Programmation Dynamique

Contexte

La programmation dynamique est une technique algorithmique puissante utilisée pour résoudre des problèmes complexes en les décomposant en sous-problèmes plus simples. Elle est particulièrement utile lorsque ces sous-problèmes se **chevauchent** et peuvent être résolus de manière récursive. En effet, au lieu de recalculer plusieurs fois les mêmes sous-problèmes, la programmation dynamique les résout une seule fois et stocke les résultats intermédiaires pour une réutilisation ultérieure (**memoïsation**). Cela permet d'améliorer significativement l'efficacité de l'algorithme, réduisant souvent un temps d'exécution exponentiel à un temps polynomial.
Nous aborderons des exemples concrets pour illustrer comment cette technique peut être utilisée pour optimiser des problèmes classiques tels que le problème du sac à dos.
Objectifs du cours :

1. Comprendre les concepts de base de la programmation dynamique.
2. Maîtriser les approches de mémoïsation et de tabulation.
3. Savoir identifier les problèmes pouvant être résolus par cette technique.

Prérequis : Avoir une bonne compréhension des algorithmes et des structures de données de base.

1 Premier exemple : calcul des coefficients binomiaux

1.1 Rappels sur les coefficients binomiaux

▲ DÉFINITION 1

On rappelle qu'étant donné un entier naturel n, on définit **factorielle** n, et on note $n!$, le nombre entier : $n! = 1 \times 2 \times \cdots \times (n-1) \times n = \prod_{k=1}^{n} k$.
On rappelle que $0! = 1$.

Étant donné deux entiers naturels n et p, on appelle **coefficient binomial** « p **parmi** n », et l'on note $\binom{n}{p}$, le nombre de parties à p éléments d'un ensemble à n éléments.

➔ PROPOSITION 1

- Formule explicite : $\binom{n}{p} = \frac{\overbrace{n(n-1)\cdots(n-p+1)}^{p \text{ termes}}}{p!} = \frac{1}{p!}\prod_{k=0}^{p-1}(n-k) = \frac{n!}{p!(n-p)!}$.
- $\binom{n}{p} = \binom{n}{n-p}$ si $p \leqslant n$,
- **Cas de base :** $\binom{n}{0} = 1$; $\binom{0}{k} = 0$ si $k \geqslant 1$; $\binom{n}{p} = 0$ si $p > n$
- **Récurrence :** $\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$ si $n \geqslant 1$ et $p \geqslant 1$ (**Relation de Pascal**).

Remarque

La **preuve** de la relation de Pascal est d'un grand interêt pour nous au vu des raisonnements que nous aurons à effectuer. Rappelons-la.

Preuve

Soit $n \geqslant 1$ et $p \geqslant 1$. Soit $E = \{x_1, \ldots, x_n\}$ un ensemble à n éléments. Il y a deux types de parties à p éléments de E :

- celles qui ne contiennent pas x_n : il s'agit en fait des parties à p éléments de $\{x_1, \ldots, x_{n-1}\}$. Il y a $\binom{n-1}{p}$ telles parties.
- celles qui contiennent x_n : mais ces parties sont exactement les parties à $p-1$ éléments de $\{x_1, \ldots, x_{n-1}\}$, auxquelles on a rajouté x_n. Il y a donc $\binom{n-1}{p-1}$ telles parties.

On a donc la relation de récurrence $\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$

1.2 Des algorithmes brutaux pour calculer des coefficients binomiaux

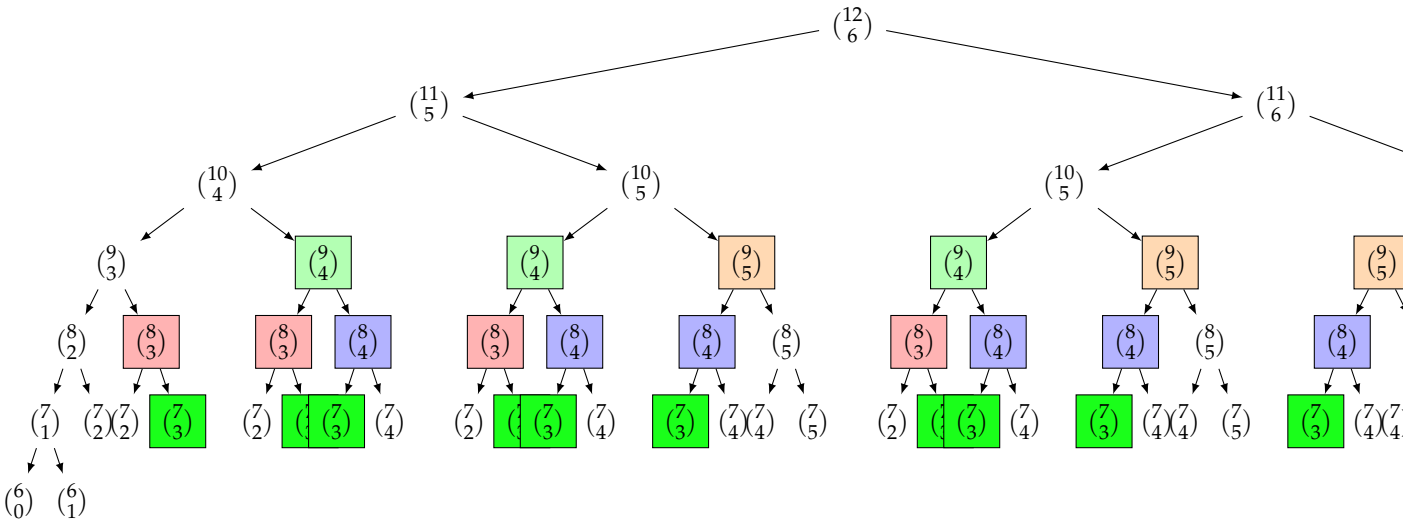
```
1
2 def  factorielle(n):
```

```
    p=1
    for k in range(1,n+1):
        p*=k
    return p

def binom_complexite_max(n,p):
    if 0<=p<=n:
        return factorielle(n)//(factorielle(p)*factorielle(n-p))
    else:
        return 0

def binom_via_produit(n,p): #on simplifie d'avance n!/(n-p)!
    if not(0<=p<=n):
        return 0
    #et sinon:
    p1=1
    for k in range(n,n-p,-1):
        p1*=k
    return p1//factorielle(p)
```

1.3 Un algorithme récursif pour le calcul des coefficients binomiaux

Il est plus élégant d'envisager un algorithme récursif :

```
def binom_via_formule_Pascal_rec(n,p):
    #intention d'utiliser la formule de Pascal
    if not(0<=p<=n): #cas de base
        return 0
    if p==0:
        return 1
    if n==0:
        return 1
    else:
        return binom_via_formule_Pascal_rec(n-1,p)+binom_via_formule_Pascal_rec(n-1,p-1)
```



Attention

Cet algorithme pose un énorme problème de complexité en temps. En effet, il refait les mêmes calculs un très grand nombre de fois comme le montre l'arbre ci–dessus.

1.4 Programmation dynamique : remplissage du triangle de Pascal

On connaît bien sûr une solution à ce problème : il suffit de stocker les valeurs que l'on a calculées.

```
def binom_via_triangle_Pascal(n,p, affiche=False):
    if not(0<=p<=n):
        return 0
    Triangle=[[0 for k in range(p+1)] for N in range(n+1)]
    #Le tableau Triangle[N][k] a vocation a contenir $\binom{N}{k}$
    for N in range(n+1):
        Triangle[N][0]=1
    #on initialise le tableau
    for N in range(1,n+1):
        for k in range(1,p+1):
            if k<=N:
                Triangle[N][k]=Triangle[N-1][k]+Triangle[N-1][k-1]
    #on remplit le tableau en utilisant pour chaque ligne la formule de Pascal
```

```
    return Triangle[n][p] #on renvoie le coeff voulu
```

Remarque

On remarque cependant que le calcul de certaines valeurs n'est pas nécessaire à l'obtention du coefficient binomial.

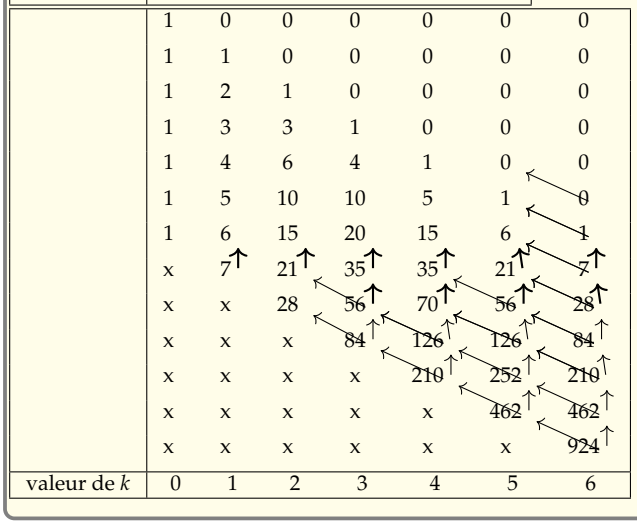
Exemple

Envisageons le calcul de $\binom{12}{6}$. On peut bien sûr compléter le triangle de Pascal ligne à ligne :

								Valeur de n
	1	0	0	0	0	0	0	0
	1	1	0	0	0	0	0	1
	1	2	1	0	0	0	0	2
	1	3	3	1	0	0	0	3
	1	4	6	4	1	0	0	4
	1	5	10	10	5	1	0	5
	1	6	15	20	15	6	1	6
	1	7	21	35	35	21	7	7
	1	8	28	56	70	56	28	8
	1	9	36	84	126	126	84	9
	1	10	45	120	210	252	210	10
	1	11	55	165	330	462	462	11
	1	12	66	220	495	792	924	12
valeur de k	0	1	2	3	4	5	6	

Cependant, on s'aperçoit que seulement une partie des valeurs « intermédiaires » sont nécessaires à l'obtention du résultat voulu :

	1	0	0	0	0	0	0
	1	1	0	0	0	0	0
	1	2	1	0	0	0	0
	1	3	3	1	0	0	0
	1	4	6	4	1	0	0
	1	5	10	10	5	1	0
	1	6	15	20	15	6	1
	x	7	21	35	35	21	7
	x	x	28	56	70	56	28
	x	x	x	84	126	126	84
	x	x	x	x	210	252	210
	x	x	x	x	x	462	462
	x	x	x	x	x	x	924
valeur de k	0	1	2	3	4	5	6



1.5 Programmation dynamique avec récursivité et mémoïsation

La constatation précédente donne l'idée, non de partir de la première ligne vers la dernière, mais de raisonner en partant du résultat voulu et de n'appeler que les calculs dont on a besoin.

```
def binom_avec_triangle_memoise_rec(n,p,affiche=False):
    if not(0<=p<=n): #cas de base
        return 0

    Triangle={}
    #fonction auxiliaire qui remplit le dictionaire triangle
    def aux_rec(N,k):
        #cas de base
        if k==0:
            Triangle[(N,k)]=1
        elif N==0:
            Triangle[(N,k)]=0
        else: # recursivite:
```

```
            aux_rec(N-1,k-1)
            aux_rec(N-1,k)
            Triangle[(N,k)]=Triangle[(N-1,k)]+Triangle[(N-1,k-1)]
        #on lance la demande:
    aux_rec(n,p)
        # on renvoie uniquement le coefficient que l'on veut:
    return Triangle[(n,p)]
```

Remarque

Le morceau de code central est :

```
aux_rec(N-1,k-1)
aux_rec(N-1,k)
Triangle[(N,k)]=Triangle[(N-1,k)]+Triangle[(N-1,k-1)]
```

On notera l'élégance du code, qui s'approche au plus près de la définition et laisse le soin de gérer les liens et les calculs à la récursivité. La structure de dictionnaire permet un stockage souple de l'information.

Méthode

Ce premier exemple contient quasiment tous les outils que nous réutiliserons.

1. On dispose au départ d'un problème :
   - Que l'on peut éventuellement résoudre brutalement, mais avec un coût exorbitant,
   - et/ou que l'on peut résoudre récursivement mais où les sous–problèmes se **chevauchent**. La récursivité brute mène donc à des calculs répétitifs.

   Dans la programmation dynmique :
2. L'idée est de stocker les résultats de tous les sous–problèmes dans une table, technique appelée **mémoïsation**.
3. Le remplissage de la table s'effectue en mettant en évidence une **relation de récurrence**, dont la démonstration sera souvent **de nature combinatoire**. On dispose aussi de « cas de base » permettant d'amorcer la relation.
4. On peut remplir toute la table, puis sélectionner la valeur qui nous intéresse. On propage alors les cas de base peu à peu.

2 Problème du sac à dos

2.1 Introduction, formulation du Problème, et notations

Le problème du sac à dos (Knapsack Problem) est un problème classique d'optimisation combinatoire. Il consiste à sélectionner un ensemble d'objets à partir d'un ensemble donné de manière à maximiser la valeur totale sans dépasser la capacité maximale du sac à dos.

Soit un ensemble d'objets $\{1, 2, \ldots, n\}$, où chaque objet i a une valeur v_i et un poids w_i. On dispose d'un sac à dos pouvant supporter un poids maximal W. L'objectif est de déterminer l'ensemble des objets à inclure dans le sac à dos de manière à maximiser la valeur totale des objets inclus, sans que la somme de leurs poids n'excède W.

- n : le nombre total d'objets.
- v_i : la valeur de l'objet i ($1 \leqslant i \leqslant n$).
- w_i : le poids de l'objet i ($1 \leqslant i \leqslant n$).
- W : la capacité maximale du sac à dos.
- x_i : une variable binaire indiquant si l'objet i est inclus dans le sac à dos ($x_i = 1$) ou non ($x_i = 0$).

2.2 Formulation Mathématique

Le problème du sac à dos peut être formulé mathématiquement comme suit :

$$\text{Maximiser} \sum_{i=1}^{n} v_i x_i$$

Sous les deux contraintes :

$$\sum_{i=1}^{n} w_i x_i \leqslant W \quad \text{et} \quad \forall i \in [\![1, n]\!] \; x_i \in \{0, 1\}$$

2.3 Première approche brutale

Exercice 1 Une **solution** est donc une liste d'objets formée d'éléments de 1 à n. Notez bien qu'une solution n'est pas optimale dans cette définition.
Ecrire un programme qui prend en entrée les paramètres du problème (les valeurs et les poids des objets, ainsi que la capacité du sac à dos) et une solution (une liste d'indices d'objets de 1 à n), et qui vérifie si la solution est acceptable (en termes de poids) et retourne la valeur totale du sac à dos si elle est acceptable, et retourne False sinon.

```python
def check_solution(values, weights, capacity, solution):
    total_weight = 0
    total_value = 0

    for item in solution:
        # Convertir l'index de 1 a n en index de 0 a n-1
        index = item - 1
        total_weight += weights[index]
        total_value += values[index]

    if total_weight > capacity:
        return False
    else:
        return True, total_value

# Exemple d'utilisation
values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50
solution = [2, 3]  # Les objets selectionnes sont les objets 2 et 3

result = check_solution(values, weights, capacity, solution)
if result is False:
    print("Solution inacceptable : poids trop lourd")
else:
    is_acceptable, total_value = result
    print(f"Solution acceptable : valeur totale du sac = {total_value}")
```

Exercice 2 Proposer un algorithme énumérant toutes les solutions et retenant la meilleure.

2.4 Algorithme avec méthode de Programmation Dynamique

On utilise une table K pour stocker les valeurs maximales réalisables pour différentes capacités du sac à dos. La table K est construite comme suit :

1. **Contenu** $K[i][w]$ représente la valeur maximale qui peut être obtenue en utilisant les i premiers objets $1, 2, \ldots, i$, avec une capacité maximale du sac égale à w. Nous cherchons $K[n][W]$.
2. **Initialisation (Cas de base)**
   - On a $K[0][w] = 0$ pour tout w en effet avec 0 objets disponibles, on peut obtenir au maximum la valeur).
   - $K[i][0] = 0$ pour tout i : en effet, avec une capacité maximale du sac égale à 0, on peut obtenir au maximum une valeur égale à .
3. **Formule de récurrence** Pour chaque objet i (1 à n) et chaque capacité w (0 à W) :

$$K[i][w] = \begin{cases} K[i-1][w] & \text{si } w_i > w \\ \max(K[i-1][w], K[i-1][w-w_i] + v_i) & \text{sinon} \end{cases}$$

Attention

Cette relation de récurrence est **fondamentale**. Elle dit ceci : Pour obtenir la valeur maximale réalisable d'un sac, avec une capacité max égale à w et en utilisant les i premier objets :

(a) **OU BIEN** l'objet i est trop lourd pour rentrer dans le sac, même seul. Dans ce cas on n'utilise que les objets numéros 1 à $i-1$. La valeur de $K[i][w]$ vaut alors $K[i-1][w]$.

(b) **OU BIEN** l'objet i peut rentrer dans le sac. On doit donc comparer deux stratégies :

- Dans **la première** on utilise le i–ème objet. Il faut alors s'arranger pour optimiser l'utilisation des objets 1 à $i-1$; cependant le poids maximum ne vaudra plus w mais il vaudra . La valeur obtenue pour une composition optimale de ce sac est alors $K[i-1][w-w_i] + v_i$.
- Dans **la seconde** on n'utilise que les objets numéros 1 à $i-1$, et cette valeur vaut .

Il faut alors retenir entre ces deux solutions possibles la meilleure, d'où la valeur $\max(K[i-1][w], K[i-1][w-w_i] + v_i)$.

La valeur maximale réalisable avec n objets et une capacité W est donnée par $K[n][W]$.

2.5 Implémentation en Python

```python
def knapsack(values, weights, capacity):
    n = len(values)
    # Initialisation de la table K avec des zeros
    K = [[0 for x in range(capacity + 1)] for x in range(n + 1)]
```

```
    # Remplissage de la table K
    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i-1] <= w:
                K[i][w] = max(K[i-1][w], K[i-1][w-weights[i-1]] + values[i-1])
            else:
                K[i][w] = K[i-1][w]

    # La valeur maximale realisable est dans K[n][W]
    return K[n][capacity]
```

Listing 1 – Implémentation de la Programmation Dynamique pour le Problème du Sac à Dos

2.6 Avec solution explicite

On peut demander non seulement la valeur maximale mais aussi une solution explicite permettant de réaliser cette valeur. Il suffit alors de modifier le programme qui précède pour **remonter** la table K, de la dernière à la première ligne, et reconstruire de proche en proche la solution.

- On initialise `selected_items` à la liste vide.
- Pour déterminer quels objets composent la solution optimale, on part de la position $K[n][W]$ et on remonte dans la table.
- Si pour l'objet i, on a $K[i][w] \neq K[i-1][w]$, cela signifie que l'objet i a été inclus.
- On ajoute alors l'objet i à la liste `selected_items` (en ajustant l'indexation pour qu'elle soit de 1 à n)
- On met à jour la capacité restante w en soustrayant le poids de l'objet inclus et on continue jusqu'à ce que la capacité soit 0 ou que tous les objets aient été considérés.

```
def knapsack(values, weights, capacity):
    n = len(values)
    # Initialisation de la table K avec des zeros
    K = [[0 for x in range(capacity + 1)] for x in range(n + 1)]

    # Remplissage de la table K
    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i-1] <= w:
                K[i][w] = max(K[i-1][w], K[i-1][w-weights[i-1]] + values[i-1])
            else:
                K[i][w] = K[i-1][w]

    # La valeur maximale realisable est dans K[n][W]
    max_value = K[n][capacity]

    # Retracer les objets inclus dans la solution optimale
    w = capacity
    selected_items = []

    for i in range(n, 0, -1):
        if K[i][w] != K[i-1][w]:
            selected_items.append(i)  # Ajout de l'objet avec un index de 1
                jusque n (et non de 0 a n-1, pour garder les notations de
                l'intro)
            w -= weights[i-1]

    return max_value, selected_items
```

Listing 2 – Implémentation de la Programmation Dynamique pour le Problème du Sac à Dos

2.7 Mémoïsation et dictionnaires

Remarques

On constate que la solution proposée remplit toute la table en utilisant une stratégie **du bas vers le haut** (des cas de base vers les résultats plus complexes que l'on cherche à atteindre). On peut écrire une stratégie du haut vers le bas en programmant récursivement, et en utilisant un dictionnaire. On s'épargnera ainsi, par la même occasion, des calculs inutiles.

Exemple

Pour 5 objets, dont les valeurs sont [10,7,25,30,50] et les poids sont [10,11,8,6,2], on trouve avec une capacité maximale de 20 :

Objets utilisés : de 1 à																					
0	0	0	0	0	0	_	0	0	0	0	0	_	0	_	0	_	_	_	0	_	0
1	_	0	_	0	0	_	0	0	_	0	10	_	10	_	10	_	_	_	10	_	10
2	_	_	_	_	0	_	0	_	_	_	10	_	10	_	10	_	_	_	10	_	10
3	_	_	_	_	_	_	_	_	_	_	_	_	25	_	25	_	_	_	35	_	35
4	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	55	_	55
5	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	105
capacité maximale :	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

```python
def knapsack_dict(values, weights, capacity):
    n = len(values)
    # Initialisation du dictionnaire K
    K = {}

    # Fonction auxiliaire pour remplir le dictionnaire
    def knapsack_recursive(i, w):

        # Cas de base: pas d'objets ou capacite nulle
        if i == 0 or w == 0:
            K[(i,w)]=0
        elif weights[i-1] > w:
            # Si l'objet est trop lourd, appeler la strategie sans i
            knapsack_recursive(i-1, w)
            K[(i,w)]=K[(i-1,w)]
        else:
            # Sinon, prendre la valeur maximale entre inclure ou non l'objet
            knapsack_recursive(i-1, w),
            knapsack_recursive(i-1, w-weights[i-1])
            #calcul des sous-problemes et mise a jour:
            K[(i,w)]=max(K[(i-1,w)],K[(i-1,w-weights[i-1])]+values[i-1])

    #remplissage de table specifiquement:
    knapsack_recursive(n,capacity)

    # Renvoyer la valeur maximale
    return K[(n, capacity)]
```

1. **Initialisation du Dictionnaire** K :
   - Le dictionnaire K stocke les valeurs maximales pour chaque paire (nombre d'objets, capacité restante).
2. **Fonction Auxiliaire** `knapsack_recursive` :
   - Cette fonction remplit le dictionaire K. A la fin, $K[(i,w)]$ ou bien n'existe pas, ou bien contient la valeur maximale pouvant être obtenue en utilisant les premiers i objets et une capacité w.
   - Les cas de base sont : pas d'objets ($i = 0$) ou capacité nulle ($w = 0$), avec une valeur maximale de 0.
   - Si l'objet actuel est trop lourd pour être inclus ($weights[i-1] > w$), la fonction ne l'inclut pas. Elle calcule le résultat pour les $i-1$ objets. On a alors un appel récursif pour faire ce calcul, et on affecte $K[(i,w)] = K[(i-1,w)]$.
   - Sinon, on calcule la valeur maximale entre inclure ou non l'objet actuel.
3. **Calcul de la Valeur Maximale** :
   - La valeur maximale pouvant être obtenue est calculée en appelant `knapsack_recursive(n, capacity)`.

2.8 Conclusion

Il y a d'autres Types de Problèmes du Sac à Dos

- **Problème du Sac à Dos (0/1 Knapsack Problem)** : Chaque objet peut être soit inclus dans le sac à dos soit exclu. Il n'est pas possible de fractionner les objets.
- **Problème du Sac à Dos Fractionnaire (Fractional Knapsack Problem)** : Les objets peuvent être fractionnés, c'est-à-dire qu'une partie d'un objet peut être incluse dans le sac à dos. Ce problème est résolu de manière optimale par un algorithme glouton.

- **Problème du Sac à Dos Multidimensionnel (Multidimensional Knapsack Problem)** : Chaque objet a plusieurs dimensions de poids, et le sac à dos a des limites de capacité pour chaque dimension. Le mot poids est alors à prendre en un sens abstrait : il s'agit de plusieurs attribut (Masse, Volume, Radioactivité) par exemple.

Exemples d'Applications

- **Logistique et Transport** : Optimisation de l'utilisation de l'espace et du poids pour le chargement de véhicules.
- **Finance** : Sélection de portefeuilles d'investissement pour maximiser le rendement sous contrainte de risque.
- **Planification de Projet** : Allocation de ressources limitées à des tâches pour maximiser le bénéfice global.

3 Ordonnancement de tâches

Description du Problème

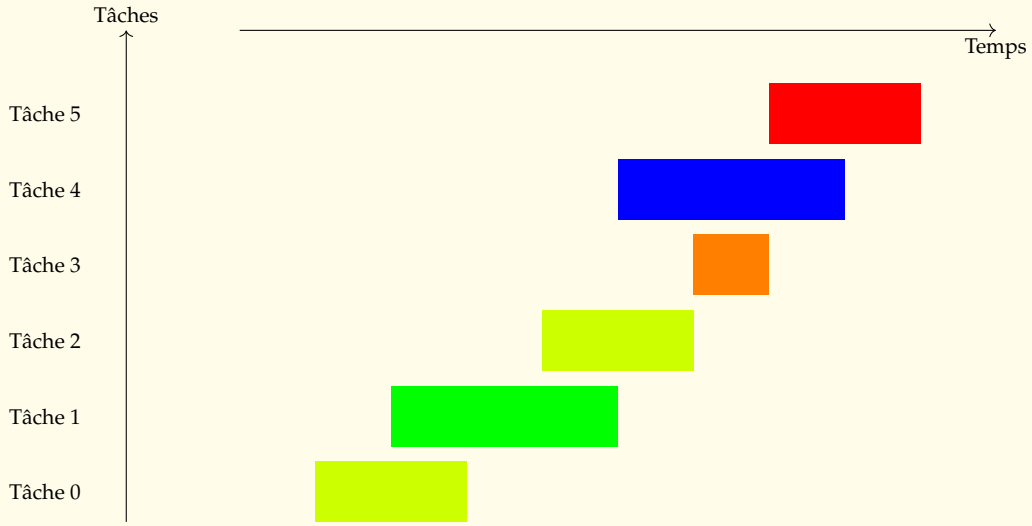
Nous avons une liste de tâches, chacune définie par un temps de début, un temps de fin, et un profit. On ne peut réaliser qu'une seule tâche à la fois, et toute tâche entamée doit être terminée avant d'en commencer une nouvelle. L'objectif est de sélectionner un sous-ensemble de tâches non chevauchantes de manière à maximiser le profit total.

On dispose plus formellement d'un ensemble de tâches (d_i, f_i, p_i) où d_i est le temps de début de la tache i, f_i est le temps de début de la tache i, et p_i le profit de la tâche i. On indice les n tâches de 0 à $n-1$. Quel est le profit maximum atteignable ?

Exemple

On peut se donner les taches suivantes, avec début, fin et un profit indiqué :

$$\{(2,5,6),(6,7,4),(1,3,5),(5,8,11),(4,6,5),(7,9,2)\}$$



On ne peut réaliser qu'une seule tâche à la fois, aussi certaines tâches ne sont pas compatibles. Par exemple considérons l'agencement

$$(1,3,5),(2,5,6),(4,6,5)$$

comme $2 \in [1,3]$, la seconde des tâches commence pendant que la première est en cours d'éxécution.
Plusieurs combinaisons sont cependant possibles pour attribuer des tâches par exemple :

$$(1,3,5),(4,6,5),(6,7,4),(7,9,2)$$

$(3 \leqslant 4\ ;\ 6 \leqslant 6\ ;\ 7 \leqslant 7)$ pour un profit total valant $5+5+4+2=16$ mais aussi

$$(2,5,6),(5,8,11)$$

pour un profit total valant $6+11=17$.
La seconde solution proposée est donc meilleure.

Attention

Voici deux solutions qui ne fonctionnent pas :

1. Stratégie exhaustive : Enumérer toutes les possibilités de solutions (réalisables), établir leur profit et prendre le maximum. **Trop long**.
2. Stratégie gloutonne : Prendre la première tâche possible, puis la première tâche compatible suivante, et ainsi de suite. On remarquera que sur l'exemple plus haut cela mènerait à conserver les tâches en **gras** suivantes :
   (1, 3, 5), (2, 5, 6), **(4, 6, 5)**, **(6, 7, 4)**, (5, 8, 11), **(7, 9, 2)** Le profit global est de 16, mais on a vu que l'on pouvait faire mieux. Aussi l'algorithme glouton ne fournit–il pas nécessairement l'optimum.

3.1 Algorithme dynamique pour maximiser le profit

1. **Tri des Tâches** : Nous trions les tâches en fonction de leur temps de **fin**. Cela nous permet de toujours considérer les tâches précédentes (qui se terminent plus tôt) avant de considérer une nouvelle tâche.
2. **Initialisation du Tableau** `dp` : Nous utilisons un tableau `dp` où `dp[i]` représente le profit maximum possible en considérant les premières tâches $0, 1, \ldots, i$.
3. **Remplissage du Tableau** `dp` : Pour chaque tâche i, nous évaluons le profit globale de deux stratégies :
   - La stratégie consistant à ne pas inclure la tâche i. Ce profit est stocké dans `dp[i-1]`
   - La stratégie consistant à inclure la tâche i. Nous calculons le profit si nous incluons cette tâche. Pour cela, nous trouvons **la dernière tâche qui ne chevauche pas la tâche** i ; appelons l cette tâche.
   - Le profit total pour inclure la tâche i est la somme du profit de la tâche i et du profit maximum jusqu'à la tâche l, c'est-à-dire `dp[l]+profit[i]`.
   - Nous mettons à jour `dp[i]` en prenant le maximum des profits entre les deux stratégies. Ainsi `dp[i]=max(dp[i-1],`

3.2 Code en Détail

Voici le code avec des commentaires détaillés pour chaque étape :

```python
# Fonction pour trouver la derniere tache non chevauchante
def find_last_non_conflicting(tasks, n):
    for j in range(n - 1, -1, -1):
        if tasks[j][1] <= tasks[n][0]:
            return j
    return -1

# Fonction pour maximiser le profit avec la programmation dynamique
def maximize_profit(tasks):
    # Trier les taches par leur temps de fin
    tasks.sort(key=lambda x: x[1])

    n = len(tasks)

    # Initialiser le tableau dp pour stocker le profit maximum
    dp = [0] * n

    # Le profit de la premiere tache est son propre profit
    dp[0] = tasks[0][2]

    for i in range(1, n):
        # Inclure le profit de la tache courante
        incl_prof = tasks[i][2]
        # Trouver la derniere tache non chevauchante
        l = find_last_non_conflicting(tasks, i)
        if l != -1:
            incl_prof += dp[l]

        # Mettre a jour le dp[i] en prenant le maximum de l'inclusion ou non
        dp[i] = max(incl_prof, dp[i - 1])

    return dp[-1]

# Exemple de taches avec (temps de debut, temps de fin, profit)
tasks = [(1, 3, 5), (2, 5, 6), (4, 6, 5), (6, 7, 4), (5, 8, 11), (7, 9, 2)]
print("Profit maximum possible: ", maximize_profit(tasks))
```

Remarque

La fonction `find_last_non_conflicting(tasks, n)` cherche la dernière tâche qui ne chevauche pas la tâche n. Elle parcourt les tâches de $n-1$ à 0 et renvoie l'index de la première tâche dont le temps de fin est inférieur ou égal au temps de début de la tâche n.

Exemple

Reprenons l'exemple vu plus haut.

1. **Tri des tâches** : Les tâches sont triées par leur temps de fin. Aussi nous avons les tâches triées suivantes :

$$\{(1,3,5),(2,5,6),(4,6,5),(6,7,4),(5,8,11),(7,9,2)\}$$

2. **Initialisation** :
   - `dp[0]` est initialisé à 5 (profit de la première tâche).
3. **Calcul du profit maximum** :
   - Pour chaque tâche i de 1 à $n-1$, nous trouvons le profit en incluant ou en excluant la tâche i :
     - Pour la tâche $i = 0$:son profit est 5, on a déjà posé `dp[0]=5`.
     - Pour la tâche $i = 1$: pas de tâche compatible avec la tache 1 ; le profit de la tache 1 est 6 donc `dp[1] = max(6, 5) = 6`

- Pour la tâche $i = 2$: la dernière tache non conflictuelle avec 2 est 0 ; le profit de la tache 2 est 5 ; et l'optimum pour 0 est 5. Ainsi `dp[2] = max(5 + 5, 6) = 10`.
- Pour la tâche $i = 3$: la dernière tache non conflictuelle avec 3 est 2 ; le profit de la tache 3 est 4 ; et l' optimum pour 2 est 10. Ainsi `dp[3] = max(4 + 10, 10) = 14`
- Pour la tâche $i = 4$: la dernière tache non conflictuelle avec 4 est 1 le profit de la tache 4 est 11 ; l'optimum pour 1 est 6. Ainsi `dp[4] = max(11 + 6, 14) = 17`
- Pour la tâche $i = 5$: la dernière tache non conflictuelle avec 5 est 3 et le profit de la tache 5 est 2. L'optimum pour 3 est 14, ainsi on a `dp[5] = max(2 + 14, 17) = 17`

Résultat

Le profit maximum possible est le dernier élément du tableau `dp`, soit `dp[-1]`, qui est 17 dans cet exemple.

Conclusion

Cet algorithme utilise la programmation dynamique pour résoudre efficacement le problème d'ordonnancement des tâches avec maximisation du profit. La clé est de trouver la dernière tâche non chevauchante pour chaque tâche courante et de décider si on inclut ou non cette tâche dans notre solution optimale.

4 Distance d'édition de Leveshtein

La distance d'édition de Levenshtein est une mesure fondamentale en traitement du langage naturel et en théorie des automates, utilisée pour quantifier la différence entre deux chaînes de caractères. Plus précisément, elle évalue le nombre minimal d'opérations nécessaires pour transformer une chaîne de caractères en une autre, où les opérations autorisées sont l'insertion, la suppression et la substitution de caractères.

4.1 Définition formelle

Soit x et y deux chaînes de caractères sur un alphabet Σ, où $x = x_1x_2 \cdots x_m$ et $y = y_1y_2 \cdots y_n$, avec x_i et y_j appartenant à Σ. La distance d'édition de Levenshtein entre x et y, notée $d(x, y)$, est définie comme le minimum du nombre d'opérations d'édition nécessaires pour convertir x en y.

Les opérations d'édition autorisées sont :

- **Insertion** : Ajouter un caractère dans x pour obtenir une nouvelle chaîne.
- **Suppression** : Supprimer un caractère de x.
- **Substitution** : Remplacer un caractère de x par un autre caractère.

Exemple

Considérons la transformation des mots CHAT en CLAIR. Voici les opérations nécessaires :

1. **Substitution de "H" par "L"**
   - CHAT → CLAT
2. **Insertion de "I" après "A"**
   - CLAT → CLAIT
3. **Insertion de "R" après "I"**
   - CLAIT → CLAIRT
4. **Suppression du dernier caractère "T"**
   - CLAIRT → CLAIR

Résumé des opérations :

- Substitution (H → L)
- Insertion (I)
- Insertion (R)
- Suppression (T)

La distance d'édition entre CHAT et CLAIR est donc au plus 4, puisque ces quatre opérations suffisent pour effectuer la transformation. Cette distance ne vaut pas 4 : on voit facilement que 3 opérations suffisent :

- Substitution (H → L)
- Insertion (I)
- Substitution (T → R)

Remarque — Applications et Importance

La distance de Levenshtein trouve des applications dans divers domaines tels que la correction orthographique, la reconnaissance de formes, la bioinformatique pour l'alignement de séquences d'ADN, et la recherche en texte.

4.2 Formalisation et algorithme de programmation dynamique

Mathématiquement, la distance d'édition de Levenshtein peut être définie par la fonction récursive suivante :

$$d(i,j) = \begin{cases} \max(i,j) & \text{si } \min(i,j) = 0, \\ d(i-1,j)+1 & \text{si } x_i \neq y_j \text{ et } \min(i,j) > 0, \\ d(i-1,j-1) & \text{si } x_i = y_j \text{ et } \min(i,j) > 0, \\ \min\{d(i-1,j)+1, d(i,j-1)+1, d(i-1,j-1)+1\} & \text{sinon.} \end{cases}$$

où $d(i,j)$ représente la distance d'édition entre les sous-chaînes $x_1x_2\cdots x_i$ et $y_1y_2\cdots y_j$.

Construisons une matrice D de dimensions $(m+1)\times(n+1)$, où $D[i][j]$ représente la distance d'édition entre les sous-chaînes $x_1x_2\cdots x_i$ et $y_1y_2\cdots y_j$.

- i varie de 0 à m
- j varie de 0 à n

On initialise les valeurs de la matrice D comme suit :

$$D[0][0] = 0$$

$$D[i][0] = i \text{ pour } i = 1,\ldots,m$$

$$D[0][j] = j \text{ pour } j = 1,\ldots,n$$

subsectionRemplissage de la Matrice

Pour chaque paire d'indices (i,j) où $1 \leqslant i \leqslant m$ et $1 \leqslant j \leqslant n$, on calcule $D[i][j]$ comme suit :

$$D[i][j] = \begin{cases} D[i-1][j-1] & \text{si } x_i = y_j \\ \min\{D[i-1][j]+1, D[i][j-1]+1, D[i-1][j-1]+1\} & \text{si } x_i \neq y_j \end{cases}$$

- Suppression : $D[i][j] = D[i-1][j]+1$
- Insertion : $D[i][j] = D[i][j-1]+1$
- Substitution : $D[i][j] = D[i-1][j-1]+1$

Exemple

Nous allons calculer la distance d'édition de Levenshtein entre "CHAT" et "CLAIR" en utilisant une matrice de distance.

Initialisation

La première ligne et la première colonne de la matrice sont initialisées comme suit :

		C	L	A	I	R
	0	1	2	3	4	5
C	1					
H	2					
A	3		?			
T	4					

Cette matrice a vocation à contenir les distances d'édition de chaque début de mot. Par exemple à la fin de l'algorithme, la case notée ? contiendra la distance d'édition de CHA à CL.

On connaît les valeurs de la première ligne et de la première collone. En effet, il s'agit des distances respectives du mot vide, aux mots C, CL, CLA, CLAI, CLAIR (en première ligne) ; et des mots C, CH, CHA, CHAT au mot vide (en première colonne). Ces distances correspondent au nombre de lettres à effacer ou insérer.

Remplissage de la Matrice

Pour connaître le contenu d'une cellule (i,j), nous voulons calculer la distance d'édition minimale de la chaine formée des i prmiers caractères de CHAT, à la chaine formée des i premiers caractères de CLAIR.

Considérons les opérations suivantes :

- **Suppression** : Distance d'édition en supprimant un caractère.
- **Insertion** : Distance d'édition en ajoutant un caractère.
- **Substitution** : Distance d'édition en remplaçant un caractère.

Pour chaque cellule (i,j), la formule est :

$$d[i][j] = \begin{cases} d[i-1][j-1] & \text{si } x_i = y_j \\ \min\{d[i-1][j]+1, d[i][j-1]+1, d[i-1][j-1]+1\} & \text{si } x_i \neq y_j \end{cases}$$

En effet on se ramène à une distance avec des sous-chaines plus petites en comparant trois stratégies :

$$d(i,j) = \min\left\{\begin{array}{l} d(i-1,j)+1, \text{ (suppression de } x_i\text{)}, \\ d(i,j-1)+1, \text{ (insertion de } y_j\text{)}, \\ d(i-1,j-1) + \text{(coût de substitution de } x_i \text{ en } y_j\text{)}, \end{array}\right\}$$

où le coût de substitution est 0 si $x_i = y_j$ et 1 sinon.

		C	L	A	I	R
	0	1	2	3	4	5
C	1	0	1	2	3	4
H	2	1	1	2	3	4
A	3	2	2	1	2	3
T	4	3	3	2	2	3

Distance Finale
La distance d'édition entre "CHAT" et "CLAIR" est trouvée dans la cellule en bas à droite de la matrice :

$$d(\text{CHAT}, \text{CLAIR}) = 3$$

Les opérations nécessaires sont :

- **Substitution** : H → L (CHAT → CLAT)
- **Insertion** : I en avant-dernière position (CLAT → CLAIT)
- **Substitution** : T → R (CLAIT → CLAIR)

4.3 Code Python

```python
def levenshtein_distance(x, y):
    m = len(x)
    n = len(y)

    # Creer une matrice de dimensions (m+1) x (n+1)
    distance_matrix = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialiser la premiere colonne et la premiere ligne
    for i in range(m + 1):
        distance_matrix[i][0] = i
    for j in range(n + 1):
        distance_matrix[0][j] = j

    # Remplir la matrice
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if x[i - 1] == y[j - 1]:
                substitution_cost = 0
            else:
                substitution_cost = 1
            distance_matrix[i][j] = min(distance_matrix[i - 1][j] + 1,          # Suppression
                                        distance_matrix[i][j - 1] + 1,          # Insertion
                                        distance_matrix[i - 1][j - 1] + substitution_cost)  # Substitution

    # La distance de Levenshtein est dans la cellule en bas a droite de la matrice
    return distance_matrix[m][n]

# Exemple d'utilisation
x = "CHAT"
y = "CLAIR"

distance = levenshtein_distance(x, y)
print(f"La distance d'edition de Levenshtein entre '{x}' et '{y}' est de {distance}.")
```

Exercice 3 Modifiez le code precedent pour renvoyer la liste des operations a effectuer.
Modifiez le code precedent pour renvoyer la liste des mots intermediaires.

5 Exercices

Exercice 4 Exo becirsphacin carré chemins

6 Annexe : suite de Fibonacci

Un exemple classique pour introduire la programmation dynamique est le calcul des nombres de Fibonacci. Les nombres de Fibonacci sont définis comme suit :

$$u_0 = 0,\ u_1 = 1$$

$$\text{et}\ \ u_n = u_{n-1} + u_{n-2} \text{ pour } n \geqslant 2$$

6.1 Calcul explicite

Exercice 5 Expliciter le terme général de la suite $(u_n)_{n\in\mathbb{N}}$. Vous devriez trouver :

$$\forall n \in \mathbb{N}\ u_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

On en déduit :

```python
def fibonacci_formule(n):
    sqrt_5 = math.sqrt(5)
    phi = (1 + sqrt_5) / 2
    psi = (1 - sqrt_5) / 2
    u_n = (1 / sqrt_5) * (phi**n - psi**n)
    return (u_n)
```

En fait ce premier algorithme ne fonctionne pas : il commet des erreurs d'arrondi. Il vaut mieux calculer successivement les termes de la suite en restant dans les entiers (integer).

6.2 Stratégie naïve

Version itérative :

```python
def fibonacci_naif(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(n - 1):
        a, b = b, a + b
    return b

# Exemple d'utilisation
print(fibonacci_naif(10))  # Sortie : 55
```

Cet algorithme simple recalcule plusieurs fois les mêmes valeurs, ce qui serait inefficace pour de grandes valeurs de n. De même, l'algorithme récursif naïf pour calculer les nombres de Fibonacci est simple à implémenter mais inefficace pour des grandes valeurs de n car il recalculera plusieurs fois les mêmes sous-problèmes, entraînant une complexité exponentielle. Voici comment cet algorithme peut être écrit en Python :

```python
def fibonacci_naif_recursif(n):
    if n <= 1:
        return n
    return fibonacci_naif_recursif(n-1) + fibonacci_naif_recursif(n-2)

# Exemple d'utilisation
print(fibonacci_naif_recursif(10))  # Sortie : 55
```

6.3 Programmation dynamique

Avec la programmation dynamique, nous **stockons les résultats intermédiaires** pour éviter ces recalculs. On dit que l'on **mémoïse** les calculs déjà faits.

Voici un exemple de code Python utilisant la programmation dynamique pour calculer les nombres de Fibonacci :

```python
def fibonacci(n):
    # Creer un tableau pour stocker les valeurs de la suite
    fib = [0] * (n + 1)
    # Initialisez les valeurs de base
    fib[0] = 0
    fib[1] = 1
    # Calculer les valeurs de Fibonacci et les stocker dans le tableau
    for i in range(2, n + 1):
        fib[i] = fib[i-1] + fib[i-2]
    return fib[n]

# Exemple d'utilisation
print(fibonacci(10))  # Sortie : 55
```

Listing 3 – Calcul des nombres de Fibonacci

6.4 Récursivité et mémoïsation

Le nec plus ultra est un **algorithme récursif avec mémoïsation** pour calculer le n-ème terme de la suite de Fibonacci.

```python
def fibonacci_memo(n):
    memo = {}

    def fibonacci_aux(k):
        if k in memo:
            return memo[k]
        if k <= 1:
            return k
        memo[k] = fibonacci_aux(k-1) + fibonacci_aux(k-2)
        return memo[k]

    return fibonacci_aux(n)
```

Listing 4 – Calcul du n-ème terme de la suite de Fibonacci avec mémoïsation

En Python cela donne :

```python
def fibonacci_memo_sans_aux(n, memo=None):
    if n == 0:
        return 0
    if n==1:
        return 1
    if memo is None:
        memo = {}
    if n in memo:
        return memo[n]

    memo[n] = fibonacci_memo_sans_aux(n-1, memo) +
       fibonacci_memo_sans_aux(n-2, memo)
    return memo[n]
```

Listing 5 – Calcul du n-ème terme de la suite de Fibonacci avec mémoïsation

Remarque — « Mémoriser » et « mémoïser »

Les termes *mémoriser* et *mémoïser* ont des significations légèrement différentes dans le contexte de l'informatique et de l'algorithmique :

1. **Mémoriser (anglais *Memorize*)** :
   - **Signification générale :** Mémoriser signifie retenir ou stocker quelque chose en mémoire.
   - **Dans l'informatique :** Lorsqu'on parle de *"mémoriser"* dans le contexte informatique, cela fait souvent référence à stocker des données ou des résultats intermédiaires pour un accès ultérieur. Par exemple, mémoriser un résultat de calcul pour éviter de le recalculer plusieurs fois.
2. **Mémoïser (anglais *Memoize*)** :
   - **Signification spécifique :** Mémoïser est un terme technique qui désigne une technique spécifique en programmation pour optimiser les performances d'algorithmes récursifs en stockant les résultats de sous-problèmes déjà résolus.
   - **Dans l'informatique :** Lorsqu'on parle de *"mémoïser"*, on fait référence à l'utilisation d'une technique de programmation dynamique où les résultats des calculs récursifs sont stockés dans une structure de données (comme un dictionnaire) afin d'éviter de les recalculer à chaque fois qu'ils sont nécessaires. Cela permet d'améliorer considérablement l'efficacité et la vitesse d'exécution de l'algorithme.

En résumé, *"mémoriser"* est un terme plus général qui implique simplement le stockage en mémoire, tandis que *"mémoïser"* est spécifique à une technique d'optimisation algorithmique utilisée pour réduire la complexité temporelle des algorithmes récursifs en évitant les recalculs.

7 Solutions

Solution

Solution 4.3

```python

def levenshtein_distance_with_traceback(x, y):
    m = len(x)
    n = len(y)

    # Creer une matrice de dimensions (m+1) x (n+1)
    distance_matrix = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialiser la premiere colonne et la premiere ligne
    for i in range(m + 1):
```

```
        distance_matrix[i][0] = i
    for j in range(n + 1):
        distance_matrix[0][j] = j

    # Remplir la matrice
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if x[i - 1] == y[j - 1]:
                substitution_cost = 0
            else:
                substitution_cost = 1
            distance_matrix[i][j] = min(distance_matrix[i - 1][j] + 1,          #
                Suppression
                                        distance_matrix[i][j - 1] + 1,          # Insertion
                                        distance_matrix[i - 1][j - 1] +
                                            substitution_cost)  # Substitution

    # Tracage des operations
    i, j = m, n
    operations = []
    while i > 0 or j > 0:
        current_cost = distance_matrix[i][j]
        if i > 0 and distance_matrix[i - 1][j] + 1 == current_cost:
            operations.append(f"Supprimer '{x[i - 1]}' a la position {i - 1}")
            i -= 1
        elif j > 0 and distance_matrix[i][j - 1] + 1 == current_cost:
            operations.append(f"Inserer '{y[j - 1]}' a la position {i}")
            j -= 1
        else:
            if x[i - 1] != y[j - 1]:
                operations.append(f"Remplacer '{x[i - 1]}' par '{y[j - 1]}' a la position
                    {i - 1}")
            i -= 1
            j -= 1

    operations.reverse()

    # Appliquer les operations pour montrer les mots intermediaires
    intermediate_words = [x]
    current_word = list(x)
    for operation in operations:
        if "Supprimer" in operation:
            pos = int(operation.split()[-1])
            del current_word[pos]
        elif "Inserer" in operation:
            char = operation.split("'")[1]
            pos = int(operation.split()[-1])
            current_word.insert(pos, char)
        elif "Remplacer" in operation:
            pos = int(operation.split()[-1])
            char = operation.split("'")[3]
            current_word[pos] = char
        intermediate_words.append("".join(current_word))

    return distance_matrix[m][n], operations, intermediate_words

# Exemple d'utilisation
x = "CHAT"
y = "CLAIR"

distance, operations, intermediate_words = levenshtein_distance_with_traceback(x, y)
print(f"La distance d'edition de Levenshtein entre '{x}' et '{y}' est de {distance}.")
print("Suite des operations:")
for operation in operations:
    print(operation)
print("Mots intermediaires:")
for word in intermediate_words:
    print(word)

def levenshtein_distance_with_intermediate_words(x, y):
    m = len(x)
    n = len(y)

    # Creer une matrice de dimensions (m+1) x (n+1)
    distance_matrix = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialiser la premiere colonne et la premiere ligne
```

```
    for i in range(m + 1):
        distance_matrix[i][0] = i
    for j in range(n + 1):
        distance_matrix[0][j] = j

    # Remplir la matrice
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if x[i - 1] == y[j - 1]:
                substitution_cost = 0
            else:
                substitution_cost = 1
            distance_matrix[i][j] = min(distance_matrix[i - 1][j] + 1,          #
                Suppression
                                        distance_matrix[i][j - 1] + 1,          # Insertion
                                        distance_matrix[i - 1][j - 1] +
                                            substitution_cost)  # Substitution

    # Tracage des operations et mots intermediaires
    i, j = m, n
    operations = []
    while i > 0 or j > 0:
        current_cost = distance_matrix[i][j]
        if i > 0 and distance_matrix[i - 1][j] + 1 == current_cost:
            operations.append(('delete', i - 1, x[i - 1]))
            i -= 1
        elif j > 0 and distance_matrix[i][j - 1] + 1 == current_cost:
            operations.append(('insert', i, y[j - 1]))
            j -= 1
        else:
            if x[i - 1] != y[j - 1]:
                operations.append(('replace', i - 1, y[j - 1]))
            i -= 1
            j -= 1

    operations.reverse()

    # Appliquer les operations pour montrer les mots intermediaires
    current_word = list(x)
    intermediate_words = [x]
    for operation in operations:
        if operation[0] == 'delete':
            _, pos, _ = operation
            del current_word[pos]
        elif operation[0] == 'insert':
            _, pos, char = operation
            current_word.insert(pos, char)
        elif operation[0] == 'replace':
            _, pos, char = operation
            current_word[pos] = char
        intermediate_words.append("".join(current_word))

    return intermediate_words

# Exemple d'utilisation
x = "CHAT"
y = "CLAIR"

intermediate_words = levenshtein_distance_with_intermediate_words(x, y)
print("Mots intermediaires:")
for word in intermediate_words:
    print(word)
```

Solution

Solution 6.1 Pour calculer explicitement le terme général de la suite de Fibonacci, repérons qu'il sagit d'une suite récurrente linéaire d'ordre deux.
Son équation quadratique est

$$r^2 - r - 1 = 0$$

nous obtenons deux racines distinctes :

$$r_1 = \frac{1+\sqrt{5}}{2} \text{ et } r_2 = \frac{1-\sqrt{5}}{2}$$

La solution générale de la relation de récurrence est une combinaison linéaire des solutions de la forme r_1^n et r_2^n :

$$u_n = Ar_1^n + Br_2^n$$

Nous utilisons les conditions initiales pour déterminer les coefficients A et B.
1. Pour $n = 0$:

$$u_0 = Ar_1^0 + Br_2^0 = A + B = 0$$

2. Pour $n = 1$:

$$u_1 = Ar_1^1 + Br_2^1 = Ar_1 + Br_2 = 1$$

Nous obtenons le système d'équations suivant :

$$A + B = 0$$
$$Ar_1 + Br_2 = 1$$

En résolvant ce système, nous trouvons :

$$A = \frac{1}{\sqrt{5}} \text{ et } B = -\frac{1}{\sqrt{5}}$$

Ainsi, la formule explicite pour u_n est :

$$u_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$