Algoritmos para Bioinformática de Secuencias
Tarea 3: Alineamiento de secuencias

Ejercicio 1

En cada iteración del algoritmo de MTP que se estudió en clase (ver la diapositiva 37), el grafo de edición se recorre en "diagonal", donde se visitan todos los vértices aún no visitados de un renglón y luego de una columna en particular. Formalmente, siendo $n \in \mathbb{N}$ el número de renglones y $m \in \mathbb{N}$ el número de columnas en el grafo de alineamiento, en cada iteración $1 \leq k \leq \min\{n, m\}$ de este algoritmo, se visitan los vértices $s_{i,k}$ tales que $k \leq i \leq n$, y los vértices $s_{k,j}$ tales que $k \leq j \leq m$. Si en lugar de esto los vértices se visitaran "columna por columna" o "renglón por renglón", ¿el algoritmo aún produciría la salida correcta? Justifica tu respuesta. .

Sean m el número de columnas y n el número de renglones. Supongamos que se recorre renglón por renglón (sería análogo columna por columna), en este caso se va a ir calculando $S_{i,j}$ de manera que i queda fija y j va variando hasta llegar a $j = m$, una vez ahí incrementamos i hasta llegar a $i = n$. Sin pérdida de generalidad supongamos que $i, j > 0$ al estar en $S_{i,j}$ como el recorrido es renglón por renglón sabemos que en el paso previo se calculó $S_{i,j-1}$. Y también calculó el renglón previo, por lo que también conocemos el valor de $S_{i-1,j}$ y $S_{i-1,j-1}$ y por lo tanto, tenemos todo lo necesario para encontrar el máximo entre $S_{i,j-1}$, $S_{i-1,j}$ y $S_{i-1,j-1} + 1$ cuando $(v_i = w_j)$.

En conclusión: **Sí, el algoritmo seguiría produciendo la salida correcta**. Esto es porque ya sea que recorramos renglón por renglón o columna por columna cuando se quiera calcular $S_{i,j}$, ya habrá sido calculado: $S_{i-1,j}$, $S_{i,j-1}$ y $S_{i-1,j-1}$, por lo tanto **se respetan las dependencias de la programación dinámica**.

Ejercicio 2

En el libro de Compeau & Pevzner, capítulo 5, sección "The changing faces of sequence alignment" (págs. 261–264), se presentan tres variantes del problema de alineamiento de secuencias.

1. Propón un algoritmo ingenuo para resolver el problema del *fitting alignment* e indica cuál es el crecimiento asintótico de su tiempo de ejecución

Sea w y v cadenas que se quieren alinear con $|w| > |v|$, y M la matriz con los scores.

Ingenuous Fitting Alignment (w, v, M):

1. Encontrar todas las subcadenas de w.
2. Hacer el alineamiento global para cada subcadena. Utilizando la matriz M para obtener el puntaje total de acuerdo a los scores y utilizando el algoritmo de Needleman–Wunsch para el alineamiento.
3. Conservar el alineamiento con mayor score.

Análisis del tiempo de ejecución:

1. Todas las subcadenas de un texto T son todas las combinaciones que se puedan obtener a partir de dos índices i y j, de tal forma que $1 \leq i \leq j \leq n$, donde n = longitud del texto. Se forman extrayendo las letras desde la posición i hasta la posición j de la cadena T, es decir, T$[i, ..., j]$.

   Observemos cuántas subcadenas se pueden formar de un texto T. Para esto fijemos el índice i y variemos j, entonces:

   - Si $i = 1$, j puede variar desde 1 hasta n. Por lo que tenemos n **opciones**.

- Si $i = 2$, j puede variar desde 2 hasta n. Por lo que tenemos $n - 1$ **opciones**.
  $\vdots$

- Si $i = n$, j sólo puede ser n. Por lo que tenemos sólo **una opción**.

Esto es equivalente a la suma de los primeros n naturales, donde la fórmula para calcular el total es $\frac{n(n+1)}{2}$, es decir, calcular todas las subcadenas de un texto T es de **orden cuadrático** $O(n^2)$.

2. Por el libro de Algotithms in bioinformatics (pág 33) sabemos que el tiempo de ejecución para el alineamiento global de dos cadenas $S[1..n]$ y $T[1..m]$ utilizando Needleman–Wunsch es de $O(nm)$.

Por lo tanto, como se va a ejecutar n^2 veces el algoritmo de alineamiento de Needleman–Wunsch se tiene que el tiempo de ejecución es $O(n^2 * nm)$, es decir, $O(n^3 m)$.

2. Propón una relación de recurrencia y, si es necesario, modificaciones al grafo de alineamiento para resolver en tiempo $O(nm)$ cada uno de los tres problemas descritos en la sección mencionada.

Edit Distance Problem
Encuentra la distancia entre dos cadenas.
Input: Dos cadenas.
Output: La distancia de edición entre esas cadenas.

Queremos el mínimo costo para transformar una cadena en otra, y tenemos 3 opciones:

- Insertar
- Eliminar
- Transformar

Sean v y w las cadenas de las que queremos obtener la distancia de edición, con $|v| = n$ y $|w| = m$. Primero pensemos en el último paso de la transformación de v a w, en este paso sólo se pudo haber: borrado un último carácter, insertado un último carácter, cambiado un último carácter o quedado igual el último carácter.

- Si en el último paso se eliminó un carácter, implica que ya se había transformado $v[1, \ldots, n-1]$ en $w[1, \ldots, m]$.
- Si en el último paso se agrego un carácter implica que ya se había transformado $v[1, \ldots, n]$ en $w[1, \ldots, m-1]$.
- Si en el último paso v_n se transformó (o se quedo igual) entonces implica que ya se había transformado $v[1, \ldots, n-1]$ en $w[1, \ldots, m-1]$.

Notemos que el problema disminuye de dimensión, es decir, **exhibe subestructura óptima** (se puede demostrar por contradicción que esto es cierto).

Así, definimos: $D(i, j)$ como el costo mínimo para transformar $v[1 \ldots i]$ en $w[1 \ldots j]$.

Caso base de la recurrencia.

El caso base es el de transformar una cadena vacía en otra, para esto se tienen dos casos:

- $D(0, j) = j$ (sólo se hacen j inserciones).
- $D(i, 0) = i$ (sólo se hacen i eliminaciones).

Recurrencia.

Para $i, j > 0$:

- Opción 1: Eliminar v_i. Entonces ya se había transformado $v[1,\ldots,i-1]$ en $w[1,\ldots,i]$. Costo total $D(i-1,j)+1$.
- Opción 2: Insertar v_i. Entonces ya se había transformado $v[1,\ldots,i]$ en $w[1,\ldots,i-1]$. Costo total $D(i,j-1)+1$.
- Opción 3: Transformar (o dejar igual) v_i. Entonces ya se había transformado $v[1,\ldots,i-1]$ en $w[1,\ldots,i-1]$. A su vez aquí hay dos opciones:
  - Si v_i se transforma entonces el costo es $D(i-1,j-1)+1$.
  - Si v_i no se transforma entonces el costo es $D(i-1,j-1)$.

Como queremos el menor costo, tomamos el mínimo entre estos, por lo tanto **la recurrencia es:**

$$D(i,j) = \min \begin{cases} D(i-1,j)+1, \\ D(i,j-1)+1, \\ D(i-1,j-1)+\text{cost}(v_i,w_j) \end{cases}$$

donde

$$\text{cost}(v_i,w_j) = \begin{cases} 0 & \text{si } v_i = w_j, \\ 1 & \text{si } v_i \neq w_j. \end{cases}$$

Complejidad.

Luego con las distancias $D(i,j)$ podemos construir una matriz D de nxm donde la entrada $(i,j) = D(i,j)$. Por lo tanto, calcular todas las entrada tomaría tiempo $O(nm)$ puesto que para calcular cada entrada sólo se hacen operaciones básicas de comparación i.e tiempo $O(1)$ para cada una.

Fitting Alignment Problem
Construye una alineación de ajuste con la puntuación más alta entre dos cadenas de texto.
Input: Cadenas v, w y una matriz de scores M.
Output: Un alineamiento por ajuste de máxima puntuación entre v y w, de acuerdo con el score. Donde $|v| < |w|$.

La idea básica es **alinear toda la cadena v con alguna subcadena de w**. Entonces, esto implica que v no se puede recortar y que w sí se puede recortar por los extremos.

Definimos $D(i,j)$ como el mejor puntaje alineando $v[1,\ldots,i]$ con $w[1,\ldots,j]$.

Caso base de la recurrencia.

- $D(0,j) = 0 \quad \forall j$. Esto significa que no hemos alineado ningún carácter de v, por lo tanto, el mejor puntaje por defecto es 0.
- Si la cadena larga está vacía pero la corta no, el alineamiento no es válido: $D(i,0) = -\infty \quad$ para $i > 0$. Escribimos $-\infty$ para que nunca se elija esa opción.

Recurrencia.

Para $i,j > 0$:

$$D(i,j) = \max \begin{cases} D(i-1,j)+\text{gap}, \\ D(i,j-1)+\text{gap}, \\ D(i-1,j-1)+\text{Score}(v_i,w_j). \end{cases}$$

Esta relación considera las tres posibles formas en que puede terminar un alineamiento: introducir un gap en w, introducir un gap en v, o alinear los caracteres v_i y w_j.

Dado que la cadena corta v debe alinearse completamente, pero la cadena larga w puede tener caracteres sobrantes al final, el puntaje óptimo del fitting alignment es:

$$\max_{0 \leq j \leq m} D(n, j).$$

Complejidad.

Podemos guardar los resultados en una matriz que tendrá tamaño $(n+1)(m+1)$, por lo que el número de subproblemas es $O(nm)$. Cada celda se calcula en tiempo constante, lo que da una complejidad de $O(nm)$.

Overlap Alignment Problem

Construir un alineamiento por superposición con la máxima puntuación entre dos cadenas.
Input: Dos cadenas v y w y una matriz de puntuación Score.
Output: Un alineamiento por superposición de máxima puntuación entre las dos cadenas, de acuerdo con Score.

La idea central de este algoritmo es alinear un sufijo de v con un prefijo de w. Por lo tanto, ninguna de las dos cadenas de alinea completa y ambas pueden sobrar, con la condición de que v sólo puede sobrar al inicio y w sólo puede sobrar al final.

Definimos el subproblema como: $D(i, j)$ como el mejor puntaje alineando $v[1, \ldots, i]$ con $w[1, \ldots, j]$.

Casos.

- $D(i, 0) = 0 \quad \forall i$, significa que se ha movido de izquierda a derecha o dicho de otro modo, se han usado caracteres de v, pero no de w. Esto no tiene penalización, pues está dentro de los casos permitidos.
- $D(0, j) = 0 \quad \forall j$, significa que se movió de derecha a izquierda, es decir, sólo se han descartado j caracteres de w, lo cual tampoco tiene penalización.

Recurrencia.

Para $i, j > 0$:

$$D(i, j) = \max \begin{cases} D(i-1, j) + \text{gap}, \\ D(i, j-1) + \text{gap}, \\ D(i-1, j-1) + \text{Score}(v_i, w_j). \end{cases}$$

El puntaje óptimo del overlap alignment corresponde al máximo valor en la última fila o en la última columna de la tabla:

$$\max \left(\max_{0 \leq i \leq n} D(i, m), \max_{0 \leq j \leq m} D(n, j) \right).$$

Complejidad.

Los resultados se pueden guardar en una tabla de $(n+1)(m+1)$, por lo que el número de subproblemas es $O(nm)$. Cada celda se calcula en tiempo constante, dando una complejidad temporal total de $O(nm)$. Por lo que la complejidad es de $O(nm)$.

Ejercicio 3

Demuestra que el problema del MTP exhibe una subestructura óptima (revisa el libro de Cormen et al., 3ra ed., págs. 379–384).

El objetivo del problema del turista de Manhattan es encontrar la ruta más corta de la fuente al sumidero que recorra el máximo número de atracciones, donde únicamente se puede viajar hacia el este o hacia el sur.

La relación de recurrencia para el problema de programación dinámica es la siguiente:

$$S_{i,j} = max \begin{cases} S_{i-1,j} + \text{peso de la arista entre (i-1, j) e (i, j)} \\ S_{i,j-1} + \text{peso de la arista entre (i, j-1) e (i, j)} \end{cases} \tag{1}$$

Demostración por contradicción

Sea P el camino con peso máximo (W_{max}) que conecta a la fuente F y el sumidero S, y p un subcamino que va de del vértice u al v dentro de el camino P.

Supongamos que el problema no tiene subestructura óptima. Esto implica que existe otro camino p^* que conecta a los vértices u y v y ese camino p^* tiene mayor peso que p, es decir, $0 \leq w(p) < w(p^*)$ lo cual implica que $w(p^*) - w(p) = x > 0$ donde x es la diferencia de pesos entre el subcamino p y el subcamino p^*.

Como ambos son subcaminos de P y ambos conectan al vértice u con v, podemos usar ahora el subcamino p^* en lugar del subcamino p sin ver a la trayectoría P afectada. Entonces el nuevo peso del camino P sería $W_{max*} = W_{max} + x$, y esto implica una contradicción ya que habíamos supuesto que W_{max} era el peso máximo. Por lo tanto el problema del MTP exhibe una subestructura óptima.

Ejercicio 4

En la pág. 30 del libro de Compeau & Pevzner (2015), vol. 2, se muestra el pseudocódigo del algoritmo NeighborJoining. *Escribe y resuelve la relación de recurrencia que expresa el tiempo de ejecución de este algoritmo. Recuerda que la matriz D es cuadrada, simétrica, que ya está dada como entrada, y que su diagonal es cero. Recuerda también que el árbol T puede manejarse como un caso especial de un árbol binario de búsqueda (ver el libro de Cormen et al., 3ra ed., págs. 286–288, 294–298).*

El pseudocódigo del algoritmo NeighborJoining es el siguiente:

```
NeighborJoining(D, n)
    if n = 2
        T ← the tree consisting of a single edge of length D_{1,2}
        return T
    D* ← the neighbor-joining matrix constructed from the distance matrix D
    find elements i and j such that D*_{i,j} is a minimum non-diagonal element of D*
    Δ ← (TotalDistance_D(i) − TotalDistance_D(j))/(n − 2)
    limbLength_i ← 1/2(D_{i,j} + Δ)
    limbLength_j ← 1/2(D_{i,j} − Δ)
    add a new row/column m to D so that D_{k,m} = D_{m,k} = 1/2(D_{k,i} + D_{k,j} − D_{i,j})
        for any k
    remove rows i and j from D
    remove columns i and j from D
    T ← NeighborJoining(D, n − 1)
    add two new limbs (connecting node m with leaves i and j) to the tree T
    assign length limbLength_i to Limb(i)
    assign length limbLength_j to Limb(j)
    return T
```

Obtención de la recurrencia

En una llamada al algoritmo con n nodos se realizan las siguientes operaciones:

- Construcción de la matriz neighbor-joining D.

  Debido a que D es simétrica y tiene diagonal cero, basta considerar únicamente las entradas no diagonales de una mitad de la matriz. No obstante, el número de entradas que deben procesarse sigue siendo proporcional a n^2 ($n^2/2$). Por lo tanto, la construcción de D tiene costo $O(n^2)$.

- Búsqueda del elemento mínimo no diagonal en D.

  Esta operación requiere recorrer las entradas relevantes de la matriz, lo cual toma tiempo $O(n^2)$.

- Actualización de la matriz D al agregar el nodo m y eliminar los nodos i y j.

  Para cada nodo restante k, se calcula la distancia $D_{k,m}$, lo cual cuesta $O(n)$.

- Actualización del árbol T.

  Las operaciones sobre el árbol (inserción de nodos y asignación de longitudes) pueden realizarse en tiempo $O(\log n)$, al manejar T como un árbol binario de búsqueda.

El costo dominante de una llamada al algoritmo con n nodos es, por tanto, $O(n^2)$. Tras estas operaciones, el algoritmo se invoca recursivamente con una matriz de tamaño $(n-1) \times (n-1)$.

De este modo, la recurrencia que describe el tiempo de ejecución es:

$$T(n) = T(n-1) + O(n^2)$$

con el caso base:

$$T(2) = O(1)$$

Solución de la recurrencia

Paso 1: Expansión de la recurrencia

Expandimos la recurrencia de forma iterativa:

$$\begin{aligned} T(n) &= T(n-1) + cn^2 \\ &= \big(T(n-2) + c(n-1)^2\big) + cn^2 \\ &= T(n-2) + c(n-1)^2 + cn^2 \\ &= T(n-3) + c(n-2)^2 + c(n-1)^2 + cn^2 \\ &\vdots \\ &= T(2) + c\sum_{k=3}^{n} k^2 \end{aligned}$$

donde c es una constante positiva.

Paso 2: Sustitución del caso base

Dado que $T(2) = O(1)$, se obtiene:

$$T(n) = O(1) + c\sum_{k=3}^{n} k^2$$

El término constante no afecta el orden de crecimiento.

Paso 3: Evaluación de la suma

Sabemos que:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

Por lo tanto:

$$\sum_{k=3}^{n} k^2 = \Theta(n^3)$$

Por lo tanto, el tiempo de ejecución del algoritmo Neighbor-Joining está dado por:

$$T(n) = \Theta(n^3)$$