2ALGO: Mini-Projet

II - Stratégie gloutonne

```python
def strategie_gloutonne(T, C, A, B):
    # Définition de la fonction avec les paramètres :
    # T est une liste de valeurs numériques,
    # C est une liste de catégories associées à chaque valeur dans T,
    # A et B sont des coefficients utilisés dans les calculs de valeurs.

    n = len(T)  # Calcul de la longueur de la liste T, qui détermine le nombre d'itérations de la boucle.
    if n == 0:  # Vérifie si la liste T est vide.
        return 0, []  # Retourne 0 et une liste vide si T est vide.

    somme_max = 0  # Initialisation de la variable qui va accumuler la somme maximale.
    chemin = []  # Initialisation de la liste qui va enregistrer les indices des éléments ajoutés à la somme_max.
    dernier_index = -1  # Variable pour suivre l'indice du dernier élément traité; -1 indique qu'aucun élément n'a été traité.
    dernier_symbole = None  # Variable pour suivre le symbole de l'élément précédemment traité.

    for i in range(n):  # Boucle pour traiter chaque élément de T et C.
        if dernier_index == -1 or C[i] != dernier_symbole:  # Condition pour utiliser le coefficient B.
            valeur_actuelle = B * T[i]  # Calcul de la valeur actuelle en utilisant le coefficient B.
        else:  # Sinon, utilise le coefficient A si l'élément courant a le même symbole que le précédent.
            valeur_actuelle = A * T[i]  # Calcul de la valeur actuelle en utilisant le coefficient A.

        if valeur_actuelle > 0:  # Vérifie si la valeur actuelle est positive.
            somme_max += valeur_actuelle  # Ajoute la valeur actuelle à la somme maximale si elle est positive.
            chemin.append(i)  # Ajoute l'indice de l'élément courant au chemin.
            dernier_index = i  # Met à jour le dernier indice traité.
            dernier_symbole = C[i]  # Met à jour le dernier symbole traité.

    return somme_max, chemin  # Retourne la somme maximale calculée et le chemin des indices ajoutés.


#EXEMPLE 1
T = [9, 7, 8, 7, 10, 7]
C = [2, 1, 1, 4, 4, 2]
A = -2
B = 5

#EXEMPLE 2
#T = [3, 9, 2, 7, 3, 1]
#C = [2, 2, 5, 4, 2, 1]
#A = 2
#B = -5

print(strategie_gloutonne(T, C, A, B))
```

La stratégie gloutonne n'est généralement pas optimale pour des problèmes où les décisions prises influencent les résultats futurs de manière significative, car elle ne prend pas en compte l'ensemble des conséquences futures de ces décisions. Dans le cas présent, choisir de maximiser la récompense à chaque étape sans considérer l'ensemble du parcours peut mener à des situations où une décision précoce non idéale bloque des options plus lucratives plus tard.

Contre-exemple :

Considérons le cas où T=[10,50,20], C=[1,2,2], A=10, et B=1.

Une stratégie gloutonne pourrait initialement choisir de visiter le premier emplacement pour une récompense de $B \times 10 = 10$, puis passer au deuxième pour $B \times 50 = 50$, ignorants une meilleure stratégie qui serait de sauter le premier emplacement pour ensuite collecter $50 + A \times 20 = 250$ en visitant seulement les deux derniers emplacements.

III - Récurrence et récursivité naïve

3.1 - Une formule de récurrence

Pour définir une formule de récurrence pour ce problème, nous pouvons utiliser la définition suivante :

Soit $V[i]$ la somme maximale que l'on peut collecter entre les emplacements d'indices ii et $n-1$ (inclus). La formulation de la récurrence pour calculer $V[i]$ est la suivante :

Formule de Récurrence : $V[i] = \max(0, B \times T[i] + V[i+1], A \times T[i] + V[k])$

où :

- 0 représente le choix de ne pas visiter l'emplacement i.
- $B \times T[i] + V[i+1]$ représente la somme si on visite l'emplacement i avec un symbole différent de celui de $i+1$ ou comme premier emplacement visité.
- $A \times T[i] + V[k]$ représente la somme si on visite l'emplacement i et que le dernier emplacement visité avant i avec le même symbole était k, où k est le plus grand indice > i avec $C[k] = C[i]$ et k est le premier tel indice après i.

Cette formule suppose que nous parcourons de droite à gauche (rétroactivement).

Justification

- **0** est inclus pour modéliser le choix de ne pas collecter la somme à l'emplacement i.
- Le deuxième terme, $B \times T[i] + V[i+1]$, est calculé en prenant l'hypothèse que le prochain emplacement $i+1$ est le premier visité après i ou avec un symbole différent, utilisant le coefficient B.
- Le troisième terme, $A \times T[i] + V[k]$, nécessite de trouver le prochain k avec le même symbole $C[i]$, et de calculer la somme en supposant que la visite se poursuit avec le même symbole, utilisant le coefficient A.

3.2 - Un algorithme récursif naïf

```python
def somme_max_recursive(T, C, A, B, i=0, dernier_symbole=None):
    # Définition de la fonction récursive pour calculer la somme maximale.
    # La fonction prend en paramètres :
    # T : une liste de valeurs numériques,
    # C : une liste de symboles ou de catégories associés à chaque valeur dans T,
    # A et B : les coefficients utilisés pour calculer les sommes maximales,
    # i : l'indice actuel à partir duquel on commence à calculer la somme maximale (défaut à 0),
    # dernier_symbole : le symbole du dernier élément visité (défaut à None).

    if i >= len(T):
        return 0
    # Cas de base : Si l'indice actuel dépasse la longueur de la liste T, on retourne 0.

    # Ne pas visiter l'emplacement i
    max_collecte = somme_max_recursive(T, C, A, B, i + 1, dernier_symbole)
    # Appel récursif pour calculer la somme maximale sans visiter l'emplacement i.

    # Visiter l'emplacement i
    if dernier_symbole is None or C[i] != dernier_symbole:
        collecte = B * T[i] + somme_max_recursive(T, C, A, B, i + 1, C[i])
    else:
        collecte = A * T[i] + somme_max_recursive(T, C, A, B, i + 1, C[i])
    # Appel récursif pour calculer la somme maximale en visitant l'emplacement i.
    # Le coefficient B est utilisé si c'est le premier élément visité ou si le symbole est différent du dernier symbole visité.
    # Sinon, le coefficient A est utilisé.

    return max(max_collecte, collecte)
    # Retourne la somme maximale entre ne pas visiter l'emplacement i et visiter l'emplacement i.

# Test de la fonction
print(somme_max_recursive(T, C, A, B))
# Appel de la fonction pour tester son fonctionnement avec les valeurs données pour T, C, A et B.
```

Le temps nécessaire pour résoudre le problème augmente de manière exponentielle avec la taille des données d'entrée.

En d'autres termes, si vous doublez la taille de votre liste T, le temps d'exécution de l'algorithme peut potentiellement quadrupler, voire plus, en raison du grand nombre de sous-problèmes distincts à résoudre à chaque étape de la récursion.

Ainsi, bien que cet algorithme puisse fonctionner rapidement pour de petites tailles de données, il peut devenir extrêmement lent voire impraticable pour des données d'entrée de taille significative, en raison de sa complexité exponentielle.

IV - Programmation dynamique

4.1 - Approche Top Down

```python
def somme_max_top_down(T, C, A, B):
    # Définition de la fonction qui calcule la somme maximale en adoptant une approche dynamique top-down.
    # Prend en paramètres :
    # T : une liste de valeurs numériques,
    # C : une liste de symboles ou de catégories associés à chaque valeur dans T,
    # A et B : les coefficients utilisés pour calculer les sommes maximales.

    n = len(T)  # Calcul de la longueur de la liste T.
    memo = {}  # Initialisation du dictionnaire pour la mémoïsation des sous-problèmes.

    def calculate(i, dernier_symbole):
        # Fonction récursive interne pour calculer la somme maximale à partir de l'indice i et avec le dernier symbole donné.

        if i >= n:
            return 0  # Cas de base : retourne 0 si l'indice dépasse la longueur de la liste T.

        if (i, dernier_symbole) in memo:
            return memo[(i, dernier_symbole)]  # Retourne la valeur mémorisée si elle existe.

        # Calcul de la somme maximale en explorant les deux options : visiter ou ne pas visiter l'emplacement i.
        max_collecte = calculate(i + 1, dernier_symbole)

        if dernier_symbole is None or C[i] != dernier_symbole:
            collecte = B * T[i] + calculate(i + 1, C[i])
        else:
            collecte = A * T[i] + calculate(i + 1, C[i])

        # Mémorisation de la somme maximale obtenue pour ce sous-problème.
        memo[(i, dernier_symbole)] = max(max_collecte, collecte)
        return memo[(i, dernier_symbole)]  # Retourne la somme maximale calculée.

    # Appel initial à la fonction récursive avec l'indice de départ 0 et aucun dernier symbole.
    return calculate(i: 0, dernier_symbole: None)

# Test de la fonction
print(somme_max_top_down(T, C, A, B))
```

La complexité de cet algorithme top-down est exponentielle. Cela signifie que le temps nécessaire pour résoudre le problème augmente de manière exponentielle avec la taille des données d'entrée.

En d'autres termes, si vous doublez la taille de votre liste T, le temps d'exécution de l'algorithme peut potentiellement doubler, quadrupler, ou augmenter encore plus, en fonction de la façon dont les sous-problèmes se répètent. Cependant, grâce à l'utilisation de la mémoïsation, cet algorithme évite de recalculer plusieurs fois les mêmes sous-problèmes, ce qui peut réduire considérablement le temps d'exécution par rapport à l'approche récursive naïve.

Bien que cet algorithme puisse fonctionner efficacement pour des petites tailles de données, il peut devenir lent voire impraticable pour des données d'entrée de taille significative en raison de sa complexité exponentielle.

4.2- Approche Bottom Up

```python
def somme_max_bottom_up(T, C, A, B):
    # Cette fonction utilise une approche dynamique "Bottom Up" pour calculer la somme maximale collectée lors du parcours,
    # en tenant compte des coefficients A et B, ainsi que des catégories associées à chaque valeur.

    n = len(T)  # Détermine la longueur de la liste T.
    if n == 0:  # Vérifie si la liste T est vide.
        return 0, []  # Si la liste T est vide, retourne 0 comme somme maximale et une liste vide pour les indices visités.

    # Initialisation des tableaux pour stocker les résultats intermédiaires et les indices des emplacements visités
    dp = [0] * (n + 1)  # Tableau pour stocker les résultats intermédiaires du calcul de la somme maximale.
    trace = [None] * n  # Tableau pour enregistrer les indices des emplacements visités.

    for i in range(n - 1, -1, -1):
        # Parcours de la liste T en partant de la fin pour calculer la somme maximale pour chaque emplacement.

        # Calcul de la somme maximale pour l'emplacement i en utilisant la formule dynamique "Bottom Up"
        dp[i] = max(dp[i + 1], (B if i == n - 1 or C[i] != C[i + 1] else A) * T[i] + dp[i + 1])

        # Mise à jour de l'indice visité dans le tableau trace
        if i == n - 1 or C[i] != C[i + 1]:  # Vérifie si l'élément actuel est le dernier ou s'il a un symbole différent du suivant.
            trace[i] = i
        else:
            trace[i] = trace[i + 1]

    # Reconstruction du chemin
    vrai_chemin = []  # Liste pour enregistrer les indices des emplacements visités dans l'ordre.
    i = 0
    while i < n and trace[i] is not None:
        vrai_chemin.append(i)  # Ajoute l'indice de l'emplacement visité à vrai_chemin.
        i += 1 if trace[i] == i else trace[i] - i  # Déplace l'indice à l'emplacement suivant du chemin.

    return dp[0], vrai_chemin  # Retourne la somme maximale et la liste des indices des emplacements visités.

# Test de la fonction
print(somme_max_bottom_up(T, C, A, B))
```

Cet algorithme a une complexité linéaire, ce qui signifie que le temps et l'espace nécessaires pour l'exécuter augmentent linéairement avec la taille de l'entrée. Cela signifie que si la taille de la liste T double, le temps nécessaire pour exécuter l'algorithme doublera également. De même, l'espace mémoire utilisé augmentera proportionnellement à la taille de l'entrée.é augmentera proportionnellement à la taille de l'entrée.