



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Ejercicio 07: Diseño de soluciones con Programación Dinámica

Unidad de aprendizaje: Análisis de Algoritmos

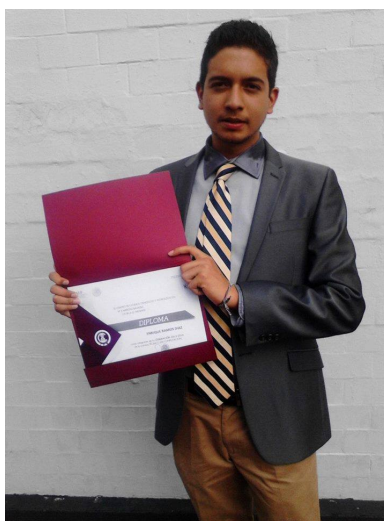
Grupo: 3CM3

Alumno:

Ramos Diaz Enrique

Profesor(a):

Franco Martínez Edgardo Adrián



19 de noviembre 2018

Índice

1	Subsecuencia Común más Larga	2
1.1	Problema	2
1.2	Análisis y solución	2
1.3	Código resultante	3
1.4	Validación del juez online	4
2	ELIS - Fácil Subsecuencia en Aumento más Larga	5
2.1	Problema	5
2.2	Análisis y solución	5
2.3	Código resultante	6
2.4	Validación del juez online	7
3	KNAPSACK - El problema de la mochila	8
3.1	Problema	8
3.2	Análisis y solución	8
3.3	Código resultante	9
3.4	Validación del juez online	10
4	Fuentes consultadas de apoyo	10

1. Subsecuencia Común más Larga

1.1. Problema

Al finalizar su viaje por cuba, Edgardo se puso a pensar acerca de problemas mas interesantes que sus alumnos podrían resolver.

En esta ocasión tu trabajo es el siguiente: Dadas 2 cadenas A y B, debes de encontrar la subsecuencia común mas larga entre ambas cadenas.

Entrada

La primera linea contendrá la cadena A. En la segunda linea vendrá la cadena B.

Salida

La longitud de la subsecuencia común mas larga.

1.2. Análisis y solución

Primero necesitamos una solución recursiva.

Si las dos cadenas A y B comienzan con el mismo carácter, siempre es seguro elegir ese carácter inicial como el primer elemento de la LCS, debido a que si luego tenemos otra subsecuencia, podemos reemplazar el carácter inicial por el otro de ésta en la LCS y comenzar desde ésta posición.

En cambio, si ambas cadenas comienzan con un carácter distinto, no pueden formar parte de la LCS, por lo que ignoramos una, otra o ambas.

Observamos que, una vez decidido qué haremos con las primeras letras de las cadenas, el subproblema restante se convierte nuevamente en otro problema de la subsecuencia más larga, que se puede resolver recursivamente.

Entonces, en el caso de que los primeros caracteres difieran, podemos determinar qué subproblema da la solución correcta resolviendo ambos y obteniendo el máximo de las longitudes de las subsecuencia resultantes, quedando la función de la siguiente manera:

- Caso base: Si ambas cadenas están vacías, la longitud de la LCS será igual a cero.
- Si $A[i] = A[j]$, la longitud de la LCS va a aumentar 1 con respecto a la longitud de la subsecuencia que resulta de quitar el elemento en esa posición de ambas cadenas. Es decir, $LCS(i, j) = 1 + LCS(i-1, j-1)$.
- Si $A[i] \neq A[j]$, debemos ignorar a alguno de estos elementos, o a ambos de la subsecuencia más larga. Es decir, $LCS(i, j) = \max(LCS(i-1, j), LCS(i, j-1))$.

Para optimizar la función recursiva anterior, empleamos Programación Dinámica, guardando los resultados calculados en una matriz, y en cada caso primero consultamos ésta antes de computar el resultado. Si el resultado no se encuentra en la matriz, lo calculamos y almacenamos.

1.3. Código resultante

Utilizando el método Bottom Up, rellenamos la matriz primero con los subproblemas más simples y luego con los más complejos. La complejidad del algoritmo es $O(\text{lenA} * \text{lenB})$, cuadrática.

```
1 //Subsecuencia Común más Larga
2 #include<stdio.h>
3 #include<string.h>
4
5 int max(int a, int b){
6     if(a>b) return a;
7     else return b;
8 }
9
10 int LCS(char *a, char *b, int lena, int lenb){
11     /*Construimos una tabla de forma Botton Up*/
12     int lcs[lena+1][lenb+1];
13     for(int i = 0; i <= lena; i++){
14         for(int j = 0; j <= lenb; j++){
15             if(a[i-1] == '\0' || b[j-1] == '\0')
16                 /*Caso base, la cadena esta vacia*/
17                 lcs[i][j] = 0;
18             else if(a[i-1] == b[j-1])
19                 /*Sumamos 1 y quitamos el elemento de ambas cadenas*/
20                 lcs[i][j] = lcs[i-1][j-1] + 1;
21             else
22                 /*Debemos ignorar a uno y sólo tomar al máximo*/
23                 lcs[i][j] = max(lcs[i-1][j], lcs[i][j-1]);
24         }
25     }
26     /*En la ultima posicion del arreglo se encuentra la LCS*/
27     return lcs[lena][lenb];
28 }
29
30 int main(void){
31     char c1[10000];
32     char c2[10000];
33     scanf("%s", c1);
34     scanf("%s", c2);
35     printf("\n%d", LCS(c1, c2, strlen(c1), strlen(c2)));
36     return 0;
37 }
```

1.4. Validación del juez online


https://omegaup.com/arena/problem/Longest-Common-Subsequence/#problems

omegaUp Arena Concursos Problemas Rank Escuelas Blog Preguntas brokenerk

Entrada
La primera línea contendrá la cadena A. En la segunda línea vendrá la cadena B.

Salida
La longitud de la subsecuencia común más larga.

Ejemplo



Entrada	Salida	Descripción
AGCT AMGXTP 	3	La subsecuencia común más larga es: <i>AGT</i>

Límites

- $1 \leq |A| \leq 10^3$
- $1 \leq |B| \leq 10^3$

Fuente: Filiberto Fuentes
 Problema subido por: galloska
[Reportar contenido inapropiado en este problema.](#)
[Calificar el problema](#)

Envíos

Enviado	GUID	Estatus	Porcentaje	Lenguaje	Memoria	Tiempo	Detalles
2018-11-17 15:11:17	13c62f6b	Respuesta correcta	100.00%	c	3.54 MB	0.01 s	
2018-11-17 15:05:58	6d7c93e2	Respuesta parcialmente correcta	13.33%	c	3.57 MB	0.00 s	

2. ELIS - Fácil Subsecuencia en Aumento más Larga

2.1. Problema

Dada una lista de números A , muestra la longitud de la subsecuencia en aumento más larga. Una subsecuencia en aumento se define como un conjunto $i_0, i_1, i_2, i_3, \dots, i_k$ tal que $0 \leq i_0 < i_1 < i_2 < i_3 < \dots < i_k < N$ y $A[i_0] < A[i_1] < A[i_2] < \dots < A[i_k]$. Una subsecuencia en aumento más larga es una subsecuencia con el máximo k (longitud).

Es decir, en la lista $\{33, 11, 22, 44\}$ la subsecuencia $\{33, 44\}$ y $\{11\}$ son subsecuencias crecientes, mientras que $\{11, 22, 44\}$ es la subsecuencia que más crece.

Entrada

La primera línea contiene un número N ($1 \leq N \leq 10$), la longitud de la lista A .

Salida

Una línea que contiene la longitud de la subsecuencia creciente más larga en A .

2.2. Análisis y solución

Para calcular la subsecuencia en aumento más larga contenida en un arreglo A , notamos que a menos que A este vacío, una LIS tendrá una longitud de al menos 1.

Una LIS que termina con un elemento x tiene la propiedad de que es una LIS que usa el último elemento de $A = A[n-1]$.

Denotamos los subarreglos del arreglo A como:

$$A_0 = \{A[0]\}$$

$$A_1 = \{A[0], A[1]\}$$

\vdots

$$A_{n-1} = \{A[0], A[1], \dots, A[n-1]\}$$

Así, para cada uno de estos subarreglos vamos a determinar una subsecuencia creciente que contenga el último elemento de este subarreglo, de manera que ya no haya una subsecuencia creciente que tenga esta propiedad. Entonces, uno de estos debe ser un LIS para la secuencia original, y se obtiene hallando el máximo de ellos.

La función recursiva queda de la siguiente manera:

- Si solo hay un elemento en el arreglo, la longitud de la LIS es 1.
- Si $A[i] > A[j]$ entonces le sumamos 1 a las LIS parciales y al final hallamos el máximo, es decir: $LIS(i) = \max(LIS(j) + 1)$.

Al pasar ésta función a Programación Dinámica, creamos un arreglo en donde iremos guardando las longitudes de los LIS parciales de cada subarreglo del arreglo A , y posteriormente hallaremos el máximo de esa serie de valores, que será la longitud del LIS del arreglo completo.

2.3. Código resultante

Nuevamente utilizamos el método Bottom Up. La complejidad del arreglo esta dada como $O(n^2)$, pues a pesar de que es un único arreglo, necesitamos dos índices de recorrido en él.

```
1  #include<stdio.h>
2  #include <stdlib.h>
3
4  int max(int a, int b){
5      if(a>b) return a;
6      else return b;
7  }
8
9  int ELIS(int arreglo[], int n){
10     //Arreglo auxiliar para hallar LIS
11     int elis[n], resultado = 0;
12     elis[0] = 1;
13     //Programacion dinamica de forma Bottom Up para hallar las
14     //↪ longitudes LIS parciales
15     for(int i = 1; i < n; i++){
16         elis[i] = 1;
17         for(int j = 0; j < i; j++){
18             if(arreglo[i] > arreglo[j] && elis[i] < elis[j] + 1)
19                 elis[i] = elis[j] + 1;
20         }
21     }
22     //Encontrar el máximo de todas las soluciones obtenidas
23     for(int i = 0; i < n; i++)
24         resultado = max(resultado , elis[i]);
25     return resultado;
26 }
27
28 int main(void){
29     int num;
30     scanf("%d", &num);
31     int *A = (int*)calloc(num, sizeof(int));
32     for(int i = 0; i < num ; i++)
33         scanf("%d", &A[i]);
34     printf("\n%d", ELIS(A, num));
35     return 0;
36 }
```

2.4. Validación del juez online

[PROBLEMS](#) [STATUS](#) [RANKS](#) [DISCUSS](#) [CONTESTS](#) [PROFILE](#)

Problems / / Easy Longest Increasing Subsequence / My submissions

Not hidden submissions All submissions

brokenerk: submissions

Easy Longest Increasing Subsequence

ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
22723029		2018-11-18 00:52:44	Easy Longest Increasing Subsequence	accepted edit Ideone It	0.00	10M	C

3. KNAPSACK - El problema de la mochila

3.1. Problema

El famoso problema de la mochila. Esta empacando para unas vacaciones en el mar y solo llevará una mochila con capacidad S ($1 \leq S \leq 2000$). También tiene N ($1 \leq N \leq 2000$) elementos que puede llevar consigo a las vacaciones. Lamentablemente, no puede colocarlos todos en la mochila, por lo que tendrá que elegir. Para cada artículo se le da su tamaño y su valor. Desea maximizar el valor total de todos los artículos que va a traer. ¿Cuál es este valor total máximo?

Entrada

En la primera línea ingresar S y N . En las siguientes N líneas se ingresan dos enteros que describen cada uno de sus elementos. El primer número es el tamaño del artículo y el siguiente es el valor del artículo.

Salida

El valor total máximo de la mejor selección de elementos para su viaje.

3.2. Análisis y solución

La solución por fuerza bruta sería ir viendo el valor total de todas las posibles combinaciones de elementos cuya tamaño sea menor o igual al de la mochila y obtener la máxima de ellas, pero la complejidad del algoritmo se dispara bastante.

Para todos los posibles conjuntos de n elementos, pueden existir 2 casos para cada elemento:

1. Si decidimos tomarlo e incluirlo en la mochila, el valor máximo total será el valor máximo de una mochila de tamaño S menos el tamaño del i -ésimo elemento y un total de $n-1$ elementos para escoger (ignoramos ese elemento), más el valor del i -ésimo elemento.
2. Si decidimos no tomarlo, el valor máximo total será el valor máximo de una mochila de tamaño S y un total de $n-1$ elementos para escoger (ignoramos ese elemento).

Para tomar la decisión correcta, simplemente aplicamos un máximo entre ambas opciones, y debemos asegurarnos que el tamaño del i -ésimo elemento sea menor al de la mochila, sino no lo tomamos. La función recursiva quedaría de la siguiente manera:

- Caso base: Si no hay ningún elemento, el valor máximo es cero.
- Si hay suficiente espacio en la mochila, el valor máximo será el máximo entre las decisiones de tomar o no tomar el i -ésimo elemento.
- Si el tamaño del i -ésimo elemento es mayor al de la mochila, tomamos la decisión de ignorarlo.







Para su implementación con Programación Dinámica, creamos una matriz de n (número de elementos en total) por s (tamaño de la mochila), en donde iremos guardando los resultados de cada cálculo.

3.3. Código resultante


Utilizamos el método Bottom Up. La complejidad de ésta solución es igual a $O(n*s)$, cuadrática, pues vamos llenando y recorriendo la matriz según los casos antes mencionados.

```
1 //S = Tamaño mochila
2 //N = Items que caben en la mochila
3 #include<stdio.h>
4 #include <stdlib.h>
5
6 int max(int a, int b){
7     if(a>b) return a;
8     else return b;
9 }
10
11 int knapsack(int tam[], int valor[], int s, int n){
12     int mochila[n+1][s+1];
13     //Programacion dinamica de forma Bottom Up
14     for(int i = 0; i <= n; i++){
15         for(int j = 0; j <= s; j++){
16             if(i == 0 || j == 0)
17                 //No hay elementos
18                 mochila[i][j] = 0;
19             else if(tam[i-1] <= j)
20                 //Aqui nos enfrentamos a la decision de tomar o no
21                 //  tomar el elemento
22                 mochila[i][j] = max(valor[i-1] +
23                                     //  mochila[i-1][j-tam[i-1]], mochila[i-1][j]);
24             else
25                 //Como ya no hay espacio, optamos por no elegirlo
26                 mochila[i][j] = mochila[i-1][j];
27         }
28     } /*En la ultima posicion se encuentra el valor máximo*/
29     return mochila[n][s];
30 }
31
32 int main(void){
33     int S, N;
34     scanf("%d %d", &S, &N);
35     int *tam = (int*)calloc(N, sizeof(int));
36     int *val = (int*)calloc(N, sizeof(int));
37     for(int i = 0; i < N; i++)
38         scanf("%d %d", &tam[i], &val[i]);
39     printf("\n%d", knapsack(tam, val, S, N));
40     return 0;
41 }
```

3.4. Validación del juez online


 **PROBLEMS**
 **STATUS**
 **RANKS**
 **DISCUSS**
CONTESTS
  **PROFILE**

Problems / / The Knapsack Problem / My submissions

 Not hidden submissions All submissions

brokenerk: submissions

The Knapsack Problem

ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
22723452		2018-11-18 04:42:24	The Knapsack Problem	accepted edit Ideone It	0.00	13M	C

4. Fuentes consultadas de apoyo

[1] (2018) Longest Common Subsequences. Accessed november 2018. [Online]. Available: <https://www.ics.uci.edu/~epstein/161/960229.html>

[2] (2018) Longest increasing subsequence. Accessed november 2018. [Online]. Available: https://wcipeg.com/wiki/Longest_increasing_subsequence

[3] (2018) Problema de la mochila. Accessed november 2018. [Online]. Available: https://es.wikipedia.org/wiki/Problema_de_la_mochila