UNIVERSIDAD DE GRANADA

Universidad De Granada

E.T.S. de Ingenierías Informática y de Telecomunicación

Práctica 4: Programación Dinámica

Algorítmica

Autores:

Noura Lachhab Bouhmadi
Eduardo Rodríguez Cao
Ramón Liria Sánchez
Quintín Mesa Romero

Mayo 2022

ÍNDICE

0. INTRODUCCIÓN

A menudo, a la hora de hacer frente a un problema, barajamos multitud de formas de resolverlo. Una forma de resolver un problema es caracterizar su solución mediante las soluciones de los subproblemas en los que lo podamos dividir. Empleando esta técnica de forma recursiva, se obtienen métodos eficientes de solución.
Pero muchas veces cuando tenemos un problema de tamaño n, solo puede obtenerse una caracterización efectiva de su solución en términos de la solución de los subproblemas de tamaño n-1. La Programación Dinámica proporciona en estos casos algoritmos eficientes.

Y de esto mismo trata la práctica que estamos abordando. Pretendemos resolver un problema que se expondrá en el siguiente apartado, aplicando programación dinámica. Tenemos como objetivos: la comprensión del enfoque de la programación dinámica, la identificación de las cuatro fases de la programación dinámica (naturaleza n-etápica, verificación del Principio de Optimalidad, el planteamiento de una recurrencia, y el cálculo de la solución), y, claramente, resolver adecuadamente el problema que se plantea.

En esta práctica se planteará el ejercicio completo a resolver, se expondrá la recurrencia que se ha utilizado para la definición del algoritmo que resuelve el problema, se expondrá su implementación y adjuntaremos un ejemplo de ejecución del algoritmo. Finalmente sacaremos algunas conclusiones al respecto.

Por último, veremos si la programación dinámica tiene sentido en la resolución de este problema en concreto.

1. LISTADO DEL HARDWARE UTILIZADO EN LA PRÁCTICA

A continuación se presenta un listado de los dispositivos, junto con sus características, con los que se ha llevado a cabo esta práctica.

- ❖ Ordenador de Noura Lachhab Bouhmadi:
  - ☐ **Nombre del dispositivo**: GF63-Thin-10SCXR
  - ☐ **CPU:** Intel® Core™ i7-10750H CPU @ 2.60GHz × 12
  - ☐ **Memoria RAM:** 16 GB
  - ☐ **Tarjeta gráfica:** Intel® UHD Graphics (CML GT2)
  - ☐ **Sistema operativo:** Ubuntu 20.04.3 LTS
- ❖ Ordenador de Eduardo Rodríguez Cao:
  - ☐ **Nombre del dispositivo**: Acer Nitro 5 AN515
  - ☐ **CPU:** Intel® Core™ i5-7300HQ CPU @ 2.50GHz × 4
  - ☐ **Memoria RAM:** 12 GB
  - ☐ **Tarjeta gráfica:** NVIDIA GeForce GTX 1050 Ti
  - ☐ **Sistema operativo:** Ubuntu 20.04.3 LTS
- ❖ Ordenador de Ramón Liria Sánchez:
  - ☐ **Nombre del dispositivo**: Asus Rog Strix G513IH-HN008
  - ☐ **CPU:** AMD Ryzen 7 4800H (8MB Cache, 2.9GHz)
  - ☐ **Memoria RAM:** 16 GB
  - ☐ **Tarjeta gráfica:** NVIDIA GeForce GTX 1650 (4GB GDDR6)
  - ☐ **Sistema operativo:** Ubuntu 20.04.3 LTS
- ❖ Ordenador de Quintín Mesa Romero:
  - ☐ **Nombre del dispositivo**: Lenovo-Yoga-S740-14IIL
  - ☐ **CPU:** Intel® Core™ i7-1065G7 CPU @ 1.30GHz × 8
  - ☐ **Memoria RAM:** 16 GB
  - ☐ **Tarjeta gráfica:** Intel® Iris(R) Plus Graphics (ICL GT2)
  - ☐ **Sistema operativo:** Ubuntu 20.04.3 LTS

2. SOFTWARE UTILIZADO

A continuación se especifica todo lo relativo al software utilizado en la práctica.

- Para la **edición del código de los algoritmos** (programas escritos en el lenguaje de programación C++), se ha empleado el editor de textos que por defecto viene instalado en Linux:

- Para la **compilación de los programas** se ha utilizado el compilador de *línea de órdenes* que compila y enlaza programas en C++, **g++**.
- Para la elaboración de la memoria de la práctica se ha utilizado la herramienta **google docs**.

3. EJERCICIO 1

3.1 PLANTEAMIENTO DEL EJERCICIO

Dos hermanos fueron separados al nacer y mediante un programa de televisión se han enterado que podrían ser hermanos. Ante esto, los dos están de acuerdo en hacerse un test de ADN para verificar si realmente son hermanos.

Deben encontrar el % de similitud que existe entre estos posibles hermanos. Lo haremos para 2 entradas posibles.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 Hermano 1 _ abbcdefabcdxzyccd
 Hermano 2 _ abbcdeafbcdzxyccd
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 Hermano 1 _ 0101110001000101010100100010010001001
 Hermano 2 _ 1100001001001010100010100100110101000
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

3.2 RECURRENCIA UTILIZADA EN EL ALGORITMO

Sean **x** e **y** dos secuencias de ADN. Nos proponemos encontrar la secuencia común a x e y de mayor longitud. Consideremos:

$$\mathbf{x} = < x_1, x_2, \ldots, x_n >$$

$$\mathbf{y} = < y_1, y_2, \ldots, y_n >$$

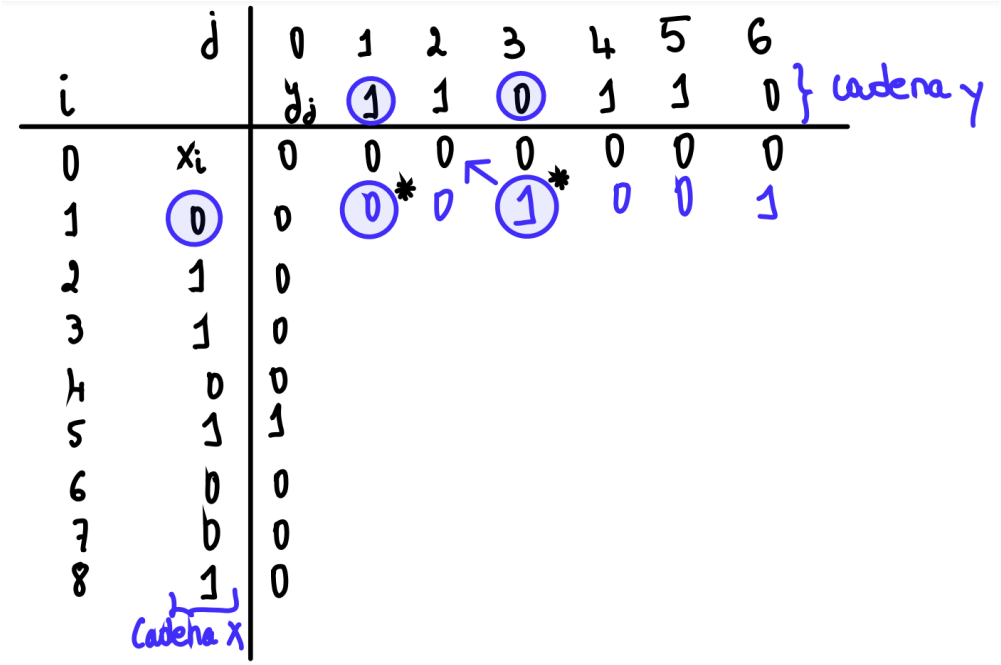
y supongamos que la secuencia común de mayor longitud es:

$$\mathbf{z} = < z_1, z_2, \ldots, z_k >$$

- ➔ Si $x_n = y_n$ entonces $z_k = x_n = y_n$ y z_{k-1} es una secuencia de mayor longitud (SML) de x_{n-1} e y_{n-1}
- ➔ Si $x_n \neq y_n$ entonces **z** es una SML de x_{n-1} e **y** o de **x** e y_{n-1}. En cuyo caso cogeríamos la máxima de las dos.

Para hallar la secuencia de mayor longitud, usaremos una tabla la cual completaremos por filas, tras lo cual podremos alcanzar el array final con la solución.

Sea el siguiente ejemplo de tabla:



0* : Comparamos x_i con y_j: $x_i \neq y_j$ entonces pongo 0.
1*: Cuando coincide x_i con y_j miramos lo que ocurre en la posición anterior (i-1,j-1) (diagonal) y ponemos en la posición (i,j) el valor de aquella + 1.
Nota: Si estamos buscando una solución, no importa a cuál apuntemos, pero si queremos calcular todas las soluciones entonces hay que almacenar todos los punteros.

A la hora de hallar la solución, mirando la tabla, partimos de la última casilla y vamos siguiendo las flechas. Las casillas que constituirán la solución serán los que tengan una flecha en diagonal.

Dicha tabla se construye de la siguiente forma:

- Se completan los bordes de la tabla con ceros.
- Si $\mathbf{x_i} = \mathbf{y_i}$ entonces ponemos un puntero en esa casilla, junto con el valor (i-1,j-1) + 1
- En cualquier otro caso, se pone el mayor valor entre el elemento en (i-1,j) y (i,j-1) en la casilla donde apunte la flecha correspondiente.

De todo esto, deducimos la siguiente solución recursiva:

$$I(i,j) = \begin{cases} 0 & si \quad i = j = 0 \\ I(i-1, j-1) + 1 & si \quad x_i = y_j \\ max(I(i-1,j), I(i,j-1)) & si \quad x_i \neq y_j \end{cases}$$

3.3 IMPLEMENTACIÓN DEL ALGORITMO

En la implementación del algoritmo que resuelve el problema de encontrar la mayor subsecuencia de ADN común a dos dadas, se ha utilizado una función auxiliar destinada al cálculo de la tabla con las longitudes de la mayor subsecuencia en la que coinciden las dos cadenas de ADN.

```
/**
 * @brief Calcula la tabla con las longitudes de la mayor subsecuencia
 * en la que coindicen los vectores a y b
 * @param a Vector con la primera cadena
 * @param a Vector con la segunda cadena
 */
vector<vector<int>> tableSequences(const vector<char> & a, const vecto
r<char> & b) {
  // Crea la matriz de longitud de subsecuencias
  int sizeA = a.size();
  int sizeB = b.size();
  vector<vector<int>> table(sizeA + 1, vector<int>(sizeB + 1));

  // Rellena la primera fila con 0
  for (int i = 0; i <= sizeB; i++)
    table[0][i] = 0;
  // Rellena la primera columna con 0
  for (int i = 1; i <= sizeA; i++)
    table[i][0] = 0;

  // Rellena la tabla con la longitud de las subsecuencias
  for (int i = 1; i <= sizeA; ++i)
    for (int j = 1; j <= sizeB; ++j){
      if (a[i-1] == b[j-1])
        table[i][j] = 1 + table[i-1][j-1];
      else
        table[i][j] = max(table[i-1][j], table[i][j-1]);
    }

  return table;
}
```

A la vista del código, podemos observar claramente que la eficiencia de la función es $O(n^2)$, pues tenemos cuatro bucles, tres de ellos al mismo nivel y uno anidado, luego, por la regla de la suma y el producto llegamos a que tiene una eficiencia cuadrática.

A partir de la tabla obtenida con la función anterior, podemos obtener la solución del problema. Para ello, hemos implementado el siguiente algoritmo:

```
/**
 * @brief Busca la mayor subsecuencia en la que coinciden los vector
 * a y b
 * @param a Vector con la primera cadena
 * @param a Vector con la segunda cadena
 * @param result Vector con la subsecuencia resultante
 */
void longestSequence(const vector<char> & a, const vector<char> & b, s
tack<char> & result) {
  auto table = tableSequences(a,b);
  int i = a.size();
  int j = b.size();
  while (i != 0 && j != 0) {
    if ((table[i][j] - 1) == table[i-1][j-1] && a[i-1] == b[j-1]) {
      result.push(a[i-1]);
      i--;
      j--;
    }
    else if (table[i][j] == table[i-1][j])
      i--;
    else
      j--;
  }
}
```

Teniendo en cuenta que la eficiencia de la función auxiliar que calcula la tabla es $O(n^2)$, y que tenemos al mismo nivel que la sentencia en la que se llama a dicha función, un bucle while que se ejecuta n veces, y por lo tanto, por la regla de la suma concluimos que la eficiencia del algoritmo que da solución al problema es $O(n^2)$.

3.4 RESULTADOS

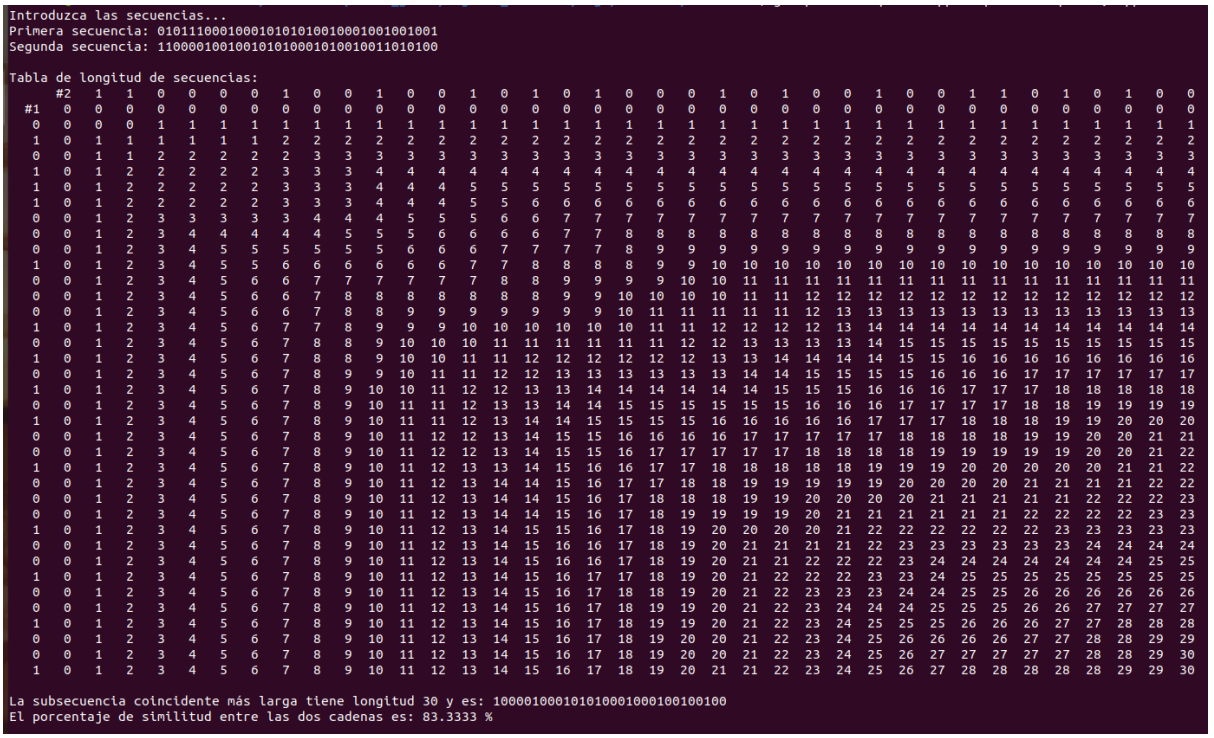
A continuación vemos el resultado de ejecutar el programa para las dos secuencias de ADN de la primera entrada:

```
Introduzca las secuencias...
Primera secuencia: abbcdefabcdxzyccd
Segunda secuencia: abbcdeafbcdzxyccd

Tabla de longitud de secuencias:
      #2   a   b   b   c   d   e   a   f   b   c   d   z   x   y   c   c   d
  #1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   a   0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
   b   0   1   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
   b   0   1   2   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3
   c   0   1   2   3   4   4   4   4   4   4   4   4   4   4   4   4   4   4
   d   0   1   2   3   4   5   5   5   5   5   5   5   5   5   5   5   5   5
   e   0   1   2   3   4   5   6   6   6   6   6   6   6   6   6   6   6   6
   f   0   1   2   3   4   5   6   6   7   7   7   7   7   7   7   7   7   7
   a   0   1   2   3   4   5   6   7   7   7   7   7   7   7   7   7   7   7
   b   0   1   2   3   4   5   6   7   7   8   8   8   8   8   8   8   8   8
   c   0   1   2   3   4   5   6   7   7   8   9   9   9   9   9   9   9   9
   d   0   1   2   3   4   5   6   7   7   8   9  10  10  10  10  10  10  10
   x   0   1   2   3   4   5   6   7   7   8   9  10  10  11  11  11  11  11
   z   0   1   2   3   4   5   6   7   7   8   9  10  11  11  11  11  11  11
   y   0   1   2   3   4   5   6   7   7   8   9  10  11  11  12  12  12  12
   c   0   1   2   3   4   5   6   7   7   8   9  10  11  11  12  13  13  13
   c   0   1   2   3   4   5   6   7   7   8   9  10  11  11  12  13  14  14
   d   0   1   2   3   4   5   6   7   7   8   9  10  11  11  12  13  14  15

La subsecuencia coincidente más larga tiene longitud 15 y es: abbcdefbcdxyccd
El porcentaje de similitud entre las dos cadenas es: 88.2353 %
```

Aquí se muestran los resultados obtenidos para la dos secuencias de ADN de la segunda entrada:



Nota: Resultados obtenidos con el ordenador de Eduardo.

4. CONCLUSIONES

En resumen, la programación dinámica aplicada a este problema proporciona una solución que no solo es óptima sino además eficiente. El algoritmo de fuerza bruta en este caso sería exponencial ($O(2^n)$), difícil de aplicar incluso para valores de n pequeños, mientras que por programación dinámica reducimos la complejidad a una cuadrática.

Sin embargo, hay que tener en cuenta que se gasta bastante memoria, de hecho la complejidad espacial también es cuadrática. Aunque la memoria no suele ser un problema, es importante tener en cuenta que esta solución tiene límites no solo temporales sino de memoria también.