step involves a linear scan of P_x, taking $O(n)$ time.
2. **Divide P_x into two sorted subsequences:**
   - Create two new arrays:
     - One for points with $x < x_p$ or $x = x_p$ (lower half-plane points).
     - One for points with $x > x_p$ (upper half-plane points).
   - Since P_x is already sorted, no additional sorting is required after dividing.

Complexity Analysis

- Iterating through P_x: $O(n)$.
- Partitioning into two arrays: $O(n)$, as each point is assigned to one of the two new arrays during the single scan.

Thus, the **worst-case time complexity** of this step is:

$$O(n)$$

3.

1. **Finding the median point:**
   - This is $O(1)$ for accessing the median since the points are pre-sorted.
2. **Dividing P_y into two subsequences:**
   - This step has a cost of $O(n)$.
3. **Dividing P_x into two subsequences:**
   - This step also has a cost of $O(n)$.

Each of the two resulting subsets, P_1 and P_2, has approximately half the points ($n/2$) in the worst case.

Thus, the recurrence relation for the **worst-case time complexity** $T(n)$ is:

$$T(n) = 2T(n/2) + O(n)$$

Solving the Recurrence Relation

Using the Master Theorem:

- The recurrence is of the form:

  $$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

  where $a = 2$, $b = 2$, and $d = 1$.
- Calculate $\log_b a$:

  $$\log_2 2 = 1$$
- Compare d with $\log_b a$:
  - Here, $d = \log_b a$, so the solution is:

$$T(n) = O(n \log n)$$

4.

1. **Preliminary Sorting:**
   - The points are sorted in ascending order along their x-coordinates to create P_x and along their y-coordinates to create P_y.
   - Sorting each array of size n takes:

     $$O(n \log n)$$
   - Sorting is done twice (once for P_x and once for P_y), so the total sorting cost is:

     $$O(n \log n)$$
2. **Building the 2-d Tree:**
   - As derived previously, the worst-case time complexity for recursively building the tree is:

     $$O(n \log n)$$
3. **Space Requirements:**
   - During sorting, the algorithm may require $O(n)$ additional space to store temporary arrays.
   - For recursion, the depth of the recursion tree is $O(\log n)$, with $O(n)$ space needed at each level to store subsequences like P_x and P_y. This results in a total space complexity of:

     $$O(n \log n)$$

Complexité des algorithmes

Maximum Subsequence Sum :

1.

```python
def sum_m(lst):
    max_i = 0
    max_j = 0
    max_sum = 0

    for i in range(len(lst)):
        current_sum = 0
        for j in range(i, len(lst)):
            current_sum += lst[j]
            if current_sum > max_sum:
                max_sum = current_sum
                max_i = i
                max_j = j

    return max_sum, max_i, max_j
```

2.

Complexité

Time Complexity

- **Outer loop** (i): Runs n times.
- **Inner loop** (j): Runs $n - i$ times for each i.
- For each pair (i, j), an addition is performed to maintain $current\_sum$, taking $O(1)$.

The total number of iterations is:

$$\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} O(1) = \sum_{i=0}^{n-1} (n - i) = O(n^2)$$

- **Worst-case time complexity**:

$$O(n^2)$$

Space Complexity

- The algorithm uses a constant amount of additional space for variables ($max\_sum, max\_i, max\_j, current\_sum$).
- **Worst-case space complexity**:

$$O(1)$$

Complexité des algorithmes

3.

```python
def max_crossing_sum(arr, left, mid, right):
    # Find the maximum sum of the left part (ending at mid)
    left_sum = float('-inf')
    current_sum = 0
    for i in range(mid, left - 1, -1):
        current_sum += arr[i]
        left_sum = max(left_sum, current_sum)

    # Find the maximum sum of the right part (starting at mid + 1)
    right_sum = float('-inf')
    current_sum = 0
    for i in range(mid + 1, right + 1):
        current_sum += arr[i]
        right_sum = max(right_sum, current_sum)

    # Combine the maximum sums from both sides
    return left_sum + right_sum


def max_subsequence_sum(arr, left, right):
    # Base case: single element
    if left == right:
        return max(0, arr[left])  # Return 0 for all-negative case

    # Divide the array into two halves
    mid = (left + right) // 2

    # Recursively find maximum subsequence sum in left and right halves
    left_max = max_subsequence_sum(arr, left, mid)
    right_max = max_subsequence_sum(arr, mid + 1, right)

    # Find the maximum subsequence sum crossing the midpoint
    cross_max = max_crossing_sum(arr, left, mid, right)

    # Return the maximum of the three cases
    return max(left_max, right_max, cross_max)
```

4.

1. **Recursive Subproblems:**
   - At each level, the array is divided into two halves, resulting in two recursive calls. This takes $2T(n/2)$ for an array of size n.
2. **Combining Results (Crossing Sum):**
   - Calculating the crossing sum involves two linear scans of the subarray, each taking $O(n)$.

Thus, the recurrence relation for the algorithm is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Using the **Master Theorem**:

- $a = 2, b = 2$, and $d = 1$ (from $O(n^d)$).
- $\log_b a = \log_2 2 = 1$.
- Since $d = \log_b a$, the time complexity is:

$$T(n) = O(n \log n)$$

Space Complexity

1. **Recursive Stack:**
   - The depth of the recursion tree is $O(\log n)$, as the array is divided in half at each level.
2. **Auxiliary Variables:**
   - Only a constant amount of space is used for variables like $left\_sum, right\_sum$, etc.

Thus, the **space complexity** is:

$$O(\log n)$$

Complexité des algorithmes

Pyramide de nombres :

1.

1. **At Each Level:**
   - From any node at level i, there are **two choices**:
     - Move to the left child.
     - Move to the right child.
2. **Total Number of Choices:**
   - There are $n-1$ steps from the top (level 1) to the bottom (level n).
   - At each step, the choice is binary (left or right).
3. **Total Paths:**
   - The total number of paths corresponds to the number of binary sequences of length $n-1$, which is:

$$2^{n-1}$$

Why?

Each path can be uniquely represented by a sequence of $n-1$ choices (e.g., $LLRR$ means two steps to the left and two to the right). Since each choice is independent and binary, the total number of sequences is 2^{n-1}.

2.

1. **Base Case:**
   - If i is the last row, return $w[i][j]$, as there are no children.
2. **Recursive Case:**
   - Compute the maximum path sum from the left child ($w[i+1][j]$) and the right child ($w[i+1][j+1]$).
   - Add $w[i][j]$ to the greater of these two values.
3. **Starting Point:**
   - Call `max_path_sum_from(0, 0)` to compute the maximum path sum starting from the top of the pyramid.

---

python Copier le code

```python
def max_path_sum(pyramid):
    def max_path_sum_from(i, j):
        # Base case: if we're at the last row, return the current cell value
        if i == len(pyramid) - 1:
            return pyramid[i][j]

        # Recursive case: compute the maximum path sum from children
        left_child = max_path_sum_from(i + 1, j)
        right_child = max_path_sum_from(i + 1, j + 1)

        # Return the current value plus the maximum of the children's path sums
        return pyramid[i][j] + max(left_child, right_child)

    # Start from the top of the pyramid
    return max_path_sum_from(0, 0)
```

3.

Complexity Analysis

1. **Time Complexity:**
   - Each call to `max_path_sum_from(i, j)` results in two recursive calls for the children.
   - This leads to exponential time complexity in the worst case:
     $$O(2^n)$$
   - n is the height of the pyramid.
2. **Space Complexity:**
   - The recursion stack depth is proportional to the height of the pyramid:
     $$O(n)$$

4.

1. **Initialize from the Bottom:**
   - The last row of the pyramid contains the base values; these are the maximum sums for the paths ending at those elements.
2. **Iterative Step:**
   - For each row i from the second-to-last row up to the top, compute the maximum sum for each element $w[i][j]$:
     $$w[i][j] = \text{value of } w[i][j] + \max(w[i+1][j], w[i+1][j+1])$$
   - This step replaces each cell $w[i][j]$ with the maximum path sum from that cell to the bottom of the pyramid.
3. **Final Result:**
   - After processing all rows, the top element $w[0][0]$ contains the maximum path sum.

Code Implementation

python Copier le code

```python
def max_path_sum(pyramid):
    n = len(pyramid)  # Height of the pyramid

    # Start from the second-to-last row and work upwards
    for i in range(n - 2, -1, -1):
        for j in range(i + 1):
            pyramid[i][j] += max(pyramid[i + 1][j], pyramid[i + 1][j + 1])

    # The top element contains the maximum path sum
```

5. **Worst-Case Time and Space Complexity**

1. **Time Complexity:**
   - The algorithm processes each element in the pyramid exactly once, performing a constant amount of work per element.
   - The total number of elements in a pyramid of height n is:

$$\text{Total elements} = \frac{n(n+1)}{2}$$

   - Thus, the time complexity is:

$$O(n^2)$$

2. **Space Complexity:**
   - The algorithm modifies the input matrix w in-place to store intermediary results.
   - No additional space is required, so the space complexity is:

$$O(1)$$

   (excluding the input storage).

Knapsack

1.

```python
def knap(w, v, c, n=None):
    """
    Fonction récursive pour résoudre le problème du sac à dos.

    Args:
    - w: Liste des poids des objets.
    - v: Liste des valeurs des objets.
    - c: Capacité maximale du sac à dos.
    - n: Nombre d'objets considérés (par défaut, tous les objets).

    Returns:
    - La valeur maximale que le sac à dos peut contenir.
    """
    if n is None:
        n = len(w)  # Si n n'est pas fourni, on considère tous les objets.

    # Cas de base : pas d'objets ou capacité nulle
    if n == 0 or c == 0:
        return 0

    # Si le poids de l'objet actuel dépasse la capacité restante, on l'exclut
    if w[n - 1] > c:
        return knap(w, v, c, n - 1)

    # Sinon, on choisit le maximum entre :
    # - Exclure l'objet actuel.
    # - Inclure l'objet actuel et soustraire son poids de la capacité.
    return max(
        knap(w, v, c, n - 1),  # Ne pas inclure l'objet actuel
        knap(w, v, c - w[n - 1], n - 1) + v[n - 1]  # Inclure l'objet actuel
    )
```

2. **Complexité**

   1. **Temps :**
      - Cette solution est **exponentielle** en temps, car elle explore toutes les combinaisons possibles d'inclusion/exclusion des objets :

        $$O(2^n)$$

      - Les sous-problèmes sont recalculés plusieurs fois, ce qui explique l'explosion de la complexité.
   2. **Espace :**
      - La profondeur maximale de la pile d'appels est $O(n)$, correspondant au nombre d'objets.

Complexité des algorithmes

3.

python Copier le code

```python
def knap_dyn(w, v, c):
    n = len(w)
    # Création d'un tableau 2D pour stocker les résultats intermédiaires
    dp = [[0] * (c + 1) for _ in range(n + 1)]

    # Remplir le tableau dp avec les solutions optimales
    for i in range(1, n + 1):  # Parcourir chaque objet
        for j in range(1, c + 1):  # Parcourir chaque capacité de 1 à c
            if w[i - 1] <= j:  # Si l'objet i peut être inclus dans le sac
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i - 1]] + v[i - 1])
            else:  # Sinon, ne pas inclure cet objet
                dp[i][j] = dp[i - 1][j]

    # La valeur maximale du sac à dos est dp[n][c]
    return dp[n][c]

# Exemple d'utilisation
if __name__ == "__main__":
    w = [1, 2, 3, 2]  # Poids des objets
    v = [10, 15, 40, 25]  # Valeurs des objets
    c = 5  # Capacité maximale du sac
    print("Valeur maximale:", knap_dyn(w, v, c))  # Output attendu : 55
```

Explication du Code

1. **Tableau `dp` :**
    - La table `dp` est de taille $(n+1) \times (c+1)$ où n est le nombre d'objets et c est la capacité du sac.
    - Chaque entrée $dp[i][j]$ représente la valeur maximale possible avec les i premiers objets et une capacité de sac de j.
2. **Initialisation :**
    - La première ligne et la première colonne de la table sont initialisées à 0, car :
        - $dp[0][j] = 0$ pour toute capacité j (pas d'objets, pas de valeur).
        - $dp[i][0] = 0$ pour tout i (sac de capacité 0, aucune valeur ne peut être stockée).
3. **Remplissage de la table :**
    - On parcourt chaque objet (de $i = 1$ à n) et chaque capacité de sac (de $j = 1$ à c).
    - À chaque étape, si le poids de l'objet actuel est inférieur ou égal à la capacité j, on choisit entre inclure ou exclure cet objet, et on garde la meilleure valeur.
    - Sinon, on exclut l'objet.
4. **Résultat :**
    - La valeur maximale est contenue dans $dp[n][c]$, qui représente la valeur maximale possible avec n objets et une capacité c.

Complexité des algorithmes

4.

Complexité Temporelle et Spatiale

1. **Complexité Temporelle :**
   - Nous remplissons une table dp de taille $(n+1) \times (c+1)$, ce qui donne une complexité de :
   $$O(n \times c)$$
   - Cette approche est beaucoup plus rapide que la méthode récursive brute, qui avait une complexité exponentielle.
2. **Complexité Spatiale :**
   - Nous utilisons une table dp de taille $O(n \times c)$, donc la complexité spatiale est également :
   $$O(n \times c)$$
   - Cependant, on peut optimiser l'espace en utilisant une table 1D si l'on remarque que pour remplir la ligne i de dp, on n'a besoin que de la ligne $i-1$.

<u>Problème du sac à dos non borné :</u>

1.

1. Fonction récursive directe

La fonction récursive directe est basée sur la relation de récurrence suivante :

$$t(j) = \max_{i \in [1,n],\, w(x_i) \leq j} (t(j - w(x_i)) + v(x_i))$$

Si aucun objet ne peut être inclus (par exemple $j = 0$), $t(j) = 0$.

Voici l'implémentation :

python Copier le code

```python
def unbounded_knapsack_recursive(w, v, c):
    # Cas de base
    if c == 0:
        return 0

    # Calculer le maximum pour tous les objets pouvant être inclus
    max_value = 0
    for i in range(len(w)):
        if w[i] <= c:  # Si l'objet peut être inclus
            max_value = max(max_value, unbounded_knapsack_recursive(w, v, c - w[i]) + v[i]

    return max_value

# Exemple d'utilisation
if __name__ == "__main__":
    w = [1, 2, 3]  # Poids des objets
    v = [10, 15, 40]  # Valeurs des objets
    c = 5  # Capacité du sac
    print("Valeur maximale (récursive):", unbounded_knapsack_recursive(w, v, c))  # Output
```

Complexité des algorithmes

2.

2. Complexités temporelle et spatiale en pire cas

Hypothèse : $w(x_i) = 1$ **pour tout** i

1. **Complexité temporelle :**
   - À chaque appel récursif, nous effectuons une boucle sur les n objets, et la profondeur maximale de la récursion est c (car on réduit c de 1 à chaque appel dans le pire cas).
   - La complexité temporelle est donc :
   $$O(n^c)$$
2. **Complexité spatiale :**
   - Chaque appel récursif utilise de l'espace sur la pile d'appels. Dans le pire cas, la profondeur de la pile est c.
   - La complexité spatiale est donc :
   $$O(c)$$

3.

3. Programmation dynamique

En utilisant la programmation dynamique, nous pouvons éviter les calculs redondants. Nous créons un tableau dp où $dp[j]$ représente la valeur maximale pour une capacité j.

L'algorithme est basé sur l'équation :

$$dp[j] = \max_{i \in [1,n],\, w(x_i) \leq j} (dp[j - w(x_i)] + v(x_i))$$

Voici l'implémentation :

python   Copier le code

```python
def unbounded_knapsack_dp(w, v, c):
    # Initialisation du tableau dp
    dp = [0] * (c + 1)

    # Remplissage du tableau
    for j in range(1, c + 1):  # Parcourir toutes les capacités
        for i in range(len(w)):  # Parcourir tous les objets
            if w[i] <= j:  # Si l'objet peut être inclus
                dp[j] = max(dp[j], dp[j - w[i]] + v[i])

    return dp[c]

# Exemple d'utilisation
if __name__ == "__main__":
    w = [1, 2, 3]  # Poids des objets
    v = [10, 15, 40]  # Valeurs des objets
    c = 5  # Capacité du sac
    print("Valeur maximale (DP):", unbounded_knapsack_dp(w, v, c))  # Output attendu : 50
```

4.

4. Complexités temporelle et spatiale pour l'algorithme dynamique

1. **Complexité temporelle :**
   - Nous avons une boucle sur toutes les capacités j (de 1 à c) et une boucle interne sur les n objets.
   - La complexité temporelle est donc :
     $$O(n \times c)$$
2. **Complexité spatiale :**
   - Nous utilisons un tableau dp de taille $c + 1$.
   - La complexité spatiale est donc :
     $$O(c)$$

Résumé

Algorithme	Complexité temporelle	Complexité spatiale
Récursif	$O(n^c)$	$O(c)$
DP	$O(n \times c)$	$O(c)$

La programmation dynamique est nettement plus efficace que la version récursive brute, en particulier pour de grandes valeurs de c et n.

Rendu de monnaie :

1.

1. Fonction récursive directe

La fonction récursive directe calcule $M(s)$ en utilisant la relation de récurrence :

$$M(s) = 1 + \min_{\forall i \in [1,n],\ v_i \leq s} M(s - v_i)$$

avec $M(0) = 0$ (aucune pièce n'est nécessaire pour rendre un montant nul).

Voici l'implémentation :

python Copier le code

```python
def coin_change_recursive(values, s):
    # Cas de base
    if s == 0:
        return 0

    # Si aucune pièce ne peut être utilisée
    if s < 0:
        return float('inf')  # Impossible de rendre le montant

    # Calculer le minimum en essayant toutes les pièces possibles
    min_coins = float('inf')
    for v in values:
        min_coins = min(min_coins, 1 + coin_change_recursive(values, s - v))

    return min_coins

# Exemple d'utilisation
if __name__ == "__main__":
    values = [1, 2, 5]
    s = 9
    print("Nombre minimal de pièces (récursif) :", coin_change_recursive(values, s))  # Ou
```

2.

2. Complexités temporelle et spatiale (pire cas)

Hypothèse : $\forall i, v_i = 1$

1. **Complexité temporelle :**
   - À chaque appel récursif, on effectue une boucle sur n pièces.
   - La profondeur de la récursion est au plus s (puisque s diminue d'au moins 1 à chaque appel dans ce cas).
   - Dans le pire cas, on effectue $O(n^s)$ appels récursifs.
   - Complexité temporelle :
   $$O(n^s)$$
2. **Complexité spatiale :**
   - La pile d'appels contient au plus s niveaux en même temps.
   - Complexité spatiale :
   $$O(s)$$

3.

3. Programmation dynamique

Avec la programmation dynamique, nous utilisons un tableau dp où $dp[k]$ représente le nombre minimal de pièces pour rendre k.

La relation de récurrence devient :

$$dp[k] = 1 + \min_{\forall i \in [1,n],\ v_i \leq k} dp[k - v_i]$$

avec $dp[0] = 0$.

Voici l'implémentation :

python Copier le code

```python
def coin_change_dp(values, s):
    # Initialisation du tableau dp
    dp = [float('inf')] * (s + 1)
    dp[0] = 0  # Cas de base : 0 pièce pour un montant 0

    # Remplir le tableau
    for k in range(1, s + 1):
        for v in values:
            if k >= v:  # Si la pièce peut être utilisée
                dp[k] = min(dp[k], 1 + dp[k - v])

    return dp[s] if dp[s] != float('inf') else -1  # -1 si le montant ne peut pas être ren

# Exemple d'utilisation
if __name__ == "__main__":
    values = [1, 2, 5]
    s = 9
    print("Nombre minimal de pièces (DP) :", coin_change_dp(values, s))  # Output attendu
```

4.

4. Complexités temporelle et spatiale (pire cas)

1. **Complexité temporelle :**
   - Nous avons une boucle pour chaque montant $k \in [1, s]$ (soit s itérations).
   - À chaque montant k, on parcourt n pièces.
   - Complexité temporelle :
   $$O(n \times s)$$
2. **Complexité spatiale :**
   - Nous utilisons un tableau dp de taille $s + 1$.
   - Complexité spatiale :
   $$O(s)$$

5.

5. Complexité en fonction de la taille k pour encoder s en base 2

La taille k nécessaire pour encoder s en base 2 est $k = \log_2(s)$. Par conséquent :

- Le nombre de montants s est exponentiel en k ($s = 2^k$).
- La complexité temporelle $O(n \times s)$ devient $O(n \times 2^k)$.

Discussion : $P = NP$?

- Cet algorithme a une complexité exponentielle en k, ce qui est cohérent avec le fait que le problème du rendu de monnaie est NP-difficile.
- La programmation dynamique ne résout pas le problème en temps polynomial en k, donc cela **ne prouve pas** que $P = NP$. Le problème reste NP-difficile, et les $1M du Clay Mathematics Institute restent intacts.

Complexité des algorithmes

BONUS :

1. Problème de la chaîne de matrices

- **Enoncé** : On a une chaîne de n matrices $A_1, A_2, \ldots, A_n$, et chaque matrice A_i a des dimensions $p_{i-1} \times p_i$. L'objectif est de déterminer l'ordre de multiplication parenthésé optimal qui minimise le nombre total d'opérations scalaires.
- **Exemple** : Pour $A_1 = 10 \times 20$, $A_2 = 20 \times 30$, $A_3 = 30 \times 40$, trouvez la manière optimale de multiplier ces matrices.
- **Indication** : Utilisez une matrice $dp[i][j]$ pour stocker le coût minimal pour multiplier A_i à A_j.

1. Problème de la chaîne de matrices

Code

python Copier le code

```python
def matrix_chain_order(dims):
    n = len(dims) - 1
    dp = [[0] * n for _ in range(n)]

    for length in range(2, n + 1):  # length of chain
        for i in range(n - length + 1):
            j = i + length - 1
            dp[i][j] = float('inf')
            for k in range(i, j):
                cost = dp[i][k] + dp[k + 1][j] + dims[i] * dims[k + 1] * dims[j + 1]
                dp[i][j] = min(dp[i][j], cost)
    return dp[0][n - 1]
```

Complexité

- **Temps** : $O(n^3)$
- **Espace** : $O(n^2)$

2. Problème de la plus longue sous-séquence commune (LCS)

- **Enoncé** : Étant donné deux chaînes X et Y, trouvez la plus longue sous-séquence commune entre les deux.
- **Exemple** : Si $X = "AGGTAB"$ et $Y = "GXTXAYB"$, la LCS est $"GTAB"$, de longueur 4.
- **Indication** : Utilisez une table $dp[i][j]$ pour stocker la longueur de la LCS entre les i-premiers caractères de X et les j-premiers de Y.

2. Problème de la plus longue sous-séquence commune (LCS)

Code

python Copier le code

```python
def lcs(X, Y):
    m, n = len(X), len(Y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i - 1] == Y[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
    return dp[m][n]
```

Complexité

- **Temps** : $O(m \cdot n)$
- **Espace** : $O(m \cdot n)$

3. Problème du découpage de barre

- **Enoncé** : Une barre d'une longueur n peut être coupée en morceaux de différentes tailles, avec un prix associé à chaque taille. Trouvez la manière optimale de découper la barre pour maximiser le profit.
- **Exemple** : Si $n = 8$ et les tailles disponibles sont $[1, 2, 3, 4]$ avec des prix $[1, 5, 8, 9]$, trouvez la découpe optimale.
- **Indication** : Utilisez une table $dp[i]$ pour stocker le profit maximal pour une barre de longueur i.

3. Problème du découpage de barre

Code

python Copier le code

```python
def rod_cut(prices, n):
    dp = [0] * (n + 1)
    for i in range(1, n + 1):
        for j in range(1, i + 1):
            dp[i] = max(dp[i], prices[j - 1] + dp[i - j])
    return dp[n]
```

Complexité

- **Temps** : $O(n^2)$
- **Espace** : $O(n)$

4. Problème du nombre de chemins dans une grille

- **Enoncé** : Trouvez le nombre de chemins possibles pour aller de la case en haut à gauche $(0, 0)$ à la case en bas à droite $(n - 1, m - 1)$ dans une grille $n \times m$, en ne pouvant se déplacer que vers la droite ou vers le bas.
- **Exemple** : Pour $n = 3$, $m = 3$, le nombre de chemins est 6.
- **Indication** : Utilisez une table $dp[i][j]$ où $dp[i][j]$ représente le nombre de chemins pour atteindre (i, j).

4. Problème du nombre de chemins dans une grille

Code

python Copier le code

```python
def unique_paths(m, n):
    dp = [[1] * n for _ in range(m)]
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1]
    return dp[m - 1][n - 1]
```

Complexité

- **Temps** : $O(m \cdot n)$
- **Espace** : $O(m \cdot n)$

5. Problème du saut de grenouille (Frog Jump)

- **Enoncé** : Une grenouille est sur une série de pierres alignées et doit sauter d'une pierre à une autre pour atteindre la dernière. Chaque saut coûte de l'énergie, et l'objectif est de minimiser le coût total. L'énergie pour aller de i à j est $|h[i] - h[j]|$, où $h[i]$ est la hauteur de la pierre i.
- **Exemple** : Si $h = [10, 30, 40, 20]$, le coût minimal est 30.
- **Indication** : Utilisez une table $dp[i]$ pour stocker le coût minimal pour atteindre la pierre i.

5. Problème du saut de grenouille (Frog Jump)

Code

python Copier le code

```python
def frog_jump(h):
    n = len(h)
    dp = [0] * n
    dp[0] = 0
    for i in range(1, n):
        dp[i] = min(dp[i - 1] + abs(h[i] - h[i - 1]),
                    dp[i - 2] + abs(h[i] - h[i - 2]) if i > 1 else float('inf'))
    return dp[n - 1]
```

Complexité

- **Temps** : $O(n)$
- **Espace** : $O(n)$

6. Problème de partition égale

- **Enoncé** : Donnez un tableau $nums$, déterminez s'il est possible de le partitionner en deux sous-ensembles dont les sommes sont égales.
- **Exemple** : Si $nums = [1, 5, 11, 5]$, la réponse est *True* ($[1, 5, 5]$ et $[11]$).
- **Indication** : Utilisez une approche dynamique similaire au problème du sac à dos.

6. Problème de partition égale

Code

python Copier le code

```python
def can_partition(nums):
    total = sum(nums)
    if total % 2 != 0:
        return False
    target = total // 2
    dp = [False] * (target + 1)
    dp[0] = True
    for num in nums:
        for j in range(target, num - 1, -1):
            dp[j] |= dp[j - num]
    return dp[target]
```

Complexité

- **Temps** : $O(n \cdot \text{sum}(\text{nums}))$
- **Espace** : $O(\text{sum}(\text{nums}))$

7. Problème de la somme cible (Target Sum)

- **Enoncé** : Donnez un tableau $nums$ et une cible S, trouvez le nombre de façons d'ajouter $+$ ou $-$ entre les éléments pour obtenir S.
- **Exemple** : Si $nums = [1, 1, 1, 1, 1]$, $S = 3$, le résultat est 5.
- **Indication** : Construisez une table dynamique où chaque $dp[i][s]$ représente le nombre de façons de créer la somme s en utilisant les i-premiers éléments.

7. Problème de la somme cible (Target Sum)

Code

python Copier le code

```python
def target_sum(nums, S):
    total = sum(nums)
    if (total + S) % 2 != 0 or total < abs(S):
        return 0
    target = (total + S) // 2
    dp = [0] * (target + 1)
    dp[0] = 1
    for num in nums:
        for j in range(target, num - 1, -1):
            dp[j] += dp[j - num]
    return dp[target]
```

Complexité

- **Temps** : $O(n \cdot \mathrm{sum}(\mathrm{nums}))$
- **Espace** : $O(\mathrm{sum}(\mathrm{nums}))$

8. Problème de l'édition de texte (Edit Distance)

- **Enoncé** : Étant donné deux chaînes A et B, trouvez le nombre minimal d'opérations nécessaires pour transformer A en B. Les opérations autorisées sont l'insertion, la suppression et la substitution.
- **Exemple** : Si $A = "horse"$ et $B = "ros"$, le résultat est 3.
- **Indication** : Utilisez une table $dp[i][j]$ pour stocker le coût minimal pour transformer les i-premiers caractères de A en les j-premiers de B.

8. Problème de l'édition de texte (Edit Distance)

Code

python   Copier le code

```python
def edit_distance(A, B):
    m, n = len(A), len(B)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif A[i - 1] == B[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])
    return dp[m][n]
```

Complexité

- **Temps :** $O(m \cdot n)$
- **Espace :** $O(m \cdot n)$

Transformée de FOURIER

L'algorithme de transformée de FOURIER rapide (FFT pour *Fast Fourier Transform*) peut s'écrire comme suit dans un pseudo-langage non-typé qui autorise les tableaux en argument et en résultat des fonctions, avec de l'arithmétique en nombre complexe :

```
FUNCTION DIF(N,f);
LOCAL N',n,fe,fo,Fe,Fo,k',F;
IF N==1
  THEN RETURN(f);
  ELSE BEGIN
    N':=N/2;
    FOR n:=0 TO N'-1 DO BEGIN
      fe[n]:=f[n]+f[n+N'];
      fo[n]:=(f[n]-f[n+N'])*T(N,n)
    END;
    Fe:=DIF(N', fe);
    Fo:=DIF(N', fo);
    FOR k':=0 TO N'-1 DO BEGIN
      F[2*k'  ] := Fe[k'];
      F[2*k'+1] := Fo[k'];
    END
    RETURN(F);
  END;
```

où T(N,n) calcule en temps constant $e^{-j2\pi n/N}$.

1. Calculer les complexités temporelle et spatiale de l'algorithme DIF.

Complexité des algorithmes

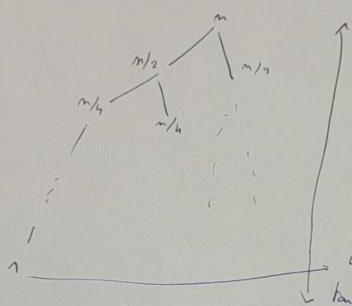
Exercices : 1) Transformée de Fourier

→ n : taille du tableau
F : tableau

On découpe le problème en 2 et après on refait 2 appels récursifs

$T(n) = 2T(n/2) + f(n)$
↳ $\in \Theta(n)$
↳ 2 boucles for

D'après le théorème Master :
$T(n) \in \Theta(n \log n)$



$h = \log_2 n$
↳ si mémoire est constante

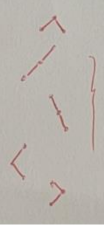
on a : $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$
$= 2n - 1$

on a au début un tableau de taille 1, ensuite taille 2, 4 ...

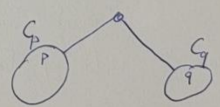
Donc, $S(n) \in \Theta(n)$

2) Catalans : nombre de possibilités d'agencer un arbre binaire,

n	0	1	2	3	4	5	6
C_n	1	1	2	5	14	42	132



5 possibilités



$n = p + q + 1$
$q = n - p - 1$

$$C_n = \sum_{\substack{p, q \\ t.q \\ p+q+1=n}} C_p \times C_q = \sum_{p=0}^{n-1} C_p \times C_{n-p-1}$$

Complexité des algorithmes

```
let rec c = fun n ->
    if n = 0 then 1 else
    let cn = ref 0 in
    for p = 0 to n-1 do
      cn := !cn + (c p) * (c (n-p-1))
    done;
    !cn
```

~~let tc = Array~~

$$T(n) = \sum_{p=0}^{n-1} \left(T(p) + T(n-p-1)\right)$$

$$= 2 \sum_{p=0}^{n-1} T(p) + cte$$

$$T(n) - T(n-1) = 2\,T(n-1) + cte$$

$$T(n) = 3\,T(n-1) + cte$$

$\Rightarrow \Theta(3^n)$ $\qquad S(n) \in \Theta(n)$

```
let c_dyn = fun n ->
    let c = Array.init (n+1) 0 in
    c.(0) <- 1;
    for i = 1 to n do
        for p = 0 to i-1 do
           c.(i) <- c.(i) + c.(p) * c.(n-p-1)
        done;
    done;
    c.(n)
```

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

$\rightarrow T(n) \in \Theta(n^2)$

$S(n) \in \Theta(n)$

3

Complexité des algorithmes

TP1 :

1. Implémentation de l'algorithme itératif

Fonction

python Copier le code

```python
def maxsubsum1(l):
    best = 0
    start, end = None, None

    for i in range(len(l)):
        current_sum = 0
        for j in range(i, len(l)):
            current_sum += l[j]
            if current_sum > best:
                best = current_sum
                start, end = i, j

    return best, start, end

# Test sur la liste donnée
l = [1, -2, 1, 2, -1, 3, -2, 1]
print(maxsubsum1(l))  # Output attendu : (5, 2, 5)
```

Explication

- **Initialisation :** La meilleure somme est `0` et les indices de la sous-séquence sont indéfinis au début.
- **Boucles imbriquées :**
  - La boucle externe parcourt tous les indices de début `i`.
  - La boucle interne calcule les sommes des sous-séquences à partir de `i` jusqu'à chaque `j`.
- **Mise à jour :** Si la somme courante dépasse la meilleure somme, on met à jour `best`, `start`, et `end`.

Complexité

- **Temps :** $O(n^2)$ en pire cas, car il y a deux boucles imbriquées qui parcourent la liste.
- **Espace :** $O(1)$, car aucune structure supplémentaire n'est utilisée.

2. Implémentation de l'algorithme "Diviser pour régner"

Fonction

python Copier le code

```python
def maxsubsum2(l):
    def maxsubsum2_rec(i, j):
        if i == j:  # Cas de base : un seul élément
            return l[i], i, i

        mid = (i + j) // 2

        # Partie gauche
        left_sum, left_start, left_end = maxsubsum2_rec(i, mid)

        # Partie droite
        right_sum, right_start, right_end = maxsubsum2_rec(mid + 1, j)

        # Partie traversant le milieu
        max_left, temp_sum = float('-inf'), 0
        cross_start = mid
        for k in range(mid, i - 1, -1):
            temp_sum += l[k]
            if temp_sum > max_left:
                max_left = temp_sum
                cross_start = k

        max_right, temp_sum = float('-inf'), 0
        cross_end = mid + 1
        for k in range(mid + 1, j + 1):
            temp_sum += l[k]
            if temp_sum > max_right:
                max_right = temp_sum
                cross_end = k

        cross_sum = max_left + max_right

        # Comparaison des trois cas
        if left_sum >= right_sum and left_sum >= cross_sum:
            return left_sum, left_start, left_end
        elif right_sum >= left_sum and right_sum >= cross_sum:
            return right_sum, right_start, right_end
        else:
            return cross_sum, cross_start, cross_end

    return maxsubsum2_rec(0, len(l) - 1)
```

Complexité des algorithmes

1. **Décomposition :**
   - Découper la liste en deux parties : gauche et droite.
   - Calculer récursivement la meilleure somme pour chaque partie.
2. **Somme traversant le milieu :**
   - Trouver la meilleure somme de la gauche en partant du milieu vers la gauche.
   - Trouver la meilleure somme de la droite en partant du milieu vers la droite.
   - Combiner les deux pour obtenir la meilleure somme traversant le milieu.
3. **Comparaison :**
   - Comparer les trois cas pour déterminer la meilleure somme globale.
4. **Arrêt de la récursion :**
   - Lorsque la tranche est réduite à un seul élément.

Complexité

- **Temps :** $O(n \log n)$, car on divise la liste à chaque étape ($O(\log n)$) et on parcourt chaque élément au moins une fois ($O(n)$).
- **Espace :** $O(\log n)$, dû à l'appel récursif.

TP3 :

1. Fonction `insert(l, i, cmp)`

Fonction

python Copier le code

```python
import operator

def insert(l, i, cmp):
    while i > 0:
        parent = (i - 1) // 2  # Calcul de l'index du parent
        if cmp(l[parent], l[i]):  # Si l'élément parent est déjà dans la bonne position
            break
        # Échange entre l'élément courant et son parent
        l[parent], l[i] = l[i], l[parent]
        i = parent  # Remonter au niveau du parent
```

Explication

- On part de l'élément à insérer et on remonte dans l'arbre jusqu'à ce que la propriété de tas soit respectée.
- La comparaison est personnalisée grâce à `cmp`, ce qui permet de choisir l'ordre (croissant ou décroissant).

---

2. Fonction `heapify(l, cmp)`

Fonction

python Copier le code

```python
def heapify(l, cmp):
    for i in range(1, len(l)):  # On insère chaque élément à partir du second
        insert(l, i, cmp)
```

Explication

- On parcourt les éléments un par un en appelant `insert` pour les insérer dans le tas.

3. Fonction `delete(l, lim, cmp)`

Fonction

python Copier le code

```python
def delete(l, lim, cmp):
    # Échanger la racine avec le dernier élément du tas
    l[0], l[lim] = l[lim], l[0]
    lim -= 1  # Réduire la taille du tas

    # Faire descendre l'élément échangé
    i = 0
    while True:
        left = 2 * i + 1
        right = 2 * i + 2
        largest = i

        if left <= lim and not cmp(l[largest], l[left]):
            largest = left
        if right <= lim and not cmp(l[largest], l[right]):
            largest = right

        if largest == i:  # L'élément est à sa place
            break

        # Échange avec l'enfant le plus grand
        l[i], l[largest] = l[largest], l[i]
        i = largest  # Descendre à l'enfant
```

Explication

- Cette fonction supprime la racine (le plus grand ou le plus petit élément selon `cmp`) et rétablit la propriété de tas.

4. Fonction `heapsort(l, cmp=operator.lt)`

Fonction

python Copier le code

```python
def heapsort(l, cmp=operator.lt):
    n = len(l)
    heapify(l, cmp)  # Construire le tas
    for i in range(n - 1, 0, -1):  # Réduire la taille du tas
        delete(l, i, cmp)  # Supprimer la racine et la placer à la fin
```

Explication

- On transforme la liste en tas.
- On retire les éléments un à un, en plaçant chaque racine à la fin.

6. Complexité

Analyse

1. **Construction du tas (`heapify`) :**
   - Chaque insertion coûte $O(\log n)$, et il y a n insertions.
   - **Complexité :** $O(n \log n)$.
2. **Phase de suppression (`delete`) :**
   - Chaque suppression coûte $O(\log n)$, et il y a n suppressions.
   - **Complexité :** $O(n \log n)$.
3. **Complexité totale :**
   - $O(n \log n)$ pour le tri complet.

Stabilité

- **Non stable :** Heapsort n'est pas stable car des éléments égaux peuvent être échangés durant les réorganisations.

Complexité pour un tableau déjà trié

- La complexité reste $O(n \log n)$, car on construit et déconstruit le tas indépendamment de l'ordre initial.