1 Un premier exemple simple : la suite de Fibonnaci

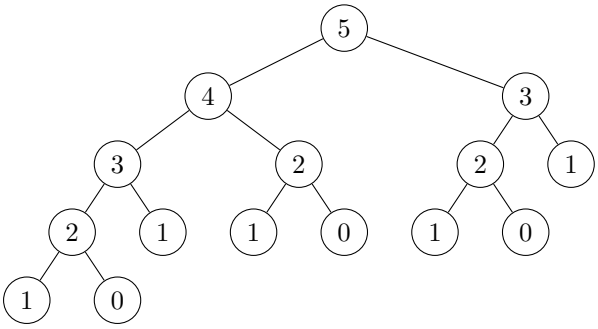
La suite de Fibonnaci est la suite définie par récurrence par les relations

$$F_0 = F_1 = 1 \qquad \text{et} \qquad \forall n \geq 2, \quad F_n = F_{n-1} + F_{n-2}$$

On cherche à écrire une fonction qui prend en argument l'entier n et renvoie la valeur de F_n. Une version récursive naïve s'écrit en quelques lignes.

```
def F(n):
    if n<=1:
        return 1
    else:
        return F(n-1)+F(n-2)
```

En revanche, on constatera immédiatement que le calcul de F_{40} prend un temps relativement important (de l'ordre de la minute), quand F_{50} prend plus d'une heure (à la louche, pas testé en fait). L'explication est classique : cette implémentation lance de nombreuses fois les mêmes appels récursifs, comme le montre le graphe ci-dessous (pour le calcul de F_5). L'arbre se lit « en profondeur d'abord », c'est-à-dire qu'on explore toujours intégralement la partie gauche d'un noeud avant sa partie droite.



Précisément, si l'on note $T(n)$ le nombre d'additions effectuées lors du calcul de `fibo n`, on a la formule de récurrence

$$\forall n \geq 2, \qquad T(n) = T(n-1) + T(n-2) + 1$$

Les méthodes de résolution des récurrences linéaires d'ordre 2 permettent d'en déduire que

$$T(n) \sim \left(\frac{1+\sqrt{5}}{2}\right)^{n+1}$$

soit une complexité exponentielle. On fait donc de l'ordre d'un milliard de calcul pour déterminer F_{50} !

Cet exemple illustre un problème fréquemment rencontré lorsque l'on cherche à résoudre des problèmes par une méthode récursive naïve. Résoudre récursivement un problème consiste dans les grandes lignes à introduire des sous-problèmes, à mettre en place une méthode simple pour résoudre les « petits » problèmes, et une seconde méthode pour déterminer la solution d'un « gros » problème à partir des solutions des sous-problèmes. La méthode diviser pour régner est basée sur ce principe, mais n'est efficace que s'il n'est jamais nécessaire de résoudre plusieurs fois le même problème (c'est le cas par exemple pour le tri-fusion, puisque qu'on ne triera jamais deux fois la même sous-liste). Lorsque les sous-problèmes se chevauchent (comme ici, où les calculs de F_3 et F_4 nécessitent tous deux le calcul de F_2, la récursivité naïve est inefficace. Une méthode simple pour éviter des appels récursifs inutile consiste à stocker au fur et à mesure les solutions des sous-problèmes, pour ne pas avoir à les recalculer. C'est le principe même de la programmation dynamique. Il existe en général deux implémentations

Méthode ascendante (bottom-up)

La méthode ascendante consiste à résoudre les sous-problèmes en partant des plus simples pour terminer par les plus compliqués. La principale contrainte est de devoir déterminer l'ordre dans lequel les sous-problèmes doivent être résolus.

En ce qui concerne la suite de Fibonnaci, c'est élémentaire. Il suffit de calculer tous les termes F_k pour k allant de 0 à n, chaque calcul utilisant les deux dernières valeurs calculées. On peut donc écrire

```
def F(n):
    L=[1,1]
    for k in range(n-1):
        L.append(L[-1]+L[-2])
    return L[-1]
```

La complexité est clairement linéaire. En pratique, il suffirait de garder en mémoire les deux derniers termes calculés, ce qui occuperait une mémoire constante (la version ci-dessus utilise un espace mémoire en $O(n)$). Mais pour une autre suite récurrente pour laquelle le n-ième terme u_n dépendrait de tous les termes $u_0, \ldots, u_{n-1}$, il faudrait stocker comme ci-dessus l'ensemble des valeurs de la suite.

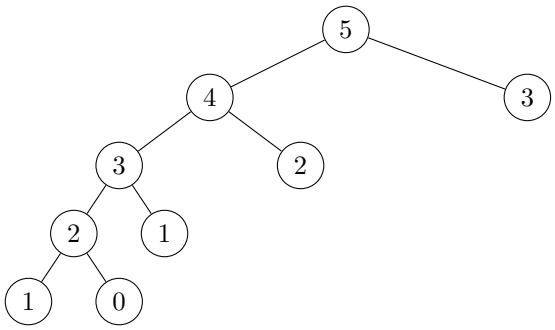
Méthode descendante (top-bottom)

Dans cette implémentation, on ne s'embarasse pas à chercher un ordre de résolution. On utilise à nouveau une fonction récursive, si ce n'est qu'en parrallèle, on stocke les résultats des appels récursifs en mémoire lorsque celui-ci s'exécute pour la première fois. En pratique, la fonction récursive commence donc par tester si la valeur a déjà été calculée. Si c'est le cas, elle renvoie directement la valeur (sans lancer de récursivité). Sinon, elle lance les appels récursifs nécessaires, calcule le résultat, **le stocke** et enfin le renvoie. Ce principe porte le nom de **mémoïsation**.

En ce qui concerne la suite de Fibonnaci, on peut donc proposer l'implémentation suivante.

```
def F(n):
    L=[None for i in range(n+1)]
    L[0]=1; L[1]=1
    def aux(k):
        if L[k]==None:
            L[k]=aux(k-1)+aux(k-2)
        return L[k]
    return aux(n)
```

Cette implémentation n'empêche pas de lancer plusieurs fois le même appel récursif, mais en supprime la majeure partie. Pour le calcul de F_5 par exemple, les appels effectués sont les suivants :

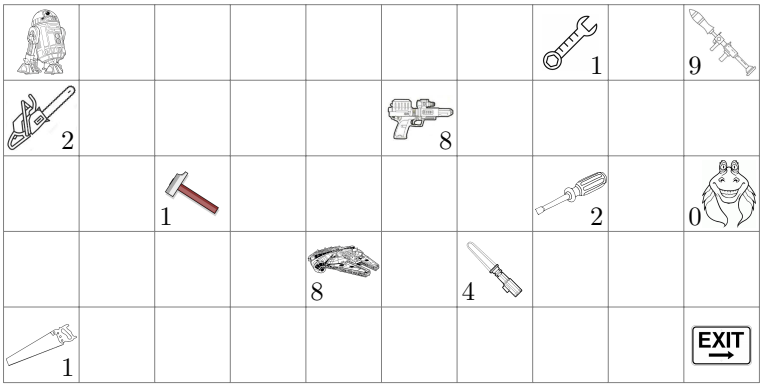


Il y a donc bien un nombre d'appels en $O(n)$ et une complexité linéaire.

2 Deplacement optimal d'un robot

2.1 Problématique

Un petit robot doit se déplacer sur un damier à p lignes et q colonnes, en partant du coin en haut à gauche (d'indices $(0,0)$) jusqu'au coin en bas à droite (d'indices $(p-1, q-1)$). Le robot ne peut effectuer des pas que de la gauche vers la droite ou du haut vers le bas. Sur le damier sont disseminés un certain nombre d'objets auxquels sont attribués une utilité représentée par une valeur entière strictement positive et que le robot peut ramasser sur son passage. L'objectif est alors de déterminer le trajet pour lequel la somme des utilités des objets ramassés est maximale.



Le nombre total de chemins jusqu'à la sortie est donné par $\binom{p+q}{p}$. En effet, un tel trajet comporte nécessairement p pas vers la droite et q vers le bas, et est entièrement déterminé par les positions des p pas vers la droite parmi les $p+q$ pas au total. Lorsque $p=q$, l'ordre de grandeur est un $O(4^p/\sqrt{p})$, donc exponentiel. Une recherche exhaustive de minimum sur l'ensemble des trajets possibles est donc inenvisageable. On doit trouver une autre méthode.

2.2 Sous-structures optimales et relations de récurrence

Pour obtenir une méthode efficace pour résoudre le problème, l'idée consiste dans un premier temps non pas à chercher le chemin optimal jusqu'à la sortie, mais plus généralement jusqu'à n'importe quelle case. Dans toute la suite, on note pour tout $(i,j) \in [\![0; p-1]\!] \times [\![0; q-1]\!]$

- $m_{i,j}$ la somme maximale des utilités que l'on peut récupérer sur un trajet jusqu'à la case (i,j).
- $C_{i,j}$ l'ensemble des chemins d'utilité maximale aboutissant à la case (i,j). Un tel chemin peut être représenté par la liste des cases traversées, ou par la liste des pas effectués (on peut représenter un pas par un caratère : "b" pour bas, "d" pour droite). Sur l'exemple précédent, on peut par exemple proposer deux chemins optimaux différents jusqu'au faucon millenium :
  - `[(0,0), (1,0), (2,0), (2,1), (2,2), (2,3), (2,4), (3,4)]` par liste des cases
  - `["b", "d", "b", "d", "b", "d", "d"]` par liste des pas

  Dans toute la suite de cette partie, on utilisera la première réprésentation.

La valeur de l'utilité de l'objet dans la case (i,j) est notée $u_{i,j}$. Notons que $u_{i,j}=0$ s'il n'y a pas d'objet sur cette case. Par convention, il n'y a pas d'objet sur la première case et la sortie, de sorte que $u_{0,0} = u_{p-1,q-1} = 0$ (ça ne change rien si on en met un). De plus, $m_{0,0}=0$. Justifions maintenant l'égalité suivante :

Proposition 1

Pour tous $i,j \geq 1$: $\quad m_{i,0} = m_{i-1,0} + u_{i,0} \qquad m_{0,j} = m_{0,j-1} + u_{0,j}$

et $\quad m_{i,j} = \max\{m_{i-1,j}, m_{i,j-1}\} + u_{i,j}$

Remarque 1

L'idée derrière cette relation de récurrence est assez simple. Fixons $i,j \geq 1$ et considérons un chemin optimal C jusqu'à la case (i,j) (il n'est pas nécessairement unique). Nécessairement, ce chemin passe par l'une des cases $(i-1,j)$ ou $(i,j-1)$. Supposons que l'on soit dans le premier cas et notons $C = [(0,0),\dots,(i-1,j),(i,j)]$. Le début du chemin $C' = [(0,0),\dots,(i-1,j)]$ est nécessairement lui aussi optimal. Sinon, on pourrait le remplacer par un chemin C'' d'utilité strictement supérieur. En ajoutant (i,j) à la fin de C'', on obtiendrait un chemin d'utilité strictement supérieure à celle de C aboutissant en case (i,j), ce qui contredit la définition de C. Puisque C' est optimal, son temps de parcours est alors $m_{i-1,j}$, ce qui prouve que le temps de parcours $m_{i,j}$ de C est $m_{i-1,j} + u_{i,j}$. Dans le deuxième cas, on aboutit à l'égalité $m_{i,j} = m_{i,j-1} + u_{i,j}$. La subtilité pour une preuve irréprochable, c'est la gestion de cette disjonction de cas pour aboutir rigoureusement à la formule avec le « max ». La preuve rigoureuse proposée ci-dessous passe par une double inégalite.

Preuve : Les deux premières formules pour $i=0$ ou $j=0$ sont évidentes. En effet, les déplacements du robots imposent que pour aller à la case $(i,0)$ (resp. $(0,j)$), on ne peut que provenir de la case $(i-1,0)$ (resp. $(0,j-1)$). On peut alors reprendre les arguments de la remarque précédente, mais il n'y a cette fois aucune disjonction de cas à traiter et donc directement l'égalité. Dans toute la suite, on considère donc $i,j \geq 1$.

$\boxed{\leq}$ Ce sens découle directement de la remarque précédente. On démontre en effet que

$$m_{i,j} = \begin{cases} m_{i-1,j} + u_{i,j} & \text{si il existe un chemin optimal vers } (i,j) \text{ passant par } (i-1,j) \\ m_{i,j-1} + u_{i,j} & \text{si il existe un chemin optimal vers } (i,j) \text{ passant par } (i,j-1) \end{cases}$$

Dans les **deux** cas, on a bien en particulier

$$m_{i,j} \leq \max\{m_{i-1,j} + u_{i,j}, m_{i,j-1} + u_{i,j}\} = \max\{m_{i-1,j}, m_{i,j-1}\} + u_{i,j}$$

$\geq$ Pour l'inégalité réciproque, considérons maintenant un chemin optimal jusqu'à la case $(i-1,j)$, qui est donc d'utilité $m_{i-1,j}$. En ajoutant (i,j) à la fin de ce parcours, on obtient un chemin jusqu'à la case (i,j) (pas nécessairement optimal à priori), d'utilité $m_{i-1,j}+u_{i,j}$. Par définition, $m_{i,j}$ est la maximum des utilités des chemins menant à (i,j). On en déduit donc que

$$m_{i,j} \geq m_{i-1,j} + u_{i,j}$$

Un raisonnement similaire à partir d'un chemin optimal jusqu'à la case $(i,j-1)$ montre que l'on a également

$$m_{i,j} \geq m_{i,j-1} + u_{i,j}$$

et donc $$m_{i,j} \geq \max\{m_{i-1,j}+u_{i,j}, m_{i,j-1}+u_{i,j}\} = \max\{m_{i-1,j}, m_{i,j-1}\} + u_{i,j}$$

Finalement $$m_{i,j} = \max\{m_{i-1,j}, m_{i,j-1}\} + u_{i,j}$$ ... □

La formule de récurrence précédente permet maintenant de calculer de proche en proche les valeurs $(m_{i,j})_{i,j\in[\![0;p-1]\!]\times[\![0;q-1]\!]}$. Elle ne donne toutefois pas immédiatement un moyen de reconstituer un chemin optimal jusqu'à la sortie. Pour cela, il suffit de remarquer que, compte tenu de la proposition précédente et de sa preuve, on en déduit que les élément de $C_{i,j}$ peuvent s'obtenir de la manière suivante :

- Si $i=j=0$, le chemin $(0,0)$ est le seul chemin jusqu'à $(0,0)$. Il est donc optimal et $C_{0,0}=\{[(0,0)]\}$.
- Si $i=0$, on obtient $C_{0,j}$ en concatènant $(0,j)$ à la fin de chaque élément de $C_{0,j-1}$. Le cas $j=0$ est symétrique.
- Pour $i,j \geq 1$, on compare $m_{i-1,j}$ et $m_{i,j-1}$. Si $m_{i,j-1} > m_{i-1,j}$, on obtient $C_{i,j}$ en ajoutant (i,j) à la fin d'un élément quelconque de $C_{i,j-1}$. Si $m_{i,j-1} < m_{i-1,j}$, il faut ajouter (i,j) à la fin de chaque élément de $C_{i-1,j}$. En cas d'égalité, on réunit $C_{i,j-1}$ et $C_{i,j}$ et on ajoute (i,j) à la fin de chaque élément.

2.3 Calcul du temps et du chemin optimal

Cette dernière partie propose l'implémentation de deux méthodes de programmation dynamique pour déterminer un plus court chemin vers la sortie. Les données $(u_{i,j})_{(i,j)\in[\![0;p-1]\!]\times[\![0;q-1]\!]}$ sont données sous la forme d'une matrice `U` (liste de listes) telle que pour tous (i,j), $U[i][j]$ contient la valeur $u_{i,j}$. Dans les deux cas, l'objectif est de calculer un dictionnaire `M` contenant les quantités $m_{i,j}$. Une fois ce dictionnaire obtenu, un dernier programme (indépendant de la méthode utilisée) permettra de reconstituer un chemin optimal vers la sortie à partir de `M`.

Méthode impérative (bottom-up)

Pour cette implémentation, il suffit de choisir un ordre de résolution des sous-problèmes consistant à arriver à une case quelconque. La formule de la proposition 1 nécessite pour calculer $m_{i,j}$ de connaître $m_{i-1,j}$ et $m_{i,j-1}$. On peut alors constater qu'un calcul ligne par ligne (ou colonne par colonne) des $(m_{i,j})_{i,j\in[\![0;p-1]\!]\times[\![0;q-1]\!]}$ est compatible avec cette relation de récurrence. On traite dans un premier temps la première ligne et la première colonne, puis le cas général.

```
1  def construit_dico(U):
2      p=len(U); q=len(U[0])
3      M={}
4      M[(0,0)]=0
5      for j in range(1,q):
6          M[(0,j)]=M[(0,j-1)]+U[0][j]
7      for i in range(1,p):
8          M[(i,0)]=M[(i-1,0)]+U[i][0]
9      for i in range(1,p):
10         for j in range(1,q):
11             M[(i,j)]=max(M[(i-1,j)],M[(i,j-1)]) + U[i][j]
12     return(M)
```

Chaque boucle de cette fonction n'exécute que des opérations élémentaires. La première boucle s'exécute q fois, la seconde p fois, la ligne 9 lance p fois une boucle qui s'exécute q fois. La complexité totale est donc un $O(p)+O(q)+O(p\cdot q)$ soit en $O(p\cdot q)$. La construction est donc de coût quadratique.

Méthode récursive (top-bottom)

La méthode top-bottom ne nécessite pas de déterminer l'ordre de résolution des sous-problèmes. On utilise une sous-fonction récursive qui se chargera de lancer la résolution des sous-problèmes nécessaires à partir du problème initial (chemin jussqu'à la sortie). On crée un dictionnaire initialement vide qui servira à stocker les solutions. La sous-fonction récursive commence par consulter ce dernier pour savoir si la solution n'a pas déjà été calculée. Si ce n'est pas le cas, elle lance les appels récursifs nécessaires, puis stocke le résultat obtenu dans le dictionnaire (principe de mémoïsation).

```
def construit_dico(U):
    p=len(U); q=len(U[0])
    M={}
    def aux(i,j):
        if M.get((i,j))!=None:
            return M[(i,j)]
        elif i==0 and j==0:
            M[(0,0)]=0
            return 0
        elif i==0:
            M[(0,j)]=aux(0,j-1)+U[i][j]
            return M[(0,j)]
        elif j==0:
            M[(i,0)]=aux(i-1,0)+U[i][j]
            return M[(i,0)]
        else:
            M[(i,j)]=max(aux(i-1,j),aux(i,j-1)) + U[i][j]
            return M[(i,j)]
    opt=aux(p-1,q-1)
    return(M)
```

Dans la version ci-dessus, la fonction `aux` renvoie l'utilité maximale pour un chemin jusqu'à (i, j). On aurait aussi pu écrire une fonction qui ne renvoie pas de résultat mais se contente de remplir le dictionnaire. On admettra que cette fonction est également de complexité $O(p \cdot q)$.

Reconstitution du chemin optimal

Il reste pour finir à reconstruire un chemin optimal jusqu'à la sortie. On procède récursivement à partir de cette case en utilisant les remarques en fin de partie 2.2

```
def reconstruit(M,i,j):
    if i==0 and j==0:
        return [(0,0)]
    elif i==0:
        return (reconstruit(M,0,j-1)+[(i,j)])
    elif j==0:
        return (reconstruit(M,i-1,0)+[(i,j)])
    elif M[(i,j-1)]>M[(i-1,j)]:
        return (reconstruit(M,i,j-1)+[(i,j)])
    else:
        return (reconstruit(M,i-1,j)+[(i,j)])
```

A chaque appel récursif, soit i diminue de 1, soit j diminue de 1, et la récursivité s'arrête lorsque $i = j = 0$. Il y a donc exactement $p + q - 1$ appels récursifs. Le reste n'étant que des opérations élémentaires, la complexité est en $O(p + q)$ soit linéaire (dès lors que le dictionnaire `M` est construit).

Il suffit pour finir de composer les fonctions.

```
def chemin_optimal(U):
    p=len(U); q=len(U[0])
    M=construit_dico(U)
    return reconstruit(M,p-1,q-1)
```

La complexité finale de la méthode est donc $O(p \cdot q) + O(p + q)$ soit $O(p \cdot q)$, donc de coût quadratique.

Exercice 1

On conserve la même problématique, mais on rajoute des obstacles sur le parcours en supprimant certaines cases. La quantité `U[i][j]` vaut maintenant (-1) lorsqu'un case est supprimée : le robot ne peut plus emprunter cette case lors de son trajet. Expliquer comment modifier l'énoncé de la proposition 1 dans ce cas de figure, et modifier l'algorithme de façon à ce fournisse à nouveau un trajet optimal en tenant compte de ces nouvelles contraintes.

3 Plus longue sous-séquence commune

Soit `E` un ensemble et $X = (x_1, x_2, \ldots, x_n)$ une suite finie d'éléments de `E`. Dans toute la suite, on parlera de **séquence** d'éléments de `E`. Une sous-séquence Z de X est une suite obtenue en supprimant des éléments de X (éventuellement zéro). Plus formellement, cela signifie qu'il existe un entier k et une suite d'indices $i_1 < \ldots < i_k$ telle que $z_r = x_{i_r}$ pour tout $r \in [\![1; k]\!]$. Etant donné une autre séquence $Y = (y_1, y_2, \ldots, y_m)$, on dit que Z est une sous-séquence commune à X et Y si c'est une sous-séquence à la fois de X et de Y. A titre d'exemple, avec `E` $= \{0, 1\}$ et

$$X = 0, 1, 1, 0, 1, 1, 0, 1, 1 \qquad \text{et} \qquad Y = 1, 1, 1, 0, 0, 0$$

les sous-séquences de X et de Y sont

$$0 \quad 1 \quad 1, 0 \quad 0, 0, 0 \quad 1, 0, 0 \quad 1, 1, 0 \quad 1, 1, 1 \quad 1, 1, 0, 0 \quad 1, 1, 1, 0$$

En biologie, on est souvent amené à comparer des morceaux d'ADN de deux organismes. Un échantillon d'ADN est une suite de molécules appelées **bases**. Il existe quatre bases différentes : l'adénine, la guanine, la cytosine et la thymine. Si l'on représente chacune de ces bases par leurs initiales, on peut représenter un brin d'ADN par une séquence prise sur l'ensemble `E = {A, C, G, T}`. Comparer des échantillons d'ADN permet de déterminer leur degrés de « similitude », qui mesure d'une certaine manière la façon dont les organismes sont apparentés. On considère donc par exemple que deux brins sont d'autant plus proches qu'ils admettent une sous-séquence commune de très grande longueur, ce qui amène à chercher un algorithme permettant de déterminer la plus longue sous-séquence commune (PLSC) de deux séquences.

Dans toute la suite, étant donné une séquence $X = (x_1, \ldots, x_n)$, on notera X_k la sous-séquence $(x_1, \ldots, x_k)$ de X, avec X_0 la séquence vide par convention.

Exercice 2 : Sous-structure optimale d'une PLSC

Soient $Z = (z_1, \ldots, z_k)$ une plus longue sous-séquence commune de deux séquences $X = (x_1, \ldots, x_n)$ et $Y = (y_1, \ldots, y_m)$. Justifier que

- Si $x_n = y_m$, alors $z_k = x_m = y_n$ et Z_{k-1} est une PLSC de X_{n-1} et Y_{m-1}.
- Si $x_n \neq y_m$, alors Z est une PLSC de X_{n-1} et Y, ou de X et Y_{m-1}.

En déduire un algorithme pour déterminer une plus longue sous-séquence commune de deux séquences X et Y données en argument sous la forme de chaînes de caractères.

Réponse : Distinguons les deux cas de l'énoncé :

- Si $x_n = y_m \neq z_k$, alors Z ne contient pas la dernière lettre commune à X et Y. On peut donc la rajouter à Z et obtenir une sous-séquence de X et Y strictement plus longue que Z. C'est absurde. Ainsi, $z_k = x_n = y_m$.

  Il est clair qu'alors Z_{k-1} est une sous-séquence de X_{n-1} et Y_{m-1}. S'il ne s'agit pas d'une sous-séquence de longueur maximale, on peut en trouver une autre W de longueur strictement plus grande, et lui rajouter la dernière lettre $x_n = y_m$. On obtient alors une sous-séquence commune à X et Y strictement plus longue que Z ce qui est à nouveau absurde. Ainsi, Z_{k-1} est une PLSC de X_{n-1} et Y_{m-1}.
- Si $x_n \neq y_m$, alors nécessairement $z_k \neq x_n$ ou $z_k \neq y_m$. Considérons par exemple le premier cas. Puisque Z ne contient pas la dernière lettre de X, c'est nécessairement une sous-séquence de X_{n-1}. C'est donc une sous-séquence commune à X_{n-1} et Y. Elle est de longueur maximale car s'il existait une sous-séquence W strictement plus longue, W serait également une sous-séquence de X et Y ce qui contredirait la maximalité de Z. Ainsi, Z est une PLSC de X_{n-1} et Y.

  Le cas $z_k \neq x_n$ est similaire et on justifie alors que Z est une PLSC de X et Y_{m-1}.

Pour résoudre le problème de la plus longue sous-séquence commune, on introduit l'ensemble des sous-problèmes consistant à trouver la plus longue sous-séquence commune à X_i et Y_j pour $(i, j) \in [\![0; n]\!] \times [\![0; m]\!]$. Notons $L_{i,j}$ cette longueur. Le résultat précédent montre que :

$$L_{i,j} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ L_{i-1,j-1} + 1 & \text{si } i, j > 0 \text{ et } x_i = y_j \\ \max(L_{i,j-1}, L_{i-1,j}) & \text{sinon} \end{cases} \qquad (\star)$$

L'algorithme suivant utilise cette relation de récurrence pour calculer $L_{n,m}$ en utilisant une sous-fonction récursive et le principe de mémoïzation (qui permet ici d'éviter de traiter de nombreux sous-problèmes inutiles). On utilise une matrice `L` pour stocker les valeurs **utiles** des $(L_{i,j})$, ainsi qu'un tableau `D` de « direction » permettant de retenir lors de l'utilisation de $(\star)$ si la valeur de $L_{i,j}$ est obtenue à partir de $L_{i-1,j-1}$, $L_{i-1,j}$ ou bien $L_{i,j-1}$.

```
def calcule_l_d(x,y):
    n=len(x); m=len(y)
    l = {}; d = {}
    def aux(i,j):
        if l.get((i,j))==None:
            if i==0 or j==0:
                l[(i,j)]=0
            else:
                if x[i-1]==y[j-1]:
                    aux(i-1,j-1)
                    l[(i,j)]=l[(i-1,j-1)]+1
                    d[(i,j)]="d"
                else:
                    aux(i-1,j); aux(i,j-1)
                    if l[(i-1,j)] < l[(i,j-1)]:
                        l[(i,j)] = l[(i,j-1)]
                        d[(i,j)]="g"
                    else:
                        l[(i,j)] = l[(i-1,j)]
                        d[(i,j)]="h"
    aux(n,m)
    return(l,d)
```

A titre d'exemple, les tableaux `L` et `D` pour les séquences $0,1,1,0,1,1,0,1,1$ et $Y = 1,1,1,0,0,0$ sont donnés par les tableaux suivants (les valeurs initiales non modifiées ne sont pas affichées pour plus de clarté).

		j	1	2	3	4	5	6
		y_j	1	1	1	0	0	0
i	x_i		0	0	0			
1	0	0	0	0		1		
2	1	0	1	1	1	1		
3	1		1	2	2	2		
4	0	0	1	2		3	3	
5	1	0	1	2	3	3	3	
6	1		1	2	3	3	3	
7	0	0	1	2		4	4	4
8	1			2	3	4	4	4
9	1				3	4	4	4

		j	1	2	3	4	5	6
		y_j	1	1	1	0	0	0
i	x_i							
1	0		h	h		d		
2	1		d	d	d	h		
3	1		d	d	d	g		
4	0		h	h		d	d	
5	1		d	d	d	h	h	
6	1		d	d	d	h	h	
7	0		h	h		d	d	d
8	1			d	d	h	h	h
9	1				d	h	h	h

Notez que certaines cases n'ont pas été modifées, car le traitement des sous-problèmes associés n'était pas nécessaire. Pour retrouver une sous-séquence de longueur optimale, il suffit de partir de la dernière case en bas à droite et de suivre les directions indiquées par les cases : haut pour h, diagonale pour d et enfin gauche pour d. A chaque fois que l'on se déplace en diagonale depuis une position (i,j), la lettre commune $x_i = y_j$ fait partie de la sous-séquence cherchée.

		j	1	2	3	4	5	6
		y_j	1	1	1	0	0	0
i	x_i		–	–	–	–	–	–
1	0	–	h	h	–	d	–	–
2	1	–	d	d	d	h	–	–
3	1	–	d	d	d	g	–	–
4	0	–	h	h	–	d	d	–
5	1	–	d	d	d	h	h	–
6	1	–	d	d	d	h	h	–
7	0	–	h	h	–	d	d	d
8	1	–	–	d	d	h	h	h
9	1	–	–	–	d	h	h	h

La fonction suivante permet d'obtenir une chaîne de caractère réprésentant la plus longue sous-séquence commune de deux chaînes `x` et `y`.

```
def extrait_plsc(x,y):
    l,d = calcule_l_d(x,y)
    def aux(i,j):
        print((i,j))
        if i==0 or j==0:
            return("")
        elif d[(i,j)]=="d":
            return (aux(i-1,j-1)+x[i-1])
        elif d[(i,j)]=="g":
            return aux(i,j-1)
        else:
            return aux(i-1,j)
    return (aux (len(x),len(y)))
```

4 Plus courts chemins pour tous couples de sommets (Floyd-Warshall)

Dans cette section, l'objectif est de déterminer pour tous les couples de sommets du graphe un plus court chemin entre ces deux éléments et son poids. Le résultat peut être naturellement rendu sous la forme d'une matrice de taille $|S| \times |S|$, ce qui incite fortement à utiliser les matrices d'adjacences. On rappelle que dans le cas de graphes pondérés, cette matrice contient en position (i, j) le poids de l'arc (i, j) s'il existe et une valeur modélisant ∞ sinon. Dans toute cette partie, on considère des graphes sans boucle.

On pourrait envisager d'appliquer $|S|$ fois l'algorithme de Dijkstra en partant de chaque sommet du graphe. On obtiendrait alors une solution de complexité en $O(|S|^3)$ mais sous réserve que tous les poids des arcs soient positifs. On présente dans cette partie deux approches par programmation dynamique. Bien que ce point ne soit pas abordé dans ce paragraphe, ces algorithmes fonctionnent également lorsque certains poids sont négatifs (contrairement à Dijkstra).

4.1 Analyse du problème

Il existe essentiellement deux approches du problème menant à une implémentation par programmation dynamiques. Pour simplifier (et rester dans le cadre du programme), on suppose que les poids des arcs sont positifs. La première donne une méthode de complexité $O(|S|^4)$, qui peut s'améliore rapidement en $O(|S|^3 \ln |S|)$ grâce à l'exponentiation rapide. La seconde est de complexité $O(|S|^3)$ et porte le nom d'algorithme de Floyd-Warshall. C'est ce dernier qui est explicitement suggéré comme exemple d'application dans le programme officiel.

Plus courts chemins de longueur bornée

Soit m un entier quelconque et i, j deux sommets quelconques. On note $E_{i,j}^m$ l'ensemble des chemins de i vers j longueur au plus m et $d_{i,j}^{(m)}$ le poids minimal des éléments de $E_{i,j}^{(m)}$. Si $E_{i,j}^{(m)}$ est vide, on pose par convention cette quantité égale à ∞. Par suite, on note D_m la matrice dont le coefficient d'indice (i, j) est égal à $d_{i,j}^{(m)}$:

- pour $m = 0$, il existe un chemin de i vers j de longueur nulle si et seulement si $i = j$, et le poids de ce chemin est nul car il ne comporte aucun arc. La matrice D_0 est donc de diagonale nulle, et tous ses autres coefficients valent ∞.
- pour $m = |S| - 1$, il existe au moins un chemin de poids minimal contenant au plus $|S| - 1$ arcs. La matrice cherchée est donc la matrice $D_{|S|-1}$.

Considérons maintenant un entier m quelconque supérieur ou égal à 1. Considérons deux sommets i et j du graphe. Un plus court chemin de longueur au plus m de i vers j est nécessairement soit un plus court chemin de longueur au plus $m-1$ vers j, soit composé d'un plus court chemin de longueur au plus $m-1$ vers un certain sommet $k \neq j$ du graphe, puis de l'arc (k, j). On en déduit que

$$d_{i,j}^{(m)} = \min_{1 \leq k \leq |S|} \left(d_{i,k}^{(m-1)} + d_{k,j} \right) \quad (1)$$

Plus courts chemins à sommets intermédiaires restreints

Etant donné un chemin $(s_0, s_1, \ldots, s_m)$ entre deux sommets s_0 et s_m, on appelle **sommets intermédiaires** du chemin les sommets $s_1, s_2, \ldots, s_{m-1}$. L'idée de l'algorithme de Floyd-Warshall consiste à calculer de proche en proche les poids des plus

courts chemins utilisant de plus en plus de sommets intermédiaires. On rappelle que les sommets du graphes sont les entiers $\{0, 1, \dots, |S|-1\}$. Soit k un entier appartenant à $[\![0; |S|-1]\!]$. On note $X_{i,j}^{(k)}$ l'ensemble des chemins de i vers j n'utilisant que des sommets intermédiaires d'indices compris entre 0 et $k-1$. Soit $m_{i,j}^{(k)}$ le poids minimal de ces chemins. On note comme précédemment M_k la matrice de terme général $m_{i,j}^{(k)}$.

- pour $k = 0$, on cherche le poids des chemins entre deux sommets i et j sans sommet intermédiaire. Si i et j sont différents, un tel chemin ne peut être constitué que d'un seul arc, et son poids est égal à la pondération de l'arc. Si i et j sont égaux, un tel chemin ne peut être que réduit au sommet i. Ainsi, M_0 est égale à la matrice d'adjacence du graphe à l'extérieur de sa diagonale, et nulle sur sa diagonale.
- pour $k = |S|$, tous les sommets du graphe sont autorisés pour sommet intermédiaire. La matrice $M_{|S|}$ est donc cette fois la matrice recherchée.

Soit maintenant k un entier compris entre 0 et $|S|-1$. et considérons un élément de $X_{i,j}^{(k+1)}$. Deux cas sont possibles :

- soit ce chemin n'utilise pas k comme sommet intermédiaire, auquel cas il n'utilise que $\{0, \dots, k-1\}$ et est donc nécessairement de poids $m_{i,j}^{(k)}$;
- soit il passe une et une seule fois par k, car il ne comporte pas de cycle. Dans ce cas, il est nécessairement constitué d'un chemin de poids minimal de i vers k et d'un chemin de poids minimal de k vers j, tous deux n'utilisant que des sommets intermédiaires parmi $\{0, \dots, k-1\}$.

On en déduit cette fois la formule de récurrence

$$m_{i,j}^{(k+1)} = \min\left(m_{i,j}^{(k)}, m_{i,k}^{(k)} + m_{k,j}^{(k)}\right) \tag{2}$$

Commentaires

La formule (**1**) est plus contraignante que la formule (**2**). En effet, la mise à jour d'un coefficient nécessite $|S|$ calculs, là où la formule (**2**) permet une mise à jour en temps constant. Calculer tous les coefficients de la matrice D_m à partir de D_{m-1} nécessite donc $|S|^3$ calculs, là où il n'en faut que $|S|^2$ pour calculer M_k à partir de M_{k-1}. L'objectif étant de calculer la matrice de rang $|S|-1$ dans chaque cas, on retrouve bien les complexités en $O(|S|^4)$ et $O(|S|^3)$.

Remarque 2

Pour améliorer la première complexité à $O(|S|^3 \ln|S|)$, il faut remarquer que la formule (**1**) est exactement la formule d'un produit matriciel où `min` joue le rôle d'une addition et `+` le rôle d'une multiplication. (l'ensemble $\mathbb{Z}$ muni de ces deux opérations forme un pseudo-anneau). La matrice D_m est ainsi égale à $(D_1)^m$, la puissance étant prise au sens de l'anneau $(\mathbb{Z}, \min, +)$, que l'on peut donc calculer en $O(\log m)$ multiplications matricielles par exponentiation rapide.

Dans les deux cas, on peut assez facilement obtenir les plus courts chemins à l'aide des techniques classiques de la programmation dynamique, à savoir le stockage supplémentaire d'information. Plus précisément, on met à jour au fur et à mesure des calculs une matrice P (matrice des pères) suivant le principe suivant : à la fin de la r-ième boucle, le coefficient $P_{i,j}$ contient la valeur du prédécesseur du sommet j dans un élément de $E_{i,j}^{(r)}$ (resp. $X_{i,j}^{(r)}$) s'il existe et une valeur symbolisant le vide sinon (l'entier (-1) fait très bien l'affaire). Ces valeurs sont mises à jour en fonction du minimum trouvé dans l'une ou l'autre des formules.

4.2 Algorithme de Floyd-Warshall

L'implémentation de l'algorithme de Floyd-Warshall utilise une fonction de copie de matrices. Cela permet de n'utiliser que 4 matrices pendant l'exécution et notamment de ne pas stocker en mémoire toute la suite des matrices.

```
1 def copie_dans(a,b):
2     n=len(a)
3     for i in range(n):
4         for j in range(n):
5             b[i][j]=a[i][j]
```

On utilisera dans cette implémentation la chaîne de caractère `"inf"` pour modéliser l'infini. D'autres choix sont possibles, par exemple le booléen `False`. On implémente donc des fonctions d'addition et de comparaison spécifiques pour prendre en compte ce cas de figure.

```python
def add(x,y):
    if x=="inf" or y=="inf":
        return "inf"
    else:
        return (x+y)

def strict_inf(x,y):
    return ((x!="inf") and ((y=="inf") or (x<y)))
```

L'initialisation de la matrice `m` se fait en copiant `g` à l'intérieur à l'exception de la diagonale qui est nulle. Le calcul des matrices $M_0, \ldots, M_{|S|}$ se fait selon la formule (**2**). En ce qui concerne la matrice des pères, on utilise les constatations suivantes :

- pour $k = 0$, les chemins qui mènent de i à j (distincts) sans sommet intermédiaire sont composés uniquement de l'arc (i, j), s'il existe. On construit donc une matrice dont le coefficient vaut i si $i \neq j$ et si l'arc (i, j) existe et (-1) sinon.
- Considérons maintenant k quelconque.
  - Si $m_{i,j}^{(k+1)} = m_{i,j}^{(k)}$, les ensembles $X_{i,j}^{(k)}$ et $X_{i,j}^{(k+1)}$ ont un chemin de poids minimal commun. On ne modifie donc pas le prédécesseur courant de j.
  - Sinon, l'ensemble $X_{i,j}^{(k+1)}$ contient un chemin de poids strictement inférieur à ceux de $X_{i,j}^{(k)}$ qui passe par k. On a vu qu'alors, le sous-chemin de k vers j est dans $X_{k,j}^{(k)}$. On modifie donc le prédécesseur de j en lui donnant son précédésseur dans un élément de $X_{k,j}^{(k)}$.

```python
def floyd_warshall(g):
    n=len(g)
    m=[["inf"]*n for i in range(n)]
    t=[["inf"]*n for i in range(n)]
    a=[[(-1)]*n for i in range(n)]
    b=[[(-1)]*n for i in range(n)]
    # initialisation des matrices
    copie_dans(g,m)
    for i in range(n):
        m[i][i]=0
        for j in range(n):
            if i!=j and g[i][j]!="inf":
                a[i][j]=i
    # m est maintenant egale a M_0
    for k in range(n):
        # calcul de M_(k+1) (on remplit la matrice t)
        for i in range(n):
            for j in range(n):
                r = add(m[i][k],m[k][j])
                if strict_inf(r,m[i][j]):
                    t[i][j]=r; b[i][j]=a[k][j]
                else:
                    t[i][j]=m[i][j]; b[i][j]=a[i][j]
        # on remplace M_k par M_(k+1) (idem pour les peres)
        copie_dans(t,m); copie_dans(b,a)
    return(m,a)
```

Une fois que la matrice des pères est calculée, on peut implémenter une simple fonction récursive pour retrouver un plus court chemin entre deux sommets quelconques. La méthode est similaire à celle utilisée pour Dijkstra ou les graphes non pondérés après un parcours en largeur.

```python
def pcc(i,j,p):
    if i==j:
        return [i]
    else:
        return pcc(i,p[i][j],p)+[j]
```

5 Compléments : quelques exemples supplémentaires

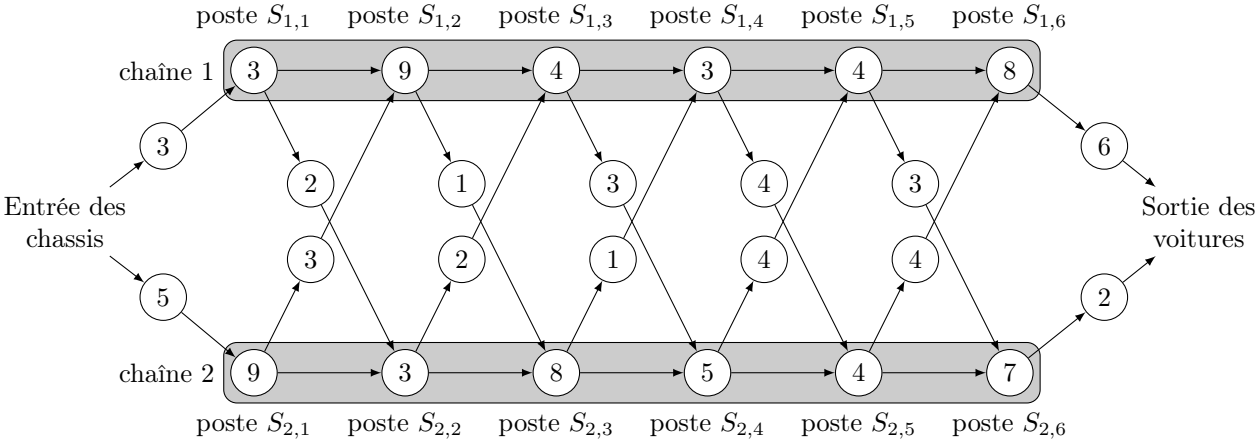
Les deux exemples qui suivent sont les premiers exemples de l'un des livres de référence en informatique : *introduction à l'algorithmique* des auteurs Cormen, Leiserson, Rivest et Stein, plus vulgairement surnommé « le Cormen ». Je les ai utilisés pendant plusieurs années avant de les remplacer par d'autres, plus simples et pédagogiques. Je les laisse en lecture pour la postérité ... En revanche, j'ai la flemme de re-rédiger les codes, donc je les laisse en version Ocaml.

5.1 La chaîne de montage

Problématique

Une entreprise produit des automobiles dans un atelier comportant deux chaînes de montage, chacune contenant n stations de travail, par lesquelles transitent les véhicules en construction. Les stations ont été installés à des périodes différentes et les technologies différentes entraînent des temps de passage variables à travers les postes, y compris ceux réalisant le même travail sur les deux chaînes.

Normalement, un châssis circule sur une seule chaîne de montage. Mais il peut arriver que l'on ait besoin de construire un véhicule en urgence et d'accélérer le délai de fabrication. On a alors la possibilité de faire transiter un véhicule partiellement construit du poste j au poste $j+1$ de l'autre chaîne de montage. Toutefois, cette opération nécessite un temps de tranferts, là où le passage d'un poste au suivant sur une même chaîne est négligeable. La figure suivant illustre un exemple de chaîne de longueur 6, où les entiers représentent les différents temps de transition, y compris ceux d'accès de sortie de la chaîne.



Le problème consiste à trouver le chemin à travers toute l'usine pour réaliser la construction d'une voiture en un minimum de temps. Sachant qu'il y a deux choix pour chaque poste, une méthode naïve testant tous les chemins possibles prendrait au minimum $O(n \cdot 2^n)$ calculs. On cherche donc une méthode plus efficace.

Sous-structures optimales et relations de récurrence

Pour obtenir une méthode efficace pour résoudre le problème, l'idée consiste dans un premier temps non pas à chercher le chemin optimal à travers tout l'atelier, mais plus généralement jusqu'à la sortie de n'importe quel poste de l'atelier. Dans toute la suite, on note pour tout $(i,j) \in \{1,2\} \times [\![1;n]\!]$

- $m_{i,j}$ le coût minimal pour aboutir à la sortie du poste $S_{i,j}$.
- $C_{i,j}$ l'ensemble des chemins de coût minimal pour aboutir à la sortie du poste $S_{i,j}$, un tel chemin étant représenté par la suite $(S_{i_1,1}, S_{i_2,2}, \ldots, S_{i_j,j})$ des postes traversés, ou plus simplement par la liste $(i_1, \ldots, i_j) \in \{1,2\}^j$ de leurs « abscisses ».

Les temps de parcourt sur la chaîne 1 (resp. 2) sont notés $t_{1,j}$ (resp. $t_{2,j}$) avec $j \in [\![0; n+1]\!]$. La valeur $t_{1,0}$ est le temps d'accès à la chaîne 1, $t_{1,n+1}$ est celui de sortie, et pour $j \in [\![1;n]\!]$, $t_{i,j}$ est le temps de traversée du poste $S_{i,j}$. Pour terminer, le temps de transfert du poste $S_{1,j}$ au poste $S_{2,j+1}$ (resp. $S_{2,j}$ à $S_{1,j+1}$) est noté $e_{1,j}$ (resp. $e_{2,j}$).

Notons qu'il n'existe qu'un seul chemin jusqu'à la sortie de $S_{1,1}$ et de $S_{2,1}$. Par définition, on a donc

$$m_{1,1} = t_{1,0} + t_{1,1} \qquad C_{1,1} = \{\{1\}\} \qquad m_{2,1} = t_{2,0} + t_{2,1} \qquad C_{2,1} = \{\{2\}\}$$

Justifions maintenant les deux égalités suivantes :

Proposition 2

Pour tout $j \in [\![2; n]\!]$,
$$m_{1,j} = \min\left(t_{1,j} + m_{1,j-1}, t_{1,j} + m_{2,j-1} + e_{2,j-1}\right)$$
$$m_{2,j} = t_{2,j} + \min\left(m_{2,j-1}, m_{1,j-1} + e_{1,j-1}\right)$$

Remarque 3

L'idée derrière cette relation de récurrence est assez simple. Considérons un chemin optimal $C = (i_1, \ldots, i_j)$ jusqu'à la sortie du poste $S_{1,j}$. Nécessairement, ce chemin passe par $S_{1,j-1}$ ou $S_{2,j-1}$. Le début du chemin $C' = (i_1, \ldots, i_{j-1})$ depuis l'entrée jusqu'à $S_{1,j-1}$ **ou** $S_{2,j-1}$ est nécessairement optimal (sinon, on pourrait le remplacer par un chemin plus rapide, et construire un chemin jusqu'à $S_{1,j}$ plus rapide que celui qu'on a choisit au départ, ce qui contredirait sa minimalité. Puisque C' est optimal, son temps de parcours est alors $m_{1,j}$ ou $m_{2,j}$, ce qui prouve que le temps de parcours de C est $m_{1,j-1} + t_{i,j}$ **ou** $m_{2,j-1} + e_{2,j-1} + t_{i,j}$ (on rajoute le cout du transfert dans ce second cas). La subtilité pour une preuve irréprochable, c'est la gestion de ce « **ou** » qu'on ne maîtrise pas pour aboutir rigoureusement à la formule avec le « min ». La preuve rigoureuse passe par une double inégalite.

Preuve : Soit $j \in [\![2; n]\!]$. On justifie uniquement la première égalité, la seconde se justifiant de manière totalement symétrique.

$\boxed{\geq}$ Considérons un chemin optimal $C = (i_1, \ldots, i_j)$ jusqu'à la sortie de $S_{1,j}$ (on a donc $i_j = 1$). On distingue deux cas suivant la valeur de i_{j-1}.

- Si $i_{j-1} = 1$, le chemin passe par le poste $S_{1,j-1}$. Le sous-chemin $C' = (i_1, \ldots, i_{j-1})$ est nécessairement optimal, comme expliqué dans la remarque ci-dessus, donc de temps de parcours $m_{1,j-1}$. Le temps de parcours de C est donc égal dans ce cas à $m_{1,j-1} + t_{1,j}$.
- Si $i_{j-1} = 2$, le chemin passe par le poste $S_{2,j-1}$. Le sous-chemin $C' = (i_1, \ldots, i_{j-1})$ est là encore nécessairement optimal donc de temps de parcours $m_{2,j-1}$. Le temps de parcours de C est donc égal dans ce cas à $m_{2,j-1} + e_{2,j-1} + t_{1,j}$ (il faut rajouter le coût du transfert)

Dans les **deux** cas, on a bien en particulier
$$m_{1,j} \geq t_{1,j} + \min\left(m_{1,j-1}, m_{2,j-1} + e_{2,j-1}\right)$$

$\boxed{\leq}$ Pour l'inégalité réciproque, notons $(i_1, \ldots, i_{j-1})$ un chemin optimal jusqu'à $S_{1,j-1}$ (donc de temps de parcours $m_{1,j-1}$) et $(k_1, \ldots, k_{j-1})$ un second chemin optimal, cette fois jusqu'à $S_{2,j-1}$ (de temps $m_{2,j-1}$). En rajoutant 1 a la fin de ces listes, on obtient deux chemins $(i_1, \ldots, i_{j-1}, 1)$ et $(k_1, \ldots, k_{j-1}, 1)$ jusqu'à $S_{1,j}$, de temps de parcours respectif $m_{1,j-1} + t_{1,j}$ et $m_{2,j-1} + e_{2,j-1} + t_{1,j}$. Or, par définition, $m_{1,j}$ est le plus petit des temps de parcours parmi tous les chemins menant à $S_{1,j}$. En particulier, il est inférieur à ces deux temps de parcours, donc à leur minimum. Ainsi,
$$m_{1,j} \leq t_{1,j} + \min\left(m_{1,j-1}, m_{2,j-1} + e_{2,j-1}\right)$$

Finalement
$$m_{1,j} = t_{1,j} + \min\left(m_{1,j-1}, m_{2,j-1} + e_{2,j-1}\right) \quad \cdots \square$$

La formule de récurrence précédente permet maintenant de calculer de proche en proche les valeurs $(m_{1,j}, m_{2,j})_{j \in [\![1;n]\!]}$, les valeurs $m_{1,1}$ et $m_{2,1}$ étant connues. Elle ne donne toutefois pas immédiatement un moyen de reconstituer un chemin optimal. Pour cela, il suffit de remarquer que, compte tenu de la proposition précédente et de sa preuve, on en déduit que les éléments de $C_{1,j}$ s'obtiennent de la manière suivante :

- Si $m_{1,j-1} < m_{2,j-1} + e_{2,j-1}$, on concatène un 1 à la fin des éléments de $C_{1,j-1}$.
- Si $m_{1,j-1} > m_{2,j-1} + e_{2,j-1}$, on concatène un 2 à la fin des éléments de $C_{1,j-1}$.
- En cas d'égalité, on réunit l'ensemble des suites finies obtenus des deux façons précédentes.

Pour finir, le temps optimal de traversée de l'atelier est donné par
$$m = \min\left(m_{1,n} + t_{1,n+1}, m_{2,n} + t_{2,n+1}\right)$$

Calcul du temps et du chemin optimal (méthode impérative)

La fonction ci-dessous utilise quatre tableaux en arguments donnant les quantités $(t_{i,j})_{(i,j)\in\{1,2\}\times[\![0;n+1]\!]}$ et les quantités $(e_{i,j})_{(i,j)\in\{1,2\}\times[\![1;n-1]\!]}$. Attention, un décalage d'indice est utilisé ici pour ces dernières quantités qui ne sont définies que pour $i \geq 1$ (on aurait aussi pu utiliser un tableau avec une case d'indice 0 non utilisée contenant une valeur arbitraire pour l'éviter). On utilise la formule de récurrence pour calculer de proche en proche les coûts minimaux $(m_{i,j})_{(i,j)\in\{1,2\}\times[\![1;n]\!]}$ ainsi qu'un élément de $C_{i,j}$ pour tous entiers (i,j). Les résultats sont stockés au fur et à mesure des calculs dans quatre tableaux. On utilise les informations à la sortie du n-ième poste pour donner la solution au problème.

```
let plus_court_chemin t1 t2 e1 e2 =
  let n = Array.length t1 - 2 in
  let m1 = Array.make (n+1) 0 in
  let m2 = Array.make (n+1) 0 in
  let c1 = Array.make n [1] in
  let c2 = Array.make n [2] in
  m1.(1) <- t1.(0)+t1.(1); m2.(1) <- t2.(0)+t2.(1);
  for i = 1 to n-1 do
    let a = m1.(i)+t1.(i+1) and b = m2.(i)+e2.(i-1)+t1.(i+1) in
    if a < b then
      (m1.(i+1) <- a; c1.(i) <- 1::c1.(i-1))
    else
      (m1.(i+1) <- b; c1.(i) <- 1::c2.(i-1));
    let c = m2.(i)+t2.(i+1) and d = m1.(i)+e1.(i-1)+t2.(i+1) in
    if c < d then
      (m2.(i+1) <- c; c2.(i) <- 2::c2.(i-1))
    else
      (m2.(i+1) <- d; c2.(i) <- 2::c1.(i-1));
  done;
  let e = m1.(n) + t1.(n+1) and f = m2.(n) + t2.(n+1) in
  if e < f then (e,List.rev c1.(n-1))
  else (f,List.rev c2.(n-1));;
val plus_court_chemin: int array -> int array -> int array -> int array -> int * int list = <fun>
```

Remarque : Le programme ci-dessus est plutôt gourmand en espace puisqu'on stocke des informations redondantes dans les tableaux `c1` et `c2` (les chemins dans les cases d'indices `i` sont des sous-chemins des cases d'indices `i+1`). Une solution consiste à stocker dans la case d'indice `i` des tableaux uniquement le $(i-1)$-ième élément d'un chemin optimum qui mène à la sortie du i-ième poste. Voici ce que donne le calcul sur l'exemple initial

m1	0	6	15	17	20	24	32
m2	0	14	11	19	24	28	34

c1	-	-	1	2	1	1	1
c2	-	-	1	2	2	1	1

Pour retrouver le chemin optimal, on procède de la manière suivante :

- Le dernier poste du chemin optimal s'obtient à nouveau en comparant `m2.(n)+t2.(n+1)` et `m1.(n)+t1.(n+1)`. Ici, il s'agit du poste de la chaîne 2 car $34+2 < 32+8$.
- On consulte alors la case `c2.(6)` pour savoir l'état dont provient un chemin optimal jusqu'au poste $S_{2,6}$. Ici, la valeur 1 indique qu'il s'agit du poste $S_{1,5}$.
- De la même manière, la valeur 1 de la case `c1.(5)` indique qu'un chemin optimal vers la sortie de l'état $S_{1,5}$ provient de l'état $S_{1,4}$.
- De proche en proche, on reconstruit donc le chemin `[1; 2; 1; 1; 1; 2]` depuis le dernier jusqu'au premier état.

Calcul du temps et du chemin optimal (méthode récursive)

La fonction suivante utilise une sous-fonction récursive qui prend en argument i et calcule un quadruplet `(m1, l1, m2, l2)` donnant les deux temps minimaux et deux chemins optimums pour arriver en sortie du i-ième poste. Elle implémente directement la relation de récurrence ci-dessus en distinguant 4 cas.

```
let plus_court_chemin t1 t2 e1 e2 =
  let rec aux = function
  |1 -> (t1.(0)+t1.(1), [1], t2.(0)+t2.(1), [2])
  |i -> let (m1, l1, m2, l2) = aux (i-1) in
        let a = m2+e2.(i-2) and b = m1+e1.(i-2) in
        if (m1 < a) && (m2 < b) then
          (m1+t1.(i), 1::l1, m2+t2.(i), 2::l2)
        else if (m1<a) then
          (m1+t1.(i), 1::l1, b+t2.(i), 2::l1)
        else if (m2 < b) then
          (a+t1.(i), 1::l2, m2+t2.(i), 2::l2)
        else
          (a+t1.(i), 1::l2, b+t2.(i), 2::l1)
  in
  let n = Array.length t1 - 2 in let (x,y,z,t) = aux n in
  let e = x+t1.(n+1) and f = z+t2.(n+1) in
    if e < f then  (e, List.rev y)
    else (f, List.rev t) ;;
val plus_court_chemin: int array -> int array -> int array -> int array -> int * int list = <fun>
```

Remarque : On aurait pu envisager de faire deux fonctions mutuellement récursives pour calculer `m1` et `m2`. Si l'idée peut paraître jolie, elle est de complexité très mauvaise puisque chaque fonction déclencherait systématiquement 2 appels récursifs, pour une complexité finale en $O(2^n)$!

5.2 Multiplication matricielle

Problématique

Soient p, q et r trois entiers. Le produit de deux éléments $A \in \mathcal{M}_{p,q}(\mathbb{K})$ et $B \in \mathcal{M}_{q,r}(K)$ se fait par la formule

$$\forall i,j \in [\![1;p]\!] \times [\![1;r]\!], \qquad (AB)_{i,j} = \sum_{k=1}^{q} A_{i,k} \cdot B_{k,j}$$

Il nécessite donc $O(pqr)$ multiplications (on néglige le temps de calcul pris par les additions).

Considérons maintenant un produit compatible de n matrices $A_0, A_2, \ldots, A_{n-1}$, dont les dimensions sont notées (p_0, p_1) pour A_1, (p_1, p_2) pour A_2 et ainsi de suite jusqu'à (p_{n-1}, p_n) pour la matrice A_{n-1}. Le produit matriciel étant associatif, il y a plusieurs façons de calculer $A_0 \cdot A_{n-1}$, et le temps de calcul dépend de l'ordre des multiplications. Par exemple, si A_0 et A_2 sont des vecteurs lignes et A_1, A_3 des vecteurs colonnes, le calcul de $A_0 \cdot A_1 \cdot A_2 \cdot A_3$ prend

- $n + n + 1 = 2n + 1$ multiplications via le parenthésage $(A_0A_1)(A_2A_3)$
- $n^2 + n^2 + n = 2n^2 + n$ multiplications via le parenthésage $(A_0(A_1A_2))A_3$.

On cherche donc dans cette partie à déterminer le parenthésage qui minimise le temps de calcul du produit des n matrices.

Remarque : Le nombre total de parenthésage possible d'un produit de n éléments est donné par le $(n-1)$-ième nombre de Catalan

$$C_{n-1} = \frac{1}{n}\binom{2n-2}{n-1} \underset{n\to+\infty}{\sim} \frac{4^{n-1}}{\sqrt{\pi} \cdot n^{3/2}}$$

Une recherche exhaustive prendrait donc comme dans l'exemple précédent un coût exponentiel et est donc à proscrire.

Analyse du problème

Pour calculer le produit $A_0 \cdots A_{n-1}$ de manière optimale, il faut nécessairement pour un certain entier k compris entre 0 et $n-2$ calculer les produits $A_0 \cdots A_k$ et $A_{k+1} \cdots A_{n-1}$ puis effectuer le calcul $(A_0 \cdots A_k)(A_{k+1} \cdots A_{n-1})$. Notons que les calculs de $A_0 \cdots A_k$ et de $A_{k+1} \cdots A_{n-1}$ doivent bien entendu se faire de manière optimale. Supposons connus pour tout entier $k \in [\![0; n-2]\!]$ le coût minimal $m_{0,k}$ pour calculer $A_0 \cdots A_k$ et celui $m_{k+1,n-1}$ pour calculer $A_{k+1} \cdots A_{n-1}$. Alors, le coût minimal pour calculer $A_0 \cdots A_{n-1}$ en passant par la décomposition $(A_0 \cdots A_k)(A_{k+1} \cdots A_{n-1})$ est donné par

$$m_{0,k} + m_{k+1,n-1} + p_0 p_{k+1} p_n$$

Le coût minimal $m_{0,n-1}$ cherché pour calculer le produit est donc donné par

$$m_{0,n-1} = \min_{k\in[\![0;n-2]\!]} (m_{0,k} + m_{k+1,n-1} + p_0 p_{k+1} p_n)$$

Plus généralement, le coût minimal $m_{i,j}$ pour calculer le produit $A_i \cdots A_j$ vaut 0 si $i = j$, et lorsque $j > i$,

$$m_{i,j} = \min_{k\in[\![i;j-1]\!]} (m_{i,k} + m_{k+1,j} + p_i p_{k+1} p_{j+1}) \qquad (\star)$$

Cette formule de récurrence permet de calculer de proche en proche le coût minimal cherché $m_{0,n-1}$. Pour trouver le parenthésage optimal, il suffit à chaque fois qu'une quantité $m_{i,j}$ est calculée de stocker dans un second dictionnaire la valeur $k_{i,j}$ de l'entier k pour lequel le minimum est atteint.

Implémentation

Commençons par définir une fonction impérative qui calcule dans l'ordre les quantités $m_{0,1}, m_{1,2}, \ldots, m_{n-2,n-1}$, puis $m_{0,2}, m_{1,3}, \ldots, m_{n-3,n-1}$ et ainsi de suite jusqu'à $m_{0,n-1}$. Les données $p_0, \ldots, p_n$ sont données en argument par un tableau `p`. Une sous-fonction permet de trouver, à partir du dictionnaire `m` partiellement construit et de `p` de calculer $m_{i,j}$, défini par $(\star)$ et l'entier $k_{i,j}$ qui réalise le minimum dans cette égalité.

```
def trouve_min(i,j,m,p):
    r = m[(i,i)]+m[(i+1,j)] +p[i]*p[i+1]*p[j+1]
    s=i
    for a in range(i+1,j):
        t = m[(i,a)]+m[(a+1,j)] +p[i]*p[a+1]*p[j+1]
        if t<r:
            r=t; s=a
    return(r,s)
```

L'ordre de calcul des $(m_{i,j})_{0\leq i\leq j\leq n-1}$ se fait dans l'ordre suivant :

- les valeurs $m_{0,0}, \ldots, m_{n-1,n-1}$ sont toutes nulles ;
- on calcule ensuite les valeurs des produits de deux matrices consécutives, à savoir $m_{0,1}$, puis $m_{1,2}, m_{2,3}, \ldots, m_{n-2,n-1}$;
- on enchaîne avec les produits de trois matrices consécutives $m_{0,2}, \ldots, m_{n-3,n-1}$, et ainsi de suite.

On vérifie facilement que cet ordre de calcul est compatible avec $(\star)$ (on aurait aussi pu implémenter en méthode descendente, mais c'est plus lourd).

```
def construit_k_m(p):
    n=len(p)-1
    m={}; k={}
    for i in range(n):
        m[(i,i)]=0
    for d in range(1,n):
        for i in range(0,n-d):
            (x,y)=trouve_min(i,i+d,m,p)
            m[(i,i+d)]=x
            k[(i,i+d)]=y
    return(m,k)
```

Pour obtenir le coût minimal, il suffit de renvoyer la valeur $m_{0,n-1}$ du dictionnaire.

Pour déterminer un parenthésage optimal, on se sert en revanche exclusivement du tableau `k`. Le retour se fait par exemple sous la forme d'une chaîne de caractère. Etant donnés deux entiers `i` et `j`, la valeur `k.(i).(j)` indique à quel endroit couper le produit $A_i \cdots A_j$ pour faire un calcul optimal. Il n'y a plus qu'à créer une sous-fonction récursive qui renvoie une chaîne de caractère en concaténant un parenthésage optimal de la partie gauche du produit et celui de la partie droite. On ne lance pas d'appel récursif lorsque la partie gauche (resp. droite) est réduite à une seule matrice (c'est le cas d'arrêt).

```
def parenthesage_optimal(p):
    (m,k)=construit_k_m(p)
    n = len(p)-1
    def aux(i,j):
        r=k[(i,j)]
        if i==r:
            gauche="A"+str(i)
        else:
            gauche="("+aux(i,r)+")"
```

```
10          if r==j-1:
11              droite="A"+str(j)
12          else:
13              droite="("+aux(r+1,j)+")"
14          return gauche+droite
15      return aux(0,n-1)
```