Programmation dynamique

Capacités exigibles

→ Programmation dynamique. Propriété de sous-structure optimale. Chevauchement de sous-problèmes.

→ Calcul de bas en haut ou par mémoïsation. Reconstruction d'une solution optimale à partir de l'information calculée.

1. Introduction

1.1. Idée générale

Commençons par expliquer ce que les termes de *programmation dynamique* veulent dire. Le premier ne signifie pas qu'on va "programmer" dans le sens de "coder" car c'est bien ce qui sera fait. Ce terme vient des années 50 lorsque cette technique algorithmique est apparue. Il s'agit de planifier/organiser le calcul (la suite du cours explique cela). Le terme "dynamique" fait référence à l'idée qu'une certaine information, non disponible au début, sera construite au fur et à mesure du déroulement de l'algorithme pour être utilisée plus tard. A ce stade, tout cela reste encore obscur. Tout devrait apparaître plus clairement lorsque vous serez arrivé la fin de ce cours.
La programmation dynamique est une technique algorithmique qui s'applique principalement à des problèmes d'optimisation. Parmi les solutions possibles, on cherche une des solutions optimales. L'optimalité consistera à trouver le minimum ou le maximum d'une fonction de coût qu'il faudra établir en fonction du problème. Comme pour la méthode "diviser pour régner", il s'agira de combiner les solutions de sous-problèmes de même nature, ce qui conduira souvent à une solution récursive. La plupart du temps ces sous-problèmes se chevauchent (ils possèdent eux aussi des problèmes en commun), ce qui peut astucieusement être exploité pour améliorer le temps de calcul.

1.2. Caractéristiques principales de la programmation dynamique

La programmation s'applique à des problèmes présentant des caractéristiques identifiables. Nous prendrons l'exemple de la fonction `Fibo()` permettant de calculer le n-ième terme de la suite de Fibonacci.

1.2.1. Version descendante

Une des manières de traiter ce petit problème est d'imaginer un version dite *top-bottom*. On cherche une fonction récursive pour le résoudre. Ici, la définition de cette suite récurrente la donne tout de suite. On rappelle que la suite est définie par le terme général $u_n = u_{n-1} + u_{n-2}$ avec $u_0 = 1$ et $u_1 = 1$.

```
def fibo(n):
    if n==0 or n==1:
        return 1
    else:
        return fibo(n-1) + fibo(n-2)
```

FIGURE 1 – Code source permettant de calculer le terme de rang n de la suite de Fibonacci

La solution d'un problème de programmation dynamique est une combinaison des solutions de sous-problèmes de même nature.

Ici c'est évident. Calculer le terme de rang n implique de calculer les deux termes de rang $n-1$ et $n-2$. On retrouvera cette idée de manière plus utile dans les exemples traités plus loin dans ce document car ici ce n'est rien de plus que ce qu'on trouve avec la recherche d'algorithmes récursifs.
Le code fourni à la figure 1 s'avère être peu efficace pour des valeurs de n même petites. A la figure 2 sont présentés les temps de calcul de différentes valeurs de la suite (bien évidemment ces temps dépendent de la puissance de la machine mais les ordres de grandeur resteront les mêmes).

rang du terme	n=10	n=15	n=20	n=25	n=30	n=35	n=40
temps de calcul	82µs	830µs	9ms	41ms	235ms	2,5s	322s

FIGURE 2 – Temps de calcul mis par la fonction `fibo()` pour calculer les termes de rang n

On constate sans difficulté que le temps croit à priori exponentiellement avec la valeur de n. La raison en est simple si on regarde l'arbre des différents appels récursifs de la figure 3.

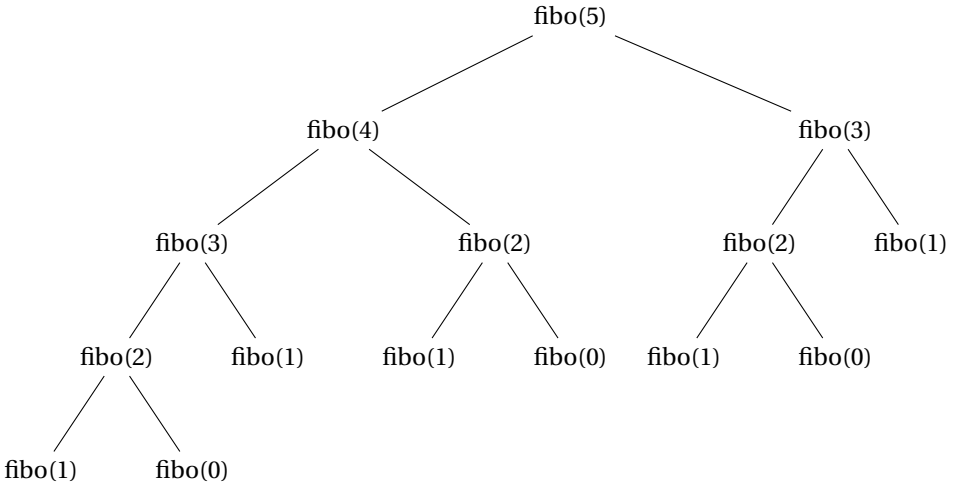


FIGURE 3 – Arbre des appels de la fonction récursive `fibo()` pour la valeur $n = 5$.

On remarque que la fonction est appelée 2 fois pour le rang 3 et 3 fois pour le rang 2. Si le rang du terme de départ était 6, alors on double ce nombre d'appels car on ajoute une branche supplémentaire quasi-identique à l'arbre présenté.

La solution d'un problème de programmation dynamique est issue de calculs de sous-problèmes identiques. On dit qu'il y a **chevauchement**.

Ces calculs sont bien évidemment inutiles et il serait plus intéressant de les garder en mémoire puis de les réutiliser quand c'est nécessaire. Pour la complexité, on peut écrire que $C(n) = C(n-1) + C(n-2)$. La suite (C_n) est une suite récurrente linéaire d'ordre 2. Mathématiquement, on montre que le terme de rang n peut se calculer par la relation $C(n) = k.r^n$ avec $k \in \mathbb{R}_+^*$ et $r > 1$. La complexité est donc exponentielle, ce qui interdit son utilisation pratique.

Solution avec la programmation dynamique par mémoïsation
L'idée est simple : puisqu'on calcule plusieurs fois la même chose, il serait utile de mémoriser le résultat afin de le réutiliser directement. Si on ne calcule plus qu'une fois les termes, l'arbre d'appels de la fonction ne contient plus que sa partie gauche.

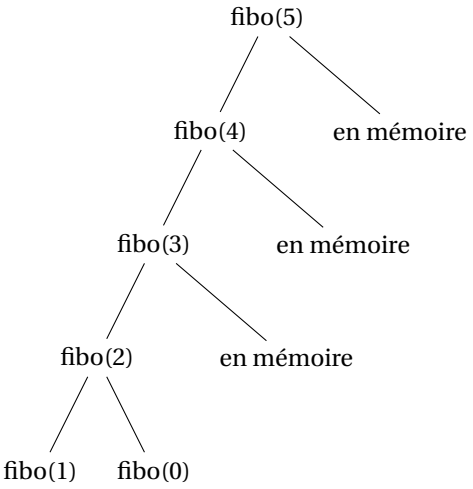


FIGURE 4 – Arbre des appels de la fonction `fibo()` avec mémoïsation et la valeur 5.

Les appels récursifs sont exécutés dans l'ordre décroissant des rangs et sont tous gardés en mémoire. La complexité chute drastiquement pour devenir linéaire (n termes à calculer donc $C(n) = \mathcal{O}(n)$).
Quel que soit le problème, on utilise une structure de données pour *mémoïser* les résultats. Naturellement on penserait plutôt au terme *mémoriser* (ce qui sera fait techniquement) mais en informatique on utilise plutôt *mémoïser* qui est un néologisme venant du mot *mémo*, indiquant qu'on garde de côté une information qu'il faudra utiliser plus tard.
Pour notre exemple, nous utiliserons une liste pour mémoriser les différents termes de la suite. Une solution est donnée figure 5.

```
def fibo_dyn(n):
    coeffs=[-1]*(n+1)
    coeffs[0]=1
    coeffs[1]=1
    def fibo_aux(n):
        if n==0 or n==1:
            return coeffs[n]
        else:
            if coeffs[n]==-1:
                coeffs[n]=fibo_aux(n-1)+fibo_aux(n-2)
                return coeffs[n]
            else:
                return coeffs[n]
    return fibo_aux(n)
```

FIGURE 5 – Code source de la fonction `fibo_dyn()`.

Commentaires du code de la figure 5

- `coeffs` : liste stockant les termes de la suite de Fibonacci. Par convention, on affecte la valeur -1 à chaque élément de la liste, ce qui veut dire qu'il n'a pas encore été calculé ;
- `coeffs[0]=1` et `coeffs[1]=1` : initialisation des termes de la suite avec les valeurs classiques ;
- `def fibo_aux(n):` : définition d'une fonction récursive interne pour cacher l'utilisation d'une liste commune aux différentes occurrences de la fonction `fibo_aux()` appelée récursivement. Sans cela il faudrait la passer en paramètre, ce qui alourdirait l'ensemble ;
- `return fibo_aux(n)` : appel de la fonction interne.

Le morceau de code où tout se fait est donné figure 6. Lors d'un appel récursif, on vérifie d'abord si le terme de rang n a été calculé. Si ce n'est pas le cas, alors on le calcule puis on l'affecte à l'élément n de la liste `coeffs`. Une fois ceci fait, on n'aura plus besoin de le calculer car pour les appels suivants le test `coeffs[n]==-1` de l'alternative sera forcément faux. Ainsi la fonction retournera directement la valeur contenue dans `coeffs[n]`.

```
if coeffs[n]==-1:     # terme de rang n non calculé
    coeffs[n]=fibo_aux(n-1)+fibo_aux(n-2) # on le calcule et on le garde pour la
        prochaine fois.
    return coeffs[n] # puis on le retourne
else:                # terme déjà calculé, donc on le retourne
    return coeffs[n]
```

FIGURE 6 – Code source montrant le cœur de la fonction `fibo_dyn()`.

Il faut énoncer aussi un autre principe de la programmation dynamique qui n'apparaît pas ici car le problème ne présente pas de choix dans le calcul de la solution. En effet, tout est forcé puisque pour un terme de rang donné c'est obligatoirement la somme des deux précédents.

Principe d'optimalité de Bellman : Toute solution optimale est une combinaison des solutions optimales de ses sous-problèmes. On dit que le problème possède une **sous-structure optimale**.

Cela veut dire que pour appliquer la programmation dynamique, il faut que le problème possède la propriété de sous-structure optimale. Pour le reformuler autrement, toute solution optimale possède en elle les solutions optimales de ses sous-problèmes. Pour le montrer, soit on procède en construisant une preuve soit on trouve une relation de récurrence mettant en évidence l'optimum choisi, ce qui, de facto, montre que le problème possède cette propriété.

Exemple de preuve avec le chemin le plus court dans un graphe non orienté et non pondéré
Soit un graphe quelconque de n sommets (avec $n \geqslant 2$ sinon c'est trivial). On suppose disposer du plus court chemin (en nombre d'arêtes) entre 2 sommets distincts D et F (un chemin est une séquence d'arêtes consécutives). Soit un sommet S quelconque par lequel passe le chemin. Alors le chemin entre S et D ainsi qu'entre S et F est optimal (principe d'optimalité).
Preuve par l'absurde : on suppose qu'il existe un autre chemin entre le sommet D et S de longueur différente. Si la longueur est plus petite, alors l'hypothèse est fausse et le chemin entre D et F n'est pas optimal. Si la longueur est plus grande, alors le nouveau chemin entre D et F ne peut être optimal puisqu'il est plus grand que celui de l'hypothèse départ. Conclusion : pour tout sommet S appartenant au chemin optimal, les chemins entre D et S et entre S et F sont optimaux. Ainsi pour trouver ce chemin optimal on pourra baser une solution sur l'application de la programmation dynamique.

1.2.2. Version ascendante

Le calcul de la solution peut être envisagé en partant directement des sous-problèmes triviaux puis de les combiner pour arriver à la solution globale. C'est une approche *Bottom-Up* qui aboutit à une version itérative connue (voir figure 7).

```
def fiboIter(n):
    L=[1]*(n+1)
    for i in range(2,n+1):
        L[i]=L[i-1]+L[i-2]
    return L[-1]
```

FIGURE 7 – Code source de la fonction `FiboIter()`.

Dans ce cas particulier, on aurait pu éviter l'utilisation d'une liste pour stocker les différentes valeurs des termes de rang intermédiaire en utilisant uniquement un jeu de deux variables. L'idée est uniquement de montrer que généralement on s'aidera d'une structure de données pour calculer les solutions optimales en partant des cas simples. Une illustration est donnée dans les exemples qui suivent.

2. Traitement de deux exemples classiques

2.1. Exemple 1 : Pyramide d'entiers

2.1.1. Présentation

```
    7           niveau 0
   2 1          niveau 1
  1 1 3         niveau 2
 3 2 2 8    niveau 3
5 1 6 2 1       niveau 4
```

FIGURE 8 – Représentation d'une pyramide de nombres

Soit une pyramide d'entiers comme représentée à la figure 8. Les niveaux de nombres sont numérotés à partir du sommet. On cherche le chemin dont la somme des éléments est maximale en partant du sommet (dont la valeur est 7) à l'un quelconque des éléments de la base (5,1,6,2 ou 1) en passant par un seul élément de chaque niveau. On peut aussi considérer le cas où on part de la base, ce qui reviendra au même. Chaque élément d'un niveau quelconque est connecté à exactement deux éléments du niveau suivant directement accessible. Par exemple, la valeur 7 du niveau 0 est connectée aux valeurs 2 et 1. La valeur 3 du niveau 2 est connectée aux deux valeurs 2 et 8 et la valeur 8 du niveau 3 est connectée aux valeurs 2 et 1 du niveau suivant.
La première solution, dite naïve, est de calculer toutes les combinaisons possibles de chemin. Si la pyramide est de hauteur h, on a alors $h-1$ niveaux à traverser et 2 choix à faire à chaque fois. Le nombre de combinaisons est alors de 2^{h-1}, ce qui donne une complexité exponentielle en $\mathcal{O}(2^h)$. Pour de petites valeurs de h, cette solution est possible mais il faut trouver mieux.
Une autre solution est d'utiliser une stratégie gloutonne, on fait le meilleur choix local en espérant qu'on atteindra un optimum global. Comme l'algorithme effectue $h-1$ choix, la complexité sera de $\mathcal{O}(h)$ on aura :

— en partant du haut la suite de valeurs : 7,2,1,3,5 ce qui donne comme somme totale 18 ;
— en partant du bas la suite de valeurs seraient : 6,2,3,1,7 soit une somme totale de 19 , ce qui est mieux mais n'est pas la valeur optimale.

La valeur optimale est obtenue par la suite de valeurs 7,1,3,8,2 soit une somme totale de 21. Dans les deux cas, commencer par le haut ou par le bas n'a pas permis de trouver la meilleure solution. Cela ne veut pas dire que ce ne sera jamais le cas, tout dépend des données du problème. Avec la pyramide légèrement différente de la figure 9 , ce type de stratégie fonctionne mais ce n'est pas généralisable. La somme optimale se trouve avec la séquence 6,8,3,1 et 7 soit 25.

```
    7
   2 1
  1 1 3
 3 2 8 2    -> inversion du 8 et du 2
5 1 6 2 1
```

FIGURE 9 – Pyramide de nombres légèrement modifiée

2.1.2. Solution descendante

Le calcul du meilleur chemin se fera en trouvant une récurrence. Tout d'abord, on choisit une liste de listes comme structure pour implémenter une pyramide d'entiers. On a ainsi le code de la figure 10.

```
P=[
    [7],          #-> correspond au niveau 0 de la pyramide
    [2,1],
    [1,1,3],
    [3,2,8,2],
    [5,1,6,2,1]
]
```

FIGURE 10 – Code source de la structure de données représentant la pyramide

Remarque : il faut noter qu'un élément d'indice (i,j) est connecté à deux autres d'indice $(i+1,j)$ et $(i+1,j+1)$.

On définit :

— les éléments de la pyramide notés e_{ij}, avec i correspondant à la ligne et j la colonne ;

— la fonction v telle que $v(e_{ij})$ retourne la valeur associée à l'élément e_{ij}. Par exemple, $v(e(3,2))$ vaut 8.

La récurrence n'est pas très difficile à trouver :

— de manière générale, la somme optimale (notée $sopt$) à partir d'un élément d'indice (i,j) est $sopt(e_{ij}) = v(e_{ij}) + max(sopt(i+1,j),sopt(i+1,j+1))$;

— un élément de la dernière ligne est un cas particulier puisque dans ce cas il n'y a plus d'éléments ensuite. On a alors $sopt(e_{ij}) = v(e_{ij})$.

La solution proposée indique que dans le cas général, pour chaque appel il y aura deux autres appels. Ce qui donne : $C(n) = 1+2^1+2^2+2^3...+2^{n-1} = \mathcal{O}(2^n)$, soit une complexité exponentielle. Encore une fois, l'algorithme n'est pas utilisable en pratique. On remarque simplement que, comme pour le premier exemple sur le calcul d'un terme de la suite de Fibonacci, il y a chevauchement des problèmes. L'idée est donc de mémoïser les résultats intermédiaires pour ne pas avoir à les recalculer. Nous utiliserons la même structure que la pyramide. Ainsi, l'élément d'indice (i,j) de la nouvelle structure contiendra la somme optimale depuis un élément de la base jusqu'à l'élément d'indice (i,j). L'implémentation est donnée à la figure 11.

```
def sommeOptimale(P):
    # S est une structure permettant de mémoriser les résultats intermédiaires.
    # Comme il sera nécessaire de savoir si la somme optimale a été calculée,
    # on l'initialise avec l'élément None.
    S=[ [None]*(i+1) for i in range(len(P)) ]

    #fonction récursive interne à la fonction sommeOptimale()
    def sopt(i,j):
        # on traite d'abord tous les cas particuliers de la fonction récursive
        if S[i][j]!=None:       # la somme optimale a été calculée
                        return S[i][j]  #-> on retourne directement la valeur

        # autre cas particulier : dernière ligne de la pyramide.
        elif i==len(P)-1:
            return P[i][j]

        # puis vient le cas général
        else:
            # on sait que la somme optimale n'a pas été calculée, donc on
                procède au calcul
            # qu'on affecte à la structure avant de retourner.
            S[i][j] = P[i][j]+max(sopt(i+1,j),sopt(i+1,j+1))
            # puis on retourne le résultat.
            return S[i][j]

    return sopt(0,0) #appel de la fonction sopt() en démarrant au sommet de la
        pyramide.
```

FIGURE 11 – Code source pour calculer une somme optimale

Une solution légèrement différente utilise une structure de dictionnaire pour mémoïser les résultats (voir figure 12). Il y a très peu de différences par rapport au code précédent, le fonctionnement n'est pas plus compliqué.

```
def sommeOptimale_v2(P):

    S={} # déclaration du dictionnaire

    def sopt(i,j):
        #on traite tous les cas particuliers
        if (i,j) in S.keys(): # vérification de la présence de la clé (i,j)
            return S[(i,j)]

        elif i==len(P)-1: #dernière ligne
            return P[i][j]

        else:
            S[(i,j)] = P[i][j]+max(sopt(i+1,j),sopt(i+1,j+1))
            return S[(i,j)]

    return sopt(0,0)
```

FIGURE 12 – Code source pour le calcul d'une somme optimale avec dictionnaire

2.1.3. Version ascendante

Comme pour l'exemple avec la suite de Fibonacci, la version ascendante va calculer les meilleurs chemins en partant des cas de base puis calculer les sommes optimales intermédiaires. Pour les garder, on utilisera le même type de structure que la pyramide initiale, soit une liste de listes. Le code source de la solution est donné figure 13.

```
from copy import deepcopy

def sommeOptimale_v3(L):
    sommes=deepcopy(L)
    n=len(L)-2
    for i in range(n,-1,-1):
        for j in range(i+1):
            sommes[i][j]+=max(sommes[i+1][j],sommes[i+1][j+1])
    return sommes[0][0]
```

FIGURE 13 – Code source de la version ascendante de la solution pour la pyramide de nombres

Commentaires du code de la figure 13

- `from copy import deepcopy` : on recopie exactement la même structure de données, ce qui permettra d'avoir les mêmes valeurs de départ. Pour des structures imbriquées, la fonction `deepcopy()` permet de réaliser une copie complète en dupliquant tous les niveaux. On rappelle que l'instruction `L2=L1` n'est une copie de liste, ni l'instruction L2=L1.copy() qui ne copie "proprement" que le premier niveau ;
- `n=len(L)-2` : le niveau 0 étant le sommet de la pyramide, on se place sur l'avant dernier niveau ;
- `for i in range(n,-1,-1)` : parcours des niveaux dans le sens décroissant à partir de n ;
- `for j in range(i+1)` : remplissage complet du niveau n ;
- `sommes[i][j]+=max(sommes[i+1][j],sommes[i+1][j+1])` : calcul de la somme optimale pour chaque élément.
- `return sommes[0][0]` : on retourne le dernier élément de la structure, qui est la somme optimale.

2.2. Exemple 2 : Découpe de barres

Le problème de la découpe de barres est un problème classique d'optimisation pour l'application de la programmation dynamique.

2.2.1. Présentation

Une entreprise achète des barres d'acier et les découpe pour les revendre ensuite. Les coupes peuvent se faire à différentes longueurs entières avec des prix dépendants d'un marché comme donné à la figure 14.

longueur l	1	2	3	4	5	6	7	8	9	10
prix p_l	1	5	8	9	10	17	17	20	24	30

FIGURE 14 – Tableau des prix en fonction de la longueur de la barre

L'entreprise souhaite optimiser la façon de couper les barres, c'est à dire optimiser son revenu.

2.3. Analyse du problème

Pour visualiser un peu mieux le problème, on considère le cas où la taille de la barre de départ vaut 4 (sans dimension). La figure 15 permet de voir toutes les manières de la découper.

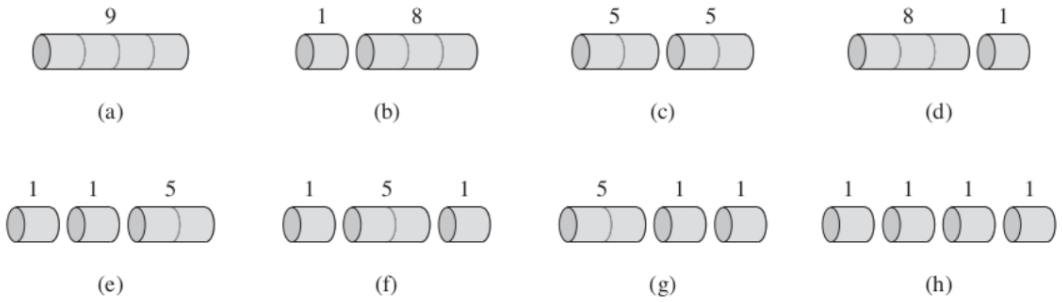


FIGURE 15 – Différentes manières de découper une barre de taille 4 au départ (le revenu de chaque morceau est indiqué au dessus).

De manière générale, on peut découper une barre de longueur n de 2^{n-1} façons différentes. La manière naïve de résoudre ce problème est donc de calculer toutes les combinaisons possibles et de calculer le prix de revient de chacune en ne gardant que le maximum. Dans notre exemple, avec $n = 4$ le revenu optimal se trouve avec la combinaison où on coupe la barre en deux morceaux de longueur 2 chacun. Un peu plus formellement, une barre de longueur n peut être découpée en k morceaux. On a alors $n = l_1 + l_2 + ... + l_k$ et le revenu associé est $r_n = p_{l_1} + p_{l_2} + ... + p_{l_1}$. En dressant une analyse de cas pour différentes longueurs de départ, on obtient les revenus optimaux suivants :

- $n = 1$: pas de choix possible, le revenu optimal est de 1 car pas de coupes ;
- $n = 2$: on peut ne pas couper et dans ce cas le revenu est de 5, ou bien couper la barre en deux ce qui donne deux morceaux de longueur 1. Le revenu est de 1+1=2. Le revenu optimal est donc de 5 ;
- $n = 3$: plusieurs combinaisons de coupes. Par exemple, 1+1+1=3, 2+1=3 ou pas de coupes par exemple. Le revenu optimal sera de 8 (pas de coupe) ;
- $n = 4$: vu précédemment, le revenu optimal est de 10.
- etc.

De cette analyse on peut constater qu'il est impossible de poser le choix le plus intéressant au départ pour une solution optimale. Par exemple, dans le cas où $n = 3$, on ne peut pas savoir si le revenu optimal sera obtenu avec une découpe commençant à 0 ou à 1. Il sera nécessaire de considérer tous les cas et ne retenir que la solution optimale. On arrive alors à la relation ci-dessous pour une barre de longueur n au départ :

$$r_n = max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_1)$$

Commentaires

- le revenu optimal peut être obtenu en ne découpant pas la barre. C'est pour cela que le terme p_n apparaît comme argument de la fonction $max()$;
- comme montré plus haut, il est nécessaire de considérer toutes les possibilités. Pour une barre de longueur k dont le prix de revient est r_k, on cherchera le revenu optimal r_{n-k} pour le reste, c'est à dire une barre de longueur $n-k$. D'où la présence de $r_1 + r_{n-1}$, de $r_2 + r_{n-2}$, etc, comme arguments de la fonction $max()$.

Principe de Bellman

Toute solution optimale est une combinaison des solutions optimales des sous-problèmes. La relation de récurrence trouvée plus haut permet d'affirmer ici que ce problème utilise le principe de Bellman. En effet, la meilleure solution retenue est forcément la combinaison d'une découpe initiale avec la solution optimale de ce qui reste. La solution est fournie dans le code de la figure 16.

```
p=[0,1,5,8,9,10,17,17,20,24,30]

def couperBarre(p,n):
        if n==0:
                return 0
        else:
                res=0
                for i in range(1,n+1): #pour toutes les longueurs
                        res=max(0,p[i]+couperBarre(p,n-i)
                return res

print(couperBarre(p,10))
```

FIGURE 16 – Code source de la fonction `couperBarre()`

Malheureusement, cette solution n'est pas utilisable pratiquement. Le nombre de découpes possibles étant 2^{n-1} pour une barre de longueur n, cela indique immédiatement une complexité exponentielle.

Application du principe de la programmation dynamique
La solution proposée pâtit du même problème que la fonction fibo() de la figure 1 : il y a chevauchement des sous-problèmes. Par exemple, avec une barre de longueur 10, il faudra rechercher le revenu optimum pour une longueur de 9, 8, 7, etc. Pour chacun de ces problèmes, il y aura à regarder le revenu optimum pour les longueurs 3,2,1, et ceci pour beaucoup de cas car pour une longueur de 7, l'optimum pour 3 sera évaluée mais aussi dans le cas ou on prend 4 comme découpe initiale. Nous allons rendre plus efficace cet algorithme en utilisant la technique de la mémoïsation. Il serait plus intéressant de garder la solution en mémoire lorsqu'elle a été calculée plutôt que de la recalculer. L'algorithme doit donc vérifier que la solution n'a pas déjà été calculée avant de lancer le calcul. Pour cela, on utilise une liste permettant de stocker les optimums une fois calculés. Le code de la solution est proposé en figure 17

```
p=[0,1,5,8,9,10,17,17,20,24,30] #tableau des prix

compteur=0

def couperBarreMemo(p,n):
    r=[-1]*(n+1) #

    def couperBarreAux(n): # fonction récursive interne
        global compteur
        compteur+=1
        if n==0:
            return 0
        else:
            if r[n]!=-1:
                return r[n]
            res=0
            for i in range(1,n+1):
                res=max(res,p[i]+couperBarreAux(n-i))
            r[n]=res
            return res
    return couperBarreAux(n)

print(couperBarreMemo(p,10)) #exemple d'appel
```

FIGURE 17 – Code source de la fonction couperBarreMemo()

Commentaires de la solution

- `compteur=0` : variable globale pour compter le nombre d'appels récursifs. Inutile pour la correction de l'algorithme bien évidemment ;
- `r=[-1]*(n+1)` : liste des revenus calculés. Initialisés avec -1. Cette liste sera utilisable depuis la fonction `couperBarreAux()` directement. Elle n'est donc pas passée en paramètre ;
- `if r[n]!=-1` : on vérifie que le revenu optimum pour la longueur n a été calculé. Si c'est le cas, alors on retourne la valeur associée ;

— `for i in range(1,n+1)` : si on arrive à ce point, c'est que le revenu correspondant n'a pas été calculé. La boucle va permettre de lancer le calcul sur toutes les longueurs restantes. une fois qu'il a été calculé, on l'affecte à l'élément n de la liste. Il reservira immédiatement pour tous les autres cas et permettra de diminuer de beaucoup la complexité ;

— `return couperBarreAux(n)` : appel de la fonction auxiliaire interne. Ce procédé a permis de cacher complètement la liste interne `r` de la fonction `couperBarreMemo()` ;

Le résultat est sans appel. Pour une barre de longueur $n = 10$, la fonction `couperBarre()` a été appelée 1024 fois (complexité exponentielle). Pour une même longueur, la fonction `couperBarreMemo()` a été appelée 56 fois.

3. Synthèse

La programmation dynamique est une technique algorithmique très puissante. Elle permet de résoudre des problèmes d'optimisation possédant les propriétés suivantes :

→ la solution est une combinaison de sous-problèmes de même nature ;

→ il y a chevauchement entre sous-problèmes, c'est à dire qu'ils possèdent des sous-sous-problèmes identiques.

Le gain en complexité se fait en utilisant une structure de données dédiée, la complexité en temps est améliorée avec une légère dégradation de la complexité en espace (à moduler en fonction de la nature des problèmes).
Deux méthodes peuvent être utilisées, une version descendante récursive, et une méthode ascendante itérative. L'utilisation de l'une ou l'autre dépend de la nature du problème.

4. Exercices

4.1. Distance d'édition

Les séquences de caractères peuvent encoder de nombreuses informations de nature différente, par exemple du texte, de la voix ou des séquences ADN. L'alignement de deux chaînes des caractères consiste à comparer deux séquences de caractères afin d'évaluer la similarité entre les deux.

La distance d'édition ou distance de Levenshtein est une mesure de la similarité entre deux chaînes de caractères. Cette distance est le nombre minimal d'opérations élémentaires à effectuer pour transformer la première chaine en la seconde, c'est-à-dire le nombre de caractères qu'il faut supprimer, insérer ou substituer pour passer d'une chaîne à l'autre.

Cette distance est majorée par la longueur de la plus grande chaine. C'est une distance au sens mathématique du terme, donc elle vérifie les propriétés :

- $\text{distance}(\text{ch}1,\text{ch}2) \geqslant 0$;
- $\text{distance}(\text{ch}1,\text{ch}2) = 0 \Leftrightarrow \text{ch}1 = \text{ch}2$;
- $\text{distance}(\text{ch}1,\text{ch}2) = \text{distance}(\text{ch}2,\text{ch}1)$;
- $\text{distance}(\text{ch}1,\text{ch}2) + \text{distance}(\text{ch}2,\text{ch}3) \leqslant \text{distance}(\text{ch}1,\text{ch}3)$.

On suppose que supprimer un caractère, insérer un caractère, substituer un caractère sont des opérations qui ont toute un coût unitaire. Si le caractère est identique, la substitution ne coûte rien.

On peut ainsi écrire :

$$d_e(ch1, ch2) = \begin{cases} \text{len(ch2)} & \text{si len(ch1)=0} \\ \text{len(ch1)} & \text{si len(ch2)=0} \\ d_e(ch1[1:], ch2[1:]) & \text{si ch1[0]=ch2[0]} \\ 1 + \min \begin{cases} d_e(ch1[1:], ch2) \\ d_e(ch1, ch2[1:]) \\ d_e(ch1[1:], ch2[1:]) \end{cases} & \text{sinon} \end{cases}$$

1. La distance d'édition de "chien" à "niche" vaut 4. Expliquer pourquoi.

2. Écrire une fonction récursive d'après l'algorithme défini précédemment `de_rec(ch1,ch2)`.

3. Pourquoi la programmation dynamique est pertinente ici?

4. Écrire une fonction récursive `de_mem(ch1,ch2,dico)` avec mémoïsation.

On souhaite utiliser la programmation dynamique de bas en haut à l'aide d'un tableau. On traduit l'algorithme précédent en exprimant le résultat d'une case du tableau en fonction de celles dont elle dépend dans le schéma dynamique.

$$T[i,j] = \begin{cases} j & \text{si i=0} \\ i & \text{si j=0} \\ T[i-1,j-1] & \text{si ch1[i]=ch2[j]} \\ 1 + \min \begin{cases} T[i-1,j] \\ T[i,j-1] \\ T[i-1,j-1] \end{cases} & \text{sinon} \end{cases}$$

5. Compléter à la main le tableau associé à la distance d'édition de "chien" à "niche". En déduire la distance d'édition.

j	0	1(n)	2(i)	3(c)	4(h)	5(e)
0						
1(c)						
2(h)						
3(i)						
4(e)						
5(n)						

6. Écrire une fonction `de_bas_haut(ch1,ch2)` qui calcule la distance d'édition de deux chaînes de caractères par programmation dynamique de bas en haut. On pourra tester sur "chien" et "niche", "AGTTC" et "AGCTC", "physique" et "informatique".

4.2. Plus longue séquence commune

La distance d'édition permet de mesurer le degré de similarité de deux chaînes. Elle ne donne pas d'information quant aux séquences maximales communes aux deux chaînes. Or, en génétique par exemple, il peut s'avérer très important de savoir quels sont les points communs de deux génomes. Le problème de la plus longue sous-chaîne permet d'apporter une réponse à cette question.

Exemples : Les chaines de caractères "AAA" et "TAA" sont les plus longues sous-chaînes communes aux chaines de ca- ractères "ATAGA" et "TAACA". La chaine de caractères "ATGC" est une plus longue sous-chaine commune aux chaines de caractères "AATGCG" et "TATTAGC".

Le problème de la plus longue sous-chaîne commune entre a et b est noté $L(a,b)$, son résultat est la longueur maximale d'une sous-chaîne commune à a et b.

1. On considère les chaines a ="AATGCG" et b ="TATTAGC" ? Donner les solutions de $L(a,b)$.
2. Écrire une fonction `est_ss(ch,sch)` où les paramètres sont deux chaînes de caractères et qui renvoie `True` si `sch` est une sous-chaîne de `ch` et `False` sinon.
3. Écrire une fonction `est_ss_commune(ch1,ch2,sch)` où les paramètres sont trois chaînes de caractères et qui renvoie `True` si `sch` est une sous-chaîne commune à `ch1` et `ch2` et `False` sinon.
4. Formuler le problème $L(a,b)$ récursivement afin de pouvoir justifier de sa sous-structure optimale.
5. Écrire un code qui résout $L(\text{ch}1,\text{ch}2)$ avec la programmation dynamique de bas en haut.
6. Résoudre $L(\text{ch}1,\text{ch}2)$ récursivement avec mémoïsation.

4.3. Ordonnancement de tâches pondérées

Vous vous rendez dans un festival de musique et souhaitez en profiter un maximum.

Le concert d'indice i est défini par l'instant de début et l'instant de fin : $[d_i, f_i]$. Vous ne pouvez bien évidemment assister qu'à un concert à la fois, et vous n'assistez qu'à des concerts entiers. L'intersection entre les intervalles de deux concerts auxquels vous assistez est l'ensemble vide ou limité à un nombre $(d_i = f_j)$.

Vous avez des préférences en terme de concerts, et avez donc attribué une valeur (niveau de satisfaction) à chaque concert. Votre objectif est de maximiser la valeur totale.

L'ensemble des concerts est noté $C = \{c_0, c_1, \ldots, c_{n-1}\}$. Chaque concert c_i a une valeur v_i. On cherche à déterminer le sous-ensemble S de C tel que $\sum_{i \in S} v_i$ est maximal.

On suppose qu'on a une liste des concerts du type `concerts=[[d0, f0 , v0 ],...]`, classés par ordre d'instant de fin croissant.

1. Écrire une fonction `compatible(C,k)` qui détermine l'indice du concert (parmi la liste `C`) compatible avec le concert d'indice `k` donné et qui se termine au plus près de ce concert `k`. La fonction renvoie -1 si aucun concert n'est compatible avec le concert `k`.
2. Proposer une fonction récursive "naïve" `sol_rec(C,k)` qui renvoie la valeur totale maximale que l'on peut obtenir. Quel est le problème de cet algorithme ?
3. L'améliorer en utilisant la mémoïsation.
4. Écrire la fonction `sol_iter(C)` qui renvoie la valeur totale en utilisant une approche ascendante.

4.4. Partition équilibrée d'un tableau d'entiers positifs

Vous partez en randonnée avec un ami. Vous avez un certain nombre, noté n, de choses à emporter : tente, victuailles, duvets, ... Vous souhaitez répartir toutes les choses à porter sur vos deux dos de la façon la plus équitable possible.

On note $C = \{i\}_{i \in [\![0,n-1]\!]}$ l'ensemble des choses à emporter, chacune de masse en gramme, m_i, avec $m_i \in N$.
On note C_1 l'ensemble des choses que vous allez devoir porter, de masse $M_1 = \sum_{k \in C_1} m_k$, et C_2 l'ensemble des choses que votre ami va devoir porter, de masse $M_2 = \sum_{k \in M_2} m_k$.
Ainsi, on cherche C_1 et C_2 tel que $C_1 \cup C_2 = C$, avec $C_1 \cap C_2 = \emptyset$, en minimisant $|M_1 - M_2|$.

Algorithme glouton

On choisit une chose, que l'on place dans C_1, puis on équilibre en plaçant les choses suivantes dans C_2, jusqu'à obtenir une somme supérieure ou égale, puis les éléments dans C_1, etc.

1. Écrire une fonction `glouton(C)` qui prend en argument la liste des masses des `n` choses à emporter, et qui renvoie les deux listes `C_1` et `C_2`.

On peut améliorer cela en commençant par trier les choses par masse décroissant.

2. Écrire la fonction `tri_rapide` qui prend en argument un liste d'entiers et qui renvoie une liste triée par ordre décroissant selon l'algorithme du tri rapide : on choisit le premier élément de la liste comme pivot, on répartit les autres éléments dans deux listes, l'une des entiers supérieurs ou égaux au pivot, et l'autre des entiers strictement inférieurs au pivot, puis on trie récursivement les deux listes que l'on concatène.

3. Écrire la fonction `glouton2(C)` qui commence par trier la liste `C`, puis applique l'algorithme glouton.

Programmation dynamique

4. Quelle est la valeur de M_1 et M_2 qui minimise $|M_1 - M_2|$ (c'est-à-dire égal à 0 , même si ce n'est pas atteignable) ? On note cette valeur M_I.

5. Combien de partitions faudrait-il évaluer si l'on voulait trouver la partition optimale en les énumérant toutes ? Commenter.

6. Soit $P(i,m)$ une fonction booléenne qui indique s'il existe un sous-ensemble de $\{0,1,..,i-1\}$ (les i premières choses) dont la masse totale est exactement m. Donner une définition récursive de $P(i,m)$. On envisage une approche ascendante, pour laquelle on stocke les résultats successifs en commençant par les plus petits sous-problèmes.

7. Calculer $P(i,m)$ pour toutes les valeurs de i et m pour l'ensemble des masses suivantes : {7,6,4,5,3} en recopiant et complétant le tableau suivant.

m_i \ m	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
7																										
6																										
4																										
5																										
3																										

8. Quelle est la complexité d'un algorithme qui compléterait un tableau donnant toutes les valeurs $P(i,m)$?

9. À partir de P et M_I, comment peut-on donner la valeur minimale de $|M_1 - M_2|$? Calculer la valeur minimale de $|M_1 - M_2|$ pour les masses {7,6,4,5,3}.

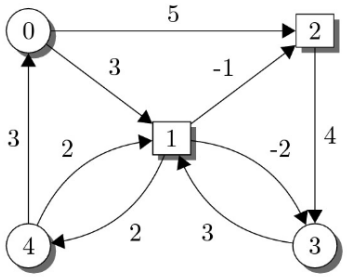
10. Expliquer à l'aide du tableau P comment trouver un groupe qui s'approche le plus du poids idéal ?

11. Écrire l'algorithme qui à partir de l'ensemble des masses des objets renvoie le tableau des valeurs (`True` ou `False`) de P.

12. Écrire l'algorithme qui à partir du tableau des valeurs de P retourne deux ensembles C_1 et C_2 tels que $|M_1 - M_2|$ soit minimal.

4.5. Plus courts chemins dans un graphe : Algorithme de Floyd-Warshall

On considère un graphe orienté pondéré, par exemple :



Sous certaines conditions, l'algorithme Floyd-Warshall calcule, pour chaque paire de sommets d'un graphe, la distance entre deux sommets. La distance entre deux sommets est définie par la longueur du plus court chemin, s'il en existe au moins un, entre les deux sommets. Un chemin est une suite d'arcs et sa longueur est la somme des poids des arcs.

Une condition d'utilisation : le graphe n'a aucun cycle de poids négatif.

Principe d'optimalité : si A-K-F est le plus court chemin entre A et F, alors A-K est le plus court chemin entre A et K et K-F est le plus court chemin entre K et F. Dans une séquence optimale, chaque sous-séquence est optimale.

À partir de la matrice d'adjacence représentant le graphe, on construit une matrice D_0, la matrice des distances. Un coefficient (i, j) égal à 0 dans la matrice d'adjacence signifie qu'il n'y a pas d'arc entre le sommet i et le sommet j. On pose alors $D_0(i, j) = \infty$.

On calcule entre les matrices D_k, après k itérations en utilisant des sommets intermédiaires dans $1,2,\ldots,k$, qui donnent les longueurs des plus courts chemins passant par ces sommets. Ce sont les sous-problèmes à résoudre.

On donne $c(s_1, s_2)$ le coût entre s_1 et s_2, soit le poids associé à l'arc (s_1, s_2), et on écrit la relation de récurrence liant les sous-problèmes : Pour $0 \leqslant k < n$:

- si $k = 0, D_k(s_1, s_2) = c(s_1, s_2)$, ($\infty$ s'il n'y a pas d'arc (s_1, s_2)) ;
- sinon, $D_k(s_1, s_2) = \min(D_{k-1}(s_1, s_2), D_{k-1}(s_1, v) + D_{k-1}(v, s_2))$ le minimum étant pris sur tous les sommets v compris entre 1 et k.

On utilise une approche ascendante pour programme la résolution du problème.

1. Écrire le dictionnaire `G` donnant les listes des couples (successeur, poids) de chaque sommet.

2. Écrire la fonction `distances(g)` qui prend un graphe représenté par un dictionnaire des listes d'adjacence et renvoie la matrice des distances initiales.

On importera `inf` de la bibliothèque `math`.

3. Écrire la fonction `floyd_warshall(D)` qui prend en entrée la matrice `D` des distances initiales, et renvoie la matrice `D` des distances minimales entre les sommets.

4. Quelle est la complexité de cet algorithme ?