UPMC
SORBONNE UNIVERSITÉS

Analyse de séquences en bioinformatique

Becirspahic Lucas
Vu Huc

[3I003] - Algorithmique - 22 novembre 2016

Partie théorique : Partie 1

Question 1.1 — Chaque alignement de longueur k est définie par le choix d'une sous-suite de longueur k dans X et d'une sous-suite de longueur k dans Y. Pour chacune des deux séquences, il y a $\binom{d}{k}$ manières différentes de choisir, soit au moins 2^n donc la complexité de l'algorithme naïf est au moins exponentielle.

Il s'agit donc de résoudre ce problème en calculant directement les coûts des solutions optimales. On considère pour la suite deux séquences $X = (x_1, \dots, x_m)$ et $Y = (y_1, \dots, y_n)$ de longueurs respectives m et n. On cherche à caractériser dans un premier temps les situations de correspondance possibles pour chaque $x_m, y_n \in M$.

Question 1.2 — Soit M un alignement de X et Y. Montrons que si $(x_m, y_n) \notin M$, alors x_m ou y_n n'apparait pas dans M.

Supposons que $(x_m, y_n) \notin M$, et (par l'absurde) supposons que x_m et y_n apparaissent dans M. Il existe donc $i \in [\![1, m]\!]$ et $j \in [\![1, n]\!]$ tel que $(x_i, y_n) \in M$ et $(x_m, y_j) \in M$.

Supposons sans perte de généralité que (x_i, y_n) est avant (x_m, y_j) dans M. Puisqu'il n'y a pas de croisement dans M, on a $n \leq j$ et donc $j = n$. De même, si on suppose que (x_m, y_j) est avant (x_i, y_n) alors $m \leq i$ et donc $i = m$. Dans les deux cas, on a donc $(x_m, y_n) \in M$. Contradiction, on a montré que si $(x_m, y_n) \notin M$ alors x_m ou y_n n'apparaît pas dans M.

Question 1.3 — On déduit de la question précédente trois cas de figure :

$$\begin{pmatrix} x_m \\ y_n \end{pmatrix} \begin{pmatrix} x_m \\ — \end{pmatrix} \begin{pmatrix} — \\ y_n \end{pmatrix}$$

— x_m correspond à y_n.
— x_m correspond à un gap, i.e. x_m n'apparaît pas dans M.
— y_n correspond à un gap, i.e. y_n n'apparaît pas dans M.

Précisons qu'il est inutile de considérer le cas où les deux séquences se terminent par un gap : il suffirait de le supprimer pour obtenir un alignement de coût inférieur.

Question 1.4 — D'après la question 1.3,

$$F(m,n) = \begin{cases} F(m-1, n-1) + \delta_{x_m, y_n} & \text{si } x_m \text{ correspond à } y_n \\ F(m-1, n) + \delta_{gap} & \text{si } x_m \text{ correspond à un gap} \\ F(m, n-1) + \delta_{gap} & \text{si } y_n \text{ correspond à un gap} \end{cases}$$

Question 1.5 — On cherche à calculer le coût minimal à chaque itération, on a donc la formule de récurrence pour $i \geq 1$ et $j \geq 1$:

$$F(i,j) = \min(F(i-1, j-1) + \delta_{x_i, y_j},\ F(i-1, j) + \delta_{gap},\ F(i, j-1) + \delta_{gap})$$

Question 1.6 — Lorsque $n = 0$, on compare la séquence X à une chaîne vide, Y ne possède que des gaps à ce stade donc $F(m, 0) = m\delta_{gap}$. De la même façon, lorsque $m = 0$, la séquence X ne possède que des gaps donc $F(0, n) = n\delta_{gap}$. Finalement, on a $F(i, 0) = i\delta_{gap}$ et $F(0, j) = j\delta_{gap}$ pour tout $i \geq 1$ et $j \geq 1$.

Question 1.7 — Construisons l'algorithme permettant de trouver la valeur d'un alignement de coût minimal pour deux séquences X et Y, à partir des réponses aux questions précédentes. Il s'agit d'un algorithme de programmation dynamique, où l'on mémorise les valeurs retournées qui seront réutilisées, et éviteront un appel récursif coûteux.

Pour ce faire, plusieurs structures de données sont envisageables : on peut par exemple utiliser une matrice à deux dimensions, mais il en existe d'autres. On supposera que cette structure, que l'on nommera M, a déjà été initialisée avant le début de l'algorithme.

La version récursive de l'algorithme est la suivante :

Algorithme : cout1(X, Y, i, j)
Entrées : X, Y : séquences ; i, j : entiers (taille des séquences)
M : matrice à deux dimensions
si (i, j) *est dans* M **alors**
  retourner $M(i, j)$
si $i = 0$ **alors**
  $M(i, j) \leftarrow j\delta_{gap}$
  retourner $M(i, j)$
si $j = 0$ **alors**
  $M(i, j) \leftarrow i\delta_{gap}$
  retourner $M(i, j)$
$M(i, j) \leftarrow \min(cout1(X, Y, i-1, j-1) + \delta_{x_i y_j},\ cout1(X, Y, i-1, j) + \delta_{gap},\ cout1(X, Y, i, j-1) + \delta_{gap})$
retourner $M(i, j)$

On utilise dans l'implémentation la version "bottom up" de l'algorithme, présentée ci-dessous :

Algorithme : cout1(X, Y)
Entrées : X, Y : séquences
F : matrice à deux dimensions $|X| \times |Y|$;
pour i *de* 0 *à* m **faire**
  $F(i, 0) \leftarrow i\delta_{gap}$
pour j *de* 1 *à* n **faire**
  $F(0, j) \leftarrow j\delta_{gap}$
pour i *de* 1 *à* m **faire**
  pour j *de* 1 *à* n **faire**
    $F(i, j) \leftarrow \min(cout1(X, Y, i-1, j-1) + \delta_{x_i y_j},\ cout1(X, Y, i-1, j) + \delta_{gap},\ cout1(X, Y, i, j-1) + \delta_{gap})$
retourner $F(m, n)$;

M étant une matrice à deux dimensions, la complexité spatiale est en $\Theta(mn)$. La complexité temporelle de l'algorithme est également en $\Theta(mn)$.

Question 1.8 — Pour construire l'alignement optimal pour les séquences X et Y, on effectue le chemin inverse en disposant des valeurs des coûts $F(i,j)$.

Algorithme : sol1(X, Y, F)
Entrées : X, Y : séquences, F : dictionnaire
M : liste vide
$i \leftarrow |X|$
$j \leftarrow |Y|$
c_1 : chaîne vide
c_2 : chaîne vide
tant que $(i,j) \neq (0,0)$ **faire**
  si $i > 0$ *et* $j > 0$ *et* $F(i,j) = F(i-1,j-1) + \delta_{x_{i-1}y_{j-1}}$ **alors**
    $c_1 \leftarrow x_{i-1} \cup c_1$
    $c_2 \leftarrow y_{j-1} \cup c_2$
    $i \leftarrow i-1$
    $j \leftarrow j-1$
  sinon si $i > 0$ *et* $F(i,j) = F(i-1,j) + \delta_{gap}$ **alors**
    $c_1 \leftarrow x_{i-1} \cup c_1$
    $c_2 \leftarrow$ "$-$" $\cup\, c_2$
    $i \leftarrow i-1$
  sinon
    $c_1 \leftarrow$ "$-$" $\cup\, c_1$
    $c_2 \leftarrow y_{j-1} \cup c_2$
    $j \leftarrow j-1$
retourner M, c_1, c_2

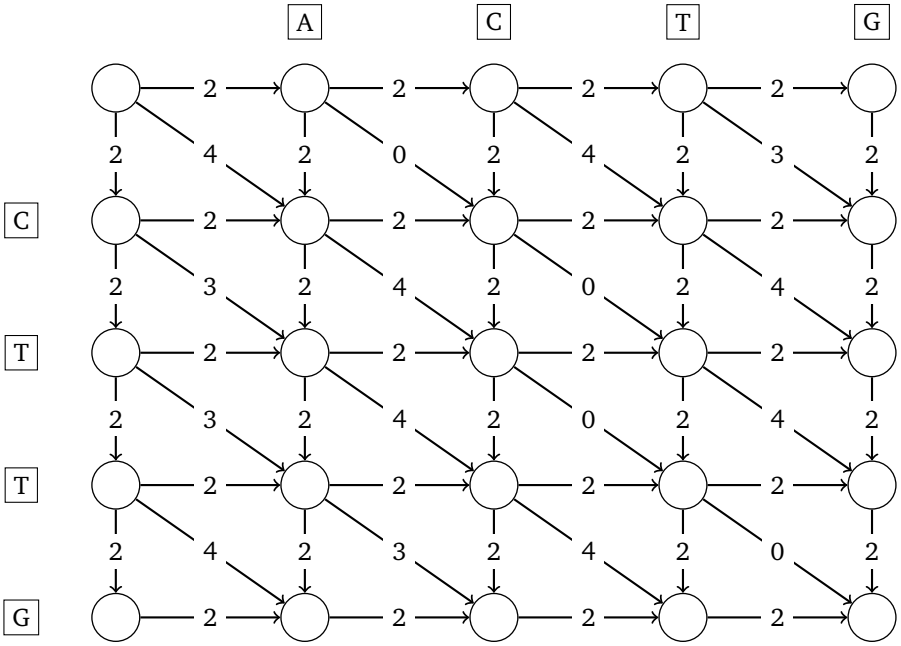
La longueur du chemin entre les cases $(0,0)$ et (m,n) ne dépasse pas $m+n$ donc la complexité temporelle est en $O(m+n)$. C'est également le cas pour la complexité spatiale.

On peut donc conclure, en considérant l'algorithme de calcul des coûts en $\Theta(mn)$ et l'algorithme "inverse" pour obtenir l'alignement en $O(mn)$, que la complexité totale est en $\Theta(mn)$.

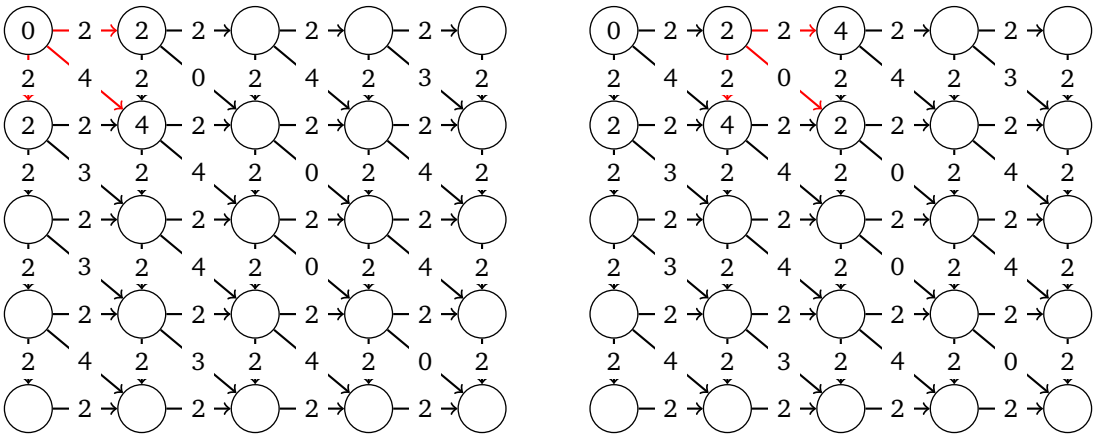
Partie théorique : Partie 2

Question 2.1 — Le graphe donne le coût de chaque arc par les règles suivantes :

$$\begin{cases} \delta_{gap} = 2 & \\ \delta_{x_i y_j} = 0 & \text{si } x_i = y_j \\ \delta_{x_i y_j} = 3 & \text{si } x_i \neq y_j \text{ et } (x_i, y_j) \in \{(A,T),(T,A),(C,G),(G,C)\} \\ \delta_{x_i y_j} = 4 & \text{si } x_i \neq y_j \text{ et } (x_i, y_j) \notin \{(A,T),(T,A),(C,G),(G,C)\} \end{cases}$$

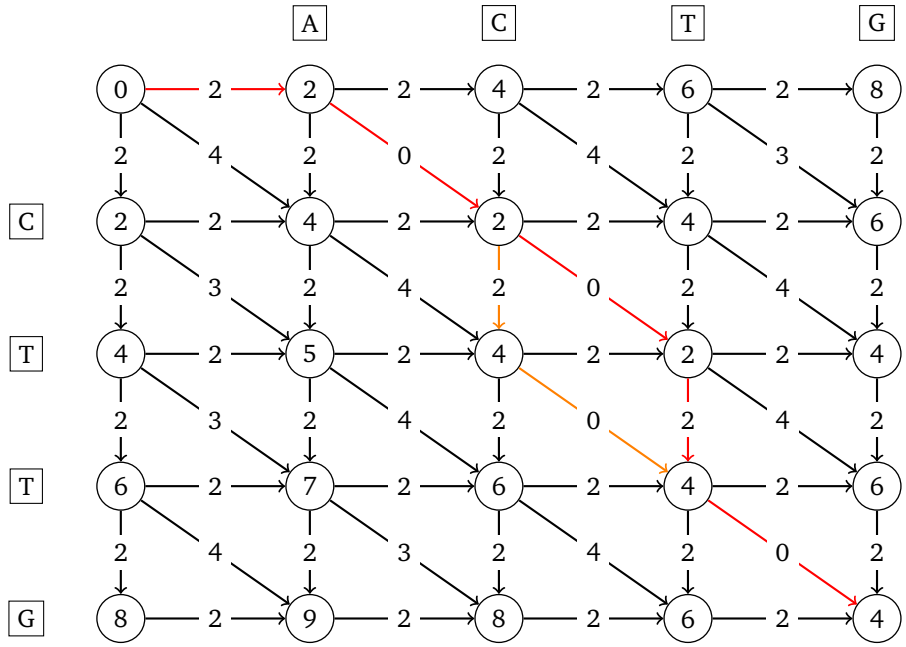


Question 2.3 — Pour déterminer un plus court chemin de (0,0) à (m,n) dans G_{XY}., on peut utiliser soit l'algorithme de Dijkstra, soit l'algorithme de Bellman. Par construction, comme G_{XY} n'a pas de circuit, on utilise l'algorithme de Bellman, qui a une meilleure complexité en $O(s+a)$ avec s le nombre de sommets de G_{XY} et a le nombre d'arêtes de G_{XY}. Comme $s = mn$, on obtient une complexité en $O(mn)$.



On suppose que la liste topologique des sommets a été construite au préalable, par exemple par le biais d'un algorithme de parcours en profondeur. Pour chaque sommet, on regarde ses successeurs et on modifie le coût du chemin vers ce successeur en conséquence s'il est meilleur que l'ancienne valeur. On réitère ainsi pour chaque sommet, et l'algorithme de Bellman nous fournit une arborescence des plus courts chemins partant du premier sommet, en haut à gauche.

En particulier, le chemin coloré en rouge dans le graphe suivant est un plus court chemin possible du sommet $(0,0)$ vers le sommet (m,n) de coût 4 (deux gaps). La variante colorée en orange est une autre possibilité de plus court chemin, également de coût 4.



Les alignements correspondants sont les suivants :

—	C	T	T	G
A	C	T	—	G

—	C	T	T	G
A	C	—	T	G

Question 2.4 — Il faut prendre en compte la complexité de la construction du graphe, qui se fait en $O(mn)$. La construction de la liste topologique pour utiliser l'algorithme de Bellman est également de complexité $O(mn)$. On a montré que les complexités des méthodes de la partie 1 et de la partie 2 sont toutes deux en $O(mn)$ en temps et en mémoire, aucune n'est donc meilleure que l'autre.

Partie théorique : Partie 3

Question 3.1 — Pour des séquences de longueur n, pour calculer $F(n,n)$, on stocke n^2 valeurs au total. On a donc, à la borne maximale, 32 GO pour stocker $n \times n$ éléments, soit $32.10^6 > n^2$. Ainsi, on peut avoir jusqu'à $n = 5656$ caractères dans chaque séquence.

Question 3.2 — Pour améliorer la complexité spatiale de l'algorithme `COUT1`, on utilise deux tableaux pour stocker les valeurs de F, un pour la ligne $i-1$ et un pour la ligne i.

Algorithme : cout2(X, Y)
Entrées : X, Y : séquences
F_1 : tableau de taille $|Y|$
F_2 : tableau de taille $|Y|$
$m \leftarrow |X|$
$n \leftarrow |Y|$
pour j *de* 1 *à* n **faire**
  $F_1[j] \leftarrow j\delta_{gap}$
pour i *de* 1 *à* m **faire**
  $F_2[0] \leftarrow i\delta_{gap}$
  pour j *de* 1 *à* n **faire**
    $F_2[j] \leftarrow \min(F_1[j-1] + \delta_{x_{i-1}y_{j-1}}, F_1[j] + \delta_{gap}, F_2[j-1] + \delta_{gap})$
  $F_1 \leftarrow F_2$
retourner $F2[n]$

La complexité temps de cet algorithme est en $O(mn)$, comme pour la partie 1. Cependant, la complexité spatiale est grandement améliorée, puisqu'on ne stocke que les lignes utiles. Ainsi, on obtient une complexité mémoire en $O(2n)$, soit $O(n)$.

Question 3.3 —

Algorithme : cout2bis(X, Y, k, l)
Entrées : X, Y : séquences, k, l : entiers
F_1 : tableau de taille $|Y|$
F_2 : tableau de taille $|Y|$
$m \leftarrow |X| - k$
$n \leftarrow |Y| - l$
$v_1 \leftarrow 1$
pour j *de* 0 *à* n **faire**
  $F_1[j] \leftarrow j\delta_{gap}$
pour i *de* k *à* m **faire**
  $F_2[0] \leftarrow v_1\delta_{gap}$
  pour j *de* 1 *à* n **faire**
    $F_2[j] \leftarrow \min(F_1[j-1] + \delta_{x_{i-1}y_{j-1}}, F_1[j] + \delta_{gap}, F_2[j-1] + \delta_{gap})$
  $F_1 \leftarrow F_2$
  $v_1 \leftarrow v_1 + 1$
retourner $F2[n]$

Question 3.4 — Soit $g(i,j)$ le plus court chemin de $(0,0)$ à (i,j) et $h(i,j)$ le plus court chemin de (i,j) à (m,n). Soit u un plus court chemin allant de $(0,0)$ à (m,n) en passant par (i,j). Par définition, on a $d(u) \leq g(i,j) + h(i,j)$.
Soit w la portion de u allant de $(0,0)$ à (i,j). Comme $g(i,j)$ est un plus court chemin, on a $d(w) \geq g(i,j)$.
Soit y la portion de u allant de (i,j) à (m,n). Comme $h(i,j)$ est un plus court chemin, on a $d(y) \geq h(i,j)$.

Alors on a $d(w) + d(y) \geq g(i,j) + h(i,j)$, d'où $d(u) \geq g(i,j) + h(i,j)$ et $d(u) \leq g(i,j) + h(i,j)$. Finalement, on obtient $d(u) = g(i,j) + h(i,j)$, ce que l'on voulait montrer.

Implémentation et analyse de complexité expérimentale

Le langage Python est utilisé pour l'implémentation des algorithmes de la partie théorique.

Question 4.1 — Fonction de lecture de fichiers contenant les séquences.

```
def lire_sequence(fichier):
    #Prend un fichier de sequence et retournes les informations dans une liste
    os.chdir('test')
    f = open(fichier,'r')
    sequence = f.read()
    L = sequence.splitlines()
    L2 = []
    L2.append(int(L[0])-1)
    L2.append(int(L[1])-1)
    L2.append(L[2].replace("␣","",""))
    L2.append(L[3].replace("␣","",""))
    if (L2[0]) < 100 and L2[1] < 100:
        print(L2[2])
        print(L2[3])
    return L2
    f.close()
```

Question 4.2 — Fonction de calcul de la valeur d'un alignement de coût optimal COUT1.

```
def cout1(x, y, gap=1,dif=1):
    m, n = len(x), len(y)
    F = np.zeros((m+1, n+1), dtype=int)
    for i in range(1, m+1):
        F[i, 0] = i * gap
    for j in range(1, n+1):
        F[0, j] = j * gap
    for i in range(1, m+1):
        for j in range(1, n+1):
            F[i, j] = min(F[i-1, j-1] + delta(x[i-1], y[j-1],dif),
                          F[i-1, j] + gap,
                          F[i, j-1] + gap)
    #print("F final :",F[len(x),len(y)])
    return F[m,n]
```

A noter qu'on récupère la valeur du coût optimal par F[len(x), len(y)].

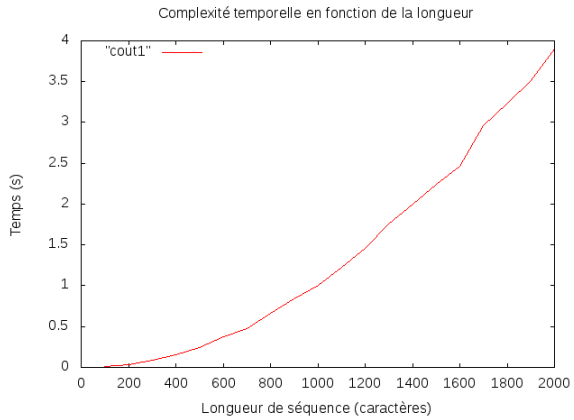
Fonction de construction de l'alignement optimal SOL1.

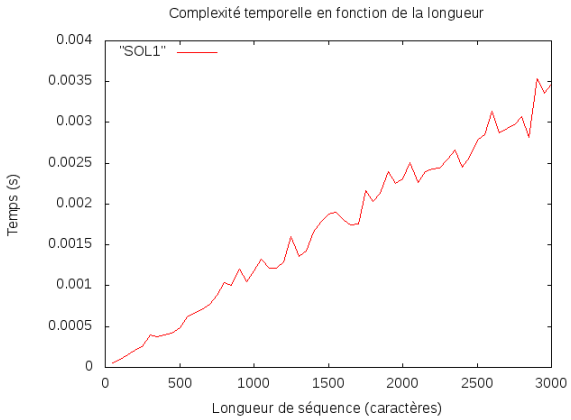
```
def sol1(x, y, F, gap=1,dif=1):
    """Renvoie l'allignement et les deux chaines avec les gap correspondant"""
    i, j = len(x), len(y)
    M = []
    c1 = ""
    c2 = ""
    M.append((len(x)+1,len(y)+1))
    while (i, j) != (0, 0):
        if i > 0 and j > 0 and F[i, j] == F[i-1, j-1] + delta(x[i-1], y[j-1],dif):
            c1 += x[i-1]
            c2 += y[j-1]
            M.append((i, j))
            i, j = i-1, j-1
        elif i > 0 and F[i, j] == F[i-1, j] + gap:
            c2 += "-"
            c1 += x[i-1]
            i -= 1
        else:
            c1 += "-"
            c2 += y[j-1]
            j -= 1
    M.reverse()
    print(c1[::-1])
    print(c2[::-1])
    return (M,c1,c2)
```

Question 4.3 — Fonction d'affichage à partir de la liste du parcours.

```
def affiche(x, y, M):
    i, j = 1, 1
    sx, sy = '', ''
    if M == []:
        sx = max(len(x),len(y))*'-'
        sy = max(len(x),len(y))*'-'
    for (u, v) in M:
        while i < u and j < v:
            sx = sx + x[i-1]
            sy = sy + y[j-1]
            i, j = i+1, j+1
        while i < u:
            sx = sx + x[i-1]
            sy = sy + '-'
            i += 1
        while j < v:
            sx = sx + '-'
            sy = sy + y[j-1]
            j += 1
    print(sx)
    print(sy)
    return (sx,sy)
```

Question 4.4 — Les graphiques ci-dessous montrent l'évolution de la complexité temporelle des algorithmes COUT1 et SOL1 en fonction de la longueur des séquences.





Pour l'algorithme COUT1, on remarque une courbe qui traduit une complexité quadratique, ce qui correspond bien à la complexité $\Theta(mn)$ trouvée dans la partie 1 ($O(m^2)$ comme n est proche de m). En ce qui concerne SOL1, si l'on considère la moyenne des points, la complexité temporelle de l'algorithme est linéaire, ce qui confirme la complexité en $O(n+m)$ (ici $O(2m) = O(m)$ pour les mêmes raisons que COUT1).

Question 4.5 — Sur une machine virtuelle dotée de 4 Go de RAM, on peut calculer un alignement de coût optimal pour une séquence de 20000 caractères. A partir de 50000 caractères, on observe un dépassement de mémoire.

Question 4.6 — Fonction de calcul de la valeur d'un alignement de coût optimal COUT2.

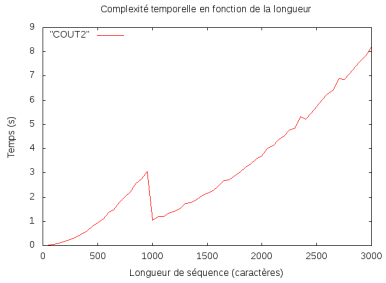
```
def cout2(x, y, gap=1,dif=1):
    m, n = len(x), len(y)
    F1 = np.zeros(n+1, dtype=int)  # ligne i-1
    F2 = np.zeros(n+1, dtype=int)  # ligne i
    for j in range(1, n+1):
        F1[j] = j * gap
    for i in range(1, m+1):
        F2[0] = i * gap
        for j in range(1, n+1):
            F2[j] = min(F1[j-1] + delta(x[i-1], y[j-1],dif),
                        F1[j] + gap,
                        F2[j-1] + gap)
        print(F1)
        F1[:] = F2[:]  # copie de F2 dans F1
    return F2[n]
```

L'algorithme COUT2 retourne les même valeurs que COUT1. On note en outre que le temps d'exécution de COUT2 est également linéaire.

Question 4.7 — Fonction de calcul de la valeur d'un alignement de coût optimal COUT2BIS

```
def cout2bis(x,y,k,l,gap = 1,dif = 1):
    #on recree les 2 tableaux a partir de la case i,j
    m,n = len(x), len(y)-l
    F1 = np.zeros(n+1, dtype=int)  # ligne i-1
    F2 = np.zeros(n+1, dtype=int)  # ligne i
    v1 = 1
    for j in range(0,n+1):
        F1[j] = j*gap
    #Pour avoir
    for i in range(k,m):
        F2[0] = v1*gap
        for j in range(1,n+1):
            F2[j] = min(F1[j-1] + delta(x[i-1], y[j-1],dif),
                        F1[j] + gap,
                        F2[j-1] + gap)
        print(F1)
        F1[:] = F2[:]  # copie de F2 dans F1
        v1 += 1
    print(F2)
    return F2[n]
```

On obtient le graphique suivant avec gnuplot, qui correspond bien à une complexité quadratique :



Question 4.8 — SOL2

```
def SOL2(k1,l1,k2,l2,L,X,Y):
    if k2 - k1 > 0 or l2 - l1:
        if k2-k1 <= 2:
            for i in range(k1,k2+1):
                X2A[i-k1] = X[i]
            for j in range(l1,l2+1):
                Y2A[j-l1] = Y[i]
            L.append(sol1(X,Y,X2A)[0])
            L.append(sol1(X,Y,Y2A)[0])
            return F2A[k2 - k1, l2 - l1]
        else:
            if l2 - l1 <= 2:
                for i in range(k1,k2+1):
                    X2B[i-k1] = X[i]
                for j in range(l1,l2+1):
                    Y2B[j-l1] = Y[i]
                L.append(sol1(X,Y,X2B)[0])
                L.append(sol1(X,Y,Y2B)[0])
                return F2B[k2 - k1, l2 - l1]
            else:
                j = l1 + (l2 - l1)//2
                ip = k1
                valmin = cout2(k1,j) + cout2bis(k1,j)
                for i in range(k+1,k2+1):
                    val = Cout2(i,j) + cout2bis(i,j)
                    if valmin > val:
                        valmin = val
                        ip = i
                return sol2(k1,l1,ip,j,L) + sol2(ip,j,k2,l2,L)
```