Programmation dynamique

MP – PC – PSI* - Lycée Leconte de Lisle

Plan

Récursif et naïf

Une solution naïve pour calculer le cœfficient binomial $\binom{n}{k}$ consiste à utiliser une fonction récursive mettant en application la formule du triangle de Pascal

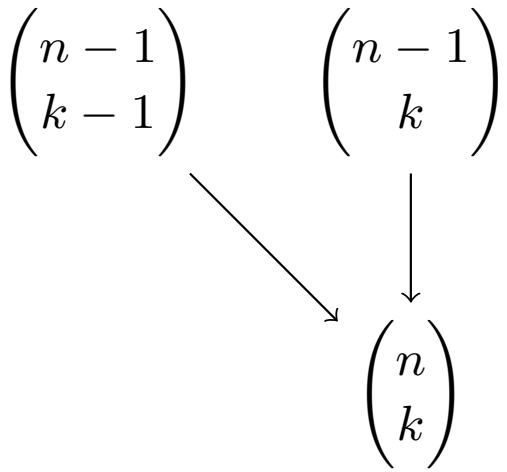
$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Récursif et naïf

Une solution naïve pour calculer le cœfficient binomial $\binom{n}{k}$ consiste à utiliser une fonction récursive mettant en application la formule du triangle de Pascal

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Schéma de dépendance :



Récursif et naïf

Python

```python
def binom(k, n):
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    return binom(k - 1, n - 1) + binom(k, n - 1)
```

Récursif et naïf

Python

```python
def binom(k, n):
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    return binom(k - 1, n - 1) + binom(k, n - 1)
```

Problème : nombre de cœfficients binomiaux sont calculés plusieurs fois.

Récursif et naïf

Python

```python
def binom(k, n):
    print(f'Calcul de {k} parmi {n}')
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    return binom(k - 1, n - 1) + binom(k, n - 1)
```

Récursif et naïf

Python

```python
def binom(k, n):
    print(f'Calcul de {k} parmi {n}')
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    return binom(k - 1, n - 1) + binom(k, n - 1)
```

Problème : nombre de cœfficients binomiaux sont calculés plusieurs fois.

Récursif et naïf

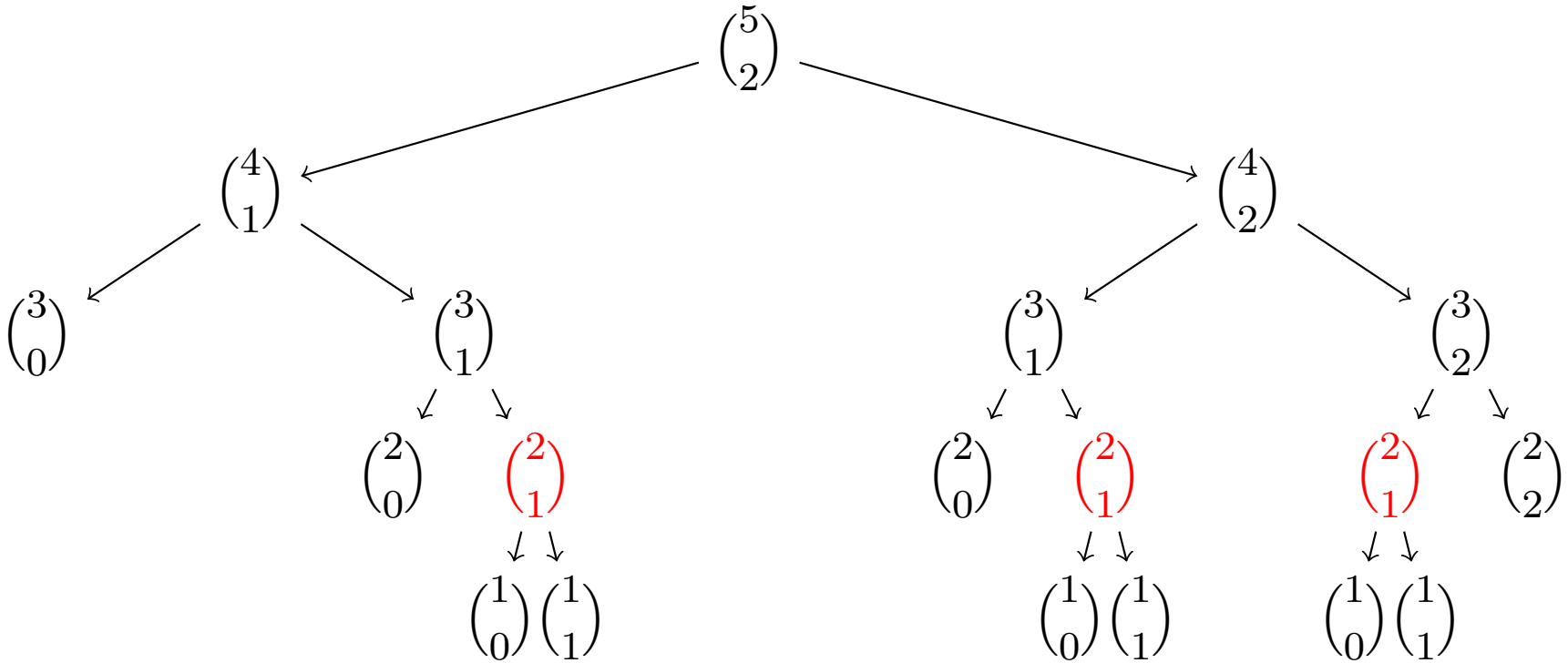
Par exemple, pour calculer $\binom{5}{2}$:

Python

```
>>> binom(2, 5)
Calcul de 2 parmi 5
Calcul de 1 parmi 4
Calcul de 0 parmi 3
Calcul de 1 parmi 3
Calcul de 0 parmi 2
Calcul de 1 parmi 2
Calcul de 0 parmi 1
Calcul de 1 parmi 1
Calcul de 2 parmi 4
Calcul de 1 parmi 3
Calcul de 0 parmi 2
Calcul de 1 parmi 2
Calcul de 0 parmi 1
Calcul de 1 parmi 1
Calcul de 2 parmi 3
Calcul de 1 parmi 2
Calcul de 0 parmi 1
Calcul de 1 parmi 1
Calcul de 2 parmi 2
10
```

Récursif et naïf

Ce qui se résume bien sur l'arbre suivant :



Récursif et naïf

Pour le vérifier formellement, on s'intéresse à la complexité temporelle (par exemple en nombre d'additions), qui vérifie alors

Récursif et naïf

Pour le vérifier formellement, on s'intéresse à la complexité temporelle (par exemple en nombre d'additions), qui vérifie alors

$$\begin{cases} T(k,n) = 0 & \text{si } k \leqslant 0 \text{ ou } k \geqslant n \\ T(k,n) = T(k-1,n-1) + T(k,n-1) + 1 & \text{sinon.} \end{cases}$$

Récursif et naïf

Pour le vérifier formellement, on s'intéresse à la complexité temporelle (par exemple en nombre d'additions), qui vérifie alors

$$\begin{cases} T(k,n) = 0 & \text{si } k \leqslant 0 \text{ ou } k \geqslant n \\ T(k,n) = T(k-1,n-1) + T(k,n-1) + 1 & \text{sinon.} \end{cases}$$

C'est une récurrence de la même forme que celle de $\binom{n}{k}$, on cherche a et b tels que $T(k,n) = a\binom{n}{k} + b$ et on trouve que $T(k,n) = \binom{n}{k} - 1$ pour $0 \leqslant k \leqslant n$.

Récursif et naïf

Pour le vérifier formellement, on s'intéresse à la complexité temporelle (par exemple en nombre d'additions), qui vérifie alors

$$\begin{cases} T(k,n) = 0 & \text{si } k \leqslant 0 \text{ ou } k \geqslant n \\ T(k,n) = T(k-1,n-1) + T(k,n-1) + 1 & \text{sinon.} \end{cases}$$

C'est une récurrence de la même forme que celle de $\binom{n}{k}$, on cherche a et b tels que $T(k,n) = a\binom{n}{k} + b$ et on trouve que $T(k,n) = \binom{n}{k} - 1$ pour $0 \leqslant k \leqslant n$.
Pour un n donné, la valeur maximale est $T\left(\left\lceil \frac{n}{2} \right\rceil, n\right)$ (au centre dans le triangle) et la formule de Stirling nous donne du

Récursif et naïf

Pour le vérifier formellement, on s'intéresse à la complexité temporelle (par exemple en nombre d'additions), qui vérifie alors

$$\begin{cases} T(k,n) = 0 & \text{si } k \leqslant 0 \text{ ou } k \geqslant n \\ T(k,n) = T(k-1,n-1) + T(k,n-1) + 1 & \text{sinon.} \end{cases}$$

C'est une récurrence de la même forme que celle de $\binom{n}{k}$, on cherche a et b tels que $T(k,n) = a\binom{n}{k} + b$ et on trouve que $T(k,n) = \binom{n}{k} - 1$ pour $0 \leqslant k \leqslant n$.

Pour un n donné, la valeur maximale est $T\left(\left\lceil \frac{n}{2} \right\rceil, n\right)$ (au centre dans le triangle) et la formule de Stirling nous donne du $\mathrm{O}\left(\frac{2^n}{\sqrt{n}}\right)$. Violent !

Une solution bien meilleure

Une meilleure idée serait d'utiliser les formules $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$ et $\binom{n}{k} = \binom{n}{n-k}$:

Une solution bien meilleure

Une meilleure idée serait d'utiliser les formules $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$ et $\binom{n}{k} = \binom{n}{n-k}$:

Python

```python
def binom(k, n):
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    if n - k < k:
        return binom(n - k, n)
    return (n * binom(k - 1, n - 1)) // k
    # Attention : surtout pas de n / k !
```

avec une complexité

Une solution bien meilleure

Une meilleure idée serait d'utiliser les formules $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$ et $\binom{n}{k} = \binom{n}{n-k}$:

Python

```python
def binom(k, n):
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    if n - k < k:
        return binom(n - k, n)
    return (n * binom(k - 1, n - 1)) // k
    # Attention : surtout pas de n / k !
```

avec une complexité linéaire cette fois,

Une solution bien meilleure

Une meilleure idée serait d'utiliser les formules $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$ et $\binom{n}{k} = \binom{n}{n-k}$:

Python

```python
def binom(k, n):
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    if n - k < k:
        return binom(n - k, n)
    return (n * binom(k - 1, n - 1)) // k
    # Attention : surtout pas de n / k !
```

avec une complexité linéaire cette fois, mais sans la formule du triangle de Pascal.

Garder en mémoire les résultats, de bas en haut (itératif)

On peut aussi mémoriser les valeurs calculées, en calculant les lignes du triangle de Pascal les unes après les autres.

En voici une première version, itérative, en stockant le triangle de Pascal dans un tableau à deux dimensions (on parle d'approche de bas à haut ou *bottom-up*).

Garder en mémoire les résultats, de bas en haut (itératif)

On peut aussi mémoriser les valeurs calculées, en calculant les lignes du triangle de Pascal les unes après les autres.

En voici une première version, itérative, en stockant le triangle de Pascal dans un tableau à deux dimensions (on parle d'approche de bas à haut ou *bottom-up*).

Python

```python
def binom(k, n):
    if k < 0 or k > n:
        return 0
    bin = [[0 for _ in range(k+1)] for _ in range(n+1)]
    bin[0][0] = 1
    for i in range(1, n + 1): # remplissage de la ligne i
        bin[i][0] = 1
        for j in range(1, k + 1):
            bin[i][j] = bin[i - 1][j - 1] + bin[i - 1][j]
    return bin[n][k]
```

Garder en mémoire les résultats, de bas en haut (itératif)

On peut aussi mémoriser les valeurs calculées, en calculant les lignes du triangle de Pascal les unes après les autres.

En voici une première version, itérative, en stockant le triangle de Pascal dans un tableau à deux dimensions (on parle d'approche de bas à haut ou *bottom-up*).

Python

```python
def binom(k, n):
    if k < 0 or k > n:
        return 0
    bin = [[0 for _ in range(k+1)] for _ in range(n+1)]
    bin[0][0] = 1
    for i in range(1, n + 1): # remplissage de la ligne i
        bin[i][0] = 1
        for j in range(1, k + 1):
            bin[i][j] = bin[i - 1][j - 1] + bin[i - 1][j]
    return bin[n][k]
```

Cette fois, nous avons une complexité temporelle en

Garder en mémoire les résultats, de bas en haut (itératif)

On peut aussi mémoriser les valeurs calculées, en calculant les lignes du triangle de Pascal les unes après les autres.

En voici une première version, itérative, en stockant le triangle de Pascal dans un tableau à deux dimensions (on parle d'approche de bas à haut ou *bottom-up*).

Python

```python
def binom(k, n):
    if k < 0 or k > n:
        return 0
    bin = [[0 for _ in range(k+1)] for _ in range(n+1)]
    bin[0][0] = 1
    for i in range(1, n + 1): # remplissage de la ligne i
        bin[i][0] = 1
        for j in range(1, k + 1):
            bin[i][j] = bin[i - 1][j - 1] + bin[i - 1][j]
    return bin[n][k]
```

Cette fois, nous avons une complexité temporelle en $\mathrm{O}(nk)$ et une complexité spatiale en

Garder en mémoire les résultats, de bas en haut (itératif)

On peut aussi mémoriser les valeurs calculées, en calculant les lignes du triangle de Pascal les unes après les autres.

En voici une première version, itérative, en stockant le triangle de Pascal dans un tableau à deux dimensions (on parle d'approche de bas à haut ou *bottom-up*).

Python

```python
def binom(k, n):
    if k < 0 or k > n:
        return 0
    bin = [[0 for _ in range(k+1)] for _ in range(n+1)]
    bin[0][0] = 1
    for i in range(1, n + 1): # remplissage de la ligne i
        bin[i][0] = 1
        for j in range(1, k + 1):
            bin[i][j] = bin[i - 1][j - 1] + bin[i - 1][j]
    return bin[n][k]
```

Cette fois, nous avons une complexité temporelle en $\mathrm{O}(nk)$ et une complexité spatiale en $\mathrm{O}(nk)$ également.

Garder en mémoire les résultats, de bas en haut (itératif)

On remarque que l'on peut faire mieux spatialement car chaque ligne est déduite uniquement de la précédente. D'où la version suivante avec seulement un tableau linéaire.

Garder en mémoire les résultats, de bas en haut (itératif)

On remarque que l'on peut faire mieux spatialement car chaque ligne est déduite uniquement de la précédente. D'où la version suivante avec seulement un tableau linéaire.

Python

```python
def binom_bul(k, n):
    if k < 0 or k > n:
        return 0
    bin = [0 for _ in range(k + 1)]
    bin[0] = 1
    for _ in range(n): # remplissage de la ligne i
        for j in range(k, 0, -1):
            # modification de droite à gauche
            bin[j] = bin[j - 1] + bin[j]
    return bin[k]
```

Garder en mémoire les résultats, de bas en haut (itératif)

On remarque que l'on peut faire mieux spatialement car chaque ligne est déduite uniquement de la précédente. D'où la version suivante avec seulement un tableau linéaire.

Python

```python
def binom_bul(k, n):
    if k < 0 or k > n:
        return 0
    bin = [0 for _ in range(k + 1)]
    bin[0] = 1
    for _ in range(n): # remplissage de la ligne i
        for j in range(k, 0, -1):
            # modification de droite à gauche
            bin[j] = bin[j - 1] + bin[j]
    return bin[k]
```

Cette fois, nous avons une complexité temporelle en

Garder en mémoire les résultats, de bas en haut (itératif)

On remarque que l'on peut faire mieux spatialement car chaque ligne est déduite uniquement de la précédente. D'où la version suivante avec seulement un tableau linéaire.

Python

```python
def binom_bul(k, n):
    if k < 0 or k > n:
        return 0
    bin = [0 for _ in range(k + 1)]
    bin[0] = 1
    for _ in range(n): # remplissage de la ligne i
        for j in range(k, 0, -1):
            # modification de droite à gauche
            bin[j] = bin[j - 1] + bin[j]
    return bin[k]
```

Cette fois, nous avons une complexité temporelle en $\mathrm{O}(nk)$ et une complexité spatiale en

Garder en mémoire les résultats, de bas en haut (itératif)

On remarque que l'on peut faire mieux spatialement car chaque ligne est déduite uniquement de la précédente. D'où la version suivante avec seulement un tableau linéaire.

Python

```python
def binom_bul(k, n):
    if k < 0 or k > n:
        return 0
    bin = [0 for _ in range(k + 1)]
    bin[0] = 1
    for _ in range(n): # remplissage de la ligne i
        for j in range(k, 0, -1):
            # modification de droite à gauche
            bin[j] = bin[j - 1] + bin[j]
    return bin[k]
```

Cette fois, nous avons une complexité temporelle en $\mathrm{O}(nk)$ et une complexité spatiale en $\mathrm{O}(k)$ seulement.

La magie de la mémoïsation, de haut en bas (récursif)

Mais on remarque que beaucoup de cœfficients binomiaux sont encore calculés inutilement.

Pour ne calculer que les cœfficients binomiaux utiles, nous pouvons changer d'approche, avec une approche dite de haut en bas (ou *top-bottom*), programmée récursivement en utilisant un dictionnaire comme variable globale. C'est la mémoïsation.

La magie de la mémoïsation, de haut en bas (récursif)

Mais on remarque que beaucoup de cœfficients binomiaux sont encore calculés inutilement.

Pour ne calculer que les cœfficients binomiaux utiles, nous pouvons changer d'approche, avec une approche dite de haut en bas (ou *top-bottom*), programmée récursivement en utilisant un dictionnaire comme variable globale. C'est la mémoïsation.

Python

```python
dico = {}

def binom(k, n):
    if k < 0 or k > n:
        return 0
    elif k == 0 or k == n:
        return  1
    elif (k, n) not in dico:
        dico[(k, n)] = binom(k - 1, n - 1)
                       + binom(k, n - 1)
    return dico[(k, n)]
```

La magie de la mémoïsation, de haut en bas (récursif)

ce qui donne

Python

```
>>> binom(2, 5)
10
>>> dico
{(1, 2): 2, (1, 3): 3, (1, 4): 4,
 (2, 3): 3, (2, 4): 6, (2, 5): 10}
```

La magie de la mémoïsation, de haut en bas (récursif)

ce qui donne

Python

```
>>> binom(2, 5)
10
>>> dico
{(1, 2): 2, (1, 3): 3, (1, 4): 4,
 (2, 3): 3, (2, 4): 6, (2, 5): 10}
```

Cette fois, seuls les cœfficients binomiaux nécessaires ont été calculés.

La magie de la mémoïsation, de haut en bas (récursif)

ce qui donne

Python

```
>>> binom(2, 5)
10
>>> dico
{(1, 2): 2, (1, 3): 3, (1, 4): 4,
 (2, 3): 3, (2, 4): 6, (2, 5): 10}
```

Cette fois, seuls les cœfficients binomiaux nécessaires ont été calculés. Le nombre d'addition devient $\mathrm{O}(n)$ et la complexité spatiale (taille du dictionnaire) également.

Déjà prévu dans Python

Il se trouve que la mémoïsation des fonctions récursives existe nativement dans Python.

On l'obtient de façon automatique, avec la fonction de départ, en utilisant

Python

```python
from functools import lru_cache

@lru_cache # lru_cache est appelé décorateur
def binom(k, n):
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    return binom(k - 1, n - 1) + binom(k, n - 1)
```

Déjà prévu dans Python

Il se trouve que la mémoïsation des fonctions récursives existe nativement dans Python.

On l'obtient de façon automatique, avec la fonction de départ, en utilisant

Python

```python
from functools import lru_cache

@lru_cache # lru_cache est appelé décorateur
def binom(k, n):
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    return binom(k - 1, n - 1) + binom(k, n - 1)
```

Il va sans dire que ce n'est pas ce qui sera attendu le jour du concours !

Plan

Quelques définitions

La programmation dynamique que l'on est en train de mettre en place consiste, pour résoudre un problème, à résoudre des sous-problèmes de taille inférieure qui ne sont pas indépendant.

Quelques définitions

La programmation dynamique que l'on est en train de mettre en place consiste, pour résoudre un problème, à résoudre des sous-problèmes de taille inférieure qui ne sont pas indépendant.

Il ne faut pas la confondre avec la méthode diviser pour régner (utilisée pour les tris rapide et fusion par exemple), pour laquelle les sous-problèmes sont indépendants les uns des autres.

Quelques définitions

La programmation dynamique que l'on est en train de mettre en place consiste, pour résoudre un problème, à résoudre des sous-problèmes de taille inférieure qui ne sont pas indépendant.

Il ne faut pas la confondre avec la méthode diviser pour régner (utilisée pour les tris rapide et fusion par exemple), pour laquelle les sous-problèmes sont indépendants les uns des autres.

La solution de la mémoïsation permet de résoudre ces dépendances.

Quelques définitions

La programmation dynamique que l'on est en train de mettre en place consiste, pour résoudre un problème, à résoudre des sous-problèmes de taille inférieure qui ne sont pas indépendant.

Il ne faut pas la confondre avec la méthode diviser pour régner (utilisée pour les tris rapide et fusion par exemple), pour laquelle les sous-problèmes sont indépendants les uns des autres.

La solution de la mémoïsation permet de résoudre ces dépendances.

La programmation dynamique est fréquemment employée pour résoudre des problèmes d'optimisation : elle s'applique lorsque la solution optimale peut être déduite des solutions optimales des sous-problèmes.

Quelques définitions

La programmation dynamique que l'on est en train de mettre en place consiste, pour résoudre un problème, à résoudre des sous-problèmes de taille inférieure qui ne sont pas indépendant.

Il ne faut pas la confondre avec la méthode diviser pour régner (utilisée pour les tris rapide et fusion par exemple), pour laquelle les sous-problèmes sont indépendants les uns des autres.

La solution de la mémoïsation permet de résoudre ces dépendances.

La programmation dynamique est fréquemment employée pour résoudre des problèmes d'optimisation : elle s'applique lorsque la solution optimale peut être déduite des solutions optimales des sous-problèmes.

Exemples

Recherche de plus court chemin, rendu de monnaie, etc.

Quelques définitions

Définition : Sous-structure

Une sous-structure est une restriction de notre problème à un ensemble plus petit.

Quelques définitions

Définition : Sous-structure

Une sous-structure est une restriction de notre problème à un ensemble plus petit.

Définition : Programmation dynamique

Les méthodes de programmation dynamique sont mises en œuvre lorsque le problème étudié possède une propriété de sous-structure optimale, et lorsque les sous-problèmes utilisés pour le résoudre se chevauchent.

Quelques définitions

Définition : Sous-structure

Une sous-structure est une restriction de notre problème à un ensemble plus petit.

Définition : Programmation dynamique

Les méthodes de programmation dynamique sont mises en œuvre lorsque le problème étudié possède une propriété de sous-structure optimale, et lorsque les sous-problèmes utilisés pour le résoudre se chevauchent.

Définition : équation de Bellman

Lorsque l'on arrive à décomposer notre problème en plusieurs sous-problèmes plus simples, la relation liant la solution optimale de notre problème à celles des sous-problème est appelée équation de Bellman.

Les méthodes de programmation dynamique garantissent d'obtenir la meilleure solution au problème étudié, mais dans un certain nombre de cas, la complexité temporelle reste trop importante pour pouvoir être utilisée dans la pratique.

Les méthodes de programmation dynamique garantissent d'obtenir la meilleure solution au problème étudié, mais dans un certain nombre de cas, la complexité temporelle reste trop importante pour pouvoir être utilisée dans la pratique.

Dans ce type de situation, on se résout à utiliser un autre paradigme de programmation, la programmation gloutonne.

Les méthodes de programmation dynamique garantissent d'obtenir la meilleure solution au problème étudié, mais dans un certain nombre de cas, la complexité temporelle reste trop importante pour pouvoir être utilisée dans la pratique.

Dans ce type de situation, on se résout à utiliser un autre paradigme de programmation, la programmation gloutonne.

Alors que la programmation dynamique se caractérise par la résolution par taille croissante de tous les problèmes locaux, la stratégie gloutonne consiste à choisir à partir du problème global un problème local et un seul en suivant une heuristique (c'est à dire une stratégie permettant de faire un choix rapide mais pas nécessairement optimal). En général, on ne peut pas garantir que la stratégie gloutonne détermine la solution optimale, mais lorsque l'heuristique est bien choisie on peut espérer obtenir une solution proche de celle-ci.

Plan

Rendu de monnaie : Le problème

Soit $S \in \mathbb{N}$ une somme à former à l'aide de pièces $P = (p_i)_{0 \leqslant i \leqslant n-1}$ (par exemple, pour les euros comptés en centimes, $(200, 100, 50, 20, 10, 5, 2, 1)$.)

Rendu de monnaie : Le problème

Soit $S \in \mathbb{N}$ une somme à former à l'aide de pièces $P = (p_i)_{0 \leqslant i \leqslant n-1}$ (par exemple, pour les euros comptés en centimes, $(200, 100, 50, 20, 10, 5, 2, 1)$.)

L'objectif est de former la somme S à l'aide des valeurs de P en utilisant un minimum de pièces.

Rendu de monnaie : Le problème

Soit $S \in \mathbb{N}$ une somme à former à l'aide de pièces $P = (p_i)_{0 \leqslant i \leqslant n-1}$ (par exemple, pour les euros comptés en centimes, $(200, 100, 50, 20, 10, 5, 2, 1)$.)

L'objectif est de former la somme S à l'aide des valeurs de P en utilisant un minimum de pièces.

Si on note m_i le nombre de pièces de valeur p_i, alors le problème consiste à minimiser le nombre total de pièces $\sum_{i=0}^{n-1} m_i$ tout en ayant $S = \sum_{i=0}^{n-1} m_i p_i$.

Méthode gloutonne

La méthode gloutonne consiste à rendre un nombre maximal de pièce par ordre décroissant de valeurs de pièces.

Méthode gloutonne

La méthode gloutonne consiste à rendre un nombre maximal de pièce par ordre décroissant de valeurs de pièces.

Python

```python
def rendu_monnaie_glouton(Somme, Pieces):
    # tri des pièces par ordre décroissant
    Pieces_triees = sorted(Pieces, reverse=True)
    rendu = {}
    somme_restante = Somme
    for piece in Pieces_triees:
        nb, somme_restante = divmod(somme_restante, piece)
        rendu[piece] = nb
     return rendu
```

Méthode gloutonne

La méthode gloutonne consiste à rendre un nombre maximal de pièce par ordre décroissant de valeurs de pièces.

Python

```python
def rendu_monnaie_glouton(Somme, Pieces):
    # tri des pièces par ordre décroissant
    Pieces_triees = sorted(Pieces, reverse=True)
    rendu = {}
    somme_restante = Somme
    for piece in Pieces_triees:
        nb, somme_restante = divmod(somme_restante, piece)
        rendu[piece] = nb
     return rendu
```

La stratégie gloutonne n'est pas optimale pour certains systèmes de pièces (dits non canoniques) : par exemple, pour rendre 6 avec un système $(1, 3, 4)$, l'algorithme glouton rend 3 pièces (une de 4 et deux de 1) alors que l'optimal est à deux pièce de 4.

Les sous-problèmes considérés ici sont des problèmes du même type mais n'utilisant que les pièces $P_k = (p_i)_{0 \leqslant i \leqslant k-1}$ pour atteindre une somme $s \leqslant S$.

Les sous-problèmes considérés ici sont des problèmes du même type mais n'utilisant que les pièces $P_k = (p_i)_{0 \leqslant i \leqslant k-1}$ pour atteindre une somme $s \leqslant S$.

Notons $c_k(s)$ le nombre minimal de pièces de valeurs parmi $P_k = (p_0, \ldots, p_{k-1})$ nécessaire pour former la somme s.

Les sous-problèmes considérés ici sont des problèmes du même type mais n'utilisant que les pièces $P_k = (p_i)_{0 \leqslant i \leqslant k-1}$ pour atteindre une somme $s \leqslant S$.

Notons $c_k(s)$ le nombre minimal de pièces de valeurs parmi $P_k = (p_0, \ldots, p_{k-1})$ nécessaire pour former la somme s.

On cherche donc à calculer $c_n(S)$.

Les sous-problèmes considérés ici sont des problèmes du même type mais n'utilisant que les pièces $P_k = (p_i)_{0 \leqslant i \leqslant k-1}$ pour atteindre une somme $s \leqslant S$.

Notons $c_k(s)$ le nombre minimal de pièces de valeurs parmi $P_k = (p_0, \ldots, p_{k-1})$ nécessaire pour former la somme s.

On cherche donc à calculer $c_n(S)$.

Comment déterminer $c_k(s)$? Il est possible de rendre m pièces de valeur p_{k-1} avec $mp_{k-1} \leqslant s$.

Les sous-problèmes considérés ici sont des problèmes du même type mais n'utilisant que les pièces $P_k = (p_i)_{0 \leqslant i \leqslant k-1}$ pour atteindre une somme $s \leqslant S$.

Notons $c_k(s)$ le nombre minimal de pièces de valeurs parmi $P_k = (p_0, \ldots, p_{k-1})$ nécessaire pour former la somme s.

On cherche donc à calculer $c_n(S)$.

Comment déterminer $c_k(s)$? Il est possible de rendre m pièces de valeur p_{k-1} avec $mp_{k-1} \leqslant s$.

Exemple : comment rendre 10 avec le système $(4, 3, 1)$?

Les sous-problèmes considérés ici sont des problèmes du même type mais n'utilisant que les pièces $P_k = (p_i)_{0 \leqslant i \leqslant k-1}$ pour atteindre une somme $s \leqslant S$.

Notons $c_k(s)$ le nombre minimal de pièces de valeurs parmi $P_k = (p_0, \ldots, p_{k-1})$ nécessaire pour former la somme s.

On cherche donc à calculer $c_n(S)$.

Comment déterminer $c_k(s)$? Il est possible de rendre m pièces de valeur p_{k-1} avec $mp_{k-1} \leqslant s$.

Exemple : comment rendre 10 avec le système $(4, 3, 1)$?

Il faut alors avoir un minimum de pièces parmi $(p_0, \ldots, p_{k-2})$ pour former la somme $s - mp_{k-1}$, et ajouter à ce nombre les m pièces de valeur p_{k-1}.

Les sous-problèmes considérés ici sont des problèmes du même type mais n'utilisant que les pièces $P_k = (p_i)_{0 \leqslant i \leqslant k-1}$ pour atteindre une somme $s \leqslant S$.

Notons $c_k(s)$ le nombre minimal de pièces de valeurs parmi $P_k = (p_0, \ldots, p_{k-1})$ nécessaire pour former la somme s.

On cherche donc à calculer $c_n(S)$.

Comment déterminer $c_k(s)$? Il est possible de rendre m pièces de valeur p_{k-1} avec $mp_{k-1} \leqslant s$.

Exemple : comment rendre 10 avec le système $(4, 3, 1)$?

Il faut alors avoir un minimum de pièces parmi $(p_0, \ldots, p_{k-2})$ pour former la somme $s - mp_{k-1}$, et ajouter à ce nombre les m pièces de valeur p_{k-1}.

D'où la formule de récurrence (équation de Bellmann) :

$$c_k(s) = \min_{m \in \mathbb{N} \mid mp_{k-1} \leqslant s} \left(c_{k-1}(s - mp_{k-1}) + m\right)$$

Ajoutons les conditions aux limites (conditions d'arrêt) :

- $c_k(0) = 0$ pour tout $k \geqslant 1$ (aucune pièce pour rendre 0) ;

Ajoutons les conditions aux limites (conditions d'arrêt) :

- $c_k(0) = 0$ pour tout $k \geqslant 1$ (aucune pièce pour rendre 0) ;
- $c_0(s) = +\infty$ pour tout $s > 0$ (pas de rendu possible si aucune pièce).

Ajoutons les conditions aux limites (conditions d'arrêt) :

- $c_k(0) = 0$ pour tout $k \geqslant 1$ (aucune pièce pour rendre 0) ;
- $c_0(s) = +\infty$ pour tout $s > 0$ (pas de rendu possible si aucune pièce).

Ajoutons les conditions aux limites (conditions d'arrêt) :

- $c_k(0) = 0$ pour tout $k \geqslant 1$ (aucune pièce pour rendre 0) ;
- $c_0(s) = +\infty$ pour tout $s > 0$ (pas de rendu possible si aucune pièce).

Nous pouvons utiliser `float('inf')` pour simuler $+\infty$.

Ajoutons les conditions aux limites (conditions d'arrêt) :

- $c_k(0) = 0$ pour tout $k \geqslant 1$ (aucune pièce pour rendre 0) ;
- $c_0(s) = +\infty$ pour tout $s > 0$ (pas de rendu possible si aucune pièce).

Nous pouvons utiliser `float('inf')` pour simuler $+\infty$.

Le plus difficile est fait.

Ajoutons les conditions aux limites (conditions d'arrêt) :

- $c_k(0) = 0$ pour tout $k \geqslant 1$ (aucune pièce pour rendre 0) ;
- $c_0(s) = +\infty$ pour tout $s > 0$ (pas de rendu possible si aucune pièce).

Nous pouvons utiliser `float('inf')` pour simuler $+\infty$.

Le plus difficile est fait.

L'implémentation va se faire par mémoïsation (récursivité, de haut en bas), bien adaptée ici.

Python

```python
def nb_pieces_rendues(Somme, Pieces)
    """nombre minimal de pièces nécessaires pour
rendre Somme à l'aide du système Pieces."""
    dico = {} # Dictionnaire pour la mémoïsation
    def c(k, s):
        """nombre minimal de pièces pour obtenir s
avec Pieces[:k]"""
        if s == 0:
            return 0
        if k == 0:
            return float('inf')
        if (k, s) not in dico:
            liste = [c(k - 1, s - m*Pieces[k - 1]) + m
                     for m in range(s//Pieces[k - 1] + 1)]
            dico[(k, s)] = min(liste)
        return dico[(k, s)]
    return c(len(Pieces), Somme)
```

Python

```python
def nb_pieces_rendues(Somme, Pieces)
    """nombre minimal de pièces nécessaires pour
rendre Somme à l'aide du système Pieces."""
    dico = {} # Dictionnaire pour la mémoïsation
    def c(k, s):
        """nombre minimal de pièces pour obtenir s
avec Pieces[:k]"""
        if s == 0:
            return 0
        if k == 0:
            return float('inf')
        if (k, s) not in dico:
            liste = [c(k - 1, s - m*Pieces[k - 1]) + m
                     for m in range(s//Pieces[k - 1] + 1)]
            dico[(k, s)] = min(liste)
        return dico[(k, s)]
    return c(len(Pieces), Somme)
```

Ce n'est pas pleinement satisfaisant dans le sens où nous avons le nombre minimal de pièces, mais nous n'avons pas la distribution desdites pièces. Nous allons y remédier.

Pour pouvoir déterminer un distribution de pièces réalisant le minimum, l'idée est de garder en mémoire les distributions réalisant les minima à chaque étape en plus des nombres de pièces.

Pour pouvoir déterminer un distribution de pièces réalisant le minimum, l'idée est de garder en mémoire les distributions réalisant les minima à chaque étape en plus des nombres de pièces.

Donnons-nous une fonction renvoyant un indice du minimum d'une liste.

Pour pouvoir déterminer un distribution de pièces réalisant le minimum, l'idée est de garder en mémoire les distributions réalisant les minima à chaque étape en plus des nombres de pièces.

Donnons-nous une fonction renvoyant un indice du minimum d'une liste.

Python

```python
def indice_min(liste):
    'renvoie un indice du minimum de liste.'
    n = len(liste)
    ind_min = 0
    for i in range(1, n):
        if liste[i] < liste[ind_min]:
            ind_min = i
    return ind_min
```

Pour pouvoir déterminer un distribution de pièces réalisant le minimum, l'idée est de garder en mémoire les distributions réalisant les minima à chaque étape en plus des nombres de pièces.

Donnons-nous une fonction renvoyant un indice du minimum d'une liste.

Python

```python
def indice_min(liste):
    'renvoie un indice du minimum de liste.'
    n = len(liste)
    ind_min = 0
    for i in range(1, n):
        if liste[i] < liste[ind_min]:
            ind_min = i
    return ind_min
```

Une fois que l'on aura rempli un dictionnaire avec des couples $(min, indice\_min)$, il faudra retrouver la solution optimale en partant de la dernière pièce et en remontant à la première (rétropropagation).

Python

```python
def nb_pieces_rendues(Somme, Pieces):
    """distribution de pièces réalisant le nombre
minimal de pièces nécessaires pour rendre Somme
à l'aide du système Pieces."""

    dico = {} # Dictionnaire pour la mémoïsation

    def c_et_m(k, s):
        """couple (c(k, s), m) où c(k, s) nombre minimal
de pièces pour obtenir s avec Pieces[:k] et m un nombre
de pièces Pieces[k - 1]  réalisant ce minimum."""
        if s == 0:
            return (0, 0)
        if k == 0:
            return (float('inf'), None) # Ne sert pas
        if (k, s) not in dico:
            liste = [c_et_m(k-1, s-m*Pieces[k-1])[0]+m
                     for m in range(s//Pieces[k-1]+1)]
            m = indice_min(liste)
            dico[(k, s)] = (liste[m], m)
        return dico[(k, s)]
```

Python

```python
# [suite de la fonction nb_pieces_rendues]
    n = len(Pieces)
    dico_solution = {} # initialisation de la solution

    # On récupère le nombre de dernière pièce.
    dico_solution[Pieces[-1]] = c_et_m(n, Somme)[1]

    # rétropropagation
    somme_restante = Somme
    for k in range(n - 1, 0, -1):
        # On enlève les dernières pièces.
        somme_restante -=
                  Pieces[k] * dico_solution[Pieces[k]]
        # Nombre de pièces précédentes.
        dico_solution[Pieces[k - 1]] =
                        c_et_m(k, somme_restante)[1]

    return dico_solution
```

Python

```python
# [suite de la fonction nb_pieces_rendues]
    n = len(Pieces)
    dico_solution = {} # initialisation de la solution

    # On récupère le nombre de dernière pièce.
    dico_solution[Pieces[-1]] = c_et_m(n, Somme)[1]

    # rétropropagation
    somme_restante = Somme
    for k in range(n - 1, 0, -1):
        # On enlève les dernières pièces.
        somme_restante -=
                  Pieces[k] * dico_solution[Pieces[k]]
        # Nombre de pièces précédentes.
        dico_solution[Pieces[k - 1]] =
                        c_et_m(k, somme_restante)[1]

    return dico_solution
```

Une alternative consiste à utiliser deux dictionnaires différents pour `c` et pour `m`.

Plan

Le problème

La distance d'édition, ou distance de Levenshtein, est une mesure de la similarité de deux chaînes de caractères : elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.

Le problème

La distance d'édition, ou distance de Levenshtein, est une mesure de la similarité de deux chaînes de caractères : elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.

Par exemple, on peut passer du mot `polynomial` au mot `polygonal` en suivant les étapes suivantes :

Le problème

La distance d'édition, ou distance de Levenshtein, est une mesure de la similarité de deux chaînes de caractères : elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.

Par exemple, on peut passer du mot `polynomial` au mot `polygonal` en suivant les étapes suivantes :

- suppression de la lettre `i` : `polynomial` $\longrightarrow$ `polynomal` ;

Le problème

La distance d'édition, ou distance de Levenshtein, est une mesure de la similarité de deux chaînes de caractères : elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.

Par exemple, on peut passer du mot `polynomial` au mot `polygonal` en suivant les étapes suivantes :

- suppression de la lettre `i` : `polynomial` $\longrightarrow$ `polynomal` ;
- remplacement du `n` par un `g` : `polynomal` $\longrightarrow$ `polygomal` ;

Le problème

La distance d'édition, ou distance de Levenshtein, est une mesure de la similarité de deux chaînes de caractères : elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.

Par exemple, on peut passer du mot `polynomial` au mot `polygonal` en suivant les étapes suivantes :

- suppression de la lettre `i` : `polynomial` $\longrightarrow$ `polynomal` ;
- remplacement du `n` par un `g` : `polynomal` $\longrightarrow$ `polygomal` ;
- remplacement du `m` par un `n` : `polygomal` $\longrightarrow$ `polygonal` ;

Le problème

La distance d'édition, ou distance de Levenshtein, est une mesure de la similarité de deux chaînes de caractères : elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.

Par exemple, on peut passer du mot `polynomial` au mot `polygonal` en suivant les étapes suivantes :

- suppression de la lettre `i` : `polynomial` $\longrightarrow$ `polynomal` ;
- remplacement du `n` par un `g` : `polynomal` $\longrightarrow$ `polygomal` ;
- remplacement du `m` par un `n` : `polygomal` $\longrightarrow$ `polygonal` ;

Le problème

La distance d'édition, ou distance de Levenshtein, est une mesure de la similarité de deux chaînes de caractères : elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.

Par exemple, on peut passer du mot `polynomial` au mot `polygonal` en suivant les étapes suivantes :

- suppression de la lettre `i` : `polynomial` $\longrightarrow$ `polynomal` ;
- remplacement du `n` par un `g` : `polynomal` $\longrightarrow$ `polygomal` ;
- remplacement du `m` par un `n` : `polygomal` $\longrightarrow$ `polygonal` ;

donc la distance d'édition entre ces deux mots est égale au plus à 3, et on se convaincra qu'il n'est pas possible de faire mieux.

Le problème

Nous allons calculer la distance d'édition entre deux mots

$$a = a_0 a_1 \cdots a_{m-1}$$

et

$$b = b_0 b_1 \cdots b_{n-1}$$

en généralisant le problème, c'est-à-dire en définissant la distance d'édition $\mathrm{d}(i, j)$ entre les mots

$$a_0 a_1 \cdots a_{i-1}$$

et

$$b_0 b_1 \cdots b_{j-1}.$$

Le problème

Nous allons calculer la distance d'édition entre deux mots

$$a = a_0 a_1 \cdots a_{m-1}$$

et

$$b = b_0 b_1 \cdots b_{n-1}$$

en généralisant le problème, c'est-à-dire en définissant la distance d'édition $\mathrm{d}(i, j)$ entre les mots

$$a_0 a_1 \cdots a_{i-1}$$

et

$$b_0 b_1 \cdots b_{j-1}.$$

Ce problème possède la propriété de sous-structure optimale.

Question 1

En séparant des cas disjoints concernant a_{i-1} et/ou b_{j-1} dans le chemin optimal reliant $a_0 a_1 \cdots a_{i-1}$ et $b_0 b_1 \cdots b_{j-1}$ (insertion, suppression, remplacement éventuel), exprimer dans chaque cas $\mathrm{d}(i,j)$ en fonction de $\mathrm{d}(i-1,j)$, $\mathrm{d}(i,j-1)$, $\mathrm{d}(i-1,j-1)$.

En déduire que

$$\mathrm{d}(i,j) = \min\left(\mathrm{d}(i-1,j)+1,\ \mathrm{d}(i,j-1)+1,\ \mathrm{d}(i-1,j-1)+\delta_{a_{i-1}\neq b_{j-1}}\right).$$

Solution

Dans un chemin optimal reliant $a_0 a_1 \cdots a_{i-1}$ et $b_0 b_1 \cdots b_{j-1}$, on fait la disjonction de cas :

- soit on a fait une suppression de la lettre a_{i-1} dans le mot de départ, et on compare $a_0 a_1 \cdots a_{i-2}$ et $b_0 b_1 \cdots b_{j-1}$, d'où une distance de $\mathrm{d}(i-1, j) + 1$;

Solution

Dans un chemin optimal reliant $a_0 a_1 \cdots a_{i-1}$ et $b_0 b_1 \cdots b_{j-1}$, on fait la disjonction de cas :

- soit on a fait une suppression de la lettre a_{i-1} dans le mot de départ, et on compare $a_0 a_1 \cdots a_{i-2}$ et $b_0 b_1 \cdots b_{j-1}$, d'où une distance de $\mathrm{d}(i-1, j) + 1$;
- soit on a fait une insertion de la lettre b_{j-1} dans le mot d'arrivée, et on compare $a_0 a_1 \cdots a_{i-1}$ et $b_0 b_1 \cdots b_{j-2}$, d'où une distance de $\mathrm{d}(i, j-1) + 1$;

Solution

Dans un chemin optimal reliant $a_0 a_1 \cdots a_{i-1}$ et $b_0 b_1 \cdots b_{j-1}$, on fait la disjonction de cas :

- soit on a fait une suppression de la lettre a_{i-1} dans le mot de départ, et on compare $a_0 a_1 \cdots a_{i-2}$ et $b_0 b_1 \cdots b_{j-1}$, d'où une distance de $\mathrm{d}(i-1, j) + 1$;
- soit on a fait une insertion de la lettre b_{j-1} dans le mot d'arrivée, et on compare $a_0 a_1 \cdots a_{i-1}$ et $b_0 b_1 \cdots b_{j-2}$, d'où une distance de $\mathrm{d}(i, j-1) + 1$;
- soit on s'occupe simultanément des lettres a_{i-1} et b_{i-1} à partir des mots $a_0 a_1 \cdots a_{i-2}$ et $b_0 b_1 \cdots b_{j-2}$ et il faut distinguer deux cas

Solution

Dans un chemin optimal reliant $a_0 a_1 \cdots a_{i-1}$ et $b_0 b_1 \cdots b_{j-1}$, on fait la disjonction de cas :

- soit on a fait une suppression de la lettre a_{i-1} dans le mot de départ, et on compare $a_0 a_1 \cdots a_{i-2}$ et $b_0 b_1 \cdots b_{j-1}$, d'où une distance de $\mathrm{d}(i-1, j) + 1$;
- soit on a fait une insertion de la lettre b_{j-1} dans le mot d'arrivée, et on compare $a_0 a_1 \cdots a_{i-1}$ et $b_0 b_1 \cdots b_{j-2}$, d'où une distance de $\mathrm{d}(i, j-1) + 1$;
- soit on s'occupe simultanément des lettres a_{i-1} et b_{i-1} à partir des mots $a_0 a_1 \cdots a_{i-2}$ et $b_0 b_1 \cdots b_{j-2}$ et il faut distinguer deux cas
  - soit $a_{i-1} \neq b_{j-1}$ et on aura remplacé la lettre a_{i-1} par la lettre b_{j-1}, ce qui correspond à une distance de $\mathrm{d}(i-1, j-1) + 1$;

Solution

Dans un chemin optimal reliant $a_0 a_1 \cdots a_{i-1}$ et $b_0 b_1 \cdots b_{j-1}$, on fait la disjonction de cas :

- soit on a fait une suppression de la lettre a_{i-1} dans le mot de départ, et on compare $a_0 a_1 \cdots a_{i-2}$ et $b_0 b_1 \cdots b_{j-1}$, d'où une distance de $\mathrm{d}(i-1, j) + 1$;
- soit on a fait une insertion de la lettre b_{j-1} dans le mot d'arrivée, et on compare $a_0 a_1 \cdots a_{i-1}$ et $b_0 b_1 \cdots b_{j-2}$, d'où une distance de $\mathrm{d}(i, j-1) + 1$;
- soit on s'occupe simultanément des lettres a_{i-1} et b_{i-1} à partir des mots $a_0 a_1 \cdots a_{i-2}$ et $b_0 b_1 \cdots b_{j-2}$ et il faut distinguer deux cas
  - soit $a_{i-1} \neq b_{j-1}$ et on aura remplacé la lettre a_{i-1} par la lettre b_{j-1}, ce qui correspond à une distance de $\mathrm{d}(i-1, j-1) + 1$;
  - soit $a_{i-1} = b_{j-1}$ et alors il n'y a pas d'opération à faire, ce qui correspond à une distance de $\mathrm{d}(i-1, j-1)$ $(+0)$.

On a donc bien en général

$$\boxed{\mathrm{d}(i,j) = \min\left(\mathrm{d}(i-1,j)+1,\ \mathrm{d}(i,j-1)+1,\ \mathrm{d}(i-1,j-1)+\delta_{a_{i-1}\neq b_{j-1}}\right).}$$

On a donc bien en général

$$\boxed{\mathrm{d}(i,j) = \min\left(\mathrm{d}(i-1,j)+1,\ \mathrm{d}(i,j-1)+1,\ \mathrm{d}(i-1,j-1)+\delta_{a_{i-1}\neq b_{j-1}}\right).}$$

Schéma de dépendance :

$$\begin{array}{ccc} \mathrm{d}(i-1,j-1) & & \mathrm{d}(i,j-1) \\ & \searrow & \downarrow \\ \mathrm{d}(i-1,j) & \longrightarrow & \mathrm{d}(i,j) \end{array}$$

Question 2

Que valent $\mathrm{d}(i, 0)$ et $\mathrm{d}(0, j)$?

Question 2

Que valent $\mathrm{d}(i, 0)$ et $\mathrm{d}(0, j)$?

Solution

Pour passer d'un mot de i lettres à un mot vide, il faut au minimum i suppressions, et pour passer d'un mot vide à un mot de j lettres, il faut au minimum j insertions.

Question 2

Que valent $\mathrm{d}(i,0)$ et $\mathrm{d}(0,j)$?

Solution

Pour passer d'un mot de i lettres à un mot vide, il faut au minimum i suppressions, et pour passer d'un mot vide à un mot de j lettres, il faut au minimum j insertions.

Donc $\boxed{\mathrm{d}(i,0) = i \text{ et } \mathrm{d}(0,j) = j.}$

Question 3

Écrire deux fonctions permettant de calculer la distance de a à b, l'une par approche de bas en haut (avec un tableau puis une simple ligne), l'autre par approche de haut en bas (avec un dictionnaire).

solution

Version tabulaire de bas en haut avec tableau bidimensionnel

solution

Version tabulaire de bas en haut avec tableau bidimensionnel

Python

```python
def distance_edition(a, b):
    m, n = len(a), len(b)
    # Tableau de format (m+1)x(n+1) pour calculer d[m][n]
    d =[[0 for j in range(n + 1)] for i in range(m + 1)]
    # Initialisation avec qu. 2
    for i in range(m + 1):
        d[i][0] = i
    for j in range(n + 1):
        d[0][j] = j
    # Remplissage du tableau
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if a[i - 1] == b[j - 1]:
                delta = 0
            else:
                delta = 1
            d[i][j] = min(d[i-1][j] + 1, d[i][j-1] + 1,
                          d[i-1][j-1] + delta)
    return d[m][n]
```

solution

Version tabulaire de bas en haut avec tableau unidimensionnel

solution

Version tabulaire de bas en haut avec tableau unidimensionnel
Une première version où on recopie la ligne à chaque tour de boucle `i`

solution

Version tabulaire de bas en haut avec tableau unidimensionnel
Une première version où on recopie la ligne à chaque tour de boucle `i`

Python

```python
def distance_edition(a, b):
    m, n = len(a), len(b)
    # Tableau-ligne de taille n + 1
    d =[j for j in range(n + 1)]
    for i in range(1, m + 1): # à la fin du tour de
        # boucle, d contient les d_{i, j} pour 0 <= j <= n.
        d1 = d.copy() # contient les d_{i - 1, j}
        d[0] = i # cf question 2
        for j in range(1, n + 1):
            if a[i - 1] == b[j - 1]:
                delta = 0
            else:
                delta = 1
            d[j] = min(d1[j] + 1, d[j - 1] + 1,
                       d1[j - 1] + delta)
    return d[n]
```

solution

Une deuxième version où on ne garde qu'une valeur entière en mémoire

solution

Une deuxième version où on ne garde qu'une valeur entière en mémoire

Python

```python
def distance_edition(a, b):
    m, n = len(a), len(b)
    # Tableau-ligne de taille n + 1
    d =[j for j in range(n + 1)]

    for i in range(1, m + 1): # à la fin du tour de
        # boucle, d contient les d_{i, j} pour 0 <= j <= n.
        dim1jm1 = d[0] # contient d_{i-1, j-1}
        d[0] = i # cf question 2
        for j in range(1, n + 1):
            if a[i - 1] == b[j - 1]:
                delta = 0
            else:
                delta = 1
            dim1jm1, d[j] = d[j], min(d[j] + 1, d[j-1] + 1,
                                      dim1jm1 + delta)
    return d[n]
```

solution

Version de haut en bas avec dictionnaire

solution

Version de haut en bas avec dictionnaire

Python

```python
def distance_edition(a, b):
    m, n = len(a), len(b)
    dico = {} # Dictionnaire pour mémoïsation
    def d(i, j):
        if i == 0:
            return i
        if j == 0:
            return j
        if (i, j) not in dico:
            if a[i - 1] == b[j - 1]:
                delta = 0
            else:
                delta = 1
            dico[(i, j)] = min(d(i-1, j) + 1,
                               d(i, j-1) + 1,
                               d(i-1, j-1) + delta)
        return dico[(i, j)]
    return d(m, n)
```

Question 4

Récrire cette dernière pour obtenir le chemin consistant à savoir, si à chaque étape, il a fallu faire une insertion, une suppression, une substitution, ou rien.

solu

Python

```python
def distance_edition_et_chemin(a, b):
    m, n = len(a), len(b)

    dico = {} # Dictionnaire pour mémoïsation
    operation = {} # Dictionnaire pour les opérations.
    # Les valeurs seront des triplets (di, dj, op) tel
    # que le minimum soit atteint en (i + di, j + dj)
    # et op une chaîne de caractère décrivant l'opération.

    def d(i, j):
        if i == 0:
            return i
        if j == 0:
            return j
        if (i, j) not in dico:
            if a[i - 1] == b[j - 1]:
                delta = 0
            else:
                delta = 1
            dico[(i, j)] = min(d(i-1, j) + 1, d(i, j-1) + 1,
                               d(i-1, j-1) + delta)
            if dico[(i, j)] == d(i - 1, j) + 1:
                operation[(i, j)] = (-1, 0,
                                     f"suppression de {a[i - 1]}")
            elif dico[(i, j)] == d(i, j - 1) + 1:
                operation[(i, j)] = (0, -1,
                                     f"insertion de {b[j - 1]}")
```

solution

Python

```python
# [suite de distance_edition_et_chemin]
        elif dico[(i, j)] == d(i - 1, j - 1) + delta:
            if delta == 1:
                operation[(i, j)] = (-1, -1,
                f"remplacement de {a[i-1]} par {b[j-1]}")
            else:
                operation[(i, j)] = (-1, -1,
                        f"{a[i - 1]} laissé inchangé")
    return dico[(i, j)]

distance = d(m, n) # remplissage des dictionnaires
# Retro-propagation
liste_operations = []

(i, j) = (m, n)
while i > 0 and j > 0:
    di, dj, op = operation[(i, j)]
    liste_operations.append(op)
    i += di
    j += dj
```

solution

Python

```python
# [suite de distance_edition_et_chemin]
    while j > 0: # Lorsque i = 0
        liste_operations.append(f"insertion de {b[j - 1]}")
        j -= 1

    while i > 0: # Lorsque j = 0
        liste_operations.append(
                                f"suppression de {a[i - 1]}")
        i -= 1

    liste_operations.reverse()

    return distance, liste_operations
```