Documentations complètes
Python3 : https://docs.python.org/3/
Nympy : https://docs.scipy.org/doc/
Matplotlib : http://matplotlib.org/
SQL : http://sql.sh/

5 Dictionnaires et Programmation Dynamique

5.1 Structure de donnée

5.1.1 Définitions

En informatique une structure de données est une manière de stocker en mémoire et d'organiser des données auxquelles on voudra accéder plus tard. en particulier elle décrit la manière dont on peut accéder ces données.

Une structure de données est munie d'un certain nombre d'opérations. Par exemple une des opérations les plus courantes est la lecture, lorsque l'on souhaite accéder à un élément stocker dans la structure. D'autres opérations sont disponibles en général comme l'insertion ou la suppression d'un élément dans la structure.

Toutes les structures de données ne permettent pas les mêmes opérations, et plus important suivant la structure les opérations n'ont pas le même coût. Par exemple, avec certaines structures, il est très rapide d'ajouter un élément, dans d'autres c'est difficile, voir impossible, et cela peut demander une réorganisation complète. Pour chaque structure de données utilisée, on essaiera d'avoir une bonne idée de la complexité des opérations que l'on effectue.

Connaître les coûts a deux intérêts : quand on veut déterminer la complexité d'un algorithme il faut connaitre les coûts des opérations utiliser sur les différentes structures utilisées. Mais surtout quand on veut écrire un algorithme, connaissant les opérations dont on a besoin, on peut choisir la structure de données la plus adaptée (celle pour laquelle ces opérations sont les moins coûteuses).

On peut distinguer deux sortes de structures de données : les structures statiques et dynamiques, suivant que la quantité de mémoire allouée à la structure est fixée à la création de celle-ci, sans pouvoir être modifiée par la suite, ou si sa taille peut varier au cours du déroulement de l'algorithme.

De même, si le contenu de la structure peut être modifiée après sa création, on dira que la structure est mutable, sinon on dira qu'elle est non-mutable.

Par exemple, en Python les classes `tuple` et `str` sont des structures statiques non-mutables, alors que la classe `list` est dynamique et mutable.

Donnons quelques familles de structures de données que l'on rencontre fréquemment :

1. Les structures linéaires : les tableaux, les piles et les files.
2. Les tableaux mutlidimensionnels : les matrices.

3. Les structures récursives : les arbres, les graphes et les listes chaînées.

5.1.2 Les tableaux

La structure de données la plus courante est certainement celle des tableaux statiques, du moins dans le langage **C** ou ses dérivés.

Un tableau statique est un structure de donnée statique, qui rassemble sous forme de suite des variables de même type qui sont stockées dans des emplacements consécutifs en mémoire.

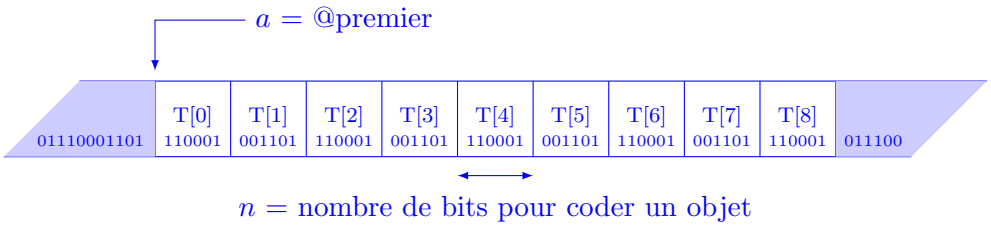

FIGURE 5.1 – Un tableau en mémoire

Comme tous les objets stockés dans le tableau sont de même type, ils sont codés sur le même nombre de bits n, si bien que pour accéder au k-ième élément du tableau il suffit de connaitre l'adresse a de la première case et de se rendre à l'adresse $a + k \times n$, c'est un modèle d'adressage direct. Ainsi l'accès à un élément du tableau se fait à coût constant $\Theta(1)$.

L'inconvénient de cette structure est son côté statique, il est impossible d'ajouter un nouvel élément après sa création, mais cette contrainte fait sa force.

En Python le module `numpy` offre la classe `array`, qui sont de vrais tableaux statiques.

Il existe aussi des tableaux dynamiques, dont la taille peut varier après avoir été créé.

5.1.3 Les listes

Le principe de cette structure est d'adjoindre à chaque donnée, du même type, contenue dans la liste l'adresse en mémoire de la donnée suivante, sauf la dernière case qui « pointe » vers une adresse particulière qui marque la fin de la liste.

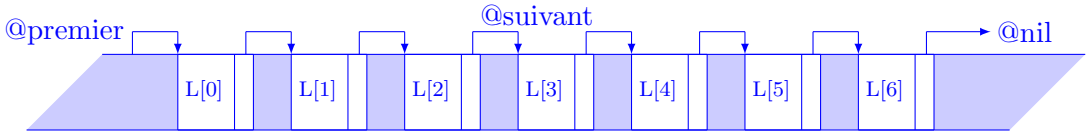

FIGURE 5.2 – Une liste chaînée en mémoire

De cette manière il n'est plus besoin lorsqu'on crée une liste de réserver des espaces contigus en mémoire, et on peut aussi la modifier après sa création (ajouter, supprimer et insérer des éléments), en un mot c'est une structure dynamique.

Là encore la force fait la faiblesse, dans une liste pour accéder au n-ème élément il faut parcourir les $n-1$ premiers éléments de la liste. En conséquence de quoi le temps de parcours n'est plus constant mais

en $O(n)$.

Il existe d'autre types de listes, les listes doublement chaînées qui permettent le parcours dans les deux sens de la liste, les listes circulaires dans lesquelles la dernière case pointe vers la première.

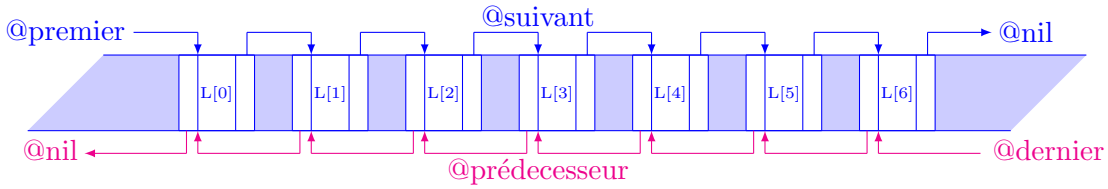

FIGURE 5.3 – Une liste doublement chaînée en mémoire

En Python, les objets de la classe `list` ne sont pas des listes... C'est une structure de données ingénieuse qui concilie les avantages des listes chaînées et des tableaux : c'est une structure dynamique dont le parcours se fait (en moyenne) à coût constant. Son secret réside dans le redimensionnement mais nous n'en dirons pas plus ici. Par contre nous allons donner les coûts (en moyenne) des opérations courantes sur les listes.

Temps d'exécution	
`list[i] = x`	$O(1)$
`liste.append(x)`	$O(1)$
`liste.pop()`	$O(1)$
`liste.insert(i,x)`	$O(n)$
`liste.pop(i)`	$O(n)$
`liste.remove(x)`	$O(n)$

5.2 Les tables d'association ou dictionnaire

Il existe de nombreux problèmes manipulant des ensembles dynamiques, donc susceptibles d'évoluer avec le temps, dans lesquels les seules opérations réalisées sont l'insertion, la suppression et la recherche. On cherche donc a mettre en place une structure de donné efficace pour ce type de problème.

Un dictionnaire ou une table d'association est un type de données associant un ensemble de clefs à un ensemble de valeurs. Plus formellement, si U désigne l'ensemble des clefs et V l'ensemble des valeurs, une table d'association est un sous-ensemble T de $U \times V$ tel que pour toute clé $u \in U$ il existe au plus un élément $v \in V$ tel que $(u, v) \in T$.

Une table d'association doit supporter les opérations suivantes :

— insérer une paire $(u, v) \in U \times V$ dans T ;

— rechercher la valeur v associée à la clef u dans T ;

— supprimer une paire $(u, v) \in U \times V$ de T.

Nous allons présenter deux méthodes d'implémentation d'une table d'association, les tables à adressage direct et les tables de hachage.

5.2.1 Tables à adressage direct

Supposons que l'univers des clefs U soit en bijection avec $\{0, \ldots, n-1\}$, avec $n \in \mathbb{N}^*$ pas trop grand, par une fonction $h : U \to [\![0, n-1]\!]$. On peut alors implémenter T sous la forme d'un tableau de taille n, dont les cases contiennent soit une donnée `Nil` qui indique que la clef n'est associée à aucune valeur, soit l'adresse où l'on peut trouver la donnée associée à la clef.

Evidemment, nous pourrions gagner de l'espace en stockant directement la valeur associée à la clef u dans la case $T[u]$, mais nous verrons plus tard pourquoi on préfère stocker un pointeur vers l'objet.

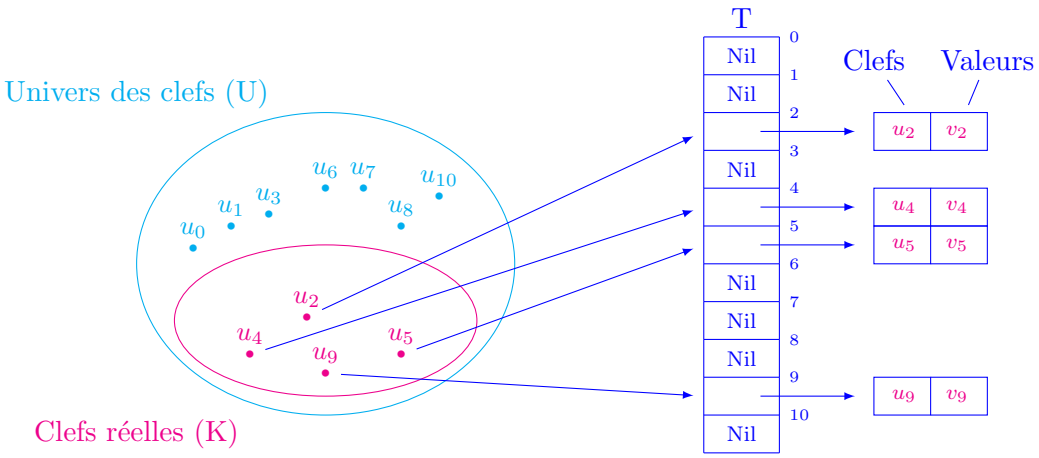

FIGURE 5.4 – Table à adressage direct

Les trois opérations d'insertion, de recherche et de suppression se font au pire en temps constant dans ce cas.

5.2.2 Tables de hachage

Le principe

L'inconvenient majeur des tables à adressage direct saute au yeux, si l'ensemble K des clefs réellement utilisées est très petit devant le nombre de clefs possibles, ce qui est très souvent le cas, la quasi-totalité de l'espace mémoire alloué à T est gaspillé.

Sans compter que l'ensemble des clefs pourrait être tout simplement de taille bien trop importante, voir infini.

Aussi, dans la pratique on utilise un tableau T de taille $m \ll n$ et une fonction $h : U \to [\![0, m-1]\!]$, dite fonction de hachage, qui permet de calculer dans quelle case[i] du tableau trouver les informations relatives à la clef. On dit que l'on a la clef k, lorsqu'on a calculé $h(k)$.

La figure (5.5) illustre ce concept.

Comme l'on choisit $m < n$, inévitablement on prend le risque qu'il y ait des collisions. Une collision se produit lorsque deux clefs distinctes $k \neq k'$ sont telles que $h(k) = h(k')$.

Si c'est inévitable, on peut au moins chercher à minimiser le nombre de collisions. Pour ça la première chose à demander à la fonction h c'est de prendre des valeurs le plus uniformément réparties possible dans $[\![0, m-1]\!]$, de sorte qu'il n'y ait pas d'accumulation.

Par ailleurs, il faudra voir comment gérer les collisions.

i. Cette case s'appelle le bucket de la clef.

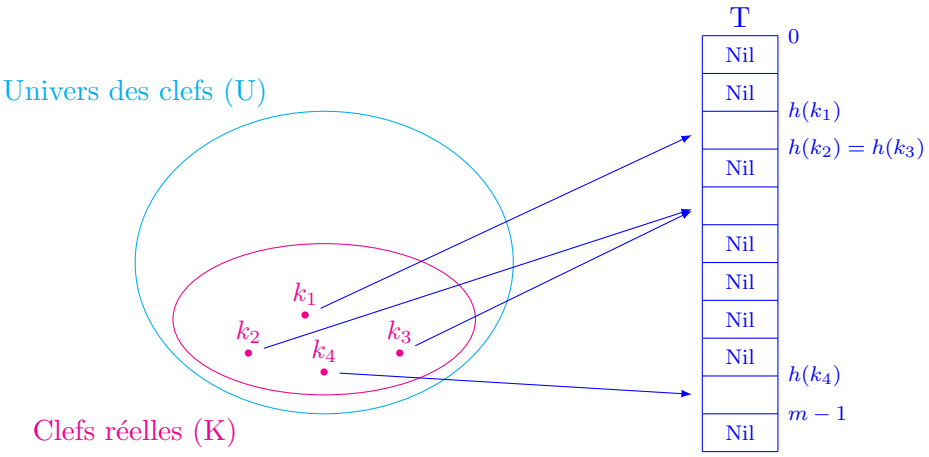

FIGURE 5.5 – Table de hachage

Résolution des collisions par chaînage

Cette méthode de résolution est très simple, il suffit de faire pointer les cases de T vers des listes (chaînées) dans lesquelles seront stockés tous les couples clef-valeur, dont la clef est associées à cette valeur de hachage, comme illustré par la figure (5.2.2).

La recherche d'une valeur, après avoir calculé la valeur de hachage de la clef, nécessite une recherche séquentielle dans la liste correspondante. Ce qui se fait en un temps proportionnel à la longueur de la liste. Mais sous certaines hypothèses on peut s'assurer qu'une recherche se fait en moyenne en temps constant. Si les listes sont doublement chaînées la suppression se fait aussi en temps constant. Et, l'insertion peut se fait en temps constant, si on fait une insertion en tête de la liste.

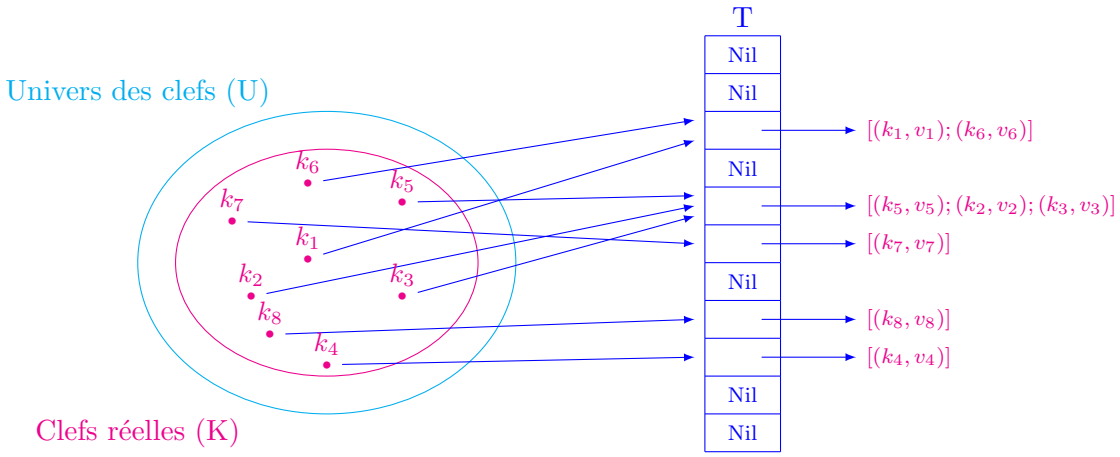

FIGURE 5.6 – Résolution des collisions par chaînage

Résolution directe

Il existe d'autres méthodes de résolution des collisions, dites d'adressage ouvert. Dans ce contexte, on suppose que les couples clef-valeur sont stockés directement dans les cases de T.

Une premiere méthode de résolution directe, appelée sondage linéaire, consiste pour enregistrer (k, v) à regarder si la case $h(k)$ est libre. Si elle l'est on enregistre le couple (k, v) dans cette case, sinon on regarde

si la case $h(k)+1 \bmod m$ est libre, sinon on inspecte la case portant le numéro $h(k)+2 \bmod m$ etc... Jusqu'à trouver une case libre. Cette méthode présente l'inconvénient de diminuer l'uniformité de la distribution.

Une seconde méthode, consiste à utiliser une seconde fonction de hachage h', et à chercher une case libre parmi celles de numéros $h(k)+h'(k) \bmod m$, $h(k)+2h'(k) \bmod m$, $h(k)+3h'(k) \bmod m$... On parle de résolution par double hachage. C'est une amélioration de la méthode par sondage linéaire mais elle n'est pas parfaite.

Evidemment ces deux méthodes nécessitent que la taille m du tableau soit supérieure au nombre de clef de K.

Fonctions de hachage

Une bonne fonction de hachage, rappelons-le, doit :

— être très simple à calculer, sa complexité doit être en $O(1)$;

— éviter, tant que faire se peut, les collisions.

Nous allons ici considérer que l'ensemble U des clefs est l'ensemble des suites de bits $\{0,1\}^{\mathbb{N}}$, donner quelques exemples de fonction de hachage.

Hachage par extraction Une fonction du type hachage par extraction détermine la valeur de hachage d'une clef en extrayant certains bits de la clef. Par exemple si $k = 1101100100011011$, et si la fonction h extrait les bits 0, 3, 4 et 7, on aura $h(k) = \underline{1110}_b = 12$ L'inconvénient de ce type de fonction est que la valeur de hachage ne dépend pas de l'intégralité de la clef, ce qui provoque de très nombreuses collisions. En revanche ces fonctions ont une répartition assez uniforme.

Hachage par division Fixons un entier m, le hachage par division s'obtient tout simplement en calculant $h(k) = k \bmod m$. Pour éviter des nombreuses collisions il faut prendre m premier et si possible éloigné d'une puissance de 2. Si cette technique offre l'avantage d'avoir une bonne répartition les collisions sont nombreuses et le choix de m est assez compliqué.

Hachage par multiplication Fixons un un nombre $0 < \theta < 1$, on pose alors $h(k) = \lfloor ((k \times \theta) \bmod 1) * m \rfloor$ [ii], avec m la taille du tableau. Il faut correctement choisir θ, en effet si θ est trop proche de 0 ou de 1 il y aura accumulation de valeurs aux extrémités du tableau. En fait, Knuth suggère que pour $\theta = \dfrac{\sqrt{5}-1}{2}$ la répartition obtenue est plutôt bonne.

5.3 Les dictionnaires en Python

Les dictionnaires sont des tableaux associatifs qui associent à chaque clef une valeur. Les clefs comme les valeurs peuvent être hétérogènes (i.e. de type différents). Seule restriction les clefs doivent être des objets hashables, donc non mutables, en particulier pas des listes ou des ensembles. Nous reviendrons plus loin sur les types hashables.

En python on définit des dictionnaires entre accolades et en déclarant les couples clef-valeur comme suit :

```
dict1 = {'bananes': 4, 'citrons': 2.5, 'pamplemousses' : 'beaucoup', 3 : [1,2,3]}
```

On accède aux différentes valeurs à l'aide de leur clef :

ii. Ici mod 1 représente la partie fractionnaire du nombre, c'est-à-dire $k \times \theta - \lfloor (k \times \theta) \rfloor$.

```
>>> dict1['bananes']
4
>>> dict1['citrons']
2.5
>>> dict1['pamplemousses']
'beaucoup'
>>> dict1[3]
[1, 2, 3]
```

Que se passe-til si l'on cherche à accéder à une valeur pour une clef qui n'existe pas :

```
>>> dict1['cerises']
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    dict1['cerises']
KeyError: 'cerises'
```

On peut éviter cela en utilisant la méthode `get` :

```
>>> # Comme second argument on donne une valeur qui sera retournée si la clef est absente.
>>> dict1.get('cerises', "Il n'y en a pas")
"Il n'y en a pas"
>>> dict1.get('cerises',0)
0
```

On obtient la longueur d'un dictionnaire avec `len` :

```
>>> len(dict1)
4
```

Il est possible d'accéder aux clefs et aux valeurs en utilisant les méthodes `keys` et `values` :

```
>>> dict1.keys()
dict_keys(['bananes', 'citrons', 'pamplemousses', 3])
>>> dict1.values()
dict_values([4, 2.5, 'beaucoup', [1, 2, 3]])
```

Il est possible d'itérer sur les clefs et/ou les valeurs :

```
>>> for key in dict1.keys(): print(dict1[key])
4
2.5
beaucoup
[1, 2, 3]
>>> for val in dict1.values(): print(val)
4
2.5
```

```
beaucoup
[1, 2, 3]
# Plus simplement
>>> for val in dict1: print(val)
bananes
citrons
pamplemousses
3
>>> for k, v in dict1.items(): print(k, v)
bananes 4
citrons 2.5
pamplemousses beaucoup
3 [1, 2, 3]
```

Les dictionnaires sont mutables :

```
>>> # On peut modifier une valeur :
>>> dict1['bananes'] = 6
>>> dict1
{'bananes': 6, 'citrons': 2.5, 'pamplemousses': 'beaucoup', 3: [1, 2, 3]}
>>> # On peut éliminer un coupl clef-valeur :
>>> dict1.pop('pamplemousses') # Comme pour la méthode de liste l'entrée est retournée.
>>> dict1
{'bananes': 6, 'citrons': 2.5, 3: [1, 2, 3]}
>>> # On peut ajouter un couple clef-valeur :
>>> dict1['pommes'] = 10
>>> dict1
{'bananes': 6, 'citrons': 2.5, 3: [1, 2, 3], 'pommes': 10}
```

Il est possible de définir un dictionnaire en compréhension comme pour les listes :

```
>>> dict2 = { x : x**3 for x in range(5)}
>>> dict2
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64}
```

Comme pour les tuples et les listes on peut tester l'appartenance d'une clef avec `in` :

```
>>> 3 in dict2
True
```

Parmi les choses intéressantes avec les dictionnaires il y l'unpacking dont le principe est illustré ci-dessous :

```
>>> def prod(a, b):
. . .      return a*b
>>> dict3 = {'a' : 2, 'b' : 3}
>>> # On utilise ** pour "unpacker" le dictionnaire :
>>> prod(**dict3)
6
```

Pour tout savoir sur les dictionnaires c'est ici Documentation : https://docs.python.org/3/tutorial/datastructures.html
Plus haut nous avons dit qu'il faut que les clefs des dictionnaires soient hashables, en particulier non mutables. Mais il faut prendre garde au fait que des types immutables peuvent contenir des mutables. Par exemple une liste dans un tuple. Dans ce genre de cas, la non-hashabilité des valeurs contenues rend non-hashable le conteneur.
Python dispose d'une fonction `hash` qui est une fonction de hashage très performante. On peut s'en servir entre autre pour tester si deux objets hashables sont différents.

5.4 Programmation dynamique

5.4.1 Un exemple : chaîne de multiplication matricielle

Considérons des matrices $A_1, \dots, A_n$ pour lesquelles on peut calculer le produit $A_1 \cdots A_n$. Le produit matricielle étant associatif tous les parenthésages possibles donnent le même résultat. On dit qu'un produit de matrice est entière parenthésé s'il est composé d'une seule matrice ou d'un produit de deux produits matriciels entièrement parenthésés.

Par exemple le produit $A_1A_2A_3A_4$ peut être entièrement parenthésé de cinq manières différentes :

$$\begin{aligned} A_1A_2A_3A_4 &= (A_1(A_2(A_3A_4))) = ((A_1A_2)(A_3A_4)) \\ &= (A_1((A_2A_3)A_4)) = (((A_1A_2)A_3)A_4) \\ &= ((A_1(A_2A_3))A_4) \end{aligned}$$

Exercice 5.1. Considérons $A_1 \in \mathcal{M}_{10,100}(\mathbb{R})$, $A_2 \in \mathcal{M}_{100,5}(\mathbb{R})$ et $A_3 \in \mathcal{M}_{5,50}(\mathbb{R})$. Combien de multiplications faut-il pour calculer $((A_1A_2)A_3)$ et $(A_1(A_2A_3))$?

L'exemple précédent montre que le parenthésage joue un rôle essentiel dans la rapidité des calculs d'un produit de matrice en chaîne.

Le problème est alors le suivant, si $A_1, \dots, A_n$ pour lesquelles on peut calculer le produit $A_1 \cdots A_n$ quel est le parenthésage qui minime le nombre de multiplications scalaires.

Brute force ou solution bovine

On peut tester tous les cas possibles. Notons $P(n)$ le nombre de parenthésages possibles pour $n \in \mathbb{N} \setminus \{0\}$.

Pour $n = 1$, il n'y a qu'une solution, donc $P(1) = 1$. Si $n > 1$, un produit entièrement parenthésé est le produit de deux sous-produits entièrement parenthésés, ce produit peut intervenir entre la matrice A_k et la matrice A_{k+1} pour un certain k compris entre 1 et $n-1$. Ce qui donne la formule de récurrence :

$$P(n) = \begin{cases} 1 & \text{si} \quad n = 1 \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{si} \quad n \geq 2. \end{cases}$$

On peut démontrer que :

$$P(n+1) = \frac{1}{n+1}\binom{2n}{n} = \frac{4^n}{n^{3/2}\sqrt{n}}(1 + O(1)).$$

Le nombre de $P(n)$ croit donc de manière exponentielle avec n. Ce qui fait que cette méthode n'est pas envisageable.

Sous-structure optimale

Un parenthésage optimal de $A_1A_2\cdots A_n$ sépare le produit entre A_k et A_{k+1} pour un certain k. Le calcul de la solution optimale commence donc par le calcul des matrices $A_1\cdots A_k$ et $A_{k+1}\cdots A_n$ puis on les multiplie pour obtenir $A_1A_2\cdots A_n$.

Le coût du calcul de $A_1A_2\cdots A_n$ est alors la somme des coûts des calculs de $A_1\cdots A_k$ et $A_{k+1}\cdots A_n$.

Il est facile de se convaincre que le parenthésage des produits $A_1\cdots A_k$ et $A_{k+1}\cdots A_n$ doivent être optimaux pour que celui de $A_1\cdots A_n$ le soit. En effet, sinon le remplacement du parenthésage de $A_1\cdots A_k$ ou $A_{k+1}\cdots A_n$ par un parenthésage plus économique fournirait une meilleure solution globale.

Une solution optimale à une instance du problème de multiplication d'une suite de matrices utilise donc uniquement des solutions optimales aux instances des sous-problèmes.

Définition récursive d'une solution optimale

Soit $m[i,j]$ le nombre minimal de multiplications scalaires pour calculer $A_i\cdot A_{i+1}\cdots A_j$, avec $1\leq i\leq j\leq n$. On cherche donc $m[1,n]$.

Notons $p_{i-1}\times p_i$ la taille de la matrice A_i. Alors :

1. Si $i=j$, le produit n'est constitué que d'une seule matrice donc $m[i,i]=0$.
2. Si $i<j$, supposons que le produit $A_i\cdots A_j$ soit coupé entre A_k et A_{k+1}, alors :

$$m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j,$$

car il faut $p_{i-1}p_kp_j$ multiplications pour calculer le produit de $A_i\cdots A_k$ et $A_{k+1}\cdots A_j$

Il s'agit donc de déterminer $i\leq k\leq j$ pout minimiser $m[i,j]$, et finalement notre récurrence pour calculer $m[1,n]$ est :

$$m[i,j]=\begin{cases}0 & \text{si} \quad i=j\\ \min_{i\leq k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & \text{si} \quad i<j\end{cases}$$

Un premier algorithme

Algorithm 1: Coût optimal

```
1  def best_cost(p, i, j):
       Data:
       p le vecteur des p_i
       i < j des entiers
2
3      if i = j then
4          return 0
5      m[i,j] ←+∞
6      for k ← i to j − 1 do
7          q ←best_cost(p, i, k) + best_cost(p, k + 1, j) + p[i − 1] × p[k] × p[j]
8          if q < m[i,j] then
9              m[i,j] ←q
10     return m[i,j]
```

Exercice 5.2. Dessiner l'arbre des appels récursifs pour $n = 4$ et en déduire un minorant de la complexité de cet algorithme.

Chevauchement des sous-problèmes

Si la complexité de notre premier algorithme est exponentielle c'est parce qu'il rencontre chaque sous-problème un nombre exponentiel de fois, alors même que les sous-problèmes sont peu nombreux. En effet, pour chaque choix de $1 \leq i \leq j \leq n$, il y en a $n + \binom{n}{2}$, soit un $\Theta(n^2)$.

Il s'agit donc d'essayer de ne traiter chaque sous problème une seule fois.

Une autre approche

Lorsque l'on détermine $m[i, j]$ le coût du calcul d'un produit enchaîné de $j - i + 1$ matrices le résultat ne dépend que des coûts de calcul de produits enchaîné de moins de $j - i + 1$ matrices. Autrement dit, pour $i \leq k \leq j - 1$ les matrices $A_i \cdots A_k$ et $A_{k+1} \cdots A_j$ sont des produits respectivement de $k - i + 1$ et $j - k$ matrices, soit strictement moins que $j - i + 1$.

Nous allons donc remplir le tableau m, de façon ascendante, c'est-à-dire résoudre le problème pour des chaînes de matrices de longueur croissante.

Nous allons utiliser deux tableaux à double entrée, le tableau m pour mémoriser les coûts $m[i, j]$ et un tableau s qui enregistre l'indice k ayant donné le coût optimal lors du calcul de $m[i, j]$, c'est-à-dire l'indice k tel que le parenthésage optimal du produit $A_i \cdots A_j$ soit $A_i \cdots A_k \cdot A_{k+1} \cdots A_j$.

L'algorithme commence par initialiser la diagonale du tableau m avec des 0, ensuite il calcule $m[i, i+1]$ pour $1 \leq i \leq n - 1$, puis $m[i, i + 2]$ pour $1 \leq i \leq n - 2$ et ainsi de suite. Autrement dit, il complète le tableau m diagonale par diagonale.

Exercice 5.3. Considérons les dimensions de matrices suivantes :

matrice	A_1	A_2	A_3	A_4	A_5
dimension	10×15	15×8	8×2	2×5	5×10

Remplir les tableaux suivants avec les valeurs de $m[i, j]$ et $s[i, j]$ suivant cet algorithme.

$m =$

i \ j	1	2	3	4	5
1					
2					
3					
4					
5					

$s =$

j \ i	1	2	3	4
2				
3				
4				
5				

Reconstruire le meilleur parenthésage de ces cinq matrices.

On peut alors construire le parenthésage optimal grâce à l'algorithme suivant. En effet la dernière multiplication effectué dans le calcul optimal de $A_1 \cdots A_n$ est le produit de $A_1 \cdots A_{s[1,n]}$ et $A_{s[1,n]+1} \cdots A_n$. Et les multiplication antérieures peuvent se calculer récursivement, en effet $s[1, s[1, n]]$ donne la dernière multiplication pour le calcul de $A_1 \cdots A_{s[1,n]}$ et $s[s[1, n] + 1, n]$ celle pour calculer $A_{s[1,n]+1} \cdots A_n$.

Algorithm 2: Affichage du parenthésage optimal

```
1  def affichage(s, i, j):
2      if i == j then
3          print (A_i)
4      else
5          print ("(")
6          affichage(s, i, s[i, j])
7          affichage(s, s[i, j] + 1, j)
8          print (")")
```

La complexité de notre algorithme est en $O(n^3)$, puisqu'il contient trois boucle imbriquées qui font chacune n itérations. Un calcul plus minutieux, montre qu'en fait elle est en $\Theta(n^3)$.

Exercice 5.4. Implémenter l'algorithme de `best_cost2`.

Dans l'autre sens

Le plus gros inconvénient de notre méthode bovine (récursive) est de retraiter de très nombreuses fois inutilement des sous-problèmes. L'idée est d'alors de sauvegarder dans un tableau tous les résultats de sous-problèmes pour ne pas les recalculer. On appelle ça la mémoïsation.

À chaque réapparition d'un sous-problème dans l'arbre des appels récursifs, la valeur stockée dans le tableau est lue et retournée au programme principal.

Algorithm 3: Avec mémoïsation

```
1  def __memo_best_cost(p, i, j):
2      if m[i, j] < +∞ then
3          return m[i, j]
4      if i == j then
5          m[i, j] ← 0
6      else
7          for k ← i to j do
8              q ← __memo_best_cost(p, i, k) + __memo_best_cost(p, k + 1, j) + p_{i-1} p_k p_j
9              if q < m[i, j] then
10                 m[i, j] ← q
11     return m[i, j]
```

Algorithm 4: Avec mémoïsation

```
1 def memo_best_cost(p):
2     n ← len(p) − 1
3     for i ← 1 to n do
4         for j ← i to n do
5             m[i, j] ← +∞
6     return __memo_best_cost(p,1,n)
```

La fonction `__memo_best_cost(p, i, j)` retourne $m[i, j]$ mais ne la calcule que si ca n'a pas déjà été fait. L'initialisation de m avec la valeur `inf` permet de savoir si une valeur a déjà été calculée ou non.

La mémoïsation permet de passer d'un algorithme récursif en $\Omega(2^n)$ à un algorithme de complexité $O(n^3)$.

Exercice 5.5. Implémenter cet algorithme en python. On utilisera un dictionnaire pour stocker les valeurs de m.

5.4.2 Un exemple : Distance de Levenshtein

Introduction

Si Σ est un alphabet, c'est-à-dire un ensemble fini dont les éléments sont appelés des lettres, un mot de longueur $n \in \mathbb{N}^*$ de Σ est une suite finie formée de n lettres. On note Σ^+ l'ensemble des mots sur Σ, les mots de longueur 1 étant les lettres de Σ. Si $u \in \Sigma^*$, on note $|u|$ la longueur du mot u.

Par exemple $ATTAGC$ est un mot sur l'alphabet $\Sigma = \{A, T, C, G\}$ et $|ATTAGC| = 6$.

On appelle mot vide le mot noté ε de longueur nulle. Enfin on pose $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.

Si $w_1, w_2 \in^*$ on appelle distance de Levenshtein[iii] de w_1 à w_2 le nombre minimal de lettres qu'il faut insérer, retirer ou remplacer dans w_1 pour le transformer en w_2. On la note $d_L(w_1, w_2)$

Par exemple, pour passer de $faire$ à $aile$, il faut supprimer le f et transformer le l en r. Donc $d_L(faire, aile) \leq 2$, mais on peut se convaincre qu'on ne peut pas faire moins de deux opérations, de sorte que $d_L(faire, aile) = 2$.

On peut définir cette distance de manière récursive :

— $\forall u \in \Sigma^*$, $d_L(u, \varepsilon) = |u|$.
— $\forall v \in \Sigma^*$, $d_L(\varepsilon, v) = |v|$.
— $\forall u, v \in \Sigma^* \setminus \{\varepsilon\}$:

$$\forall a, b \in \Sigma,\ d_L(ua, vb) = \begin{cases} d_L(u, v) & \text{si} \quad a = b \\ 1 + \min\{d_L(ua, v), d_L(u, vb), d_L(u, v)\} & \text{si} \quad a \neq b \end{cases}$$

En effet, il n'y a que trois opérations : l'insertion, la suppression et le remplacement :

$$ua \begin{cases} \nearrow\ uab & \text{insertion} \quad \text{et} \quad d_L(ua, vb) = 1 + d_L(uab, vb) = 1 + d_L(ua, v) \\ \rightarrow\ ub & \text{remplacement} \quad \text{et} \quad d_L(ua, vb) = 1 + d_L(ub, vb) = 1 + d_L(u, v) \\ \searrow\ u & \text{suppression} \quad \text{et} \quad d_L(ua, vb) = 1 + d_L(u, vb) \end{cases}$$

Un premier algorithme récursif

Il est très simple d'implémenter directement le calcul de $d_L(u, v)$ pour deux mots u et v, il suffit d'écrire une fonction récursive :

```
def lev_rec(u, v):
      n, m = len(u), len(v)
      if n*m == 0:
            return max(n,m)
      if u[-1] == v[-1]:
            return lev_rec(u[:-1], v[:-1])
      else:
            return 1 + min(lev_rec(u[:-1], v),lev_rec(u, v[:-1]),
                           lev_rec(u[:-1], v[:-1]))
```

iii. On l'appelle aussi distance d'édition.

Complexité et chevauchement des sous-problèmes

Là encore la simplicité de l'implémentation se heurte à la complexité obtenue. En effet, observons combien d'appels récursifs sont faits pour calculer $d(bob, sophie)$, et sur quelle chaine sont faits ces appels. Pour ça on modifie légèrement le code :

```
calls = 0
d = {}
def lev_rec(u, v):
    global calls
    calls += 1
    d[(u, v)] = d.get((u, v),0)+1
    n, m = len(u), len(v)
    if n*m == 0:
          return max(n,m)
    if u[-1] == v[-1]:
          return lev_rec(u[:-1], v[:-1])
    else:
           return 1 + min(lev_rec(u[:-1], v),lev_rec(u, v[:-1]),
                          lev_rec(u[:-1], v[:-1]))
print("Distance : ", lev_rec('bob', 'sophie'))
print("Nombre d'appels recursifs : ", calls)
print("Nombre de sous-problemes : ", len(d))
```

On obtient alors :

```
Distance :  5
Nombre d'appels recursifs :  439
Arguments des appels  {('bob', 'sophie'): 1, ('bo', 'sophie'): 1,
('b', 'sophie'): 1, ('', 'sophie'): 1, ('b', 'sophi'): 5,
('', 'sophi'): 6, ('b', 'soph'): 13, ('', 'soph'): 18, ('b', 'sop'): 25,
('', 'sop'): 38, ('b', 'so'): 32, ('', 'so'): 57, ('b', 's'): 43,
('', 's'): 75, ('b', ''): 45, ('', ''): 43, ('bo', 'sophi'): 3, ('bo', 'soph'): 5,
('bo', 'sop'): 7, ('bo', 'so'): 9, ('bob', 'sophi'): 1,
('bob', 'soph'): 1, ('bob', 'sop'): 1, ('bob', 'so'): 1, ('bob', 's'): 1,
('bo', 's'): 2, ('bo', ''): 3, ('bob', ''): 1}
Nombre de sous-problemes :  28
```

On peut voir que l'algorithme calcule 75 fois la distance de s au mot vide, 43 fois celle de b à s et 25 fois celle de b à sop.

Si $T(n)$ est la complexité au pire de la fonction `lev_rec(u, v)` en fonction de $n = |u| + |v|$, alors $T(n)$ vérifie :

$$T(n) = \Theta(1) + 2T(n-1) + T(n-2).$$

Ce qui permet facilement d'encadrer $T(n)$:

$$\Theta(1) + 2T(n-1) \leq T(n) \leq \Theta(1) + 3T(n-1),$$

car $T(n-2) \leq T(n-1)$. Par substitution on trouve alors que $T(n) = O(3^n)$ et $T(n) = \Omega(2^n)$, ce qui est assez large mais suffisant pour nous décourager d'utiliser cet algorithme.

Là encore, cette complexité résulte du nombre très important de fois où les mêmes sous-problèmes sont retraités.

Exercice 5.6. Quelle est la complexité au meilleur ?

Mémoïsation

Comme pour le problème du parenthésage optimal des chaines de produits de matrices, nous pouvons réduire le nombre des appels récursifs en utilisant un cache.

Voici une implémentation possible avec un dictionnaire, dont les clefs sont les couples de mots sur lesquels on fait les appels.

```
calls = 0
cache = {}
d = {}
def lev_rec2(u, v):
      global calls
      calls += 1
      d[(u, v)] = d.get((u, v),0)+1
      n, m = len(u), len(v)
      if n*m == 0:
            return max(n,m)
      if u[-1] == v[-1]:
            cache[(u, v)] = lev_rec2(u[:-1], v[:-1])
                if (u[:-1], v[:-1]) not in cache else cache[(u[:-1], v[:-1])]
            return cache[(u, v)]
      else:
            cache[(u[:-1], v)] = lev_rec2(u[:-1], v)
                if (u[:-1], v) not in cache else cache[(u[:-1], v)]
            cache[(u, v[:-1])] = lev_rec2(u, v[:-1])
                if (u, v[:-1]) not in cache else cache[(u, v[:-1])]
            cache[(u[:-1], v[:-1])] = lev_rec2(u[:-1], v[:-1])
                if (u[:-1], v[:-1]) not in cache else cache[(u[:-1], v[:-1])]
            cache[(u, v)] = 1 + min(cache[(u[:-1], v)],
                    cache[(u, v[:-1])], cache[(u[:-1], v[:-1])])
            return cache[(u, v)]



print("Distance : ", lev_rec2('bob', 'sophie'))
print("Nombre d'appels récursifs : ", calls)
print("Arguments des appels ", d)
print("Nombre de sous-problèmes : ", len(d))
```

Vérifions alors l'efficacité de cette solution :

```
Distance :  5
Nombre d'appels récursifs :  28
Arguments des appels  {('bob', 'sophie'): 1, ('bo', 'sophie'): 1,
```

```
('b', 'sophie'): 1, ('', 'sophie'): 1, ('b', 'sophi'): 1, ('', 'sophi'): 1,
('b', 'soph'): 1, ('', 'soph'): 1, ('b', 'sop'): 1, ('', 'sop'): 1, ('b', 'so'): 1,
('', 'so'): 1, ('b', 's'): 1, ('', 's'): 1, ('b', ''): 1, ('', ''): 1, ('bo', 'sophi'): 1,
('bo', 'soph'): 1, ('bo', 'sop'): 1, ('bo', 'so'): 1, ('bob', 'sophi'): 1,
('bob', 'soph'): 1, ('bob', 'sop'): 1, ('bob', 'so'): 1, ('bob', 's'): 1,
('bo', 's'): 1, ('bo', ''): 1, ('bob', ''): 1}
Nombre de sous-problèmes :  28
```

Il apparait bien que de la sorte nous avons réduit le nombre d'appels récursifs au nombre de sous-problèmes. Nous justifierons précisément ce gain plus loin.

Complexité de la version récursive et version itérative

Notre dictionnaire de cache est indexé par des clefs qui sont des chaînes de caractères, ce n'est pas très pratique. Nous allons donc introduire une autre manière de calculer la distance qui nous intéresse entre deux mots, de manière à utiliser des coordonnées entières pour mieux visualiser comment ce cache se remplit.

Prenons deux mots $u = u_1 \dots u_m$ et $v = v_1 \dots v_n$, avec $m, n \in \mathbb{N}^*$ et posons :

$$d(i, j) = d_L(u_1 \dots u_i, v_1 \dots v_j).$$

Avec la convention que $d(0, j) = d_L(\varepsilon, v_1 \dots v_j)$ et $d_L(i, 0) = d(u_1 \dots u_i, \varepsilon)$.

Alors pour tout $(i, j) \in [\![1, i]\!] \times [\![1, j]\!]$:

$$d(i,j) = \begin{cases} d(i-1, j-1) & \text{si} \quad u_i = v_j \\ 1 + \min\{d(i, j-1), d(i-1, j), d(i-1, j-1)\} & \text{si} \quad u_i \neq v_j \end{cases}$$

Et nous avons $d_L(u, v) = d(m, n)$.
Pour calculer $d(m, n)$, nous remplissons un un tableau qui contiendra les nombres $d(i, j)$ calculés récursivement depuis la case $d(m, n)$ suivant le schéma de la figure (5.7).

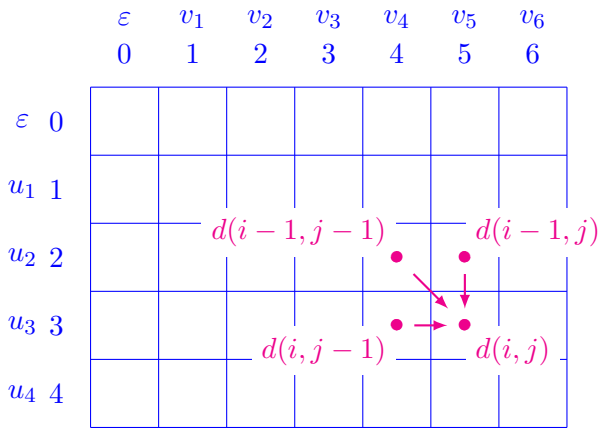

FIGURE 5.7 – Schéma récursif du calcul de $d(i, j)$.

Avec notre algorithme récursif nous remplissons le cache en partant de la case en bas à droite puis nous remettons jusqu'à avoir compléter toutes les cases.

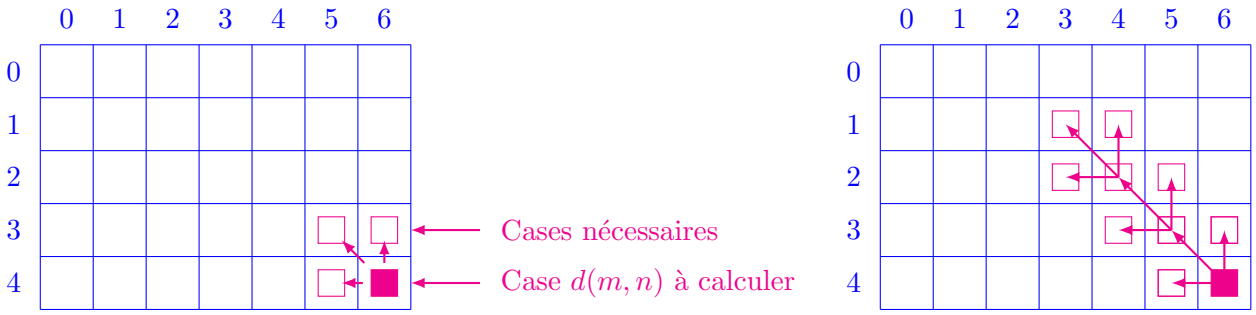

FIGURE 5.8 – Sens du remplissage de notre cache.

Chaque nombre $d(i,j)$ n'est calculé qu'une seule fois et le temps de calcul nécessaire est en $\Theta(1)$, autrement dit la complexité de notre algorithme est en $\Theta((1+|u|)\cdot(1+|v|)) = \Theta(|u|\cdot|v|)$. Au passage, on voit que sa complexité spatiale est aussi en $\Theta(|u|\cdot|v|)$ puisque l'on utilise essentiellement un tableau de $(1+|u|)\cdot(1+|v|)$ cases. Si la complexité au pire est largement meilleure (quadratique contre exponentielle), la complexité au meilleur est passée de linéaire à quadratique.

Quand on regarde le tableau, il apparait que la première ligne et la première colonne ne nécessitent aucun calcul pour être initialisées. En effet il s'agit de la distance au mot vide. Donc nous pouvons déjà les remplir, et à l'aide de ces valeurs nous pouvons remplir le tableau ligne par ligne jusqu'à arriver au calcul de $d(m,n)$.

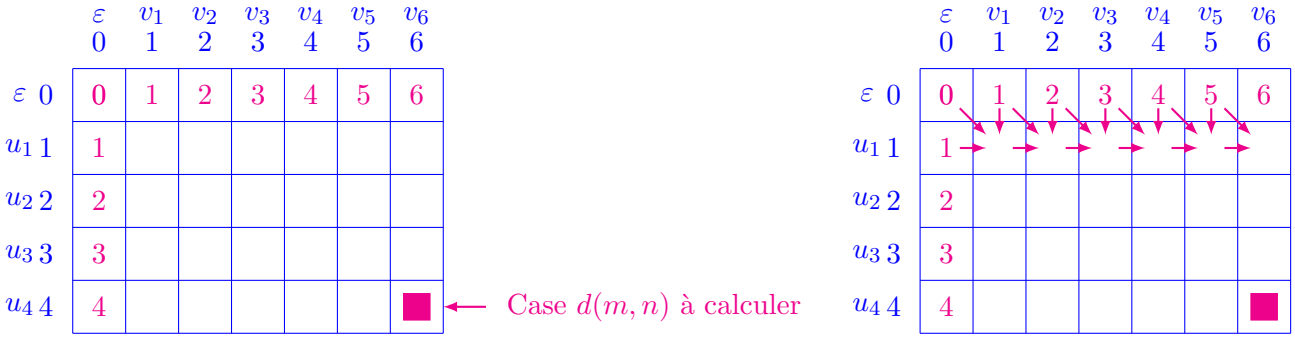

FIGURE 5.9 – Remplissage itératif du tableau.

Exercice 5.7. Calculer $d_L(bobs, sophie)$ en utilisant cette méthode.

		ε	s	o	p	h	i	e
		0	1	2	3	4	5	6
ε	0	0	1	2	3	4	5	6
b	1	1						
o	2	2						
b	3	3						
s	4	4						

Cette remarque nous permet d'écrire une version itérative du calcul de la distance de Levenshtein.

```
def lev_iter(u, v):
      m, n = len(u), len(v)
      tab = [[0 for __ in range(n+1)] for __ in range(m+1)]
      for j in range(n+1):
            tab[0][j] = j
      for i in range(m+1):
            tab[i][0] = i
      for i in range(1,m+1):
            for j in range(1,n+1):
                  if u[i-1] == v[j-1]:
                        tab[i][j] = tab[i-1][j-1]
                  else:
                        tab[i][j] = 1 + min(tab[i][j-1],tab[i-1][j],tab[i-1][j-1])
      return tab[m][n]
```

Là encore les complexités temporelle et spatiale sont en $\Theta(|u| \cdot |v|)$, pour s'en convaincre il suffit de remarquer que les calculs pour remplir une case se font en $\Theta(1)$.

Exercice 5.8. La complexité spatiale de cet algorithme peut être ramener à un $O(\min\{|u|, |v|\})$ en remarquant que l'on peut calculer chaque ligne en écrasant la précédente. Proposer une implémentation.

5.4.3 Élements de programmation dynamique

La programmation dynamique, comme la méthode « diviser-pour-régner », sert à résoudre des problèmes en combinant des solutions de sous-problèmes.

Elle fût introduite par Bellman dans les années 1950. Il faut comprendre le terme « programmation » dans le sens de « plannification » ou « optimisation ». Cette technique s'applique aux situations pour lesquelles il existe de sous-structures optimales. C'est-à-dire que l'on peut construire une solution optimale à partir des solutions optimales de sous-problèmes.

L'écriture d'un algorithme de programmation dynamique se divise en quatre étapes :

1. Caractériser la structure d'une solution optimale et casser le problème en sous-problèmes strictement plus petits.
2. Définir récursivement la valeur d'une solution optimale, et trouver des solutions aux sous-problèmes.
3. Calculer la valeur d'une solution optimale, à partir de de celles des sous-problèmes.
4. Reconstruire une solution optimale à partir des informations calculées.

Sous-structure optimale

Un problème admet une sous-structure optimale si une solution optimale au problème contient en elle-même des solutions optimales à des instances strictement plus petites du même problèmes.

Par exemple dans le cas des chaînes de multiplications de matrice, imaginons que la solution optimale du problème de la multiplication des matrices $A_1, \ldots, A_6$, soit $(A_1(A_2A_3))(A_4A_5)A_6)$. Alors nécessairement $(A_1(A_2A_3))$ et $(A_4A_5)A_6)$ sont les solutions optimales aux produits de $A_1, |dots, A_3$ et $A_4, \ldots, A_6$, sans quoi une meilleure solution serait envisageable pour $A_1, \ldots, A_6$.

Chevauchement des sous-problèmes

La grande différence avec la méthode de diviser-pour-régner qui s'applique lorsque les sous-problèmes sont indépendants, la programmation dynamique va s'appliquer lorsqu'au contraire les sous-problèmes vont se chevaucher.

Comme nous l'avons vu dans l'exemple du calcul de la distance de Levenshtein, le nombre de sous-problèmes qui réapparaissaient lors du traitement par récurrence était très importants.

Les deux approches

Une fois identifié une relation de récurrence pour résoudre le problème, on a deux choix :
L'approche Top-Down : qui consiste à traduire directement la relation de récurrence et à utiliser une fonction récursive et à mémoïser les appels. C'est souvent l'approche la plus simple à implémenter.

L'approche Bottom-Up : qui consiste à résoudre d'abord toutes les instances de taille 1, puis celles de taille 2, etc... Ce qui revient à remplir le cache de manière itérative.

Le coût entre les deux approches est le même, puisqu'il correspond à peu près à la taille du cache multiplié par le coût de calcul d'une case. Seulement les constantes ne sont généralement pas les mêmes pour les deux méthodes, ceux de la méthode Bottom-Up étant souvent bien meilleures.

5.4.4 Gérer le cache en Python

Python n'est pas un langage optimisé pour la récursivité, par exemple il ne s'est pas détecter qu'une fonction est récursive terminale[iv] et conserve les données de chaque niveau d'appel dans la pile. Son créateur Guido van Rossum pensait que le récursivité n'est pas à la base de la programmation. Malgré tout, le module `functools`, offre un décorateur `cache`[v] qui englobe une fonction avec un appelable mémoïsant qui enregistre les appels récents. Ce décorateur utilise un dictionnaire pour enregistrer le cache. Il s'utilise très simplement :

```
from functools import lru_cache

@cache
def ma_fonction_recursive(...):
        bla bla
        bla bla
        return ma_fonction_recursive(...)
```

Aux concours, il faudra certainement que gérer le cache à l'aide d'un dictionnaire ou d'un tableau. À ce sujet il faut bien faire attention d'optimiser au maximun l'utilisation du cache. Prenons un exemple :

```
def fibo(n):
    if n in [0,1]:
        return 1
    else:
        return fibo(n-1)+fibo(n-2)
```

iv. Une fonction est dite récursive terminale si la dernière instruction évaluée est l'appel récursif
v. C'est en fait équivalent à $\texttt{lru}_{c}ache(maxsize = None)$

Si l'on se contente d'écrire :

```
cache = {0:1,1:1}
def fibo_cache(n):
    if n in cache:
        return cache[n]
    else:
        cache[n] = fibo_cache(n-1)+fibo_cache(n-2)
        return cache[n]
```

Alors on n'a pas tiré profit du cache à plein. Il faut penser à remplacer un maximum d'appel récursif par une lecture du cache :

```
cache = {0:1,1:1}
def fibo_cache(n):
    if n in cache:
        return cache[n]
    else:
        cache[n] = fibo_cache(n-1)+cache[n-2]
        return cache[n]
```

5.5 Exercices

Exercice 5.9. Écrire une fonction utilisant le back-tracking pour trouver une chaine de transformation pour passer du mot u au mot v.

Exercice 5.10.
1. Ecrire une fonction récursive qui calcule $\binom{n}{k}$. Quelle est sa complexité au pire ?
2. Utiliser la programmation dynamique pour calculer $\binom{n}{k}$ par bottom-up. Quelle est la complexité de l'algorithme obtenu ?
3. Trouver un moyen de calculer $\binom{n}{k}$ en temps linéaire.

Exercice 5.11. Si $u = u_1 \dots u_m$ est un mot sur Σ^*, une sous-séquence du mot u est un mot $w = u_{i_1} \dots u_{i_k}$ avec $i_1 < i_2 < \dots < i_k$.

Étant donnée deux mots $u = u_1 \dots u_m$ et $v = v_1 \dots v_n$, on cherche la longueur d'une plus longue sous-séquence commune aux deux mots u et v.

1. Exhiber une sous-structure optimale.
2. Donner une définition récursive de la longueur d'une plsc.
3. Écrire une fonction `plsc(u: str, v: str)-> int` qui calcule, par la méthode bottom-up, cette longueur.
4. Modifier votre fonction `plsc` pour qu'elle retourne en plus de la longueur d'une plsc, une plsc.

Exercice 5.12. Soit $G = (S, A)$ un graphe orienté avec $S = \{s_1, \dots, s_n\}$. On dit que G est ordonné si il vérifie les deux propriétés suivantes :

1. Chaque arc de ce graphe est de la forme $(i \to j)$ si $i < j$
2. Tous les sommets sauf le sommet s_n ont au moins un arc sortant.

Ici, par souci de simplification, nous supposerons qu'il existe un chemin allant de s_i vers s_n pour tout $i = 1, \ldots, n$. L'objectif est de trouver le chemin le plus long entre les sommets s_1 et s_n.

1. L'algorithme glouton suivant résout-il correctement le problème ?

Algorithm 5: Algorithme Glouton

```
1 def Longueur_plus_long_chemin_glouton(G):
2     s ← s_1
3     ℓ = 0
4     while Γ⁺(s) ≠ ∅ do
5         j ← min{i | (s, s_i) ∈ A}
6         s ← s_j
7         ℓ = ℓ + 1
8     return ℓ
```

2. (a) Si $\mu(s_1, s_\ell)$ est le plus long chemin de s_1 à s_ℓ, et si s_j est l'avant dernier sommet de ce chemin. Expliquer pourquoi le sous-chemin de $\mu(s_1, s_\ell)$ d'extrémités s_1 à s_j est le plus long chemin de s_1 à s_j dans G.

   (b) Si $\text{lplc}(j)$ désigne la longueur du plus long chemin de s_1 à s_j, avec la convention que $\text{lplc}(j) = -1$ s'il n'existe pas de chemin de s_1 à s_j, justifier les relations :

$$\text{lplc}(\ell) = \begin{cases} 1 + \max\{\text{lplc}(j) \mid s_j \in \Gamma^-(s_\ell)\} & \text{si} \quad \ell \neq 1 \\ 0 & \text{si} \quad \ell = 1 \end{cases}$$

   (c) Écrire un algorithme qui calcule $\text{lplc}(n)$ en utilisant ces relations.

   (d) Modifier votre algorithme pour qu'il retourne le chemin le plus long.

5.6 Corrections

Corrigé 5.3 On a :

i	0	1	2	3	4	5
p	10	15	8	2	5	10

Un exemple de calcul :

$$m[2,4] = \min_{i \le k < j} \begin{cases} m[2,2] + m[3,4] + p_1 p_2 p_4 & = & 0 + 80 + 15 \times 8 \times 5 & = & 680 \\ m[2,3] + m[4,4] + p_1 p_3 p_4 & = & 240 + 0 + 15 \times 2 \times 5 & = & 390 \end{cases} = 390 \text{ et } s[2,4] = 3$$

$m =$

i \ j	1	2	3	4	5
1	0	1200	540	640	840
2		0	240	390	640
3			0	80	260
4				0	100
5					0

et $s =$

i \ j	2	3	4	5
1	1	1	3	3
2		2	3	3
3			3	3
4				4

Le meilleur parenthésage est $((A_1(A_2A_3))(A_4A_5))$.

Corrigé 5.4 Le code :

```
def best_cost2(p):
    inf = float('inf')
    n = len(p)-1
    print(n)
    m = [[inf for __ in range(n+1)] for __ in range(n+1)]
    s = [[0 for __ in range(n+1)] for __ in range(n+1)]
    for i in range(1,n+1):
        m[i][i] = 0
    for l in range(2,n+1):
        for i in range(1,n-l+2):
            j = i+l-1
            for k in range(i,j):
                q = m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j]
                if q < m[i][j]:
                    m[i][j] = q
                    s[i][j] = k
    return m, s
```

```
def affichage(s,i,j):
    if i == j:
        print("A{}".format(i), end='')
    else:
        print("(", end='')
        affichage(s, i, s[i][j])
```

```
        affichage(s, s[i][j]+1, j)
        print(")", end='')
```

Corrigé 5.5 Le code :

```
m = dict()
def memo_best_cost(p):

    inf = float('inf')
    n = len(p)-1
    for i in range(1,n+1):
        for j in range(i,n+1):
            m[(i,j)] = inf
    return __memo_best_cost(p,1,n,n)

def __memo_best_cost(p, i, j,n):
    inf = float('inf')

    if m[(i,j)] < inf:
        return m[(i,j)]
    if i == j:
        m[(i,j)] = 0
    else:
        for k in range(i,j):
            q = __memo_best_cost(p,i,k,n)+__memo_best_cost(p,k+1,j,n)+p[i-1]*p[k]*p[j]
            if q < m[(i,j)]:
                m[(i,j)] = q
    return m[(i,j)]
```

Corrigé 5.9 Le code :

```
def lev_iter_diff(u, v):
      path = []
      m, n = len(u), len(v)
      tab = [[0 for __ in range(n+1)] for __ in range(m+1)]
      for j in range(n+1):
            tab[0][j] = j
      for i in range(m+1):
            tab[i][0] = i
      for i in range(1,m+1):
            for j in range(1,n+1):
                  if u[i-1] == v[j-1]:
                        tab[i][j] = tab[i-1][j-1]
                  else:
                        tab[i][j] = 1 + min(tab[i-1][j-1],tab[i-1][j],tab[i][j-1])
```

```
    i = m
    j = n

    while i>0 or j>0:

        if tab[i][j] == tab[i-1][j-1]:
            path.append("Conserver {}. Cout 0".format(v[j-1]))
            i -= 1
            j -= 1

        else:
            ins = tab[i][j-1]
            sup = tab[i-1][j]
            rep = tab[i-1][j-1]
            if tab[i][j] - 1 == ins:
                path.append("Insérer {}.Cout 1".format(v[j-1]))
                j -= 1
            elif tab[i][j] - 1 == sup:
                path.append("Supprimer {}. Cout 1".format(u[i-1]))
                i -= 1
            else :
                path.append("Remplacer {} par {}. Cout 1".format(u[i-1], v[j-1]))
                i -= 1
                j -= 1

    for p in path[-1::-1]:
        print(p)

    return tab[m][n]
```

Corrigé 5.11

1. Si $u = u_1 \dots u_m$ et $v = v_1 \dots v_n$ et si $z = z_1 \dots z_k$ est une plsc à u et v alors :
   - Si $u_m = v_n$ alors $z_k = u_m = v_n$ et $z_1 \dots z_{k-1}$ est une plsc de $u_1 \dots u_{m-1}$ et $v_1 \dots v_{n-1}$.
   - Si $u_m \neq v_n$ et $z_k \neq u_m$ alors z est une plsc de $u_1 \dots u_{m-1}$ et de v.
   - Si $u_m \neq v_n$ et $z_k \neq v_n$ alors z est une plsc de $v_1 \dots v_{n-1}$ et de u.
2. Pour $0 \leq i \leq m$ et $0 \leq j \leq n$ on a :

$$\ell[i,j] = \begin{cases} 0 & \text{si} \quad i = 0 \text{ ou } j = 0 \\ \ell[i-1, j-1] + 1 & \text{si} \quad i,j > 0 \text{ et } u_i = v_j \\ \max\left(\ell[i, j-1], \ell[i-1, j]\right) & \text{si} \quad i,j > 0 \text{ et } u_i \neq v_j \end{cases}$$

3. Le code :

```
def plcs(u : str, v: str)-> int:
    m =len(u)
    n = len(v)
    l = [[0 for __ in range(m+1)] for __ in range(n+1)]
    for i in range(m):
        for j in range(n):
```

```
            if u[i] == v[j]:
                l[i+1][j+1] = l[i][j] + 1
            else:
                l[i+1][j+1] = max(l[i][j+1],l[i+1][j])

    return l[m][n]
```

4. Le code :

```
def plcs2(u, v):
    m =len(u)
    n = len(v)
    l = [[0 for __ in range(m+1)] for __ in range(n+1)]
    b = [[0 for __ in range(m+1)] for __ in range(n+1)]
    for i in range(m):
        for j in range(n):
            if u[i] == v[j]:
                l[i+1][j+1] = l[i][j] + 1
                b[i+1][j+1] = 0
            elif l[i][j+1] >= l[i+1][j]:
                l[i+1][j+1] = l[i][j+1]
                b[i+1][j+1] = 1
            else:
                l[i+1][j+1] = l[i+1][j]
                b[i+1][j+1] = 2
    path = []
    def build_plcs(b, u, i, j):
        nonlocal path
        if i*j == 0:
            return
        if b[i][j] == 0:
            build_plcs(b,u, i-1,j-1)
            path.append(u[i-1])
        elif b[i][j] == 1:
            build_plcs(b,u, i-1,j)
        else:
            build_plcs(b,u, i,j-1)

    build_plcs(b, u, m,n)

    return l[m][n], "".join(path)
```