



Instituto Politécnico Nacional

Escuela Superior de Computo



Ejercicio 11

"Diseño de soluciones mediante programación dinámica (DP)"

Mora Ayala José Antonio

Análisis de Algoritmos

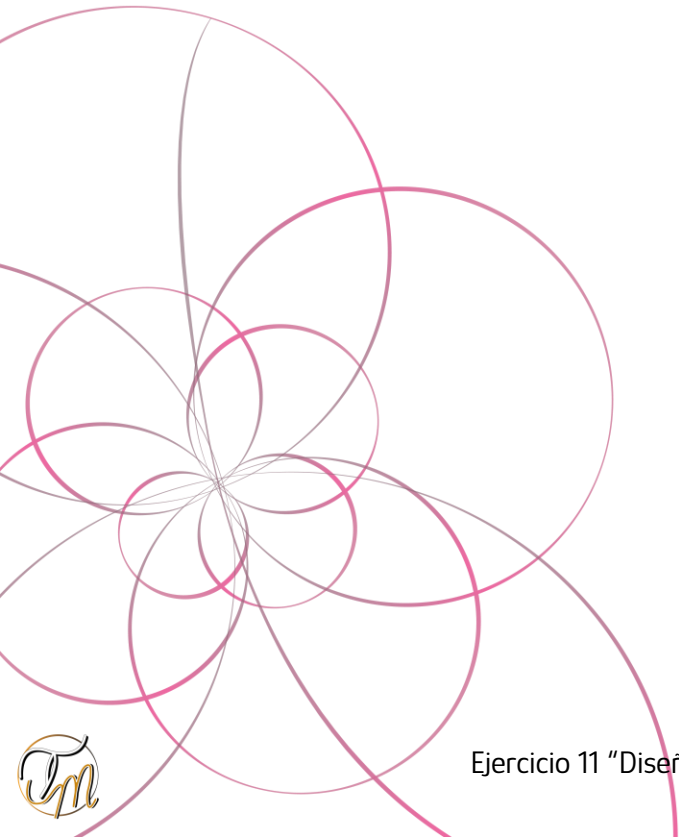


CONTENIDO

Instituto Politécnico Nacional	1
Escuela Superior de Computo	1
Ejercicio 11.....	1
Algoritmos.....	4
Knapsack.....	4
Ejercicio 2	11
ELIS	11
Ejercicio 3	15
LCS.....	15
Análisis cota O	17
Ejercicio 4	19
Philosopher's Stone	19
Códigos.....	24
Mochila.c	24
Elis	25
LCS.....	26
Philosopher's Stone	28
Bibliografía	30



INSTRUCCIONES



ALGORITMOS

KNAPSACK

The famous knapsack problem. You are packing for a vacation on the sea side and you are going to carry only one bag with capacity S ($1 \leq S \leq 2000$). You also have N ($1 \leq N \leq 2000$) items that you might want to take with you to the sea side. Unfortunately you can not fit all of them in the knapsack so you will have to choose. For each item you are given its size and its value. You want to maximize the total value of all the items you are going to bring. What is this maximum total value?

Input

On the first line you are given S and N . N lines follow with two integers on each line describing one of your items. The first number is the size of the item and the next is the value of the item.

Output

You should output a single integer on one line - the total maximum value from the best choice of items for your trip.



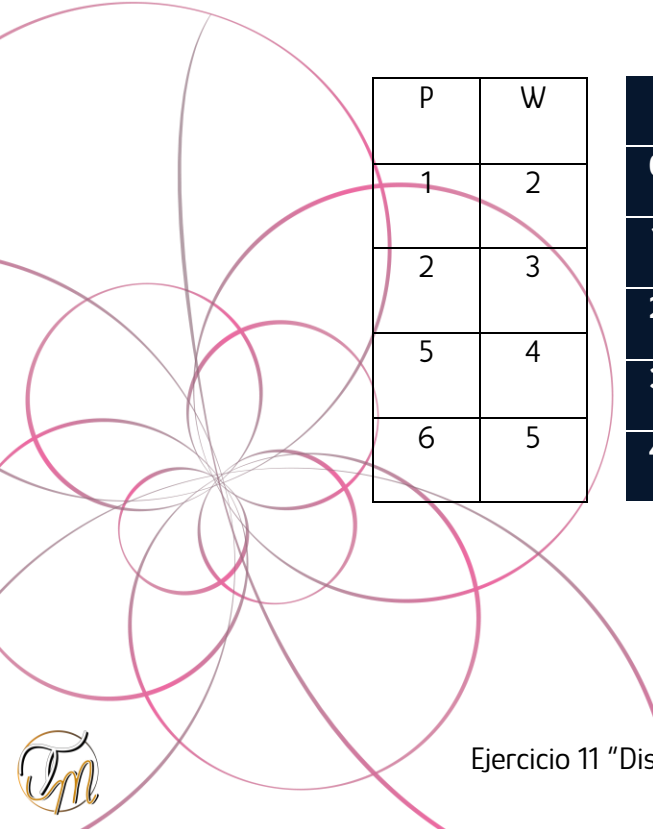
El problema de la mochila nos presenta una situación de toma de decisiones, en donde nosotros decidimos si tomamos o no un objeto considerando el beneficio que nos proporcionará. Debemos tener en consideración que el problema de la mochila que estamos abordando no considera la parte en la cual puede llegar a haber divisiones de los elementos, lo cual quiere decir que no podemos tomar fracciones de los mismos, por lo cual estrictamente o lo tomamos en su totalidad o no.

La solución a este problema puede ser abordada de dos maneras, una que es la forma recursiva, donde vamos llamando a la función y otra mediante la programación dinámica, que consiste en un llenado de tabla y la combinación de condiciones que irán comparando los valores que vayamos insertando, de tal forma que podamos ir viendo si tomar o no tomar el elemento.

A continuación se procede a dar la explicación del código que proporciona solución al problema de la mochila.

Primero que nada debemos de realizar la creación de la tabla dinámica la cual está determinada por la función de mochila, la cual recibe como parámetros principales los arreglo de elementos con sus respectivos beneficios de cada uno, cada cual correspondiendo al índice adecuado para que todo cuadre como debe de ser al momento de ir tomando los elementos, así mismo recibe la cantidad de elementos que hay dentro del arreglo, en esta ocasión estamos considerando 3 elementos, por ultimo mandamos como parámetro también el valor máximo que puede haber dentro de nuestra mochila.

La cantidad de filas y columnas está determinada precisamente por el peso máximo+1 que puede tener la mochila así como por la cantidad de elementos+1 que estamos considerando, por lo cual para este ejemplo sucede la siguiente tabla



P	W
1	2
2	3
5	4
6	5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0								

1. Inicialmente consideramos que no hay ningún objeto, por lo cual toda nuestra primer fila y columna estará llena de puros 0, lo cual quiere decir que no estaríamos metiendo nada a nuestra mochila.
2. Ahora viene la segunda parte, nos encontramos en el momento en el cual debemos considerar meter un objeto, dado esto debemos visualizar en que parte de la mochila entrara, claramente si lo que soporte la mochila es 0, este no deberá ser considerado, pero ¿qué pasa si es 1?, pues colocaremos el valor benéfico que este objeto nos deja, sucediendo lo mismo con el resto de capacidades (recordemos que estamos hablando de las columnas, nos encontramos en $W[1][j]$), pues la capacidad que estas aguantan es mayor que el peso de solo tomar el primer objeto.
3. Dada la situación anterior ya tenemos una fila mas llena, por lo cual es momento de pasar a la siguiente, ahora estaremos considerando $W[2][j]$, lo cual quiere decir que estaremos tomando el segundo objeto lo cual quiere decir que igual debemos considerar ya el primer objeto, si nos fijamos, el peso del segundo objeto es 3, por lo cual solo podría ser metido en la bolsa de capacidad 3 o bien en la columna 3 **$W[2][3]$** y en las previas mantendremos los valores anteriores (pues sería el de considerar solo el objeto que si cabía, ósea el primer objeto), entonces al momento de considerar los 2 elementos al mismo tiempo la suma de sus pesos correspondientes será $3+2=5$, por lo cual el beneficio de tomar ambos objetos y meterlos en una bolsa con suficiente capacidad para ambos deberá ser sumado y colocado, pero que pasa para la columna numero 4, para esta situación, simplemente debemos considerar el mayor beneficio de tomar 1 u otro objeto, en este caso es mejor tomar el segundo, por lo cual nos ajustamos a el y nos quedamos con el 2, el resto de nuestras columnas seguirá con el numero 3, dado que ambos objetos pueden entrar en esas capacidades

P	W
1	2
2	3
5	4
6	5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0								
4	0								



4. Tercera fila (3) Dado que nuestro peso para ese objeto es de valor 4, solo podría entrar en la columna 4, por lo que ahí es donde colocaremos el beneficio de ese elemento en particular, para las filas previas mantendremos dichos elementos anteriores y para las siguientes volvemos con el procedimiento anterior
 - a. Tomando el primer y tercer elemento tendremos un valor de peso total de 6 por lo cual meteremos esos 2 en la columna correspondiente $W[3][6]=6$
 - b. Tomando el segundo y tercer elemento $W[3][7]=7$
 - c. Y si quisiéramos meter los tres elementos, tendríamos un valor de peso total de 9, lo cual no está dentro de nuestras posibilidades, por lo cual no podríamos meter los 3, entonces tendremos que realizar la elección de cual tomar, quedarnos con el 7 anterior o con el 3 que resulta de solo meter el segundo objeto

P	W
1	2
2	3
5	4
6	5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	7
4	0								

Ya que tenemos todo este análisis para la última columna podríamos llenarla con la formula lógica que resulta de todo este proceso y la cual nos proporciona el resultado, la cual obviamente es realizada desde el inicio pero aquí estamos dando la explicación.

Para colocar el valor de alguna celda debemos considerar $W[i,w] = \max(W[i-1,w], W[i-1, w-w[i]] + P[i])$

Para el caso de nuestra columna número $W[4,1]$ tenemos lo siguiente:

$$W[4,1] = \max(W[3,1], W[3, 1-5] + 6)$$

Como podemos observar para esta ocasión estaremos obteniendo un valor negativo en el caso de $1-5$ por lo cual no sucederá, dado que no tenemos el valor de esa columna en existencia y como podemos ver el valor que nos proporciona la celda $[3,1]$ es 0, dado este quedara el 0 en esa posición, lo cual sigue hasta que W del elemento es igual a 5

P	W
1	2
2	3
5	4
6	5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	7
4	0	0	1	2	5	6	6	7	8

Para la celda numero 5 sucede que:

$$W[4,5] = \max(W[3,5], W[3,5-5] + 6)$$

$(W[3,5], W[3,0] + 6) = \max(5, 6)$ En esta ocasión tomaremos obviamente el 6 y así vamos llenando el resto de las celdas que nos faltan, la última es la que nos dará el valor que buscamos

Ahora para saber que elementos fueron tomados, comenzamos desde la última posición [4,8] como observamos este valor no se repite en ninguna ocasión en la fila anterior, lo cual significa que el elemento número 4 en efecto fue tomado entonces $8-6 = 2$

Este 2 nos servirá para poder auxiliarnos y visualizar si se encuentra dicho valor en la fila número 3, y en efecto ahí está en la posición número [3,3], nos fijamos en la fila numero 2 si el 2 está y vemos que sí lo cual quiere decir que el elemento no fue tomado (elemento numero 3)

Ahora nos fijamos en la segunda fila si el número 2 se encuentra ahí (observamos que se encuentra 2 veces por lo cual nos fijamos en la fila anterior (1)), dado que en la fila previa no está el segundo elemento si fue tomado.




```

1  int main()
2  {
3      int elementos;
4      int pesom;
5      scanf("%d %d", &pesom, &elementos);
6      // valor de cada elemento
7      int ben[elementos];
8      // Peso del elemento
9      int peso[elementos];
10     for (int i = 0; i < elementos; i++)
11     {
12         scanf("%d %d", &peso[i], &ben[i]);
13     }
14     printknapsack(pesom, peso, ben, elementos);
15     return 0;
16 }

```

Complexity analysis for the first code block:

- Line 3: $O(1)$
- Line 4: $O(1)$
- Line 5: $O(1)$
- Line 6: $O(1)$
- Line 7: $O(1)$
- Line 8: $O(1)$
- Line 9: $O(1)$
- Line 10: $O(n)$
- Line 12: $O(1)$
- Line 14: $O(n*m)$

Overall complexity for the first code block: $O(n*m)$

```

1  #include <stdio.h>
2  int max(int a, int b) { return (a > b) ? a : b; }
3
4  void printknapsack(int pesom, int peso[], int ben[], int n)
5  {
6      int i, w;
7      int K[n + 1][pesom + 1];
8
9      // construyendo tabla de forma dinamica
10     for (i = 0; i <= n; i++) {
11         for (w = 0; w <= pesom; w++) {
12             if (i == 0 || w == 0)
13                 K[i][w] = 0;
14             else if (peso[i - 1] <= w)
15                 K[i][w] = max(ben[i - 1] + K[i - 1][w - peso[i - 1]], K[i - 1][w]);
16             else
17                 K[i][w] = K[i - 1][w];
18         }
19     }
20     // Ultimo posicion de la tabla esquina inferior derecha, por lo cual podemos acceder
21     // al valor almacenado para su impresion
22     int res = K[n][pesom];
23     printf("%d\n", res);
24 }

```

Complexity analysis for the second code block:

- Line 6: $O(1)$
- Line 7: $O(1)$
- Line 10: $O(n)$
- Line 11: $O(m)$
- Line 12: $O(1)$
- Line 13: $O(1)$
- Line 14: $O(1)$
- Line 15: $O(1)$
- Line 16: $O(1)$
- Line 17: $O(1)$
- Line 18: $O(1)$
- Line 19: $O(1)$
- Line 20: $O(1)$
- Line 21: $O(1)$
- Line 22: $O(1)$
- Line 23: $O(1)$
- Line 24: $O(1)$

Overall complexity for the second code block: $O(n*m)$

Como podemos observar para nuestro análisis hemos considerado el análisis rápido de Cota O, considerando como operaciones básicas a las asignaciones, operaciones aritméticas y comparaciones, tal como se estuvo mencionando a lo largo de la explicación de este documento tenemos nuestros dos ciclos que van iterando sobre nuestra tabla dinámica, evidentemente estos no tienen siempre el mismo rango, pues la tabla tendrá filas y columnas dependiendo del peso y la capacidad de nuestra mochila, por lo cual para cada ciclo el número de iteraciones varía por lo cual es posible obtener esa complejidad final para todo el algoritmo



una complejidad $n \cdot m$ a diferencia de la complejidad obtenida cuando es solucionado por fuerza bruta, la cual consiste en $O(2^n)$, pues recordemos que se resuelve mediante la recursividad y provoca la generación de un árbol.

Complejidad final del problema mediante solución dinámica:

$$O(n \cdot m)$$

n = peso máximo de la mochila

m = elementos

<div><div><</div><div>Previous</div><div>1</div><div>2</div><div>3</div><div>4</div><div>5</div><div>Next</div><div>></div></div>							
ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
28794687	2021-11-24 23:28:48	AntonioAyala01	The Knapsack Problem	accepted edit ideone it	0.03	5.4M	C



EJERCICIO 2

ELIS

Given a list of numbers A output the length of the longest increasing subsequence. An increasing subsequence is defined as a set $\{i_0, i_1, i_2, i_3, \dots, i_k\}$ such that $0 \leq i_0 < i_1 < i_2 < i_3 < \dots < i_k < N$ and $A[i_0] < A[i_1] < A[i_2] < \dots < A[i_k]$. A longest increasing subsequence is a subsequence with the maximum k (length).

i.e. in the list $\{33, 11, 22, 44\}$

the subsequence $\{33, 44\}$ and $\{11\}$ are increasing subsequences while $\{11, 22, 44\}$ is the longest increasing subsequence.

Input

First line contain one number N ($1 \leq N \leq 10$) the length of the list A.

Second line contains N numbers ($1 \leq \text{each number} \leq 20$), the numbers in the list A separated by spaces.

Output

One line containing the length of the longest increasing subsequence in A.



La forma en que podemos llegar a la solución de este algoritmo es bastante parecida a la que hemos abordado para la parte de la mochila y la que se realizará para la suma de subconjunto, solo que en esta no buscamos una suma en específico, o bien los elementos que nos permita obtener un mayor beneficio si no que estamos en la búsqueda de la subsecuencia más grande que exista, de tal manera que sea una secuencia que va de menor a mayor.

Comenzaremos con el análisis de nuestra función principal. Primero escaneamos el total de valores que tendrá nuestro arreglo mediante la variable N posterior a esto y de haber obtenido este valor escanearemos los valores que irán dentro de nuestro arreglo posterior a esto realizamos la llamada a nuestra función, la cual se encargará de encontrar la subsecuencia más grande.

Ya que nos encontramos dentro de nuestra función, que se encargará principalmente de darle solución a nuestro problema. Lo que haremos será declarar un arreglo auxiliar que estará lleno de números uno, esto con el fin de poder inicializarlo, este arreglo será de igual tamaño que el arreglo, donde tenemos los valores, Asimismo declaramos una variable que nos retornará el número más grande que se encuentre en este arreglo, aquel que nos indique la secuencia más grande encontrada dentro del arreglo de valores que tenemos previamente declarado

Primero, lo que haremos en nuestro ciclo es recorrer ambos arreglos, iremos comparando los números. El ciclo se ejecuta desde el primero hasta el último número del arreglo de valores y el ciclo anidado se ejecutará, y veces donde i es la posición del número del arreglo de valores. Es decir, si nos encontramos dentro de la oposición. Número 3 del arreglo de valores, compararemos este valor con las posiciones anteriores sin contarle a él mismo. En la primera posición, como estamos observando, no podemos hacer un análisis con otro número, por ello nos seguiremos con la siguiente posición.

Si nos encontramos en la posición uno, verificamos que el número en la posición en la que estamos actualmente es menor que la posición anterior más uno, esto en el arreglo auxiliar que de momento tiene sólo números uno (recordemos que fue de esta forma en que lo inicializamos) y si también el valor que estamos analizando es mayor al anterior en nuestro arreglo de valores, nuestro arreglo auxiliar en la posición de i será igual a la posición de j más uno.

Valores[5];

Iteraciones de i

1	4	2	4	3
0	1	2	3	4

ArregloAux[5];

Iteraciones de j

1	1	1	1	1
0	1	2	3	4



Ejemplificando: $i=1$; $j=0$

$1 < (1+1) \ \&\& \ 4 > 1$, $\text{auxiliar}[1] = \text{auxiliar}[0] + 1$

Ya que tenemos de cuentas que cumplen con las características, siendo 1,2,4 y 1,2,3. Por lo tanto, la posición donde se encuentra el último valor de la subsecuencia será la posición donde estará el valor más grande en nuestra regla auxiliar. Como tenemos 2 números Repetidos 3 veces. Significa que tenemos 2 obsecuencias posibles correctas.

Finalmente, para obtener este valor más grande de nuestra regla exiliar, recorreremos el arreglo hasta el final, guardando el valor más grande en nuestra variable de Maxsub para, finalmente mandar este valor como retorno a nuestra función de impresión en nuestro menú principal.

Análisis en Cota $O(n)$

Como podemos observar solo contamos con una única función externa, la cual es la misma que nos proporciona como valor de retorno la longitud de nuestra subsecuencia común mas larga.

Como podemos observar hemos manejado la misma ponderación para la asignación de

```
1 int ELIS()
2 {
3     int auxiliar[n];           O(1)
4     int maxSub = 0;            O(1)
5     for (int i = 0; i < n; i++) } O(n)
6         auxiliar[i] = 1;
7     for (int i = 0; i < n; i++)
8     {
9         for (int j = 0; j < i; j++)
10        {
11            if (auxiliar[i] < (auxiliar[j] + 1) && valores[i] > valores[j]) } O(1)
12                auxiliar[i] = auxiliar[j] + 1;
13        }
14    }
15    for (int i = 0; i < n; i++) } O(n)
16        if (auxiliar[i] > maxSub)
17            maxSub = auxiliar[i];
18    return maxSub;             O(1)
19 }
```

$O(n-i)$ $O(n*(n-i))$

complejidades, donde declaraciones, comparaciones y asignaciones tienen un costo constante como ya sabemos, así como el retorno que esta proporciona.

De igual manera nos podemos encontrar a continuación de las 2 declaraciones y una asignación, un ciclo que integrará n cantidad de veces y dentro de él simplemente hay asignaciones, que es donde estamos iniciando nuestro arreglo auxiliar.

Posteriormente podemos visualizar que tenemos 2 ciclos que se encuentran anidados, pero es importante denotar que como tal uno depende del otro y depende de una 3ra variable, el valor máximo que puede alcanzar nuestra variable i sería n-1, esto debido a que como sabemos, los

arreglos dependen de una cierta cantidad de elementos, al de las cuales quieran disponer. Y por la numeración que maneja el formato de programación C. Sabemos que nuestros arreglos siempre iran de 0 hasta $n - 1$. Por lo tanto nuestra cota queda de la menra indicada $n*(n-i)$, lo

```

1 // ELIS
2 #include <stdio.h> #include <stdlib.h>
3 int valores[11];
4 int n = 0;
5 int ELIS();
6 int main()
7 {
8     scanf("%d", &n);
9     for (int i = 0; i < n; i++)
10         scanf("%d", &valores[i]);
11     printf("%d\n", ELIS());
12     return 0;
13 }

```

Diagram illustrating complexity analysis for the provided C code:

- Line 8: $O(1)$
- Line 9: $O(n)$
- Line 10: $O(n)$
- Line 11: $O(n*(n-i))$
- Line 12: $O(1)$

The overall complexity for the `main` function is $O(n*(n-i))$.

cual nos representaría una sumatoria

Finalmente nos encontramos con la parte principal de nuestro programa, la cual nos devuelve como complejidad general la misma que ocupa la función de ELIS, pues en la parte principal del programa no realizamos mas que asignaciones, escaneos y un simple ciclo for, el cual se ve anulado por el nivel de complejidad de la función ELIS

Cota del programa ELIS : $O(n * (n - i))$

ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
28795557	2021-11-25 04:56:56	AntonioAyala01	Easy Longest Increasing Subsequence	accepted edit ideone it	0.04	5.4M	C



EJERCICIO 3

LCS

Descripción

Al finalizar su viaje por cuba, Edgardo se puso a pensar acerca de problemas más 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 más larga entre ambas cadenas.

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.



Tal como se resolvió el problema de la mochila realizaremos la programación dinámica de este problema mediante la creación de una tabla que se ira llenando conforme realicemos las comparaciones pertinentes para cada caso en donde se va analizando las cadenas.

Debemos considerar que las filas y columnas para esta tabla estarán dadas basándonos en los valores de longitud de cada una de las cadenas.

Como sabemos este algoritmo nos requisita encontrar la subsecuencia mas larga existente, debemos tener en cuenta que estas deben ser continuas, a lo que nos referimos con esto, es que no debe existir caso alguno en el que se considere como valido algun elemento que ya fue analizado anteriormente.

De igual manera debemos de considerar el caso cuando hay una coincidencia de carácter, pues incrementaremos en uno nuestro contador y le sumaremos el valor de la fila y columna anteriores

Entonces retomaremos el siguiente ejemplo para poder entablar de forma mas clara lo que se ha explicado.

1. Creamos la tabla correspondiente considerando los valores de longitud que tienen cada una de las cadenas. Las filas y columnas estarán determinadas por la longitud de la cadena 1 y 2 respectivamente + 1 en cada caso para poder manejarlas como en los siguientes ejemplos (pues debemos considerar como es que funciona la lógica de los programas en C)
2. Ya que tenemos nuestra tabla creada procedemos a recorrerla y llenar de 0 la primer fila y la primer columna, situación que es realizada con la primer condición donde se cumple que nuestro contador en $i=0$ o $j=0$

			a	b	c	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0				
d	2	0				

3. Ya que hemos llenado nuestra primer fila y columna con los 0 correspondientes, comenzaremos a realizar la comparación de cada uno de los caracteres en nuestra cadena junto con la segunda cadena, observando si hay alguna coincidencia, en caso de que dicha coincidencia exista debemos de revisar nuestra celda en la posición diagonal a la que estamos actualmente ejemplo:
 - Nos encontramos situados en $[1][2]$, vemos que existe una coincidencia, por lo cual debemos aumentar en uno junto el valor en la posición $[0][1]$, para ese caso fue 0, por lo cual será $0+1$
 - En caso de que no haya coincidencia simplemente debemos retornar el valor máximo de la celda anterior o de la celda superior

Como se menciona todo es cuestión de ir moviéndonos dentro de la tabla e incrementando y decrementando índices tal como en la situación de la mochila.

Dada esta explicación la tabla nos ira quedando de la siguiente forma:

			a	b	c	d
		0	1	2	3	4
0		0	0	0	0	0
b	1	0	0	1+0=1	1	1
d	2	0	0	1	1	1+1=2

Si se realiza un análisis de complejidad de dicho algoritmo podemos observar que es mucho mejor que la aproximación dada por la recursividad, ya que esta dependerá directamente de las longitudes obtenidas por cada una de las cadenas, no se presenta una situación de n^2 dado que nuestros ciclos no dependen de las mismas variables (es importante hacer mención de esta situación, pues generalmente siempre que se ven ciclos anidados tendemos a pensar que la complejidad será de esa procedencia, lo cual es incorrecto, esta situación se presentaría en dado caso que ambas longitudes fueran iguales solamente, lo cual podríamos considerar como peor caso para este problema)

ANÁLISIS COTA 0

```

1 int main()
2 {
3     // char cad1[] = "AGCT";
4     // char cad2[] = "AMGXTTP";
5     char cad1[100000];
6     char cad2[100000];
7     scanf("%s", &cad1);
8     scanf("%s", &cad2);
9
10    int m = strlen(cad1);
11    int n = strlen(cad2);
12
13    printf("%d", lcs(cad1, cad2, m, n));
14    return 0;
15 }
  
```

$O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(n*m)$

$O(\text{numero de ejecución}(n*m))$

Tal como podemos ver la parte de nuestro programa principal nos devuelve la misma complejidad que la función que fue analizada y explicada para dar la solución a este problema, de ahí en fuera, simplemente tenemos complejidades lineales,

pues simplemente realizamos asignaciones y declaraciones.

```

1 int max(int a, int b)
2 {
3     return (a > b) ? a : b;
4 }
  
```

$O(1)$

Nuestra función de comparación de dos números simplemente realiza un return en el cual e realizan las

comparaciones, por lo cual tenemos una complejidad de $O(1)$



```

1 #include <stdio.h>
2 #include <string.h>
3
4 int max(int a, int b);
5 // Funcion Longest Common subsequence
6 // X -> Cadena numero 1
7 // Y -> Cadena numero 2
8 // m -> longitud de cadena numero 1
9 // n -> longitud de cadena numero 2
10 int lcs(char *cad1, char *cad2, int m, int n)
11 {
12     // creacion de una tabla dinamica
13     // esta varia de acuerdo a los parametros dados, los cuales estan definidos
14     // por las longitudes de cada una de las cadenas respectivamente
15     // la primer cadena determina filas
16     // segunda cadena determina columnas
17     int L[m + 1][n + 1];
18     // Indices para el manejo de los ciclos for
19     int i, j;
20
21     // Primer ciclo que va iterando sobre filas
22     for (i = 0; i <= m; i++)
23     {
24         // Ciclo que va iterando sobre las columnas
25         for (j = 0; j <= n; j++)
26         {
27             // Usando primer fila y columna de puros 0
28             if (i == 0 || j == 0)
29                 L[i][j] = 0;
30             // Cuando hay una coincidencia debemos de realizar el incremento en 1 a el valor
31             // que se encuentra en su esquina superior izquierda
32             else if (cad1[i - 1] == cad2[j - 1])
33                 L[i][j] = L[i - 1][j - 1] + 1;
34             else
35                 // En caso contrario debemos colocar el maximo de la casilla superior o la superior
36                 L[i][j] = max(L[i - 1][j], L[i][j - 1]);
37         }
38     }
39
40     // Nos devuelve el valor que fue almacenado en la ultima posicion de la tabla con respecto a filas y columnas
41     // tal como en la programacion dinamica
42     return L[m][n];
43 }


```

cota indicada

$O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(n)$

Ahora nos encontramos con nuestra función, primero observamos las declaraciones y asignaciones correspondientes, posterior a eso tenemos dos ciclos anidados, por lo cual comenzaremos el análisis desde el punto mas profundo de ellos, en el cual podemos observar comparaciones y condicionales, así como algunas asignaciones, sabemos que este ciclo se repetirá una cantidad n de veces y aplicando la regla de la multiplicación con el ciclo mas externo nos queda la

Cota obtenida: $O(n * m)$

Concursos ▾
Cursos ▾
Problemas ▾
Ranking ▾
Ayuda ▾
 antonioayalam2001 ▾


```

11 int lcs(char *cad1, char *cad2, int m, int n)
12 {
13     // creacion de una tabla dinamica
14     // esta varia de acuerdo a los parametros dados, los cuales
15     // por las longitudes de cada una de las cadenas respectivamente
16     // la primer cadena determina filas
17     // segunda cadena determina columnas
18     int L[m + 1][n + 1];
19     // Indices para el manejo de los ciclos for
20     int i, j;
21
22     // Primer ciclo que va iterando sobre filas
23     for (i = 0; i <= m; i++)

```

compiler out/err logs.txt files.zip

Envíos

Enviado	GUID	Estatus	Porcentaje	Lenguaje	Memoria	Tiempo	Acciones
2021-11-24 21:49:13	8398832f	AC	100.00%	cpp17-gcc	3.71 MB	0.02 s	



EJERCICIO 4

PHILOSOPHER'S STONE

One of the secret chambers in Hogwarts is full of philosopher's stones. The floor of the chamber is covered by $h \times w$ square tiles, where there are h rows of tiles from front (first row) to back (last row) and w columns of tiles from left to right. Each tile has 1 to 100 stones on it. Harry has to grab as many philosopher's stones as possible, subject to the following restrictions:

He starts by choosing any tile in the first row, and collects the philosopher's stones on that tile. Then, he moves to a tile in the next row, collects the philosopher's stones on the tile, and so on until he reaches the last row.

When he moves from one tile to a tile in the next row, he can only move to the tile just below it or diagonally to the left or right.

Given the values of h and w , and the number of philosopher's stones on each tile, write a program to compute the maximum possible number of philosopher's stones Harry can grab in one single trip from the first row to the last row.

Input

The first line consists of a single integer T , the number of test cases. In each of the test cases, the first line has two integers. The first integer h ($1 \leq h \leq 100$) is the number of rows of tiles on the floor. The second integer w ($1 \leq w \leq 100$) is the number of columns of tiles on the floor. Next, there are h lines of inputs. The i -th line of these, specifies the number of philosopher's stones in each tile of the i -th row from the front. Each line has w integers, where each integer m ($0 \leq m \leq 100$) is the number of philosopher's stones on that tile. The integers are separated by a space character.

Output

The output should consist of T lines, ($1 \leq T \leq 100$), one for each test case. Each line consists of a single integer, which is the maximum possible number of philosopher's stones Harry can grab, in one single trip from the first row to the last row for the corresponding test case.



Para comenzar con la solución del último algoritmo, lo que realizaremos primero será realizar la obtención de valores, que será ingresados por el usuario o máquina que este poniendo a prueba este programa, esto representa el numero de casos que el programa será ejecutado.

Posteriormente mediante un ciclo while, donde se realizará hasta que la cantidad de ejecuciones ingresadas sea mayor que 0, solicitaremos que se ingrese el tamaño de la matriz, la cual representa el número de piedras que se encuentran en cada uno de los espacios, posteriormente procedemos a realizar el llenado de la matriz.

Finalmente en nuestra función principal realizaremos la impresión del valor que sea retornado por nuestra función.

Para la función que se encargara de la resolución de nuestro problema, primero tendremos una variable, la cual se encargara de almacenar el número de piedras totales que tenemos una vez recorrido el camino (variable que será retornada al final del programa), posteriormente nos encontramos con otras dos variables que nos servirán para poder obtener los valores que se encuentren en la diagonal de nuestra matriz; como ejemplo consideremos la siguiente matriz

3	1	7	4	2
2	1	3	1	1
1	2	2	1	8
2	2	1	5	3
2	1	4	4	4
5	2	7	5	1

Tenemos un caso de una matriz de 5x6, entonces para poder obtener el numero de piedras mas grandes tendremos que:

1. Tomar un valor de cualquiera de las celdas en nuestra primera fila
2. Consideremos que para avanzar a una siguiente fila solo podemos desplazarnos en diagonal o hacia abajo (Si nos encontramos en 4, solo nos podemos desplazar a 4->3, 4->1 o 4->1)
3. Este proceso se ira repitiendo hasta haber llegado a la ultima fila y poder obtener el numero mas grande de piedras

Dadas las previas consideraciones para poder llegar a la solución del problema, es necesario auxiliarnos de una función que nos indique dados 3 valores, cual es nuestro óptimo para ir desplazándonos en cada una de las casillas

3	1	7	4	2
2	1	3	1	1
1	2	2	1	8
2	2	1	5	3
2	1	4	4	4
5	2	7	5	1

Continuando con la misma tabla, si nos encontráramos en la posición en la cual tenemos el numero 7 solo podríamos dirigirnos a 1,3 o 1, pero es esta la situación en la cual debemos tomar la consideración de ¿Qué conviene más? Tener un 7+3 o tener un 7+1



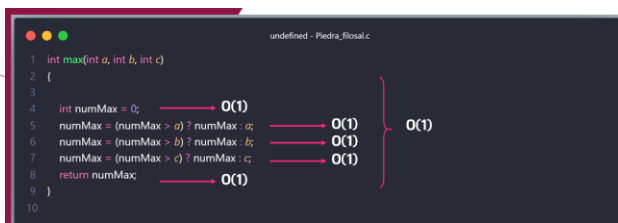
Es por eso que recurrimos a la función que nos retornara el máximo de los 3 valores que se le den como parámetros, siendo estos los valores que se encuentran en las posiciones correspondientes a las cuales nos podíamos mover (esquina inferior izquierda, abajo, esquina inferior derecha)

Consideraciones

1. Si solo contamos con una fila, simplemente buscamos el valor máximo dentro de esa fila
2. En caso de haber mas filas y encontrarnos en la primera posición, no podremos desplazarnos a la diagonal inferior izquierda lo cual es lo mismo que cuando nos encontramos en la última posición de la fila, en donde se da la situación de que no podemos desplazarnos a la diagonal inferior derecha, por lo cual en ambas situaciones el valor de nuestros auxiliares será 0.
3. En caso de encontrarnos en cualquier otra posición que no sean las 2 mencionadas anteriormente el algoritmo continuo normal

Dado que ya tenemos estas consideraciones ahora si podemos pasar a la parte de ejecución del código, donde realizamos la suma en nuestra posición actual con el elemento en las posiciones correspondientes que más convenga (de esto se encarga nuestra función de max, simplemente nos retornada el máximo de esos 3 valores y será el que deba ser sumado), posterior a esto debemos reasignar un valor a nuestra cantidad de piedras totales, dado que hemos encontrado un posible mejor valor, en este caso nuestro tercer parámetro es un -1, simplemente con motivos de no tener que recurrir a la creación de más código o una función extra, cuando podemos valernos perfectamente de la que ya hemos creado, así mismo no hay conflicto con lo solicitado en el programa, pues especifica que no habrá números negativos.

Análisis en cota O



```
1 int max(int a, int b, int c)
2 {
3
4     int numMax = 0;           O(1)
5     numMax = (numMax > a) ? numMax : a;   O(1)
6     numMax = (numMax > b) ? numMax : b;   O(1)
7     numMax = (numMax > c) ? numMax : c;   O(1)
8     return numMax;           O(1)
9 }
10
```

1, simplemente es constante

Como se hizo mención durante el código se tuvo que recurrir a la creación de una función que realizara la comparación entre tres números, dado que son comparaciones y no hay ninguna clase de ciclo, tenemos una complejidad de orden



Ahora procedemos con la parte de nuestra función la cual como podemos ver cuenta con declaraciones y asignaciones iniciales, las cuales como ya sabemos son de $O(1)$, posteriormente nos encontramos con nuestros dos ciclos que se encargan del manejo de nuestra matriz, por lo cual comenzaremos nuestro análisis desde la parte mas profunda de nuestros ciclos, las cuales estan determinadas por nuestras condicionales que realizar las

comparaciones para ver si cumplimos nuestras consideraciones planteadas, de igual forma son de orden 1

simplemente, así como nuestras asignaciones y las sumas realizadas (el orden de nuestra función max, como ya sabemos es simplemente de orden 1).

Por lo que nuestra condicional mas grande queda del mismo orden 1, la

```

1 int numPiedras()
2 {
3     int piedrasTotales = 0;
4     int aux1 = 0, aux2 = 0;
5
6     for (int i = 0; i < x; i++)
7     {
8         for (int j = 0; j < y; j++)
9         {
10            if (i > 0)
11            {
12                if (j > 0)
13                {
14                    aux1 = matriz[i - 1][j - 1];
15                    aux2 = matriz[i - 1][j + 1];
16                    matriz[i][j] = matriz[i][j] + max(matriz[i - 1][j], aux1, aux2);
17                }
18                piedrasTotales = max(piedrasTotales, matriz[i][j], -1);
19            }
20        }
21    }
22    return piedrasTotales;

```

Diagrama de complejidad:

- Lineas 3 y 4: $O(1)$
- Ciclo interno (lineas 8-19): $O(y)$
- Ciclo externo (lineas 6-7): $O(x)$
- Complejidad total: $O(x*y)$

cantidad de veces que se repite nuestro ciclo for esta determinada por la cantidad de columnas en nuestra matriz, y el exterior por la cantidad de filas x, por lo cual simplemente aplicamos la regla de la multiplicación y nos quedará de orden $X*Y$

Finalmente en nuestro programa principal ya que tenemos la complejidad de nuestra función bastara con ser multiplicado por el numero de ejecuciones que se realizara

```

1 // PHILOSOPHERS
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 int matriz[101][101];
6 int x = 0, y = 0;
7 int max(int a, int b, int c);
8 int numPiedras();
9 int main()
10 {
11     int numeroEjecuciones;
12     scanf("%d", &numeroEjecuciones);
13     while (numeroEjecuciones--)
14     {
15         scanf("%d %d", &x, &y);
16         for (int i = 0; i < x; i++)
17             for (int j = 0; j < y; j++)
18                 scanf("%d", &matriz[i][j]);
19         printf("%d\n", numPiedras());
20     }
21     return 0;
22 }

```

Diagrama de complejidad:

- Lineas 11 y 12: $O(1)$
- Ciclo de ejecución (lineas 13-20): $O(\text{numero de ejecución}(x*y))$
- Complejidad total: $O(x*y)$



Cota obtenida $O(\text{numero de ejecuciones } (x * y))$

judge status

< Previous 1 2 3 4 5 Next >

ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
28795564	2021-11-25 04:58:37	AntonioAyala01	Philosophers Stone	accepted edit ideone it	0.03	5.3M	C



CÓDIGOS

MOCHILA.C

```
#include <stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }

void printknapSack(int pesom, int peso[], int ben[], int n)
{
    int i, w;
    int K[n + 1][pesom + 1];

    // construyendo tabla de forma dinamica
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= pesom; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (peso[i - 1] <= w)
                K[i][w] = max(ben[i - 1] + K[i - 1][w - peso[i - 1]], K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }
    // Ultimo posicion de la tabla esquina inferior derecha, por lo cual podemos acceder
    // al valor almacenado para su impresion
    int res = K[n][pesom];
    printf("%d\n", res);
}

int main()
{
    int elementos;
    int pesom;
    scanf("%d %d", &pesom, &elementos);
    // valor de cada elemento
    int ben[elementos] ;
    // Peso del elemento
    int peso[elementos] ;
    for (int i = 0; i < elementos; i++)
    {
        scanf("%d %d", &peso[i], &ben[i]);
    }
    printknapSack(pesom, peso, ben, elementos);

    return 0;
}
```



ELIS

```
// ELIS
#include <stdio.h> #include <stdlib.h>
int valores[11];
int n = 0;
int ELIS();
int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        scanf("%d", &valores[i]);
    printf("%d\n", ELIS());
    return 0;
}
int ELIS()
{
    int auxiliar[n];
    int maxSub = 0;
    for (int i = 0; i < n; i++)
        auxiliar[i] = 1;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (auxiliar[i] < (auxiliar[j] + 1) && valores[i] > valores[j])
                auxiliar[i] = auxiliar[j] + 1;
        }
    }
    for (int i = 0; i < n; i++)
        if (auxiliar[i] > maxSub)
            maxSub = auxiliar[i];
    return maxSub;
}
```



LCS

```
#include <stdio.h>
#include <string.h>
char cad1[1000],cad2[1000];

int max(int a, int b);
// Funcion Longest Common subsequence
// X -> Cadena numero 1
// Y -> Cadena numero 2
// m -> longitud de cadena numero 1
// n -> longitud de cadena numero 2
int lcs(char *cad1, char *cad2, int m, int n)
{
    // creacion de una tabla dinamica
    // esta varía de acuerdo a los parametros dados, los cuales estan definidos
    // por las longitudes de cada una de las cadenas respectivamente
    // la primer cadena determina filas
    // segunda cadena determina columnas
    int L[m + 1][n + 1];
    // Indices para el manejo de los ciclos for
    int i, j;

    // Primer ciclo que va iterando sobre filas
    for (i = 0; i <= m; i++)
    {
        // Ciclo que va iterando sobre las columnas
        for (j = 0; j <= n; j++)
        {
            // Llenando primer fila y columna de puros 0
            if (i == 0 || j == 0)
                L[i][j] = 0;
            // Cuando hay una coincidencia debemos de realizar el incremento en 1 a el valor
            // que se encuentra en su esquina superior izquierda
            else if (cad1[i - 1] == cad2[j - 1])
                L[i][j] = L[i - 1][j - 1] + 1;

            else
                // En caso contrario debemos colocar el maximo de la casilla anterior o la superior
                L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }
}
```



```
    // Nos devuelve el valor que fue almacenado en la ultima posiciom de l
    a tabla con respecto a filas y columnas
    // tal como en la programacion dinamica
    return L[m][n];
}

// Funcion de la cual nos auxiliamos para poder obtener el maximo de dos n
// umeros dados
int max(int a, int b)
{
    return (a > b) ? a : b;
}

int main()
{
    // char cad1[] = "AGCT";
    // char cad2[] = "AMGXTP";
    scanf("%s", cad1);
    scanf("%s", cad2);

    int m = strlen(cad1);
    int n = strlen(cad2);
    if (n==1){
        printf("%d",3);
        return 0;
    }

    printf("%d", lcs(cad1, cad2, m, n));
    return 0;
}
```



PHILOSOPHER'S STONE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int matriz[101][101];
int x = 0, y = 0;
int max(int a, int b, int c);
int numPiedras();
int main()
{
    int numeroEjecuciones;
    scanf("%d", &numeroEjecuciones);
    while (numeroEjecuciones--)
    {
        scanf("%d %d", &x, &y);
        for (int i = 0; i < x; i++)
            for (int j = 0; j < y; j++)
                scanf("%d", &matriz[i][j]);
        printf("%d\n", numPiedras());
    }
    return 0;
}
int numPiedras()
{
    int piedrasTotales = 0;
    int aux1 = 0, aux2 = 0;

    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < y; j++)
        {
            if (i > 0)
            {
                if (j > 0)
                    aux1 = matriz[i - 1][j - 1];
                if (j < y - 1)
                    aux2 = matriz[i - 1][j + 1];
                matriz[i][j] = matriz[i][j] + max(matriz[i - 1][j], aux1,
aux2);
            }
            piedrasTotales = max(piedrasTotales, matriz[i][j], -1);
        }
    }
    return piedrasTotales;
}
int max(int a, int b, int c)
{
    int numMax = 0;
```



```
numMax = (numMax > a) ? numMax : a;  
numMax = (numMax > b) ? numMax : b;  
numMax = (numMax > c) ? numMax : c;  
return numMax;  
}
```



BIBLIOGRAFÍA

- [1] «Hirok19,» 22 Noviembre 2016. [En línea]. Available:
<http://hirok19.blogspot.com/2016/11/bytesm2-philosophers-stone.html>. [Último acceso: 25
Noviembre 2021].
- [2] «GeeksForGeeks,» 21 Septiembre 2021. [En línea]. Available:
<https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/>. [Último acceso: 25
Noviembre 2021].
- [3] F. Fuentes, «OmegaUp,» 3 Marzo 2017. [En línea]. Available:
<https://omegaup.com/arena/problem/Longest-Common-Subsequence/#problems>. [Último
acceso: 22 Noviembre 2021].
- [4] O. ElAzazy, «SPOJ,» 17 Marzo 2012. [En línea]. Available:
<https://www.spoj.com/problems/ELIS/>. [Último acceso: 22 Noviembre 2021].
- [5] N. P. Borisov, «SPOJ,» 2008 Noviembre 10. [En línea]. Available:
<https://www.spoj.com/problems/KNAPSACK/>. [Último acceso: 22 Noviembre 2021].

