PROGRAMMATION DYNAMIQUE

B. Landelle

Table des matières

I Dictionnaires

L'enjeu de cette partie est d'exposer comment stocker efficacement des couples de données qu'on appelle *(clé,valeur)* comme par exemple `("chien","dog")` pour une traduction français/anglais ou (n, u_n) pour une suite numérique. Il s'agit donc de permettre à l'ordinateur de réaliser une correspondance entre un ensemble de clés et des emplacements en mémoire, les *alvéoles*, pour le stockage des valeurs de ces clés.

1 Fonctions de hachage

Définition 1. *Une* fonction de hachage *est une fonction qui à une clé associe une alvéole. Si cette fonction est injective, on dit que la fonction de hachage est* parfaite. *Si cette fonction renvoie toujours le même résultat pour une entrée donnée, la fonction de hachage est dite* déterministe.

Traditionnellement, l'ensemble des clés possibles est appelé *univers des clés* noté $\mathcal{U}$ et l'ensemble des clés réellement utilisées est noté $\mathcal{K}$.

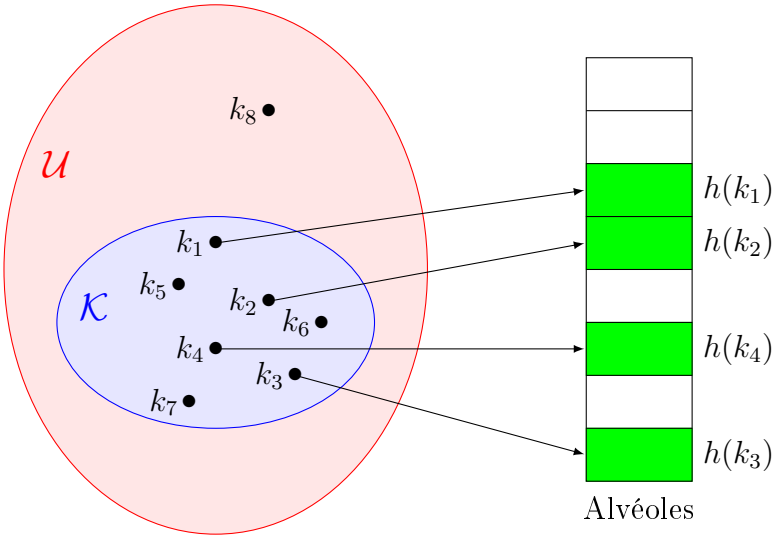


FIGURE 1 – Fonction de hachage h

Si l'univers des clés était petit, on pourrait envisager en pratique une correspondance directe entre les clés et les alvéoles réalisée par une fonction de hachage parfaite.

Mais si l'univers des clés est grand ce qui est le plus souvent le cas en pratique, on ne peut plus garantir l'injectivité d'une fonction de hachage.

Définition 2. *Étant donnée une fonction de hachage, on appelle* collision *un couple de clés ayant même image par la fonction de hachage.*

Illustration : La fonction `hash()` est une implémentation d'une fonction de hachage. Pour une architecture 64 bits (la plupart des machines actuelles), elle renvoie une valeur dans l'ensemble $[\![-2^{63} ; 2^{63} - 1]\!]$. Évidemment, cette fonction n'est pas parfaite puisque l'ensemble des clés, laissé au bon vouloir de l'utilisateur, est potentiellement infini. On peut facilement observer des collisions

```
>>> hash(1)
1
>>> hash(2**61)
1
```

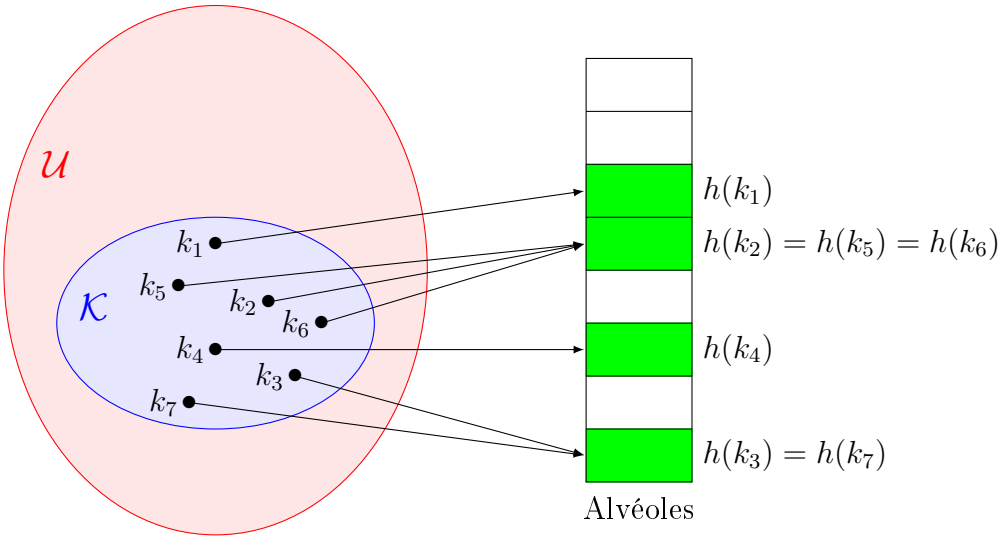


FIGURE 2 – Phénomène de collisions

En pratique, les collisions sont donc inévitables. Toutefois, on cherche à les rendre aussi rares que possible et le choix d'une bonne fonction de hachage est déterminant. On fait en sorte, en général, que cette fonction de hachage balaye l'ensemble des alvéoles de manière à peu près équiprobable. En effet, si une alvéole est choisie plus fréquemment qu'une autre, celle-ci sera le théâtre de nombreuses collisions. Le choix d'une fonction de hachage est un sujet d'étude en soi (voir [1]).

On peut « hacher » d'autres types que des entiers mais tous les types ne sont pas « hachables » :

```
>>> hash(1.234)
5395672641560043 53
>>> hash("bonjour")
-3789520816923425902
>>> hash((1,"python"))
1315902330207892874
>>> hash([0])
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    hash([0])
TypeError: unhashable type: 'list'
```

Proposition 1. *En python, les types non mutables sont hachables par la fonction* `hash` *tandis que les types mutables ne sont pas hachables.*

On peut également observer que la fonction `hash()` n'est pas déterministe. Entre deux sessions d'utilisation, on n'obtient pas les mêmes résultats pour des chaînes de caractères par exemple :

```
>>> hash("bonjour")
-8503926400129448945

= RESTART: D:\ITC2\COURS\CH04_PROG_DYN\PROG_DYN.py =

>>> hash("bonjour")
7548442157190083004
```

2 Tables de hachage

Définition 3. *Une* table de hachage *est une structure de données qui réalise une association clé-valeur.*

Une telle structure utilise une fonction de hachage pour identifier l'alvéole stockant la valeur associée à une clé. Si le coût de la fonction de hachage est en O(1), alors on dispose d'une structure de données extrêmement performante pour la lecture et l'écriture dans une alvéole.

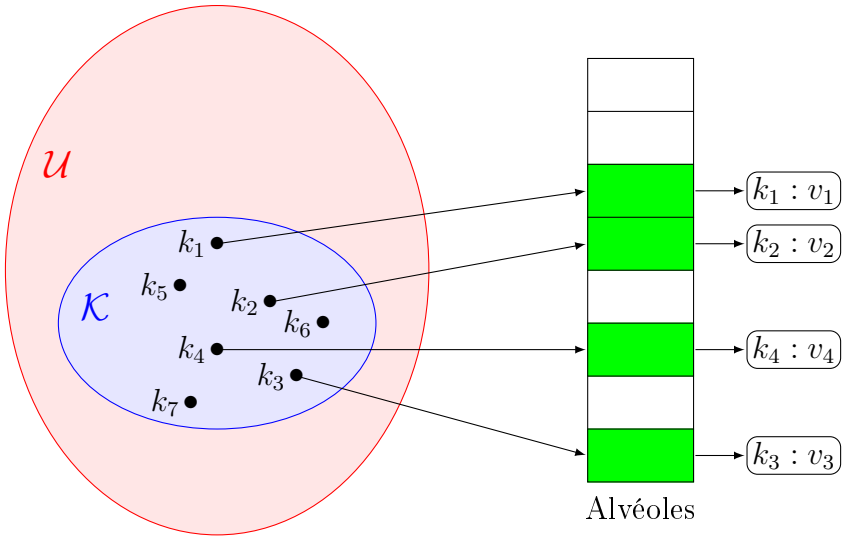


FIGURE 3 – Table de hachage

On l'a vu ci-avant, les collisions sont inévitables dès que l'univers des clés est grand ce qui est le cas en pratique. Il faut donc s'efforcer de résoudre ces collisions.

Pour résoudre les collisions, deux stratégies sont employées :
- le hachage avec chaînage,
- l'adressage ouvert.

Définition 4. *Le hachage avec* chaînage *consiste à utiliser des* listes chaînées *pour les alvéoles qui permettent le stockage de plusieurs valeurs au lieu d'une.*

Une liste chaînée est constituée de couples *(valeur, pointeur)* où l'information *pointeur* désigne l'adresse du successeur de ce couple. Ceci garantit des performances élevées de lecture/écriture.

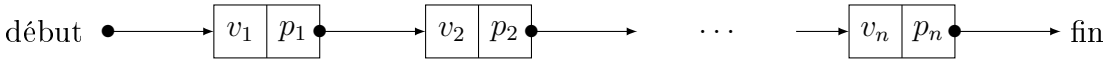


FIGURE 4 – Liste chaînée

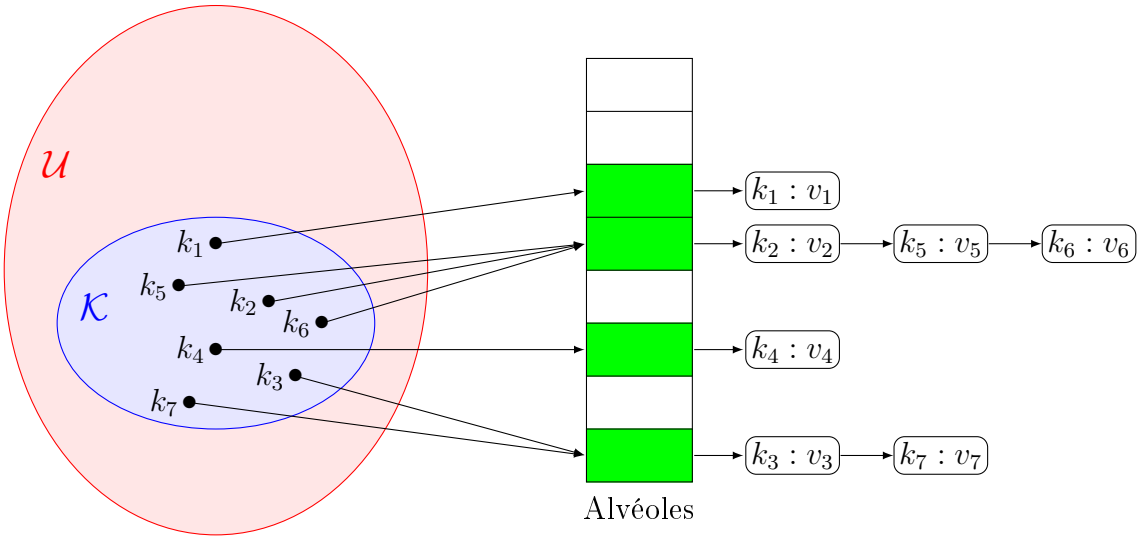


FIGURE 5 – Hachage avec chaînage

Définition 5. *Le hachage par* adressage ouvert *consiste à réaliser une séquence de sondages lors de l'attribution d'une alvéole, jusqu'à en trouver une libre.*

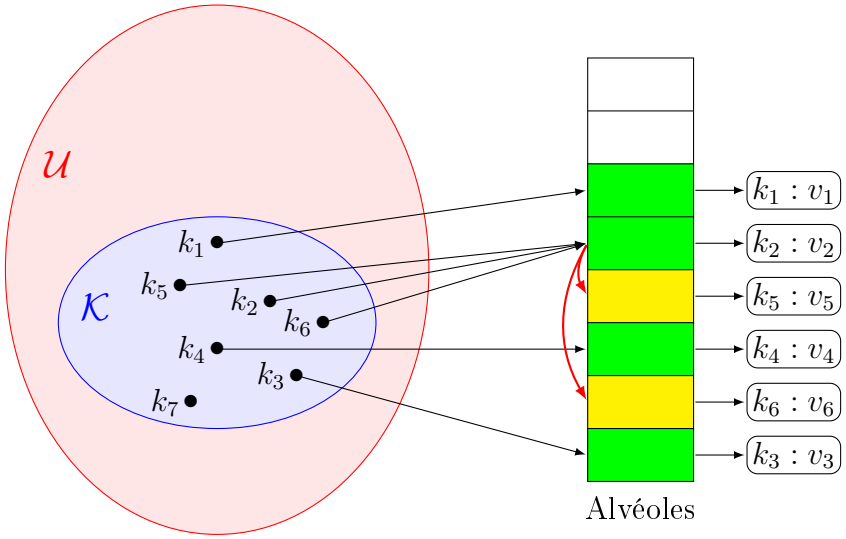


FIGURE 6 – Hachage avec adressage ouvert

3 Implémentation

Définition 6. *En python, les* dictionnaires, *de type* `dict`, *sont une implémentation des tables de hachage et servent donc à réaliser des associations clé-valeurs.*

Expérimentation :

```
>>> dico={}
>>> type(dico)
<class 'dict'>
>>> dico["chat"]="cat"
>>> dico[0]=False
```

```
>>> dico
{'chat': 'cat', 0: False}
>>> "chat" in dico
True
>>> dico["chat"]
'cat'
>>> "dog" in dico
False
>>> dico["SPE"]=["MP","PC"]
>>> dico
{'chat': 'cat', 0: False, 'SPE': ['MP', 'PC']}
>>> dico["SPE"].append("PSI")
>>> dico
{'chat': 'cat', 0: False, 'SPE': ['MP', 'PC', 'PSI']}
>>> len(dico)
3
```

Proposition 2. *On peut parcourir un dictionnaire :*
- *par clés, directement ou avec la liste renvoyée par la méthode* `keys` *;*
- *par valeurs, avec la liste renvoyée par la méthode* `values` *;*
- *par couples (clé,valeur), avec la liste renvoyée par la méthode* `items`*.*

Expérimentation :

```
>>> for k in dico:
        print(k)
chat
0
SPE
>>> list(dico)
['chat', 0, 'SPE']
>>> for k in dico:
        print(k,dico[k])
chat cat
0 False
SPE ['MP', 'PC', 'PSI']
>>> dico.keys()
dict_keys(['chat', 0, 'SPE'])
>>> dico.values()
dict_values(['cat', False, ['MP', 'PC', 'PSI']])
>>> dico.items()
dict_items([('chat', 'cat'), (0, False), ('SPE', ['MP', 'PC', 'PSI'])])
>>> for k,v in dico.items():
        print(k,v)
chat cat
0 False
SPE ['MP', 'PC', 'PSI']
```

Les dictionnaires sont mutables :

```
>>> memo=dico
>>> memo["dog"]="chien"
>>> memo
{'chat': 'cat', 0: False, 'SPE': ['MP', 'PC', 'PSI'], 'dog': 'chien'}
>>> dico
{'chat': 'cat', 0: False, 'SPE': ['MP', 'PC', 'PSI'], 'dog': 'chien'}
```

Proposition 3. *Les tests d'appartenance et les temps d'accès en lecture/écriture à un couple (clé,valeur) d'un dictionnaire sont en* $O(1)$.

Ces prouesses de complexité, résultats d'une implémentation réussie de la structure de table de hachage, font des dictionnaires un type parfaitement adapté à certains aspects de la *programmation dynamique*.

II Programmation dynamique

1 Introduction

La *programmation dynamique* est une méthode générale pour concevoir des algorithmes qui permettent de résoudre efficacement des problèmes en combinant des solutions de sous-problèmes (voir [1] et [2]). Son concept est formalisé et popularisé par Richard Bellman [1] dans les années 1950 (on peut trouver des approches de programmation dynamique antérieures à Bellman mais c'est réellement lui qui pose un socle conceptuel à cette démarche).

Le terme *programmation* s'entend au sens d'une méthode ou planification et non au sens informatique du terme.

Cette méthode s'applique en général aux problèmes d'optimisation. Pour un problème d'optimisation, on peut envisager des algorithmes gloutons mais ceux-ci procèdent par optimisation locale ce qui ne garantit pas d'atteindre l'optimum global (c'est d'ailleurs ce qui arrive dans la plupart des cas).

On connaît également l'approche *diviser pour régner* qui consiste à diviser le problème initial en sous-problèmes indépendants que l'on résout (le règne). À la différence de l'approche *diviser pour régner*, la programmation dynamique s'applique à des sous-problèmes qui se recouvrent, c'est-à-dire des sous-problèmes ayant eux-même en commun d'autres sous-problèmes.

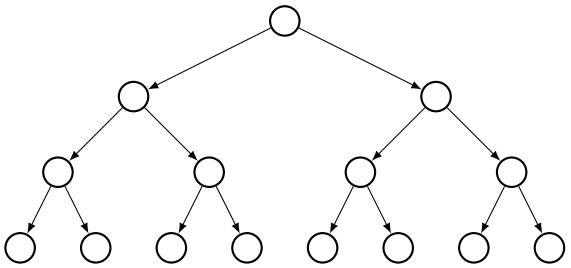


FIGURE 7 – Diviser pour régner

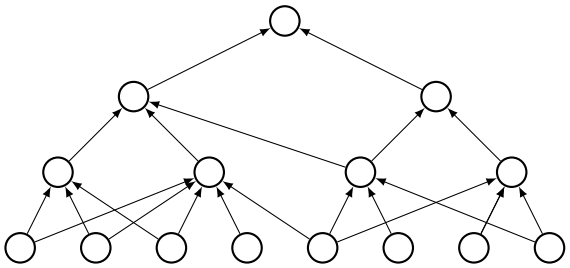


FIGURE 8 – Programmation dynamique

1. Richard Bellman, 1920-1984, mathématicien américain.

Par exemple, pour la suite de Fibonacci $(u_n)_n$ définie par

$$u_0 = 0, \qquad u_1 = 1 \qquad \forall n \in \mathbb{N} \qquad u_{n+2} = u_{n+1} + u_n$$

on constate que le calcul de u_5 fait intervenir plusieurs fois les mêmes sous-problèmes, à savoir de valeurs de la suite pour de mêmes indices inférieurs à 5.

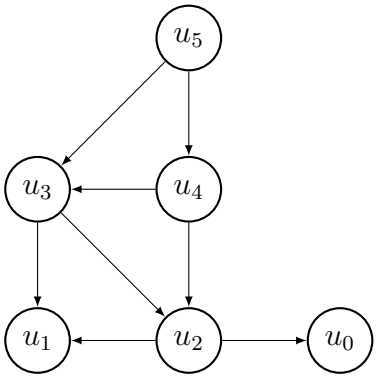


FIGURE 9 – Recouvrement de sous-problèmes pour le calcul de u_5

La conception d'un algorithme de programmation dynamique pour un problème d'optimisation se décompose en quatre étapes :

1. caractérisation de la structure d'une solution optimale ;
2. définition récursive de la valeur d'une solution optimale ;
3. calcul *ascendant* de la valeur d'une solution optimale,
4. construction d'une solution optimale.

Définition 7. *Pour un problème d'optimisation, le* principe d'optimalité de Bellman *s'énonce ainsi : une solution optimale au problème contient les solutions optimales des sous-problèmes. Un problème qui vérifie le principe d'optimalité de Bellman possède une* sous-structure optimale.

Exemples : 1. Dans un graphe, si un plus court chemin relie les sommets x et y en passant par z, alors les sous-chemins de x à z et de z à y sont optimaux.

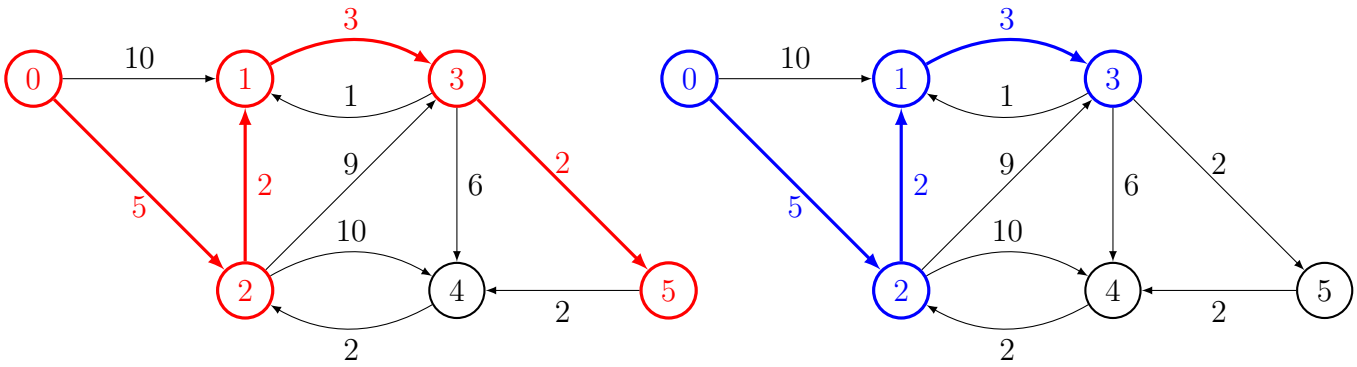


FIGURE 10 – Principe d'optimalité de Bellman

2. Pour rendre la monnaie sur 13€ de manière optimale, on rend 10€ puis 2€ puis 1€. Ainsi, pour rendre $13 - 2 = 11$€ de manière optimale, on rend 10€ puis 1€.

Plus généralement, si $S_n = [c_1, \ldots, c_n]$ désigne un système de monnaie (ensemble de jetons disponibles) et $M(S_n, v)$ le nombre minimal de jetons pour rendre le montant v, on peut établir la relation parfois appelée *équation de Bellman*[2] :

$$M(S_n, v) = 1 + \min_{1 \leqslant k \leqslant n} M(S_n, v - c_k)$$

En effet, si pour rendre le montant v on utilise le jeton c_k, alors le rendu optimal est $1 + M(S_n, v - c_k)$. On lit dans cette relation la sous-structure optimale du problème.

Pour que la programmation dynamique soit applicable à un problème d'optimisation, celui-ci doit présenter deux caractéristiques :

— la présence de sous-structures optimales ;
— le recouvrement de sous-problèmes.

Les sous-problèmes sont de taille « un peu plus petite » que le problème initial pour qu'il y ait recouvrement. On peut aussi l'interpréter ainsi : l'ensemble des sous-problèmes est « petit » ce qui fait qu'on rencontre plusieurs fois les mêmes sous-problèmes.

2 Approche descendante, mémoïsation

Définition 8. *L'approche* descendante *(*top-down *en anglais) en programmation dynamique consiste à partir du problème initial et à résoudre récursivement les sous-problèmes de la formulation récursive.*

Exemple : Une situation très simple (qui n'est pas un problème d'optimisation) où la programmation dynamique s'applique est le calcul de la suite de Fibonacci $(u_n)_n$ définie par

$$u_0 = 0, \qquad u_1 = 1 \qquad \forall n \in \mathbb{N} \qquad u_{n+2} = u_{n+1} + u_n$$

Approche descendante

L'approche descendante consiste en l'implémentation récursive suivante :

```
def fibo1(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fibo1(n-1)+fibo1(n-2)
```

Cette approche n'est pas du tout performante : sa complexité est en $O(\varphi^n)$ avec $\varphi = \dfrac{1+\sqrt{5}}{2}$.
Il y a une très forte redondance des calculs comme on peut le voir dans l'arbre des appels récursifs :

2. Selon cette dénomination, il y a donc autant d'équations de Bellman que de formulations de problèmes vérifiant le principe de Bellman ce qui en fait une appellation un peu galvaudée.

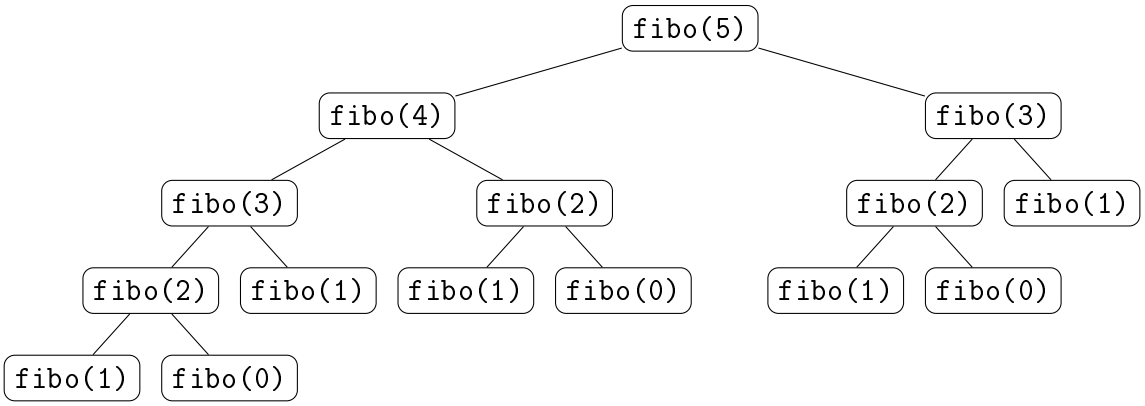


FIGURE 11 – Arbre des appels de `fibo(5)`

On voit que les branches issues de `fibo(2)` apparaissent trois fois ce qui signifie trois fois la résolution du même problème, les branches issues de `fibo(3)` apparaissent deux fois, etc. ...

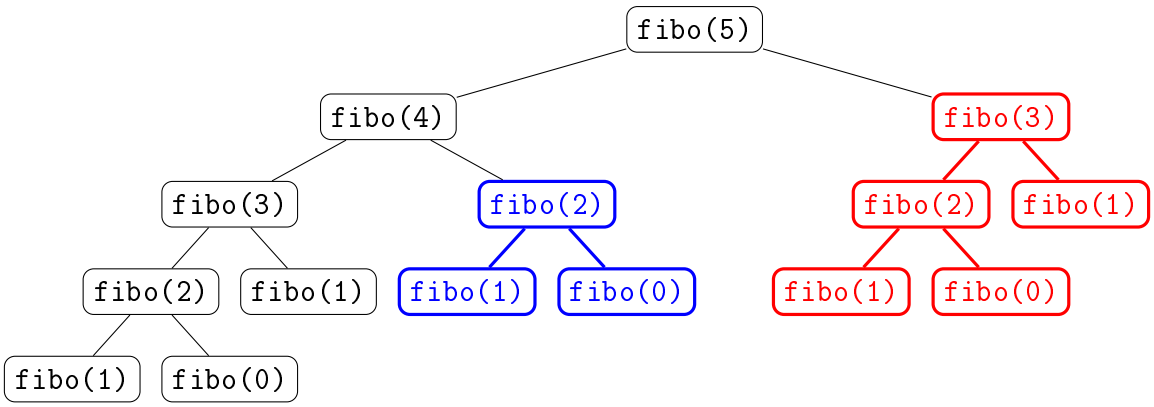


FIGURE 12 – Redondance des branches

Définition 9. *La* mémoïsation *consiste, pour une fonction récursive, à enregistrer tout résultat qui n'a pas été déjà traité lors des récursions.*

Remarques : (1) On utilise très souvent des dictionnaires, parfois des listes, pour effectuer cette mémoïsation.
(2) Ce procédé va donc, au prix d'un accroissement de la complexité spatiale, permettre une amélioration notable de la complexité temporelle.

Approche descendante avec mémoïsation

On reprend l'exemple de la suite de Fibonacci :

```
def fibo2(n):
    memo=[None]*(n+1)
    def fibo_memo(k):
        # fonction locale, memo vue comme globale ici
        if memo[k]==None:
            if k==0:
                memo[0]=0
            elif k==1:
                memo[1]=1
            else:
                memo[k]=fibo_memo(k-1)+fibo_memo(k-2)
        return memo[k]
    return fibo_memo(n)
```

ou avec un dictionnaire :

```
def fibo2(n):
    memo={0:0,1:1}
    def fibo_memo(k):
        # fonction locale, memo vue comme globale ici
        if not k in memo:
            memo[k]=fibo_memo(k-1)+fibo_memo(k-2)
        return memo[k]
    return fibo_memo(n)
```

L'appel de `fibo2(100)` n'est pas rédhibitoire alors que `fibo1(100)` l'est clairement.

Exercice : Déterminer la complexité temporelle de `fibo2(n)`.

Corrigé : Grâce à la mémoïsation, une valeur de `fibo_memo(k)` est calculée une fois et une seule. Lors de l'appel de `fibo_memo(n)`, on empile les appels `fibo_memo(n-1)`, `fibo_memo(n-2)`, ... jusqu'à `fibo_memo(2)` puis `fibo_memo(1)`, cas de base qu'on dépile en stockant la valeur pour ensuite empiler `fibo_memo(0)`, cas de base qu'on dépile en stockant la valeur et on va ensuite dépiler tous les appels en stockant successivement leurs valeurs sans qu'il y ait de nouvel empilement. On effectue donc $n+1$ empilements/dépilements pour un coût en $\mathrm{O}(n)$ auquel il faut éventuellement ajouter le coût de création de `[None]*(n+1)` qui est également en $\mathrm{O}(n)$. On en déduit une complexité temporelle en $\mathrm{O}(n)$ pour le calcul de `fibo2(n)`.

3 Approche ascendante

Définition 10. *L'approche* ascendante *(*bottom-up *en anglais) en programmation dynamique consiste à résoudre les sous-problèmes par taille croissante, des plus petits au plus grands, en stockant leurs résultats successivement.*

Exemple : On reprend l'exemple de la suite de Fibonacci $(u_n)_n$ définie par

$$u_0 = 0, \qquad u_1 = 1 \qquad \forall n \in \mathbb{N} \qquad u_{n+2} = u_{n+1} + u_n$$

L'approche ascendante consiste en l'implémentation itérative suivante :

```
def fibo3(n):
    res=[0,1]
    for k in range(n):
        res.append(res[k]+res[k+1])
    return res[n]
```

La performance de `fibo3` est meilleure que celle de `fibo2` (pas d'empilement de récursions) et cette version itérative n'est pas contrainte pas la limitation de hauteur de pile de récursions : l'appel `fibo2(994)` provoque un dépassement de pile.

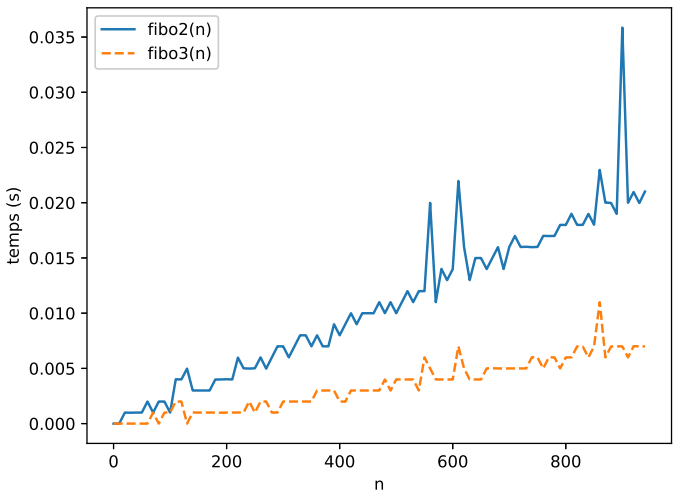


FIGURE 13 – Comparatif de `fibo2` et `fibo3`

Lors de l'appel de `fibo3(n)`, la totalité des termes $[u_0, \ldots, u_n]$ est stockée alors que la k-ième itération ne requiert que u_{k-1} et u_{k-2} autrement dit, les deux derniers termes de la suite calculés précédemment. On peut améliorer notablement la complexité spatiale du programme en ne manipulant que deux variables qui servent à stocker à chaque itération, les deux derniers termes consécutifs de la suite.

```
def fibo4(n):
    if n==0:
        return 0
    u,v=0,1
    for k in range(2,n+1):
        u,v=v,u+v
    return v
```

Cette optimisation spatiale n'est pas une quête systématique. Pour un problème d'optimisation, on souhaite en général construire une (ou toutes) solution optimale et pour cela, on « remonte

l'historique » des calculs qui ont permis d'aboutir à la valeur d'une solution optimale, d'où la nécessité de stocker les calculs intermédiaires.

Remarque : Cette dernière version d'implémentation du calcul de $(u_n)_n$ peut encore être améliorée en recourant à l'exponentiation rapide ...

III Algorithme de Floyd-Warshall

1 Présentation

Soit $G = (S, A, \nu)$ un graphe valué, orienté, sans circuit absorbant avec $n = \text{Card } S$ l'ordre du graphe et $S = [\![0\,;\,n-1]\!]$. Pour $(k,i,j) \in [\![0\,;\,n-1]\!]^3$, on note $w_{i,j}^{(k)}$ le poids d'un plus court chemin du sommet i au sommet j de sommets intermédiaires dans $[\![0\,;\,k]\!]$ s'il existe un tel chemin et $+\infty$ sinon. On pose

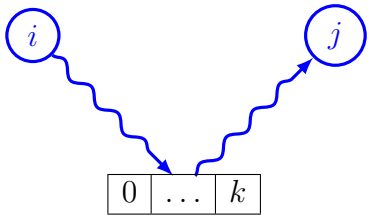
$$\forall k \in [\![0\,;\,n-1]\!] \qquad \mathrm{W}^{(k)} = \left(w_{i,j}^{(k)}\right)_{0\leqslant i,j\leqslant n-1}$$

Pour $k = -1$, la matrice $\mathrm{W}^{(-1)}$ est la matrice d'adjacence du graphe G.

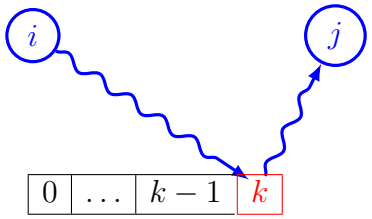
Proposition 4. *On a*

$$\forall (i,j,k) \in [\![0\,;\,n-1]\!]^3 \qquad w_{i,j}^{(k)} = \min\left(w_{i,j}^{(k-1)}, w_{i,k}^{(k-1)} + w_{k,j}^{(k-1)}\right)$$

Démonstration. Soit $(i,j,k) \in [\![0\,;\,n-1]\!]^3$. Considérons un plus court chemin de i à j de sommets intermédiaires dans $[\![0\,;\,k]\!]$. Soit ce chemin ne passe pas par le sommet k auquel cas son poids est $w_{i,j}^{(k-1)}$.



Soit ce chemin passe par le sommet k. On peut alors considérer qu'il passe exactement une fois par ce sommet. Sinon, on aurait la présence d'un circuit de coût nul qu'on peut donc éliminer (le circuit n'est pas absorbant et si son coût était strictement positif, l'élimination de ce circuit donnerait un chemin de coût strictement inférieur).



Le coût du chemin de i à j passant exactement une fois par k est donc bien la somme des coûts des chemins de i à k et de k à j de sommets intermédiaires dans $[\![0\,;\,k-1]\!]$.

□

2 Coût d'un plus court chemin

On déduit de la proposition 4 l'implémentation suivante :

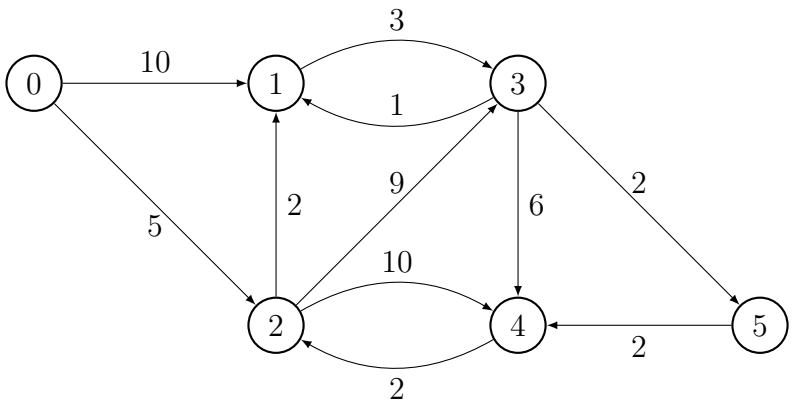
```
def fw_cost(M):
    """Plus courts chemins :
    M matrice d'adjacence
    Renvoie la matrice de l'algorithme de Floyd-Warshall"""
    n=len(M)
    W=[[M[i][j] for j in range(n)] for i in range(n)]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                W[i][j]=min(W[i][j],W[i][k]+W[k][j])
    return W
```

La complexité temporelle de `fw_cost` est clairement en $O(n^3)$. C'est moins bien que la complexité de l'algorithme de Bellman-Ford mais l'algorithme de Floyd-Warshall est exhaustif : on a les plus courts chemins entre tous les couples de sommets.

Exercice : Déterminer la complexité spatiale de `fw_cost`.

Corrigé : Pour `M` matrice d'adjacence d'ordre n, on construit une liste de listes `W` de n sous-listes chacune de taille n. Les autres variables locales sont des entiers. On en déduit que la complexité spatiale de `fw_cost(M)` est en $O(n^2)$.

On teste `fw_cost` sur le graphe décrit par liste d'adjacence suivant :



On commence par construire la matrice d'adjacence :

```
S=list(range(6))
A={0:[(1,10),(2,5)],1:[(3,3)],2:[(1,2),(3,9),(4,10)],
   3:[(1,1),(4,6),(5,2)],4:[(2,2)],5:[(4,2)]}

def matr_adj(S,A):
    """ Construit la matrice d'adjacence
    du graphe de sommets S et de liste d'adjacence A"""
    n=len(S)
    M=[[float('inf')]*n for j in range(n)]
    for x in S:
        for y,nu in A[x]:
            M[x][y]=nu
    return M
```

On teste :

```
>>> cost=fw_cost(M)
>>> for x in cost: print(x)

[inf, 7, 5, 10, 14, 12]
[inf, 4, 9, 3, 7, 5]
[inf, 2, 11, 5, 9, 7]
[inf, 1, 6, 4, 4, 2]
[inf, 4, 2, 7, 11, 9]
[inf, 6, 4, 9, 2, 11]
```

3 Un concurrent : Bellman-Ford

Cette section peut être évitée en première lecture.

L'algorithme de Bellman-Ford réalise, par relâchements successifs, la recherche d'un plus court chemin dans un graphe orienté valué sans circuit absorbant depuis un sommet source s_0. Contrairement à ce qui était fait dans l'algorithme de Dijkstra, chaque arc va être relâché plusieurs fois. On relâche une première fois tous les arcs ce qui permet de déterminer tous les plus courts chemins de longueur (en nombre d'arcs parcourus) égale à 1 depuis s_0. On relâche ensuite une deuxième fois ce qui permet de déterminer tous les plus courts chemins de longueur égale à 2 depuis s_0. On répète le processus $n-1$ fois avec n le nombre de sommets du graphe. Pour tout sommet accessible depuis s_0, il existe un plus court chemin depuis s_0 de longueur $\leqslant n-1$; en effet, considérant un plus court chemin depuis s_0 vers un sommet, s'il n'est pas élémentaire, alors on peut en extraire un chemin élémentaire de même coût puisque le circuit éliminé sera de coût positif et donc nul, du fait du caractère « le plus court ».

On conserve les notations introduites pour l'algorithme de Dijkstra. Le principe est le suivant :
- on initialise $\mathrm{d}[s_0] \leftarrow 0$ et $\mathrm{d}[u] \leftarrow \infty$ pour tout $u \in \mathrm{S} \setminus \{s_0\}$;
- Pour tout k de 1 à $n-1$, on répète :
  - Pour tout arc $(x,y) \in \mathrm{A}$, on effectue l'opération de *relâchement* :

$$\text{si } \mathrm{d}[y] > \mathrm{d}[x] + \nu(x,y), \text{ alors } \mathrm{d}[y] \leftarrow \mathrm{d}[x] + \nu(x,y)$$

```
def BF(S,A,s0):
    cout,predec={},{}
    for s in S:
        cout[s]=np.inf
    cout[s0]=0
    for _ in range(1,len(S)):
        # pour chaque arc
        for x in A:
            for y,nu in A[x]:
                # on relâche
                if cout[y]>cout[x]+nu:
                    cout[y]=cout[x]+nu
                    predec[y]=x
    return cout,predec
```

Avec $n = \text{Card S}$ et $m = \text{Card A}$, la complexité temporelle de `BF` est en $\mathrm{O}(nm)$ avec $m \leqslant n^2$.

Exercice : Si on compare le résultat fourni par la fonction `fw_cost` à celui par la fonction `BF`, on obtient :

```
>>> for k in range(6):
        print(BF(S,A,k)[0])


{0: 0, 1: 7, 2: 5, 3: 10, 4: 14, 5: 12}
{0: inf, 1: 0, 2: 9, 3: 3, 4: 7, 5: 5}
{0: inf, 1: 2, 2: 0, 3: 5, 4: 9, 5: 7}
{0: inf, 1: 1, 2: 6, 3: 0, 4: 4, 5: 2}
{0: inf, 1: 4, 2: 2, 3: 7, 4: 0, 5: 9}
{0: inf, 1: 6, 2: 4, 3: 9, 4: 2, 5: 0}
```

Les résultats sont conformes exceptés pour les nœuds diagonaux. Expliquer ce point.

Corrigé : Dans la fonction `BF`, un sommet est considéré à distance nulle de lui même alors que dans `fw_cost`, un nœud diagonal est le coût minimal pour partir d'un sommet et y revenir.

4 Plus court chemin et prédécesseurs

Pour déterminer les prédécesseurs dans les plus courts chemins calculés par l'algorithme de Floyd-Warshall, on initialise une matrice pour les prédécesseurs avec en position (i, j) la valeur i si le sommet j est adjacent au sommet i et $+\infty$ sinon puis, à chaque minimisation, on met à jour le prédécesseur en retenant la dernière étape si celle-ci abaisse le coût.

```
def fw_cost_path(M):
    """Plus courts chemins :
    M matrice d'adjacence
    Renvoie la matrice de l'algorithme de Floyd-Warshall
    et la matrice des prédécesseurs"""
```

```
    n=len(M)
    W=[[M[i][j] for j in range(n)] for i in range(n)]
    predec=[[None]*n for i in range(n)]
    for i in range(n):
        for j in range(n):
            if W[i][j]<np.inf:
                predec[i][j]=i
    for k in range(n):
        for i in range(n):
            for j in range(n):
                aux=min(W[i][j],W[i][k]+W[k][j])
                if aux<W[i][j]:
                    predec[i][j]=predec[k][j]
                W[i][j]=aux
    return W,predec
```

On teste :

```
>>> cost,path=fw_cost_path(M)
>>> for x in path: print(x)

[None, 2, 0, 1, 5, 3]
[None, 3, 4, 1, 5, 3]
[None, 2, 4, 1, 5, 3]
[None, 3, 4, 1, 5, 3]
[None, 2, 4, 1, 5, 3]
[None, 2, 4, 1, 5, 3]
```

IV Rendu de monnaie

On opte pour la convention $\min \varnothing = +\infty$ (celle-ci nous permet d'écrire des min plutôt que des inf dans ce qui suit).

1 Présentation

Définition 11. *Soit* $\mathrm{S}_n = [c_1, \ldots, c_n]$ *un système de monnaie, les* c_i *désignant des montants entiers de pièces ou de billets qu'on appellera* jetons *de manière générique. Le problème du* rendu de monnaie *consiste, étant donné un montant entier* v *à rendre, à minimiser le nombre de jetons pour réaliser ce rendu, autrement dit à réaliser*

$$\mathrm{M}(\mathrm{S}_n, v) = \min\left\{\sum_{i=1}^{n} x_i \ : \ (x_1, \ldots, x_n) \in \mathbb{N}^n, \ \sum_{i=1}^{n} x_i c_i = v\right\}$$

Remarque : Les montants sont supposés entiers pour simplifier mais on pourrait tout à fait intégrer des centimes : il suffirait de formuler les montants en centimes pour ne traiter que des entiers.

Le parcours exhaustif de l'ensemble potentiel des solutions est de taille majorée par

$$\prod_{i=1}^{n}\left(1+\left\lfloor\frac{v}{c_i}\right\rfloor\right) \underset{v\to+\infty}{=} \mathrm{O}(v^n)$$

avec typiquement $n = 9$ pour $\mathrm{S} = [1, 2, 5, 10, 20, 50, 100, 200, 500]$ ce qui invite assez clairement à se lancer dans une recherche de stratégies plus performantes.

La démarche gloutonne naturelle consiste à rendre en priorité le plus gros jeton. Ainsi, pour rendre un montant de 123 avec le système $\mathrm{S} = [1, 2, 5, 10, 20, 50, 100, 200, 500]$, on choisit un jeton de 100 puis de 20 puis de 2 et enfin de 1.

Définition 12. *Un système de monnaie est dit* canonique *si l'algorithme glouton naturel de choix du plus gros jeton en priorité résout le problème du rendu de monnaie.*

Le système `S` considéré dans la plupart des exemples qui suivent est

```
S = [1, 2, 5, 10, 20, 50, 100, 200, 500]
```

```
def rendu_glouton(S,v):
    nb=0
    ind=len(S)-1
    reste=v
    while reste>0 and ind>=0:
        if reste>=S[ind]:
            nb+=1
            reste-=S[ind]
        else:
            ind-=1
    if reste>0:
        return float('inf')
    return nb
```

Quelques essais :

```
>>> rendu_glouton([2,3],1)
inf
>>> rendu_glouton(S,1989)
11
>>> rendu_glouton(S,111111)
225
>>> rendu_glouton([1,3,4],6)
# système non canonique, le glouton échoue sur 6=3+3
3
```

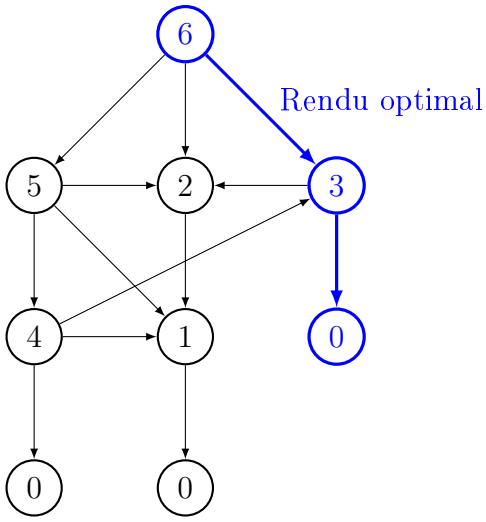


FIGURE 14 – Rendus possibles

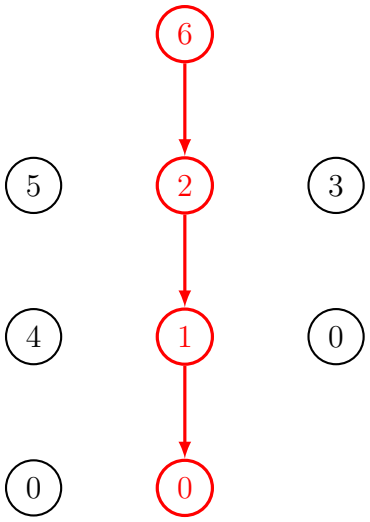


FIGURE 15 – Rendu glouton

2 Première formulation

Proposition 5. *Soit* S_n *un système de monnaie et* v *un montant à rendre. On a*

$$\mathrm{M}(\mathrm{S}_n, v) = 1 + \min_{1\leqslant k\leqslant n} \mathrm{M}(\mathrm{S}_n, v - c_k)$$

avec $\mathrm{M}(\mathrm{S}_n, 0) = 0$ *et* $\forall v \in [\![1\,;\, \min(\mathrm{S}_n) - 1]\!]$ $\mathrm{M}(\mathrm{S}_n, v) = +\infty$

Démonstration. On décompose

$$\begin{aligned}\mathrm{M}(\mathrm{S}_n, v) &= \min_{1\leqslant k\leqslant n} \min \left\{ 1 + \sum_{i=1}^{n} (x_i - \delta_{i,k}) \;:\; (x_i - \delta_{i,k})_{1\leqslant i\leqslant n} \in \mathbb{N}^n,\ \sum_{i=1}^{n} (x_i - \delta_{i,k}) c_i = v - c_k \right\} \\ &= 1 + \min_{1\leqslant k\leqslant n} \mathrm{M}(\mathrm{S}_n, v - c_k)\end{aligned}$$

□

Approche descendante sans mémoïsation

On propose l'implémentation descendante :

```
def rendu1(S,v):
    if v==0:
        return 0
    mini=float('inf')
    for c in S:
        if v>=c:
            aux=rendu1(S,v-c)
            if aux<mini:
                mini=aux
    return 1+mini
```

On se rend très vite compte après quelques essais que cette version est complètement inefficace. La complexité temporelle vérifie une relation de la forme

$$\mathrm{T}(v) = \sum_{k=1}^{n} \mathrm{T}(v - c_k) + \mathrm{O}(1)$$

ce qui laisse augurer une complexité catastrophique. Par ailleurs, en considérant l'arbre des appels de `rendu1(S,5)` (on note les nœuds `r(x)` pour alléger la taille de l'arbre), on observe une très forte redondance des calculs.

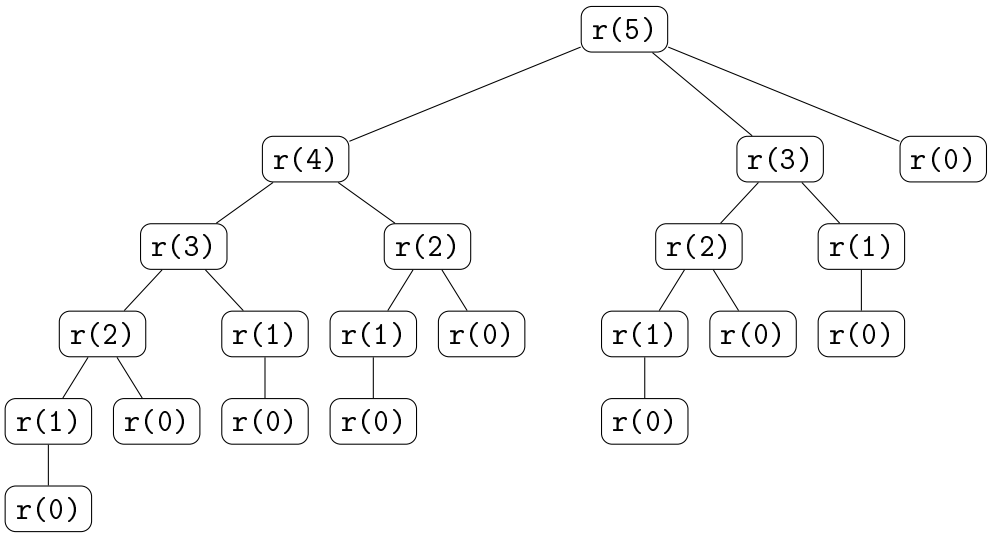


FIGURE 16 – Arbre des appels de `rendu1(S,5)`

La branche issue de `r(1)` apparaît cinq fois, les branches issues de `r(2)` apparaissent trois fois et les branches issues de `r(3)` apparaissent deux fois.

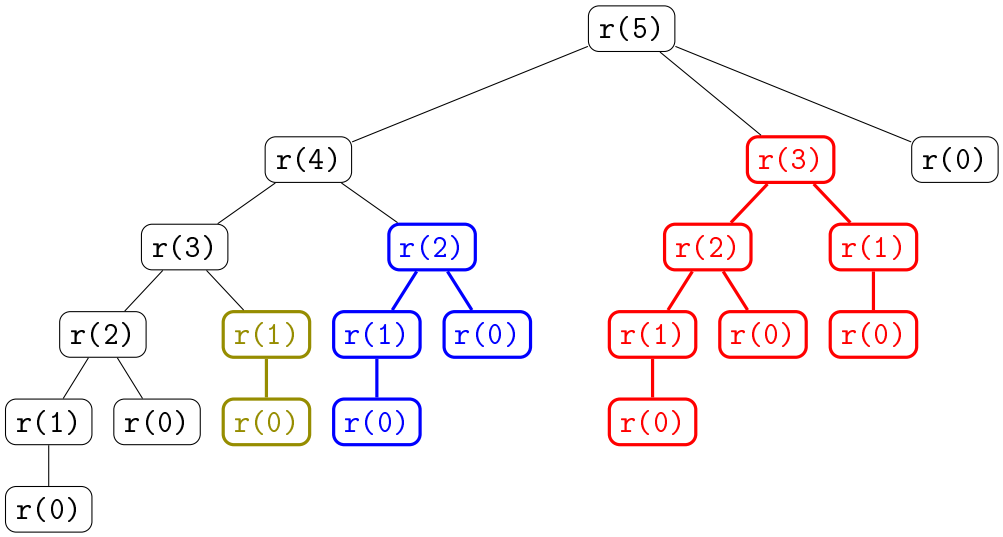


FIGURE 17 – Redondance des branches

Approche descendante avec mémoïsation

On améliore le procédé avec de la mémoïsation dans un dictionnaire utilisé par le biais d'une fonction locale :

```
def rendu2(S,v):
    def M(S,v):
        if v==0:
            return 0
```

```
        else:
            if v not in memo:
                mini=float('inf')
                for c in S:
                    if v>=c:
                        aux=M(S,v-c)
                        if aux<mini:
                            mini=aux
                memo[v]=1+mini
            return memo[v]
    memo={}
    return M(S,v)
```

Quelques essais :

```
>>> rendu2([2,3],1)
inf
>>> rendu2([1,3,4],6) # système non canonique
2
>>> rendu2(S,989) # S=[1,2,5,10,...,500]
9
>>> rendu2(S,1989)
...
RecursionError: maximum recursion depth exceeded in comparison
```

Cette version avec mémoïsation est bien meilleure mais demeure contrainte par la limitation de hauteur de la pile de récursions.

Approche ascendante

Une implémentation ascendante permet d'éviter les lenteurs des récursions multiples et de s'affranchir de la limitation de hauteur de piles de récursions :

```
def rendu3(S,v):
    T=[0]+v*[float('inf')]
    for j in range(1,v+1):
        mini=float('inf')
        for c in S:
            # si j peut être rendu avec une pièce (cas d'égalité)
            # alors T[0] est pris en compte d'où mini=0
            if j>=c and T[j-c]<mini:
                mini=T[j-c]
        T[j]=1+mini
    return T[v]
```

Quelques essais :

```
>>> rendu3([2,3],1)
inf
```

```
>>> rendu3([1,3,4],6)
2
>>> rendu3(S,1989) # S=[1,2,5,10,...,500]
11
>>> rendu3(S,111111)
225
```

Exercice : Déterminer la complexité temporelle et spatiale de `rendu3`.

Corrigé : La variable locale `T` est une liste de nombres de taille $v+1$ et les autres variables locales sont de tailles fixées. On effectue deux boucles imbriquées avec respectivement v et n passages. Par conséquent, la complexité spatiale est en $\mathrm{O}(v)$ et la complexité temporelle est en $\mathrm{O}(nv)$.

3 Deuxième formulation

Proposition 6. *Soit S_n un système de monnaie et v un montant à rendre. On a*

$$\mathrm{M}(\mathrm{S}_n, v) = \min_{0 \leqslant x \leqslant v/c_n} (x + \mathrm{M}(\mathrm{S}_{n-1}, v - xc_n))$$

avec $\mathrm{M}(\mathrm{S}_n, 0) = 0 \quad et \quad \forall v \in \mathbb{N}^* \qquad \mathrm{M}([\,], v) = +\infty$

Démonstration. On a

$$\begin{aligned}\mathrm{M}(\mathrm{S}_n, v) &= \min_{0 \leqslant x \leqslant v/c_n} \min \left\{ x + \sum_{i=1}^{n-1} x_i \ : \ (x_1, \ldots, x_{n-1}) \in \mathbb{N}^{n-1}, \ \sum_{i=1}^{n-1} x_i c_i = v - xc_n \right\} \\ &= \min_{0 \leqslant x \leqslant v/c_n} (x + \mathrm{M}(\mathrm{S}_{n-1}, v - xc_n))\end{aligned}$$

□

Approche descendante avec mémoïsation

On propose l'implémentation descendante avec mémoïsation :

```
def rendu4(S,u):
    def M(n,v):
        if v==0:
            return 0
        elif n==0:
            return float('inf')
        else:
            if (n,v) not in memo:
                mini=float('inf')
                for x in range(v//S[n-1]+1):
                    mini=min(mini,x+M(n-1,v-x*S[n-1]))
                memo[(n,v)]=mini
            return memo[(n,v)]
    memo={}
    return M(len(S),u)
```

La mémoïsation et la non création de sous-liste en argument de `M(n,v)` (on ne transmet que l'indice `n` en tant qu'indice d'arrêt) rend cette version plus performante qu'une implémentation récursive naïve. Toutefois, la complexité temporelle n'est pas satisfaisante. Le nombre de récursions est donné par

$$\sum_{0\leqslant x_n\leqslant v/c_n}\left(\sum_{0\leqslant x_{n-1}\leqslant (v-x_n)/c_{n-1}}\dots\left(\sum_{0\leqslant x_1\leqslant (v-x_n-\dots-x_2)/c_1}1\right)\dots\right)=\mathrm{O}(v^n)$$

Avec $\mathrm{S}=[1,2,5,10,20,50,100,200,500]$, on a $n=9$ d'où une complexité médiocre pour de grandes valeurs de v.

Exercice : Déterminer une implémentation ascendante suivant cette approche et la comparer à `rendu3`.

Corrigé : On propose :

```
def rendu(S,v):
    T=[0]+v*[float('inf')]
    for c in S:
        for j in range(1,v+1):
            mini=float('inf')
            for x in range(v//c+1):
                if x+T[j-x*c]<mini:
                    mini=x+T[j-x*c]
            T[j]=mini
    return T[v]
```

La complexité temporelle est en

$$\sum_{k=1}^{n}\sum_{i=1}^{v}\sum_{x=0}^{\lfloor v/c\rfloor}\mathrm{O}(1)=\mathrm{O}(nv^2)$$

ce qui est moins bon que celle de `rendu3` en $\mathrm{O}(nv)$.

4 Troisième formulation

Proposition 7. *Soit* S_n *un système de monnaie et* v *un montant à rendre. On a*

$$\mathrm{M}(\mathrm{S}_n,v)=\min\left(\mathrm{M}(\mathrm{S}_{n-1},v),1+\mathrm{M}(\mathrm{S}_n,v-c_n)\right)$$

avec $\mathrm{M}(\mathrm{S}_n,0)=0 \quad et \quad \forall v\in\mathbb{N}^* \qquad \mathrm{M}([\,],v)=+\infty$

Démonstration. On distingue selon que x_n est nul ou pas dans un n-uplet $(x_1,\dots,x_n)\in\mathbb{N}^n$ tel que $\sum_{i=1}^{n}x_ic_i=v$. Si $x_n=0$, on a donc $\sum_{i=1}^{n-1}x_ic_i=v$ et sinon, on a $\sum_{i=1}^{n}(x_i-\delta_{i,n})c_i=v-c_n$ d'où la relation

$$\mathrm{M}(\mathrm{S}_n,v)=\min\left(\mathrm{M}(\mathrm{S}_{n-1},v),1+\mathrm{M}(\mathrm{S}_n,v-c_n)\right)$$

□

Remarque : En vue d'une implémentation, on distinguera

$$\mathrm{M}(\mathrm{S}_n,v)=\begin{cases}\mathrm{M}(\mathrm{S}_{n-1},v)) & \text{si } v<c_n\\ \min\left(\mathrm{M}(\mathrm{S}_{n-1},v),1+\mathrm{M}(\mathrm{S}_n,v-c_n)\right) & \text{sinon}\end{cases}$$

On pose $\forall (i,j) \in [\![0\,;\, n]\!] \times [\![0\,;\, v]\!] \qquad m(i,j) = \mathrm{M}([x_1, \ldots, x_i], j)$

Corollaire 1. *On a*

$$m(0,0) = 0 \qquad \forall j \in [\![1\,;\, v]\!] \qquad m(0,j) = +\infty$$

et $\forall (i,j) \in [\![1\,;\, n]\!] \times [\![0\,;\, v]\!] \qquad m(i,j) = \mathrm{Min}\,(m(i-1,j), 1 + m(i, j-c_i))$

Démonstration. Conséquence immédiate de la proposition 7. □

Approche ascendante

Une implémentation descendante avec mémoïsation présente des faiblesses similaires à celles des versions précédemment exposées : lenteur, limitation de hauteur de pile. On propose l'implémentation ascendante suivante :

```
def rendu5(S,v):
    n=len(S)
    M=[[0] +[float('inf')]*v for k in range(n)]
    for i in range(n):
        for j in range(v+1):
            if j<S[i]:
                M[i][j]=M[i-1][j]
            else:
                M[i][j]=min(M[i-1][j],1+M[i][j-S[i]])
    return M[-1][-1]
```

On est performant pour le calcul de `rendu5(S,111111)` comme avec `rendu3`. Le coût spatial est plus élevé puisque toutes les étapes sont stockées. On pourrait se contenter de n'utiliser que deux lignes puisque la construction d'une nouvelle ligne ne requiert que la précédente mais nous verrons l'intérêt de l'accès à la totalité de la matrice ultérieurement.

```
>>> rendu5([2,3],1)
inf
>>> rendu5(S,1989) # S=[1,2,5,10,...,500]
11
>>> rendu5(S,111111)
225
```

5 Composition du rendu

On conserve les notations précédemment introduites. Désormais, on veut déterminer

$$\arg\min \left\{ \sum_{i=1}^{n} x_i \;:\; (x_1, \ldots, x_n) \in \mathbb{N}^n,\ \sum_{i=1}^{n} x_i c_i = v \right\}$$

La complexité résultant du parcours exhaustif est là encore en $\mathrm{O}(v^n)$. Comme pour le nombre de jetons, un algorithme glouton permet, pour un système de monnaie canonique, de résoudre le problème.

```
def argmin_rendu_glouton(S,v):
    ind=len(S)-1
    res=[]
    reste=v
    while reste>0 and ind>=0:
        if reste>=S[ind]:
            res.append(S[ind])
            reste-=S[ind]
        else:
            ind-=1
    if reste>0:
        return False
    return res
```

Quelques essais :

```
>>> argmin_rendu_glouton([2,3],1)
False
>>> argmin_rendu_glouton(S,1989) # S=[1,2,5,10,...,500]
[500, 500, 500, 200, 200, 50, 20, 10, 5, 2, 2]
>>> argmin_rendu_glouton(S,111111)
[500, ..., 500, 500, 500, 100, 10, 1]
>>> argmin_rendu_glouton([1,3,4],6)
# système non canonique, le glouton échoue sur 6=3+3
[4, 1, 1]
```

Y compris avec un système canonique, il n'y a pas unicité d'un rendu optimal. Ainsi, avec le système[3] $\mathrm{S}=[1,3,5]$, on a

$$6=5+1=3+3 \quad 7=5+1+1=3+3+1 \quad 8=5+3 \quad 9=5+3+1 \quad \textit{etc.}$$

On a vu que pour la détermination du nombre de jetons, les versions ascendantes avec variation d'un seul des deux paramètres simultanément (soit n, soit v) étaient les plus performantes. On peut modifier/compléter ces programmes pour garder la mémoire des pièces intervenant lors des minimisations.

On peut adapter le programme `rendu3` implémentant la formulation

$$\mathrm{M}(\mathrm{S}_n,v)=1+\min_{1\leqslant k\leqslant n}\mathrm{M}(\mathrm{S}_n,v-c_k)$$

avec $\mathrm{M}(\mathrm{S}_n,0)=0 \quad \text{et} \quad \forall v\in[\![\,1\,;\,\min(\mathrm{S}_n)-1\,]\!] \qquad \mathrm{M}(\mathrm{S}_n,v)=+\infty$

Pour cela, on garde la mémoire de la dernière pièce intervenant lors de la minimisation. Il suffit ensuite de remonter le parcours des rendus successifs en lisant la dernière pièce rendue à chaque étape. On ne peut réaliser cette reconstitution de manière ascendante puisqu'il faut partir de la dernière pièce du rendu pour amorcer la remontée.

```
def rendu_tab1(S,v):
    memo_nb=[0]+[float('inf')]*v
```

3. Vérification du caractère canonique avec l'algorithme de Kozen-Zaks.

```
    memo_piece=[0]+[None]*v
    for j in range(1,v+1):
        mini=float('inf')
        piece=None
        for c in S:
            if j>=c and memo_nb[j-c]<mini:
                mini,piece=memo_nb[j-c],c
        memo_nb[j]=1+mini
        memo_piece[j]=piece
    return memo_nb,memo_piece


def argmin_rendu1(S,v):
    M_nb,M_piece=rendu_tab1(S,v)
    res=[]
    if M_piece[v]==None:
        return False
    while M_nb[v]>0:
        res.append(M_piece[v])
        v-=res[-1]
    return res
```

Quelques essais :

```
>>> argmin_rendu1([2,3],1)
False
>>> argmin_rendu1(S,1989) # S=[1,2,5,10,...,500]
[2, 2, 5, 10, 20, 50, 200, 200, 500, 500, 500]
>>> argmin_rendu1(S,111111)
[1, 10, 100, 500, 500, ..., 500]
```

Exercice : Déterminer la complexité temporelle de `argmin_rendu1`.

Corrigé : La complexité temporelle de `rendu_tab1` est en $\mathrm{O}(nv)$. Ensuite, la boucle `while` réalise la remontée des indices de `M_nb`, liste de taille $v+1$, en au plus v étapes avec des opérations à coût constant. La complexité temporelle est donc en

$$\mathrm{O}(nv) + \mathrm{O}(v) = \mathrm{O}(nv)$$

Avec la troisième formulation, on peut exploiter la matrice $\mathrm{RM} = \big(m(i,j)\big)_{1\leqslant i\leqslant n, 0\leqslant j\leqslant v}$ où $(i,j) \mapsto m(i,j)$ est la fonction introduite à la section 4. En remontant les étapes dans cette matrice RM, on peut reconstituer un rendu de monnaie.

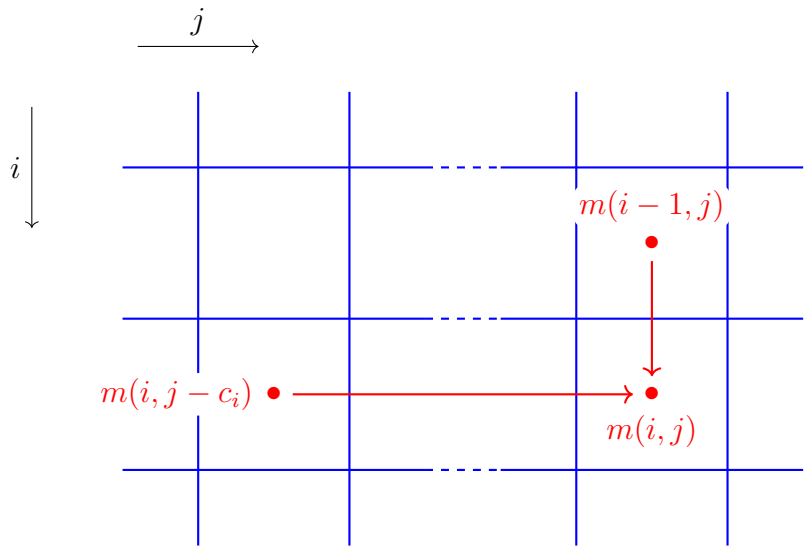


FIGURE 18 – Transitions pour le rendu de monnaie

Cette reconstitution ne peut être réalisée de manière ascendante : il faut partir du nombre de jetons $m(n,v)$ pour effectuer cette remontée.

```
def rendu_tab2(S,v):
    n=len(S)
    M=[[0] +[float('inf')]*v for k in range(n)]
    for i in range(n):
        for j in range(v+1):
            if j<S[i]:
                M[i][j]=M[i-1][j]
            else:
                M[i][j]=min(M[i-1][j],1+M[i][j-S[i]])
    return M

def argmin_rendu2(S,v):
    tab=rendu_tab2(S,v)
    i,j=len(S)-1,v
    if tab[i][j]==float('inf'):
        return []
    res=[]
    while j>0:
        if tab[i][j]==1+tab[i][j-S[i]]:
            j-=S[i]
            res.append(S[i])
        else:
            i-=1
    return res
```

Compositions de rendu de monnaie :

	0	1	2	3	4	5
3	0	∞	∞	1	∞	∞
7	0	∞	∞	1	∞	∞

Rendu de 5 avec $[3, 7]$: $\varnothing$

	0	1	2	3	4	5	6
1	0	1	2	3	4	5	6
2	0	1	1	2	2	3	3
5	0	1	1	2	2	1	2
10	0	1	1	2	2	1	2

Rendu de 6 avec $[1, 2, 5, 10]$: $[5, 1]$

En remontant la matrice T et en plaçant dans une file chaque trajet en construction conforme pour réaliser un rendu optimal, on peut obtenir tous les rendus optimaux possibles.

Compositions de rendus de monnaie :

	0	1	2	3	4	5	6
1	0	1	2	3	4	5	6
3	0	1	2	1	2	3	2
5	0	1	2	1	2	1	2

Rendu de 6 avec $[1, 3, 5]$: $[5, 1]$

	0	1	2	3	4	5	6
1	0	1	2	3	4	5	6
3	0	1	2	1	2	3	2
5	0	1	2	1	2	1	2

Rendu de 6 avec $[1, 3, 5]$: $[3, 3]$

V Problème du sac-à-dos

1 Introduction

On cherche à remplir un sac-à-dos d'une capacité maximale notée C (poids maximal supporté par le sac-à-dos) avec des objets caractérisés par leur utilité et leur poids en maximisant le remplissage, c'est-à-dire en maximisant la somme des utilités des objets placés dans le sac. Tous les poids considérés sont des entiers.

On note $\mathrm{X}_n = [(v_1, p_1), \ldots, (v_n, p_n)]$ la liste des objets considérés avec n entier non nul. Pour $k \in [\![1\,;\, n]\!]$, la valeur v_k désigne l'utilité du k-ième objet et p_k désigne son poids. On suppose C entier et

$$\forall k \in [\![1\,;\, n]\!] \qquad v_k \geqslant 0 \quad \text{et} \quad p_k \in \mathbb{N}^*$$

Le *problème du sac-à-dos* (*KP*, *Knapsack Problem*) consiste à réaliser

$$\mathrm{KP}(\mathrm{X}_n, \mathrm{C}) = \mathrm{Max}\left\{ \sum_{k=1}^{n} x_k v_k,\ (x_1, \ldots, x_n) \in \{0,1\}^n \ :\ \sum_{k=1}^{n} x_k p_k \leqslant \mathrm{C} \right\}$$

Le parcours exhaustif de l'ensemble des solutions est potentiellement le parcours de $\{0,1\}^n$ qui est de taille 2^n. Il est donc pertinent d'envisager des stratégies de résolution plus performantes.

On note $\mathrm{X}_i = [(v_1, p_1), \ldots, (v_i, p_i)]$ pour $i \in [\![1\,;\, n]\!]$ et $\mathrm{X}_0 = \varnothing$. On a le résultat suivant :

Proposition 8. *On a*

$$\mathrm{KP}(\mathrm{X}_n, \mathrm{C}) = \begin{cases} 0 & \text{si } n = 0 \\ \mathrm{KP}(\mathrm{X}_{n-1}, \mathrm{C}) & \text{si } p_n > \mathrm{C} \\ \max(\mathrm{KP}(\mathrm{X}_{n-1}, \mathrm{C}), v_n + \mathrm{KP}(\mathrm{X}_{n-1}, \mathrm{C} - p_n)) & \text{sinon} \end{cases}$$

Démonstration. On note

$$\Lambda(\mathrm{X}_n, \mathrm{C}) = \left\{ \sum_{k=1}^{n} x_k v_k,\ (x_1, \ldots, x_n) \in \{0,1\}^n \ :\ \sum_{k=1}^{n} x_k p_k \leqslant \mathrm{C} \right\}$$

Comme une somme vide est nulle, on a $\Lambda(\mathrm{X}_0, \mathrm{C}) = \{0\}$. Soit n entier non nul. Si $p_n > \mathrm{C}$, alors l'objet numéro n ne peut être emmené. Sinon, on distingue selon qu'on l'emmène ou pas. Si on ne l'emmène pas, on passe à la configuration $(\mathrm{X}_{n-1}, \mathrm{C})$. Si on l'emmène, on tient compte de l'utilité de l'objet et on déduit son poids de la capacité maximale restante. Ainsi, on a

$$\Lambda(\mathrm{X}_n, \mathrm{C}) = \begin{cases} \Lambda(\mathrm{X}_{n-1}, \mathrm{C}) & \text{si } p_n > \mathrm{C} \\ \Lambda(\mathrm{X}_{n-1}, \mathrm{C}) \bigsqcup (v_n + \Lambda(\mathrm{X}_{n-1}, \mathrm{C} - p_n)) & \text{sinon} \end{cases}$$

Le résultat suit. □

Remarque : En optant pou la convention $\mathrm{Max}\ \varnothing = -\infty$, on peut aussi formuler la relation ainsi

$$\mathrm{KP}(\mathrm{X}_n, \mathrm{C}) = \begin{cases} -\infty & \text{si } \mathrm{C} < 0 \\ 0 & \text{si } n = 0 \\ \max\left(\mathrm{KP}(\mathrm{X}_{n-1}, \mathrm{C}), \mathrm{KP}(\mathrm{X}_{n-1}, \mathrm{C} - p_n) + v_n\right) & \text{sinon} \end{cases}$$

2 Approche descendante

Approche descendante sans mémoïsation

On propose l'implémentation descendante :

```
def KP_rec(X,C):
    if len(X)==0:
        return 0
    X_extr=X[:-1]
    v,p=X[-1]
    if p>C:
        return KP_rec(X_extr,C)
    else:
        return max(KP_rec(X_extr,C),KP_rec(X_extr,C-p)+v)
```

ou avec la deuxième formulation :

```
def KP_rec(X,C):
    if C<0:
        return -float('inf')
    if len(X)==0:
        return 0
    X_extr=X[:-1]
    v,p=X[-1]
    return max(KP_rec(X_extr,C),KP_rec(X_extr,C-p)+v)
```

On testera les différentes versions de résolution du problème du sac-à-dos avec la liste d'objets :

```
X=((1,2),(2,5),(3,7),(7,12),(10,9),(11,15),(1,1),(2,1))
```

On saisit `X` en tant que `tuple` qui est un type hachable (pour la suite). On obtient par exemple :

```
>>> KP_rec(X,5)
4
# une composition possible : [(2, 1), (1, 1), (1, 2)]
>>> KP_rec(X,11)
13
# une composition possible : [(2, 1), (1, 1), (10, 9)]
```

On note $\mathrm{R}(n, \mathrm{C})$ le nombre de récursions lors d'un appel de `KP_rec(X,C)` avec $\mathrm{X} = \mathrm{X}_n$. On admet (hypothèse raisonnable) la croissante de $n \mapsto \mathrm{R}(n, \mathrm{C})$ et $\mathrm{C} \mapsto \mathrm{R}(n, \mathrm{C})$. Dans le pire des cas, avec la décroissance la plus faible sur le deuxième argument, on a la relation

$$\mathrm{R}(n, \mathrm{C}) = 1 + \mathrm{R}(n-1, \mathrm{C}) + \mathrm{R}(n-1, \mathrm{C}-1) \geqslant 1 + 2\mathrm{R}(n-1, \mathrm{C}-1)$$

Avec $\alpha = -1$ solution de $\alpha = 2\alpha + 1$, on trouve

$$\forall n \in \mathbb{N} \qquad \mathrm{R}(n, \mathrm{C}) - \alpha \geqslant 2\left(\mathrm{R}(n-1, \mathrm{C}-1) - \alpha\right)$$

d'où

$$\mathrm{R}(n, \mathrm{C}) \geqslant \alpha + 2^{\min(n, \mathrm{C})} \mathrm{C}^{\mathrm{te}}$$

Si $\mathrm{C} = n$, on retrouve une complexité de l'ordre de 2^n qui correspond à un parcours exhaustif de $\Lambda(\mathrm{X}_n, \mathrm{C})$ ce qui est évidemment très lourd.

Exercice : Expliquer pourquoi l'implémentation qui suit est fausse :

```
def KP_wrong(X,C):
    if len(X)==0:
        return 0
    if C<0:
        return -float('inf')
    X_extr=X[:-1]
    v,p=X[-1]
    return max(KP_wrong(X_extr,C),KP_wrong(X_extr,C-p)+v)
```

Corrigé : Si Card $\mathrm{X} = 1$ et $\mathrm{C} - p < 0$, le résultat est $\max(0, 0 + v) = v$ au lieu d'être 0.

Approche descendante avec mémoïsation

On suppose que la liste d'objets `X` est au format `tuple` pour le hachage. On améliore l'approche précédente avec de la mémoïsation :

```
def KP_memo(X,C):
    def KP(X,C):
        if (X,C) not in memo:
            if len(X)==0:
                return 0
            v,p=X[-1]
            X_extr=X[:-1]
```

```
            if p>C:
                res=KP(X_extr,C)
            else:
                res=max(KP(X_extr,C),KP(X_extr,C-p)+v)
            memo[(X,C)]=res
        return memo[(X,C)]
    memo={}
    return KP(X,C)
```

ou avec la deuxième formulation :

```
def KP_memo(X,C):
    def KP(X,C):
        if (X,C) not in memo:
            if C<0:
                return -float('inf')
            if len(X)==0:
                return 0
            v,p=X[-1]
            X_extr=X[:-1]
            memo[(X,C)]=max(KP(X_extr,C),KP(X_extr,C-p)+v)
        return memo[(X,C)]
    memo={}
    return KP(X,C)
```

Cette version avec mémoïsation est meilleure que la précédente mais est contrainte par la limitation liée à la hauteur de la pile d'appels récursifs.

3 Approche ascendante

Soit une liste d'objets $\mathrm{X}_n = [(v_1, p_1), \ldots, (v_n, p_n)]$ de taille n et une capacité maximale entière C. On précise que l'on note $\mathrm{X}_i = [(v_1, p_1), \ldots, (v_i, p_i)]$ pour $i \in [\![1\,;\, n]\!]$ avec $\mathrm{X}_0 = \varnothing$. on définit la matrice $\mathrm{T} = \left(t_{i,j}\right)_{(i,j)\in[\![0\,;\, n]\!]\times[\![0\,;\, \mathrm{C}]\!]}$ par

$$\forall (i,j) \in [\![0\,;\, n]\!] \times [\![0\,;\, \mathrm{C}]\!] \qquad t_{i,j} = \mathrm{KP}(\mathrm{X}_i, j)$$

Proposition 9. *On a les relations*

$$\forall j \in [\![0\,;\, \mathrm{C}]\!] \qquad t_{0,j} = 0$$

et

$$\forall (i,j) \in [\![1\,;\, n]\!] \times [\![0\,;\, \mathrm{C}]\!] \qquad t_{i,j} = \begin{cases} t_{i-1,j} & \text{si } j < p_i \\ \max(t_{i-1,j}, t_{i-1,j-p_i} + v_i) & \text{sinon} \end{cases}$$

Démonstration. Conséquence immédiate de la proposition 8. □

On en déduit l'implémentation ascendante suivante :

```
def KP_asc(X,C):
    n=len(X)
    tab=[[0]*(C+1) for i in range(n+1)]
```

```
    for i in range(1,n+1):
        for j in range(C+1):
            v,p=X[i-1]
            if j>=p:
                tab[i][j]=max(tab[i-1][j],tab[i-1][j-p]+v)
            else:
                tab[i][j]=tab[i-1][j]
    return tab[n][C]
```

Quelques essais :

```
>>> KP_asc(X,5)
4
>>> KP_asc(X,11)
13
```

Exercice : Déterminer la complexité temporelle et spatiale de `KP_asc`.

Corrigé : La variable `tab` est une liste de $n+1$ listes d'entier de taille $C+1$. Les autres variables sont en nombre fixe et de tailles fixées. On effectue deux boucles imbriquées avec repsectivement n et C passages puis des opérations à coût constant. On en déduit une complexité spatiale et temporelle en $O(nC)$.

4 Extraction d'une composition

En conservant l'intégralité de la matrice T et pas seulement $t_{n,C}$, on peut, en remontant les étapes depuis la cible (n, C) vers un bord, reconstituer une composition optimale du sac.

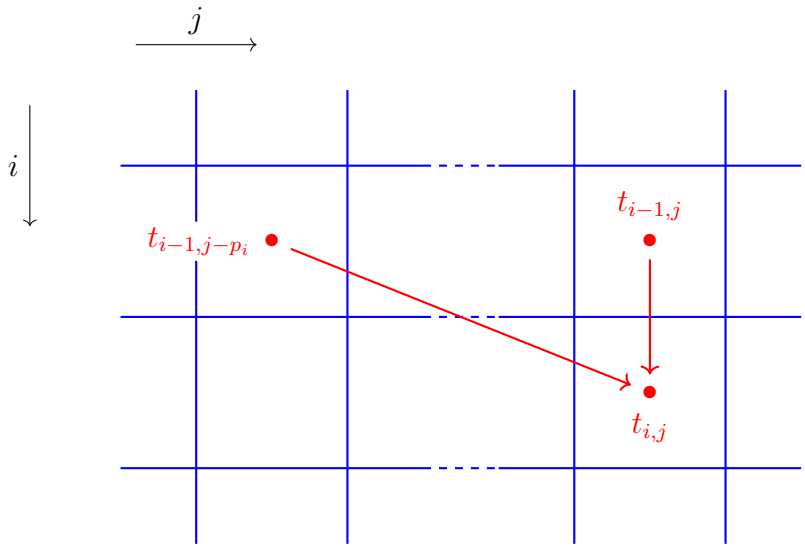


FIGURE 19 – Transitions pour un remplissage de sac

Étant donné $(i, j) \in [\![1 ; n]\!] \times [\![1 ; C]\!]$, on détermine une position parmi $(i-1, j)$ et $(i-1, j-p_i)$ depuis laquelle la transition est possible pour l'obtention de $t_{i,j}$. Il y a au moins une position satisfaisante (peut-être plusieurs) par définition de la matrice T.

Cette reconstitution ne peut être réalisée de manière ascendante : il faut partir de la valeur maximale $t_{n,C}$ (étape finale de `KP_tab`) pour effectuer cette remontée.

```
def KP_tab(X,C):
    n=len(X)
    tab=[[0]*(C+1) for i in range(n+1)]
    for i in range(1,n+1):
        for j in range(C+1):
            v,p=X[i-1]
            if j>=p:
                tab[i][j]=max(tab[i-1][j],tab[i-1][j-p]+v)
            else:
                tab[i][j]=tab[i-1][j]
    return tab
```

```
def KP_comp(X,C):
    tab=KP_tab(X,C)
    n=len(X)
    i,j=n,C
    res=[]
    while i>0 and  j>0:
        if tab[i-1][j]==tab[i][j]:
            i-=1
        else:
            v,p=X[i-1]
            i-=1
            j-=p
            res.append((v,p))
    return res
```

Quelques essais :

```
>>> KP_comp(X,5)
[(2, 1), (1, 1), (1, 2)]
>>> KP_comp(X,11)
[(2, 1), (1, 1), (10, 9)]
```

5 Extraction de toutes les compositions

En remontant la matrice T et en plaçant dans une file chaque trajet en construction conforme pour réaliser une composition optimale de sac, on peut obtenir toutes les compositions optimales possibles.

```
def KP_all(X,C):
    n=len(X)
    tab=KP_tab(X,C)
    file_KP=deque()
    res=[]
```

```
    file_KP.append([(n,C),[]])
    while len(file_KP)>0:
        coord,contenu=file_KP.popleft()
        i,j=coord
        if i==0 or j==0:
            res.append(contenu)
        else:
            v,p=X[i-1]
            if tab[i-1][j]==tab[i][j]:
                file_KP.append([(i-1,j),contenu])
            if j-p>=0 and tab[i-1][j-p]+v==tab[i][j]:
                # pas de append sur contenu
                # sinon modification sur tous les colocaires de contenu
                file_KP.append([(i-1,j-p),contenu+[(v,p)]])
    return res
```

Quelques essais :

```
>>> KP_all(X,8)
[[(2, 1), (3, 7)], [(2, 1), (2, 5), (1, 2)], [(2, 1), (1, 1), (2, 5)]]
>>> KP_all(X,15)
[[(2, 1), (10, 9), (2, 5)], [(2, 1), (1, 1), (10, 9), (1, 2)]]
```

VI Partition équilibrée

1 Présentation

Pour L une liste d'entiers, on note $s(\mathrm{L}) = \sum_{x\in \mathrm{L}} x$.

Définition 13. *Soit* L *une liste d'entiers. On appelle* partition équilibrée *de* L *un couple de listes* $(\mathrm{L}_1, \mathrm{L}_2)$ *tel que* $\mathrm{sorted}(\mathrm{L}) = \mathrm{sorted}(\mathrm{L}_1 + \mathrm{L}_2)$ *avec* $|s(\mathrm{L}_1) - s(\mathrm{L}_2)|$ *minimale.*

Remarques : (1) Cette notion est bien définie. L'ensemble

$$\{|s(\mathrm{L}_1) - s(\mathrm{L}_2)|\,,\ \mathrm{L}_1 \subset \mathrm{L},\ \mathrm{L}_2 \subset \mathrm{L}\ :\ \mathrm{sorted}(\mathrm{L}_1 + \mathrm{L}_2) = \mathrm{sorted}(\mathrm{L})\}$$

est une partie non vide de $\mathbb{N}$ (considérer $(\mathrm{L}_1, \mathrm{L}_2) = ([\,]\,, \mathrm{L})$ par exemple) et admet donc un plus petit élément. Un couple correspondant est une partition équilibrée.
(2) Soit $\mathrm{L} = [x_0, \ldots, x_{n-1}]$ une liste d'entiers et $(\mathrm{L}_1, \mathrm{L}_2)$ une partition équilibrée. Notant $\mathrm{L}_1 = \left[x_{i_1}, \ldots, x_{i_p}\right]$ et $\mathrm{L}_2 = \left[x_{j_1}, \ldots, x_{j_q}\right]$, on a l'union disjointe

$$[i_1, \ldots, i_p] \sqcup [j_1, \ldots, j_q] = [\![\, 0\,;\, n-1\,]\!]$$

d'où la notion de *partition* équilibrée.

Exemple : Avec la liste $\mathrm{L} = [3, 7, 6, 2, 3, 4, 7, 8, 4, 2, 9, 8, 7, 3, 2, 3, 3, 1, 6, 4]$, le couple

$$(\mathrm{L}_1, \mathrm{L}_2) = ([2, 4, 8, 7, 4, 3, 2, 6, 7, 3]\,, [9, 8, 7, 3, 2, 3, 3, 1, 6, 4])$$

est une partition équilibrée puisque $s(\mathrm{L}_1) = s(\mathrm{L}_2) = 46$.

Remarque : Il n'y a pas unicité (sans tenir compte de l'ordre au sein des listes) d'une partition équilibrée d'une liste d'entiers. Par exemple, avec la liste $\mathrm{L} = [1, 1, 2, 3]$, les couples

$$([1, 1, 2], [3]) \quad \text{et} \quad ([1, 2], [1, 3])$$

sont des partitions équilibrées.

Proposition 10. *Soit* L *une liste d'entiers. La détermination d'une partition équilibrée de* L *équivaut à la détermination de*

$$\arg\min \left\{ \left| s(\mathrm{K}) - \frac{s(\mathrm{L})}{2} \right|, \ \mathrm{K} \subset \mathrm{L} \right\}$$

Démonstration. Soit un couple de listes $(\mathrm{L}_1, \mathrm{L}_2)$ tel que $\mathrm{sorted}(\mathrm{L}) = \mathrm{sorted}(\mathrm{L}_1 + \mathrm{L}_2)$. Avec l'égalité $s(\mathrm{L}) = s(\mathrm{L}_1) + s(\mathrm{L}_2)$, on constate que minimiser $|s(\mathrm{L}_1) - s(\mathrm{L}_2)|$ est équivalent à minimiser $|2s(\mathrm{L}_1) - s(\mathrm{L})|$. Pour $\mathrm{L}_1 \subset \mathrm{L}$ qui minimise cette quantité, on construit la partition en plaçant dans L_2 les éléments dont les indices sont ceux des éléments de L qui ne sont pas indices des éléments de L_1. Le résultat suit. □

2 Approches gloutonnes

On peut envisager l'algorithme glouton suivant :

- on ordonne (ou pas) les éléments de `L` ;
- on initialise `L1` et `L2` à `[ ]` ;
- on place le premier élément de `L` dans `L1` puis on place les suivants dans `L2` jusqu'à ce que `sum(L1)<=sum(L2)` ;
- on itère ce procédé tant que la liste `L` n'a pas été parcourue intégralement.

```
def part_equib_glouton(S):
    S1,S2=[],[]
    tab=sorted(S)
    n=len(tab)
    ind=0
    while ind <n:
        S1.append(tab[ind])
        ind+=1
        while ind<n and sum(S2)<sum(S1):
            S2.append(tab[ind])
            ind+=1
    return S1,S2
```

Des résultats parfois convenables :

```
>>> L=list(rd.randint(1,10,20))
>>> L
[5, 9, 8, 1, 3, 1, 3, 1, 3, 8, 6, 7, 2, 6, 9, 7, 8, 7, 8, 3]
>>> a,b=part_equib_glouton(L)
>>> a,b
([1, 1, 3, 3, 5, 6, 7, 8, 8, 9], [1, 2, 3, 3, 6, 7, 7, 8, 8, 9])
>>> sum(a),sum(b)
(51, 54)
```

et parfois très insuffisants :

```
>>> L=[1,1,1,1,4]
>>> a,b=part_equib_glouton(L)
>>> a,b
([1, 1, 4], [1, 1])
>>> sum(a),sum(b)
(6, 2)
```

On peut envisager l'autre algorithme glouton qui suit :

- on ordonne la liste `L` ;
- on initialise `L1` à `[ ]` ;
- on choisit le plus grand élément `x in L` tel que `x+sum(L1)<sum(L)/2` et on le rajoute à `L1` ;
- on répète ce procédé tant qu'on peut.

```
def part_equib_glouton2(S):
    s=sum(S)
    tab=sorted(S)
    n=len(tab)
    S1=[]
    pasfini=True
    while pasfini:
        ind=0
        while ind<len(tab) and tab[ind]+sum(S1)<s/2:
            ind+=1
        if ind==0:
            pasfini=False
        else:
            ind-=1
            S1.append(tab[ind])
            del tab[ind]
    return S1,tab
```

Là encore, les résultats ne sont pas toujours au rendez-vous :

```
>>> L=[5, 9, 3, 8, 2, 5]
>>> a,b=part_equib_glouton2(L)
>>> sum(a),sum(b)
(14, 18)
```

3 Approche descendante

Proposition 11. *Soit* L *une liste d'entiers et* $a \in \mathrm{L}$. *On a*

$$\arg\min\left\{\left|s(\mathrm{K})-\frac{s(\mathrm{L})}{2}\right|,\ \mathrm{K}\subset\mathrm{L}\right\} =$$
$$\arg\min\left(\min\left\{\left|s(\mathrm{K})+a-\frac{s(\mathrm{L})}{2}\right|,\ \mathrm{K}\subset\mathrm{L}\smallsetminus\{a\}\right\},\min\left\{\left|s(\mathrm{K})-\frac{s(\mathrm{L})}{2}\right|,\ \mathrm{K}\subset\mathrm{L}\smallsetminus\{a\}\right\}\right)$$

avec $$\arg\min\left\{\left|s(\mathrm{K})-\frac{s([\,])}{2}\right|,\ \mathrm{K}\subset[\,]\right\}=[\,]$$

Démonstration. Pour $\mathrm{K}\subset\mathrm{L}$, on distingue selon que a appartient à K ou pas. Si $a\in\mathrm{K}$, notant $\mathrm{K}'=\mathrm{K}\smallsetminus\{a\}$, on a $\mathrm{K}'\subset\mathrm{L}\smallsetminus\{a\}$ et $s(\mathrm{K})=s(\mathrm{K}')+a$ et sinon $a\notin\mathrm{K}$ auquel cas $\mathrm{K}\subset\mathrm{L}\smallsetminus\{a\}$. L'égalité attendue s'ensuit. □

Approche descendante sans mémoïsation

On en déduit l'implémentation récursive suivante :

```
def part_rec(L,s):
    if len(L)==0:
        return []
    a=L[0]
    p1=part_rec(L[1:],s-a)
    p2=part_rec(L[1:],s)
    s1=a+sum(p1)
    s2=sum(p2)
    if abs(s1-s)<abs(s2-s):
        return [a] + p1
    else:
        return p2


def part(L):
    L1=part_rec(L,sum(L)/2)
    L2=L[:]
    for x in L1:
        L2.remove(x)
    return L1,L2
```

Notant n la taille de `L`, la complexité temporelle de `part_rec` vérifie une relation de la forme

$$\mathrm{T}(n)=2\mathrm{T}(n-1)+\mathrm{O}(n)$$

d'où
$$\begin{aligned}\mathrm{T}(n)&=2^n\mathrm{T}(0)+2^n\sum_{k=1}^{n}\left[\frac{\mathrm{T}(k)}{2^k}-\frac{\mathrm{T}(k-1)}{2^{k-1}}\right]\\&=2^n\mathrm{T}(0)+2^n\sum_{k=1}^{n}\mathrm{O}\left(\frac{k}{2^k}\right)=2^n\left(\mathrm{T}(0)+\mathrm{O}\left(\sum_{k=1}^{n}\frac{k}{2^k}\right)\right)=\mathrm{O}(2^n)\end{aligned}$$

ce qui est désastreux pour de grandes valeurs de n.

Approche descendante avec mémoïsation

Les appels récursifs vont donner lieu à des calculs redondants. On peut améliorer nettement les performances du programme avec de la mémoïsation dans un dictionnaire par une fonction locale. Les listes ne sont pas hachables et il faut donc convertir la donnée `(L,s)` en chaîne pour la hacher :

```
def part_memo(L):
    def part_rec_memo(L,s):
        id_L_s=str((L,s))
        if id_L_s not in memo:
            if len(L)==0:
                return []
            a=L[0]
            p1=part_rec_memo(L[1:],s-a)
            p2=part_rec_memo(L[1:],s)
            s1=a+sum(p1)
            s2=sum(p2)
            if abs(s1-s)<abs(s2-s):
                res=[a]+p1
            else:
                res=p2
            memo[id_L_s]=res
        return memo[id_L_s]
    memo={}
    L1=part_rec_memo(L,sum(L)/2)
    L2=L[:]
    for x in L1:
        L2.remove(x)
    return L1,L2
```

Quelques essais :

```
>>> a,b=part_memo(list(rd.randint(1,20,100)))
>>> sum(a),sum(b)
(506, 507)
>>> a,b=part_memo(list(rd.randint(1,100,1000)))
...
RecursionError: maximum recursion depth exceeded while calling a Python object
```

La version avec mémoïsation fait mieux mais reste contrainte par la hauteur de pile de récursions.

4 Approche ascendante

On propose une autre approche, plus efficace mais moins intuitive. On note $\mathrm{L} = [x_1, \ldots, x_n]$ et $m = \left\lfloor \dfrac{s(\mathrm{L})}{2} \right\rfloor$. On va construire une matrice de booléens $\mathrm{T} = \left(t_{i,j}\right)_{0 \leqslant i \leqslant n, 0 \leqslant j \leqslant m}$ avec

$$\forall (i,j) \in [\![0\,;\, n]\!] \times [\![0\,;\, m]\!] \qquad t_{i,j} = \begin{cases} \texttt{True} & \text{si } \exists \mathrm{K} \subset \{x_1, \dots, x_i\} \mid \sum_{x \in \mathrm{K}} x = j \\ \texttt{False} & \text{sinon} \end{cases}$$

Proposition 12. *On a*

$$\forall i \in [\![0\,;\, m]\!] \qquad t_{i,0} = \texttt{True} \qquad \forall j \in [\![1\,;\, m]\!] \qquad t_{0,j} = \texttt{False}$$

et

$$\forall (i,j) \in [\![1\,;\, n]\!] \times [\![0\,;\, m]\!] \qquad t_{i,j} = t_{i-1,j} \vee t_{i-1,j-x_i}$$

Démonstration. Pour $j = 0$, la partie $\varnothing$ permet de réaliser la somme nulle d'où $t_{i,0} = \texttt{True}$ pour tout $i \in [\![0\,;\, n]\!]$. Pour $i = 0$, la somme porte sur l'ensemble vide et est donc nulle d'où $t_{0,j} = \texttt{False}$ pour tout $j \in [\![1\,;\, m]\!]$. Pour $(i,j) \in [\![1\,;\, n]\!] \times [\![1\,;\, m]\!]$, on a

$$t_{i,j} = \texttt{True} \iff \exists \mathrm{K} \subset \{x_1, \dots, x_i\} \mid \sum_{x \in \mathrm{K}} x = j$$

Or, pour une telle partie $\mathrm{K} \subset \{x_1, \dots, x_i\}$, celle-ci peut contenir x_i ou pas d'où

$$\begin{aligned} t_{i,j} = \texttt{True} &\iff \exists \mathrm{K} \subset \{x_1, \dots, x_{i-1}\} \mid \sum_{x \in \mathrm{K}} x = j - x_i \quad \text{ou} \quad j \\ &\iff t_{i-1,j} = \texttt{True} \quad \text{ou} \quad t_{i-1,j-x_i} = \texttt{True} \end{aligned}$$

d'où

$$\forall (i,j) \in [\![1\,;\, n]\!] \times [\![0\,;\, m]\!] \qquad t_{i,j} = t_{i-1,j} \vee t_{i-1,j-x_i}$$

□

On en déduit l'implémentation ascendante suivante :

```
def tab_flag(L):
    """construire tab (n+1)*(m+1) avec n=len(L), m=sum(L)//2 tel que
    tab[i][j]= True s'il existe K inclus dans L pour sum_{x in K} x=j
                False sinon"""
    n,m=len(L),sum(L)//2
    tab=[[True]+[False for j in range(m)] for i in range(n+1)]
    for i in range(1,n+1):
        for j in range(1,m+1):
            if j>=L[i-1]:
                tab[i][j]=tab[i-1][j-L[i-1]] or tab[i-1][j]
            else:
                tab[i][j]=tab[i-1][j]
    # il se peut qu'on n'atteigne pas demi_s
    # on détermine l'indice max réalisant la plus grande somme <= demi_s
    ind=m
    while not tab[n][ind]:
        ind-=1
    return tab,ind
```

Ensuite, par remontées successives dans ce tableau, on va pouvoir constituer une sous-liste répondant au problème.

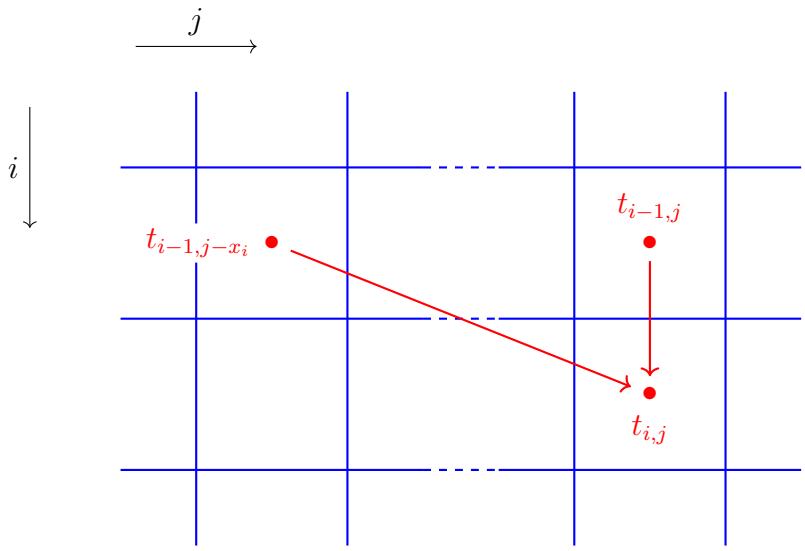


FIGURE 20 – Transitions pour une partition équilibrée

On amorce la construction d'une suite d'indices $(j_k)_k$, $(i_k)_k$ avec

$$i_{-1} = n \qquad j_0 = \max\{j \in [\![0\,;m]\!] \mid t_{n,j} = \texttt{True}\}$$

et pour k entier, si $j_k > 0$, on définit

$$i_k = \min\{i \in [\![1\,;i_{k-1}]\!] \mid t_{i,j_k} = \texttt{True}\} \quad \text{et} \quad j_{k+1} = j_k - x_{i_k}$$

Justifions que la suite $(i_k)_k$ est bien définie. On a $t_{i_k,j_k} = \texttt{True}$ d'où $t_{i_k,j_{k+1}} = \texttt{True}$ puisqu'on peut réaliser la somme j_{k+1} en omettant x_{i_k} dans la somme. L'entier i_{k+1} est donc bien défini comme minimum d'une partie non vide de $\mathbb{N}^*$ ce qui prouve l'existence de la suite $(i_k)_k$. Par ailleurs, par minimalité de i_k, on a $t_{i_k-1,j_k} = \texttt{False} \neq t_{i_k,j_k}$ ce qui prouve $x_{i_k} > 0$ et par conséquent, la suite $(j_k)_k$ décroît strictement ce qui prouve que la suite $(j_k)_k$ atteint zéro (sinon, elle stationne sur un entier non nul ce qui contredit sa décroissance stricte). Ainsi, l'algorithme se termine.

```
def part_asc(L):
    L1=[]
    tab,j=tab_flag(L)
    i=len(L)
    while j>0:
        while i>0 and tab[i][j]:
            i-=1
        j=j-L[i]
        if j>=0:
            L1.append(L[i])
        i-=1
    L2=L[:]
    for x in L1:
        L2.remove(x)
    return L1,L2
```

Un essai un peu ambitieux :

```
>>> L=list(rd.randint(1,100,1000))
>>> a,b=part_asc(L)
>>> sum(a),sum(b)
(24441, 24441)
```

On a un résultat dans un délai très raisonnable.

Chemin associé à une partition équilibrée

	0	1	2	3	4	5	6	7	8	9	10
0	True	False	False	False	False	False	False	False	False	False	False
1	True	True	False	False	False	False	False	False	False	False	False
1	True	True	True	False	False	False	False	False	False	False	False
5	True	True	True	False	False	True	True	True	False	False	False
2	True	True	True	True	True	True	True	True	True	True	False
3	True	True	True	True	True	True	True	True	True	True	True
8	True	True	True	True	True	True	True	True	True	True	True

$$\text{sorted}([1,1,5,2,3,8]) = \text{sorted}([3,5,1,1]+[2,8])$$

Les stationnements horizontaux donnent la composition de la première partie de la partition.

Exercice : Déterminer la complexité temporelle et spatiale de `part_asc`.

Corrigé : On note n la taille de L et $m = \left\lfloor \frac{s(\mathrm{L})}{2} \right\rfloor$. La variable `tab` est une liste de $n+1$ sous-listes de taille $m+1$ d'où un coût spatiale en $\mathrm{O}(nm)$. Les variables `L1` et `L2` contiennent au plus n éléments puisque qu'elles sont extraites de `L`. Par conséquent, la complexité spatiale de `part_tab` est en $\mathrm{O}(nm)$. Dans `tab_flag`, on a la présence de deux boucles `for` imbriquées induisant nm répétitions et d'une boucle `while` avec au plus m passages pour le balayage de la liste `tab[b]` d'où une complexité temporelle en $\mathrm{O}(nm)$. Les deux boucles `while` imbriquées de `part_asc` construisent les suites $(i_k)_k$ et $(j_k)_k$. Comme on a $i_{-1} = n$ et $j_0 \leqslant m$, il faut au plus $n+1+m+1$ passages pour atteindre la première colonne de `tab`. On conclut que la complexité temporelle de `part_asc` est en $\mathrm{O}(nm)$.

VII Distance de Levenshtein

1 Présentation

Sur une chaîne de caractères, on considère les opérations élémentaires suivantes :
- substitution d'un caractère ;
- insertion d'un caractère ;
- suppression d'un caractère.

On va s'intéresser au nombre minimal d'opérations élémentaires pour transformer une chaîne de caractères en une autre.

Exemple : Pour transformer `abcd` en `bcef`, la séquence

$$\begin{array}{l}\texttt{abcd}\\ \texttt{bcef}\end{array} \xrightarrow[\text{suppression}]{} \begin{array}{l}\texttt{bcd}\\ \texttt{bcef}\end{array} \xrightarrow[\text{substitution}]{} \begin{array}{l}\texttt{bce}\\ \texttt{bcef}\end{array} \xrightarrow[\text{insertion}]{} \begin{array}{l}\texttt{bcef}\\ \texttt{bcef}\end{array}$$

est minimale en le nombre d'opérations élémentaires. Il n'y a pas unicité d'une telle séquence puisqu'on peut changer l'ordre des opérations effectués.

Pour S une chaîne de caractères, on note |S| sa taille.

Définition 14. *On définit la* distance de Levenshtein *ou* distance d'édition *notée* lev *entre deux chaînes de caractères* X *et* Y *par*

$$\text{lev}(\text{X},\text{Y}) = \begin{cases} \max(|\text{X}|,|\text{Y}|) & \text{si } \min(|\text{X}|,|\text{Y}|) = 0 \\ \text{lev}(\text{X}[1:],\text{Y}[1:]) & \text{si } \text{X}[0] = \text{Y}[0] \\ 1 + \min\left(\text{lev}(\text{X}[1:],\text{Y}),\text{lev}(\text{X},\text{Y}[1:]),\text{lev}(\text{X}[1:],\text{Y}[1:])\right) & \text{sinon} \end{cases}$$

Proposition 13. *La distance de Levenshtein est le nombre minimal d'opérations élémentaires pour transformer une chaîne de caractères en une autre.*

Démonstration. Soient X et Y deux chaînes de caractères. Si l'une des deux chaînes est vide, alors on transforme celle-ci en l'autre par insertion des caractères de l'autre chaîne et ceci est optimal. Si les deux premiers caractères coïncident, alors le nombre minimal d'opérations élémentaires pour transformer une chaîne en l'autre est celui correspondant aux chaînes tronquées de leurs premiers caractères. Sinon, il faut tenir compte, après troncature du premier caractère, d'une substitution ou d'une insertion ou d'une suppression d'un caractère. En effet, les configurations restantes sont les suivantes :

- suppression de X[0] puis transformation de X[1 :] en Y ;
- insertion de Y[0] puis transformation de Y[0] + X en Y ce qui équivaut à considérer la suppression de Y[0] puis transformation de X en Y[1 :] ;
- substitution de X[0] en Y[0] puis transformation de X[1 :] en Y[1 :].

□

Proposition 14. *La distance de Levenshtein est une distance.*

Démonstration. Soient X, Y des chaînes de caractères. L'expression de lev(X, Y) est symétrique en X et Y. Si les chaînes sont distinctes, il faut au moins une opération élémentaire pour transformer une chaîne en l'autre, d'où

$$\text{X} \neq \text{Y} \quad \Longrightarrow \quad \text{lev}(\text{X},\text{Y}) > 0$$

et par contraposée

$$\text{lev}(\text{X},\text{Y}) = 0 \quad \Longrightarrow \quad \text{X} = \text{Y}$$

Enfin, étant données X, Y et Z trois chaînes de caractères, le nombre minimal d'opérations élémentaires pour transformer X en Z est inférieur au nombre minimal pour transformer X en Y additionné au nombre minimal pour transformer Y en Z, autrement dit

$$\text{lev}(\text{X},\text{Z}) \leqslant \text{lev}(\text{X},\text{Y}) + \text{lev}(\text{Y},\text{Z})$$

□

2 Approche descendante

Approche descendante sans mémoïsation

La définition de la distance de Levenshtein se prête à une implémentation récursive :

```
def lev_rec(X,Y):
    """Distance de Levenshtein entre X et Y
    récursivement"""
    n,m=len(X),len(Y)
    if min(n,m)==0:
        return max(n,m)
    elif X[0]==Y[0]:
        return lev_rec(X[1:],Y[1:])
    else:
        X_trunc=X[1:]
        Y_trunc=Y[1:]
        return 1+min(lev_rec(X_trunc,Y),lev_rec(X,Y_trunc),
                     lev_rec(X_trunc,Y_trunc))
```

Quelques essais :

```
>>> lev_rec("bonjour","bonsoir")
2
>>> lev_rec("mathématique","informatique")
5 # LENT!
```

Le deuxième exemple qui, bien que simple, requiert déjà quelques efforts à la machine suggère d'envisager une approche plus performante.

On note $\mathrm{R}(n,m)$ le nombre de récursions lors d'un appel de `lev_rec(X,Y)` avec X, Y chaînes de tailles respectives n et m. On admet (hypothèse raisonnable) la croissance de $n \mapsto \mathrm{R}(n,m)$ pour m entier fixé et de $m \mapsto \mathrm{R}(n,m)$ pour n entier fixé. Dans le pire des cas, on a

$$\mathrm{R}(n,m) = 1 + \mathrm{R}(n-1,m) + \mathrm{R}(n,m-1) + \mathrm{R}(n-1,m-1)$$

Par récurrence sur $n+m$, on vérifie sans difficulté

$$\forall (n,m) \in \mathbb{N}^2 \qquad \mathrm{R}(n,m) \geqslant \binom{n+m}{n}$$

d'où une complexité temporelle rédhibitoire pour de grandes valeurs de n et m.

Approche descendante avec mémoïsation

La mémoïsation pour éviter les redondances de calcul n'apporte pas une amélioration significative.

```
def lev_memo(X,Y):
    """Distance de Levenshtein entre X et Y
    récursivement avec mémoïsation"""
    def lev(X,Y):
        if (X,Y) not in memo:
```

```
            n,m=len(X),len(Y)
            if min(n,m)==0:
                res=max(n,m)
            elif X[0]==Y[0]:
                res=lev(X[1:],Y[1:])
            else:
                X_trunc=X[1:]
                Y_trunc=Y[1:]
                res=1+min(lev(X_trunc,Y),lev(X,Y_trunc),lev(X_trunc,Y_trunc))
            memo[(X,Y)]=res
        return memo[(X,Y)]
    memo={}
    return lev(X,Y)
```

Un essai :

```
>>> lev_memo("mathématique","informatique")
5
```

Remarque : Le choix d'une troncature à gauche est arbitraire. On peut tout à fait implémenter récursivement la distance de Levenshtein avec une troncature à droite, sur le dernier caractère des chaînes.

3 Approche ascendante

Proposition 15. *Soient* X *et* Y *des chaînes de caractères non vides. On a*

$$\mathrm{lev}(\mathrm{X},\mathrm{Y}) = 1 + \min\left(\mathrm{lev}(\mathrm{X}[:-1],\mathrm{Y}), \mathrm{lev}(\mathrm{X},\mathrm{Y}[:-1]), \mathrm{lev}(\mathrm{X}[:-1],\mathrm{Y}[:-1]) - \delta_{\mathrm{X}[-1],\mathrm{Y}[-1]}\right)$$

Démonstration. C'est la remarque faite précédemment. D'après la proposition 13, on a

$$\mathrm{lev}(\mathrm{X},\mathrm{Y}) = \mathrm{lev}(\mathrm{X}[::-1],\mathrm{Y}[::-1])$$

En effet, une séquence optimale transformant X en Y transforme également X[::−1] en Y[::−1] et réciproquement. Le résultat suit. □

Soient X et Y deux chaînes de caractères de tailles respectives n et m. On pose

$$\forall (i,j) \in [\![\,0\,;\,n\,]\!] \times [\![\,0\,;\,m\,]\!] \qquad \mathrm{d}(i,j) = \mathrm{lev}(\mathrm{X}[:i],\mathrm{Y}[:j])$$

Corollaire 2. *Pour* $(i,j) \in [\![\,1\,;\,n\,]\!] \times [\![\,1\,;\,m\,]\!]$*, on a*

$$\mathrm{d}(i,j) = 1 + \min\left(\mathrm{d}(i-1,j), \mathrm{d}(i,j-1), \mathrm{d}(i-1,j-1) - \delta_{\mathrm{X}[i-1],\mathrm{Y}[j-1]}\right)$$

Démonstration. Conséquence immédiate de la proposition précédente. □

On peut alors opter pour une approche ascendante en construisant la matrice $\mathrm{D} = \big(\mathrm{d}(i,j)\big)_{0\leqslant i\leqslant n, 0\leqslant j\leqslant m}$ et en renvoyant $\mathrm{d}(n,m)$.

```
def lev_asc(X,Y):
    n,m=len(X),len(Y)
    tab=[[0]*(m+1) for i in range(n+1)]
    for i in range(n+1):
```

```
        tab[i][0]=i
    for j in range(m+1):
        tab[0][j]=j
    for i in range(1,n+1):
        for j in range(1,m+1):
            tab[i][j]=1+min(tab[i-1][j],tab[i][j-1],
                            tab[i-1][j-1]-int(X[i-1]==Y[j-1]))
    return tab[n][m]
```

Quelques essais :

```
>>> lev_asc("mathématique","informatique")
5
>>> lev_asc("algèbre","analyse")
5
>>> lev_asc("algèbre","probabilités")
11
>>> lev_asc("analyse","probabilités")
10
```

4 Alignement optimal

Définition 15. *Un* alignement *de deux chaînes de caractères* X *et* Y *est formé d'un couple de chaînes* $\widehat{\text{X}}$ *et* $\widehat{\text{Y}}$ *de même taille, obtenues respectivement à partir des chaînes* X *et* Y *avec éventuellement des insertions d'espaces (caractère* `'-'`*) mais pas sur des positions identiques.*

Exemples : Des alignements des chaînes `bond` et `bonjour` :

```
bon---d     b-on--d     -bo-n-d     b-o-n---d
bonjour     bonjour     bonjour     -b-onjour
```

Définition 16. *Un* désaccord *dans un alignement est une différence entre deux caractères sur un indice donné.*

Exemple : Sur l'alignement

```
indice: 0123456
        bon---d
        bonjour
```

les indices 3 jusqu'à 6 sont en désaccord.

En vue de réaliser le nombre minimal d'opérations élémentaires permettant de passer de `bond` à `bonjour`, il est intuitif de choisir le premier parmi les quatre alignements proposés précédemment puis de procéder à des substitutions sur les quatre derniers caractères.

Définition 17. *Un* alignement optimal *de deux chaînes de caractères* X *et* Y *est un alignement de ces chaînes dont le nombre de désaccords est minimal.*

Exemple : Les alignements optimaux des chaînes `'bond'` et `'bonjour'` sont :

```
bond---     bon-d--     bon--d-     bon---d
bonjour     bonjour     bonjour     bonjour
```

Proposition 16. *La distance de Levenshtein est le nombre de désaccords dans un alignement optimal.*

Démonstration. Soient X et Y deux chaînes de caractères et une séquence d'opérations élémentaires permettant de passer de X à Y. En remplaçant une suppression dans X par une insertion de `'-'` dans Y aux mêmes indices et une insertion dans X par une insertion de `'-'` toujours dans X au même indice, le nombre de désaccords entre les deux chaînes correspond au nombre d'insertions, de suppressions et de substitutions pour passer de X à Y. Optimiser une séquence d'opérations élémentaires permettant de passer de X à Y équivaut donc à optimiser le nombre de désaccords dans un alignement de ces chaînes. □

Exemple : La séquence pour transformer `abcd` en `bcef`

```
abcd  ——suppression→  bcd  ——substitution→  bce  ——insertion→  bcef
bcef                  bcef                  bcef               bcef
```

correspond à la construction de l'alignement optimal

```
abcd  ——suppression→  abcd  ——substitution→  abcd  ——insertion→  abcd-
bcef                  -bcef                 -bcef               -bcef
```

5 Extraction d'un alignement optimal

Soient X et Y deux chaînes de caractères et $\mathrm{D} = \big(\mathrm{d}(i,j)\big)_{0\leqslant i\leqslant n, 0\leqslant j\leqslant m}$ la matrice précédemment définie des distances de Levenshtein des chaînes extraites. En conservant l'intégralité de la matrice D et pas seulement $\mathrm{d}(n,m)$, on peut, en remontant les étapes depuis la cible (n,m) vers $(0,0)$, reconstituer les chaînes associées à un alignement optimal.

On rappelle que pour tout $(i,j) \in [\![1\,;n]\!] \times [\![1\,;m]\!]$, on a

$$\mathrm{d}(i,j) = 1 + \min\big(\mathrm{d}(i-1,j), \mathrm{d}(i,j-1), \mathrm{d}(i-1,j-1) - \delta_{\mathrm{X}[i-1],\mathrm{Y}[j-1]}\big)$$

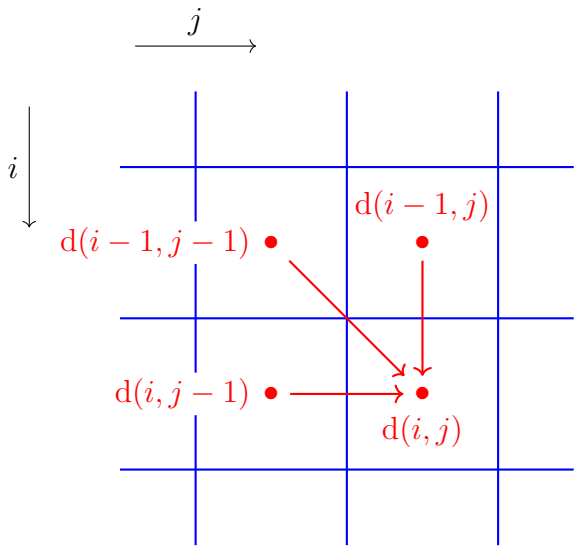


FIGURE 21 – Transitions pour la distance de Levenshtein

Étant donné $(i,j) \in [\![1\,;n]\!] \times [\![1\,;m]\!]$, on détermine une position parmi $(i,j-1)$, $(i-1,j)$ et $(i-1,j-1)$ depuis laquelle la transition est possible pour l'obtention de $\mathrm{d}(i,j)$. Il y a au moins une position satisfaisante (peut-être plusieurs) par définition de la matrice D.

Cette reconstitution ne peut être réalisée de manière ascendante : il faut partir de la distance de Levenshtein $\mathrm{d}(n,m)$ (étape finale de `lev_asc`) pour effectuer cette remontée.

```
def lev_tab(X,Y):
    n,m=len(X),len(Y)
    tab=[[0]*(m+1) for i in range(n+1)]
    for i in range(n+1):
        tab[i][0]=i
    for j in range(m+1):
        tab[0][j]=j
    for i in range(1,n+1):
        for j in range(1,m+1):
            tab[i][j]=min(tab[i-1][j]+1,tab[i][j-1]+1,
                          tab[i-1][j-1]+int(X[i-1]!=Y[j-1]))
    return tab

def lev_align(X,Y):
    """Détermination d'un alignement optimal des chaînes X et Y"""
    n,m=len(X),len(Y)
    tab=lev_tab(X,Y)
    i,j=n,m
    res="",""
    while i>0 or j>0:
        if tab[i-1][j-1]+int(X[i-1]!=Y[j-1])==tab[i][j]:
            i-=1
            j-=1
            res=X[i]+res[0],Y[j]+res[1]
        elif i>0 and tab[i-1][j]+1==tab[i][j]:
            i-=1
            res=X[i]+res[0],"-"+res[1]
        else:
            j-=1
            res="-"+res[0],Y[j]+res[1]
    return res
```

Quelques essais :

```
>>> lev_align("bond","bonjour")
('bon---d', 'bonjour')
>>> lev_align("informatique","paradigme")
('informatique', '--par-adigme')
```

Exercice : Déterminer la complexité temporelle et spatiale de `lev_align`.

Corrigé : On note n la taille de X et m la taille de Y. La variable `tab` est une liste de $n+1$ listes de taille $m+1$ d'où un coût spatial en $\mathrm{O}(nm)$. La variable `res` est un tuple qui reçoit un alignement optimal $\left(\widehat{\mathrm{X}}, \widehat{\mathrm{Y}}\right)$. Comme les éventuelles insertions de caractère `'-'` ne sont pas sur des positions identiques, les tailles de $\widehat{\mathrm{X}}$ et $\widehat{\mathrm{Y}}$ sont majorées par $n+m$. Par conséquent, le complexité spatiale de `lev_align` est en $\mathrm{O}(nm)$. La complexité temporelle de `lev_tab` est en $\mathrm{O}(nm)$ du fait des deux boucles `for` imbriquées. Le nombre de passages dans la boucle `while` de `lev_align` est le nombre de cases parcourues dans `tab` lors de la détermination d'un alignement optimal, nombre majoré par $n+1+m+1$ puisque les transitions sont soit horizontales, soit verticales, soit diagonales. On conclut que la complexité temporelle de `lev_align` est en $\mathrm{O}(nm)$.

Chemins associés aux alignements optimaux

En remontant la matrice D et en plaçant dans une file chaque trajet en construction conforme pour réaliser la taille maximale, on peut obtenir tous les alignements optimaux de deux chaînes. L'idée est identique à celle mise en œuvre pour déterminer toutes les plus longues sous-suites communes à deux chaînes.

	∅	b	o	n	j	o	u	r
∅	0	1	2	3	4	5	6	7
b	1	0	1	2	3	4	5	6
o	2	1	0	1	2	3	4	5
n	3	2	1	0	1	2	3	4
d	4	3	2	1	1	2	3	4

Alignement = `('bond---', 'bonjour')`

	∅	b	o	n	j	o	u	r
∅	0	1	2	3	4	5	6	7
b	1	0	1	2	3	4	5	6
o	2	1	0	1	2	3	4	5
n	3	2	1	0	1	2	3	4
d	4	3	2	1	1	2	3	4

Alignement = `('bon-d--', 'bonjour')`

	∅	b	o	n	j	o	u	r
∅	0	1	2	3	4	5	6	7
b	1	0	1	2	3	4	5	6
o	2	1	0	1	2	3	4	5
n	3	2	1	0	1	2	3	4
d	4	3	2	1	1	2	3	4

Alignement = `('bon--d-', 'bonjour')`

	∅	b	o	n	j	o	u	r
∅	0	1	2	3	4	5	6	7
b	1	0	1	2	3	4	5	6
o	2	1	0	1	2	3	4	5
n	3	2	1	0	1	2	3	4
d	4	3	2	1	1	2	3	4

Alignement = `('bon---d', 'bonjour')`

Les stationnements horizontaux donnent les caractères `'-'` pour la première chaîne, les stationnements verticaux donnent les caractères `'-'` pour la deuxième chaîne et les transitions diagonales donnent les lettres correspondantes pour chaque chaîne.

Références

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, Third Edition MIT Press and McGraw-Hill, 2009

[2] Jeff Erickson, *Algorithms* , `https://jeffe.cs.illinois.edu/teaching/algorithms/`