UNIVERSIDAD DE GRANADA

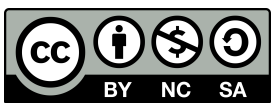
Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Práctica 4: Programación Dinámica

Doble Grado Ingeniería Informática y Matemáticas

Autores:
Jose Alberto Hoces Castro
Javier Gómez López
Moya Martín Castaño

Mayo 2022



Índice

1. Introducción

El objetivo de esta práctica es aprender a implementar y utilizar algoritmos que utilizan la técnica de programación dinámica. Para ello hemos tenido que resolver el siguiente ejercicio:

Enunciado. 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.

Dadas las 2 entradas:

PRIMERA
Hermano 1 - abbcdefabcdxzyccd
Hermano 2 - abbcdeafbcdzxyccd

SEGUNDA
Hermano 1 - 0101110001000101010100100010010010001
Hermano 2 - 1100001001001010100010100100110101000

- Deben encontrar el % de similitud que existe entre estos posibles hermanos, como es un ejemplo lo haremos para 2 entradas posibles.
- Dar las salidas (secuencia más larga) de las 2 entradas.
- Dar la matriz de los cálculos de la primera entrada.

2. Requisitos de la PD

Como ya sabemos, la programación dinámica se aplica en cuatro fases que vamos a identificar:

- **Naturaleza n-etápica:** Para hallar la mayor subsecuencia entre dos secuencias dadas, comenzamos hallando la subsecuencia más larga para el primer carácter de las dos secuencias, después la más larga teniendo en cuenta los dos primeros caracteres, y así hasta tener en cuenta los caracteres de ambas secuencias en su totalidad.
- **Verificación del POB (Principio de Optimalidad de Bellman):** Sea $s_i, s_2, ..., s_k, s_j$ la mayor subsecuencia del índice i hasta el j. Comenzando en el elemento s_i correspondiente al índice i, entonces hemos encontrado, el siguiente elemento común en ambas entradas, que es s_2. Por tanto, ahora necesitamos encontrar la mayor subsecuencia común entre el índice que le corresponde a s_2, es decir, $i+1$, llamémoslo l, y el índice j. Es claro que entre l y j la subsecuencia $s_2, ..., s_k$ debe ser la de mayor longitud. De no serlo, entonces existiría una subsecuencia $s_2, t_3, t4, ..., t_r, d_j$ con $r > k$ entre los índices l y j que sería de mayor longitud. Entonces, tendríamos que $s_i, s_2, t_3, t4, ..., t_r, s_j$ es una subsecuencia de mayor longitud que $s_i, s_2, ..., s_k, s_j$ del índice i al j.
- **Planteamiento de una recurrencia:** Para resolver este problema, vamos a hacer uso de una matriz de dimensión $n \cdot m$, siendo n y m las longitudes de las secuencias de nuestro problema:

  1. Comenzaremos rellenando la primera fila y la primera columna de ceros.

  2. Rellenamos el resto de la matriz según el siguiente criterio:

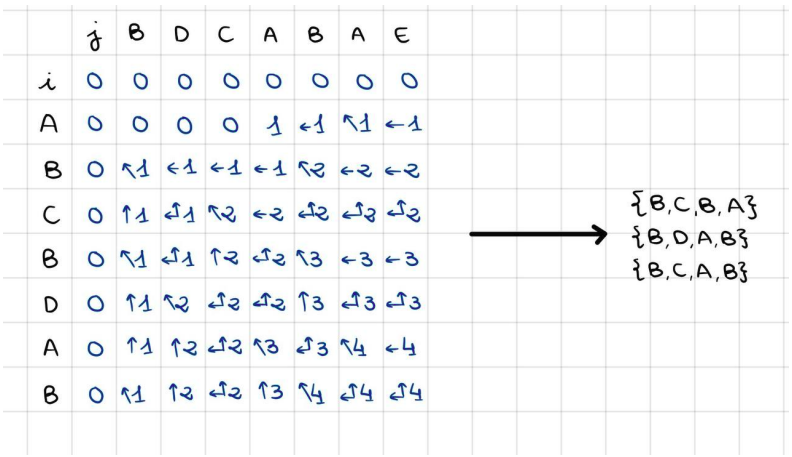
  Si el carácter i de la primera secuencia coincide con el carácter j de la segunda secuencia, la casilla $< i, j >$ se rellenará con el valor de la casilla $< i-1, j-1 >$ más 1. Así, estamos teniendo en cuenta si ha habido coincidencias anteriormente a esos índices para poder formar la subsecuencia. En caso de no haber coincidencia entre esos índices, se toma el valor máximo entre los de las casillas $< i-1, j >$ y $< i, j-1 >$, ya que debemos tener en cuenta las coincidencias que ha habido antes de esos índices a pesar de no haber coincidencia entre ellos.

  3. Por último, hallamos el máximo de la matriz (que estará situado en la última fila) deshaciendo el camino que hemos ido haciendo por las casillas y teniendo en cuenta si estamos en una posición de coincidencia o no.

  Gracias a esta forma de ir anotando las coincidencias que se van hallando entre ambas secuencias, estamos almacenando de forma numérica la secuencia más larga dados dos índices de ambas secuencias. En resumen, nuestra matriz se rellenará según la siguiente expresión:

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

Dado un ejemplo pequeño de secuencias, sean $< A, B, C, B, D, A, B >$ y $< B, D, C, A, B, A, E >$, la matriz usando nuestro algoritmo quedaría así:



- **Cálculo de una solución:** Una vez explicado el algoritmo que vamos a usar, mostramos cómo realizamos la entrada de datos en nuestro programa, al igual que la implementación de los 3 pasos descritos anteriormente:

1. Entrada de datos

```
ifstream file;
file.open(argv[1]);
string entrada1;
file >> entrada1;

string entrada2;
file >> entrada2;
file.close();

int n = entrada1.size();
int m = entrada2.size();
```

2. Creación e inicialización de la matriz

```
int ** matriz_calculos = nullptr;

// Reserva de memoria

matriz_calculos = new int * [n+1];
for(int i=0; i < n+1; i++){
        matriz_calculos[i] = new int [m+1];
}

// Inicialización a 0 de la matriz

for(int i = 0; i < n+1; i++){
        for(int j = 0; j < m+1; j++){
                matriz_calculos[i][j] = 0;
        }
}
```

3. Cálculo de la matriz

```
// Cálculo de la matriz de cálculos

for(int i = 1; i < n+1; i++){
```

```
        for(int j = 1; j < m+1; j++){
                if(entrada1[i-1] == entrada2[j-1]){
                        matriz_calculos[i][j] = matriz_calculos[i-1][j-1]+1;
                }
                else{
                if(matriz_calculos[i][j-1] >= matriz_calculos[i-1][j]){
                                matriz_calculos[i][j] = matriz_calculos[i][j-1];
                        }
                        else{
                                matriz_calculos[i][j] = matriz_calculos[i-1][j];
                        }
                }
        }
}
```

4. Localización del máximo y obtención de la subsecuencia más larga

```
// Buscamos el máximo en la última fila de la matriz de cálculos

int max = 0;
int posj = 0;

for(int j = 1; j < m+1; j++){
        if(matriz_calculos[n][j] >= max){
                max = matriz_calculos[n][j];
                posj = j;
        }
}

// Realizamos el camino hacia atrás y almacenamos la subsecuencia en una lista

list<char> resultado;
int i = n;

while(i >= 1){

        // Si hay coincidencia, almacenamos la letra al principio de la cadena
        // Nos vamos a la anterior fila en diagonal

        if(entrada1[i-1] == entrada2[posj-1]){
                resultado.push_front(entrada2[posj-1]);
                posj--;
                i--;
        }
        else{

                // Si no hay coincidencia, vemos cuál es el máximo de entre la
                // casilla de la izquierda y la de arriba a la actual, y nos
                // movemos hacia donde esté el máximo

                if(matriz_calculos[i][posj-1] == matriz_calculos[i][posj]){
                        posj--;
                }
                else{
                        i--;
                }
        }
}
```

Para el cálculo del **porcentaje de similitud**, simplemente dividimos el tamaño de la subsecuencia más larga hallada entre el tamaño de cualquiera de las dos cadenas, ya que son de la misma longitud, y multiplicamos el valor del cociente por 100:

```
double similitud = (double)max/n * 100
```

‘

3. Solución

Una vez expuesto nuestro algoritmo para resolver el problema y su implementación en C++, mostramos los resultados que se nos pedían:

- **1. Secuencias más largas de ambos casos**

SECUENCIA MÁS LARGA LETRAS: a b b c d e a b c d z y c c d
SECUENCIA MÁS LARGA NÚMEROS: 1 1 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0

- **2. Porcentaje de similitud**

PORCENTAJE DE SIMILITUD LETRAS: 88.2353 %
PORCENTAJE DE SIMILITUD NÚMEROS: 83.3333 %

- **3. Matriz de cálculos del primer problema**

		a	b	b	c	d	e	a	f	b	c	d	z	x	y	c	c	d
	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

4. Conclusiones

- Si hubiésemos resuelto este problema usando la **fuerza bruta**, es decir, enumerando todas las subsecuencias comunes existentes, obtendríamos un algoritmo de orden $O(d^n)$, siendo d la longitud de las cadenas.
- Gracias a la **programación dinámica**, evitamos muchos de los cálculos que se harían en **fuerza bruta**, obteniendo así un algoritmo de **orden polinomial**, en este caso concretamente $O(n \cdot m)$, siendo n y m las longitudes de las secuencias de entrada.
- Obtenemos la **solución optimal**, cosa que no habríamos logrado con otras técnicas ya vistas como **Greedy**
- También ganamos ventaja respecto a **Divide y Vencerás**, pues en **programación dinámica** tenemos problemas que están **encajados**, por lo que **no repetimos cálculos**.
- Como principal **inconveniente** de la **programación dinámica**, se hace un **gran uso de recursos**, en nuestro caso, la **memoria** reservada para nuestra **matriz de cálculos**.