# Archivo:Logo Instituto Politécnico Nacional.png - Wikipedia, la enciclopedia libreArchivo:EscudoESCOM.png - Wikipedia, la enciclopedia libreINSTITUTO POLITÉCNICO NACIONAL

# ESCUELA SUPERIOR DE CÓMPUTO

Vargas Hernández Carlo Ariel

Análisis de algoritmos

3CM15

Ejercicios 11



Contenido

[**Problema 01: Longest Common Subsequence. [1]** 1](#_Toc104672729)

[Descripción 1](#_Toc104672730)

[Explicación: 1](#_Toc104672731)

[Complejidad: 2](#_Toc104672732)

[Código: 2](#_Toc104672733)

[Captura: 3](#_Toc104672734)

[**Problema 02: ELIS - Easy Longest Increasing Subsequence. [2]** 3](#_Toc104672735)

[Descripción: 3](#_Toc104672736)

[Explicación: 4](#_Toc104672737)

[Complejidad: 5](#_Toc104672738)

[Código: 5](#_Toc104672739)

[Captura: 6](#_Toc104672740)

[**Problema 03: Bacterias. [3]** 6](#_Toc104672741)

[Descripción: 6](#_Toc104672742)

[Explicación: 7](#_Toc104672743)

[Complejidad: 8](#_Toc104672744)

[Código: 9](#_Toc104672745)

[Captura: 10](#_Toc104672746)

[**Problema 07: KNAPSACK - The Knapsack Problem. [4]** 10](#_Toc104672747)

[Descripción: 10](#_Toc104672748)

[Explicación:} 10](#_Toc104672749)

[Complejidad: 12](#_Toc104672750)

[Código: 12](#_Toc104672751)

[Captura: 13](#_Toc104672752)

[**Referencias:** 14](#_Toc104672753)

# **Problema 01: Longest Common Subsequence. [1]**

## 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.

AGCT, AMGXTP, y la susecuencia común más larga es AGT.

## Explicación:

Anteriormente en clase habíamos visto la solución para este problema de manera recursiva:

0 si i=0 o j=0

LCS(Xi,Yj)= 1+LCS(Xi-1,Yj-1) si xi=yj

max(LCS(Xi-1,Yj), LCS(Xi,Yj-1)) si xi!=yj

Aplicando topdown estos valores se guardaban en una tabla y aquí encontramos un patrón que daba los valores máximos, para explicarlo usaremos el ejemplo de OmegaUp. Para llenar la tabla primero sabemos por la recursividad que cuando j=0 o i=0 es 0. Después de llenar esto escribimos en la tabla si la letra coincide o no:

Cuando coinciden sumamos 1 ya que esta letra la podemos tomar. Sin embargo podemos arrastrar el valor máximo de subsecuencia mientras se itera si no se encuentran más coincidencias, siempre y cuando no haya un mayor número en el renglón de arriba:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Yj | A | M | G | X | T | P |  |
| Xi | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Renglón base |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 | Como no hay nada mayor se arrastra el 1 |
| G | 0 | 1 | 1 | 2 |  |  |  | No coincide G con A por lo que es 0, pero arriba hay 1 y se baja  En ese caso cuando vuelve a coincidir una letra se toma el valor de la diagonal superior izquierda y se le suma 1 (1+1) |
| C | 0 |  |  |  |  |  |  |
| T | 0 |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Yj | A | M | G | X | T | P |
| Xi | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| G | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| C | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| T | 0 | 1 | 1 | 2 | 2 | 3 | 3 |

En la recursividad habíamos dicho que si Xi y Yj son iguales se sumaba 1 y se llamaba recursivamente la función. En el caso de omega up las celdas [1,1], [2,3] y [4,5] se ve este caso y podemos ver que la casilla en la diagonal superior izquierda (la azul) más uno da como resultado.

## Complejidad:

Al tener que llenar una matriz con los resultados y teniendo en cuenta que las otras operaciones hechas son constantes este algoritmo tienen una complejidad de O(nm) donde n y m son la cantidad de caracteres de la primer y segunda palabra.

## Código:

#include <bits/stdc++.h>

using namespace std;

int LCS(string c1, string c2){

  int i,j;

    int m=c1.size();

    int n=c2.size();

    int L[m+1][n+1];

    for(i=0;i<=m;i++){

      L[i][0]=0;

  }

  for(i=0;i<=n;i++){

      L[0][i]=0;

  }

  for(i=1;i<=m;i++){

    for(j=1;j<=n;j++){

      if(c1[i-1]==c2[j-1]){

L[i][j]=L[i-1][j-1]+1;

      }else{

        L[i][j]=max(L[i-1][j],L[i][j-1]);

      }

    }

  }

  return L[m][n];

}

int main(){

  string c1,c2;

  cin>>c1;

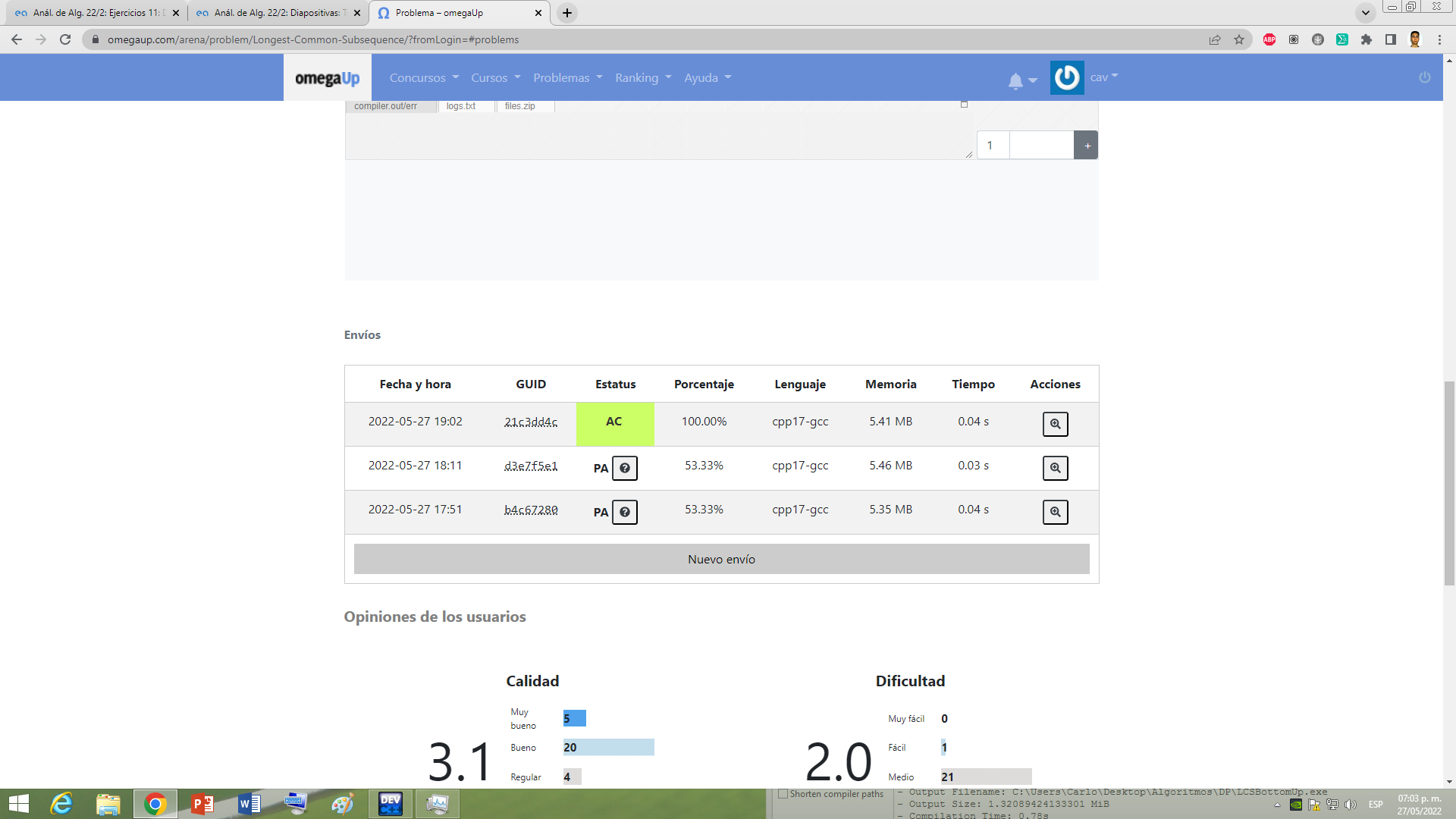
  cin>>c2;

    cout<<LCS(c1, c2);

    return 0;

}

## Captura:



# **Problema 02: ELIS - Easy Longest Increasing Subsequence. [2]**

## Descripción:

Dada una lista de A números, obtenga el tamaño de la subsecuencia creciente más grande. Una subsecuencia creciente se define como un arreglo donde {i0 , i1 , i2 , i3 , ... , ik} tal que 0 <= i0 < i1 < i2 < i3 < ... < ik < N y A[ i0 ] < A[ i1 ] < A[ i2 ] < ... < A[ ik ].

Entrada:

La primer línea contiene un numero N (1 <= N <= 10) que es el tamaño de la lista A.

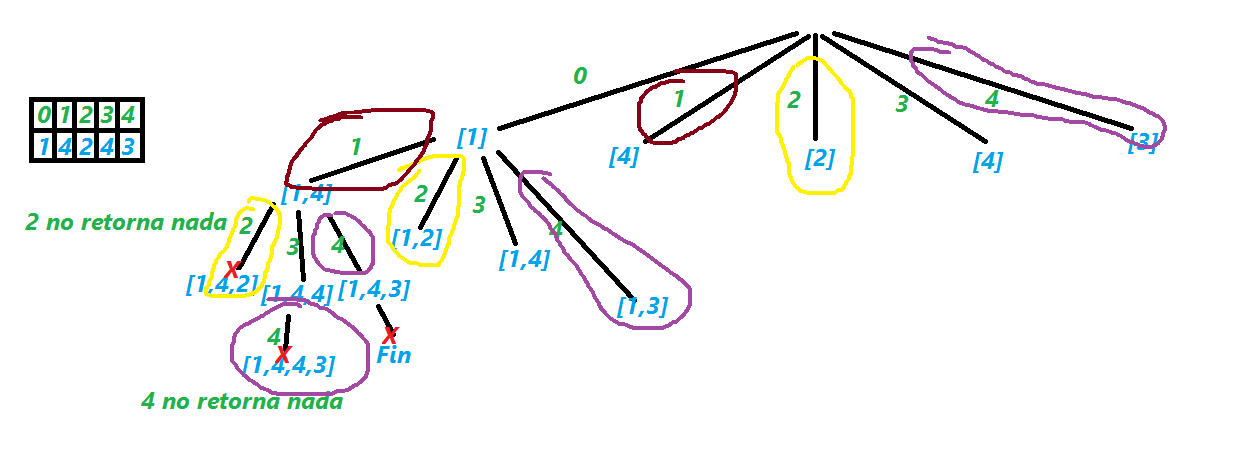
La segunda línea contiene N números (1 <= cada numero <= 20), de la lista A separados por espacios.

Salida:

Una línea conteniendo el tamaño de la subsecuencia creciente más larga.

## Explicación:

La primer forma de resolver este problema sería la fuerza bruta, parecido a knapsack tomando o no tomando un elemento del arreglo que habíamos visto era 2n. Al igual que knapsack se puede mejorar implementando memoización, veamos el ejemplo que viene en SPOJ:



Sin expandir todo el árbol, ya podemos ver que hay valores que se repiten que podríamos memoizar una vez los calculemos. Con lo que el ELIS para los índices (verdes) seria:

ELIS(4)=1 El mismo 3 del final y ya no hay más a la derecha

ELIS(3)=1 El mismo 4, el 3 no vale porque es menor que 4

ELIS(2)=2 2,4 o 2,3 cualquiera es valido

ELIS(1)=2 4,4 es el único

ELIS(0)=3 1,4,4 o 1,2,3 y este es el más grande

Como se puede ver, la forma en que se resuelve el árbol da como resultado que los valores de subsecuencia incrementen si tomamos los elementos del arreglo dado de atrás hacia adelante. Usando la lógica con lo que resolvimos lo que previamente, tenemos que comparar todos los que estén enfrente del elemento seleccionado, si son menores que este entonces aumentamos la longitud de la subsecuencia. Estos valores se guardan para no recalcularlos cada vez y solo escogemos el máximo. En código:

for(int i=n-1;i>=0;i--){//Se empieza del ultimo

  for(int j=i+1;j<n;j++){//Para todos los valores delante del seleccionado

    if(A[i]<A[j]){//Si es mas pequeño es subsecuencia

      subsec[i]=max(subsec[i],1+subsec[j]);//Si el anterior +1 es mas grande  guardalo si no usa el que ya estaba

    }

  }

}

## Complejidad:

Si bien no se está llenando una matriz si se pasa n veces por los n elementos del arreglo con lo que sería O(n2).

## Código:

#include<bits/stdc++.h>

using namespace std;

int ELIS(int A[], int n){

  int subsec[n];

  for(int i=0;i<n;i++){

    subsec[i]=1;

  }

  int maximo=1;

  for(int i=n-1;i>=0;i--){

    for(int j=i+1;j<n;j++){

      if(A[i]<A[j]){

        subsec[i]=max(subsec[i],1+subsec[j]);

      }

    }

  }

  for(int i=0;i<n;i++){

    if(maximo<subsec[i]){

      maximo=subsec[i];

    }

  }

  return maximo;

}

int main(){

  int n;

  cin>>n;

  int A[n];

  for(int i=0;i<n;i++){

    cin>>A[i];

  }

  cout<<ELIS(A,n);

}

## Captura:



# **Problema 03: Bacterias. [3]**

## Descripción:

Un grupo de biólogos computacionales ha diseñado un experimento para decidir si una colonia de microbios es capaz de resolver problemas cuando se le estimula de ciertas formas específicas. En este experimento se construye un recipiente con la forma de una cuadricula rectangular con m renglones y n columnas. En cada uno de los cuadritos de la cuadricula se coloca cierta cantidad de un compuesto químico que le es muy desagradable a los microbios y que, por lo tanto, los microbios preferirían evitar lo más posible. El recipiente está inclinado de tal forma que los microbios solo puede avanzar hacia el este o hacia el sur. Por supuesto, los microbios tampoco pueden salir del recipiente. Al principio la colonia de microbios está localizada en el cuadrito correspondiente al primer renglón y primera columna del recipiente. Al final se espera que la colonia de microbios termine en el cuadrito correspondiente al último renglón y última columna del recipiente. En términos de un sistema de coordenadas, los microbios comienzan en la coordenada (1, 1) y terminan en la coordenada (m, n). Antes de llevar a cabo el experimento, los científicos desean calcular la cantidad c de unidades del compuesto químico que deberá soportar la colonia en su trayecto, esto es, la suma de todas las cantidades del compuesto químico que fueron depositadas en todos los cuadritos por los que pasen. En el ejemplo se muestra un recipiente donde el mínimo valor posible de c es 17.

Entrada:

Dos enteros m y n separados por un espacio, seguidos de m renglones cada uno con n enteros separados por espacios. Estos enteros representan las cantidades del compuesto químico. Puedes suponer que 2 ≤ m ≤ 100, 2 ≤ n ≤ 100 y que todas las demás cantidades están entre 1 y 9, incluyéndolos.

Salida:

Deberás escribir en pantalla un entero c el cual representa la cantidad mínima del compuesto químico al que puede estar expuesto la colonia durante su recorrido. Consideraciones Tu programa se evaluará con varios casos de prueba

## Explicación:

Para explicar este problema hay que tener en cuenta algunas cosas: La primera es que las bacterias solo se pueden mover de izquierda a derecha de arriba a abajo pero no en diagonal y la segunda es que no se pueden salir del recipiente. Voy a usar el ejemplo que yo idee para resolver el problema:

Entrada:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 5 | 9 | 8 | 7 |
| 9 | 3 | 2 | 6 | 6 |
| 1 | 1 | 7 | 8 | 9 |
| 4 | 5 | 2 | 7 | 1 |

4 5

1 5 9 8 7

9 3 2 6 6

1 1 7 8 9 La solución bruta seria sacar todos los caminos posibles de la casilla verde

4 5 2 7 1 a la azul, sumar las casillas de esos caminos y escoger el menor lo que

sería exponencial.

Empecemos a ver mi razonamiento en la primer fila y columna (en el proceso de pensar la solución me di cuenta que habría sido más fácil empezar así), de inicio ya sabemos que el valor que si o si va a tener es el de la celda [0,0]. Hay que ver como se vería si solo estuviera la primera fila y columna:

Para una sola fila la única cosa que se podría hacer es sumar con la de la izquierda ya que el problema dice que solo se pueden mover de izquierda a derecha. Para una sola columna solo se puede sumar lo de arriba porque se mueve de arriba abajo.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 5 | 9 | 8 | 7 |
| 9 |
| 1 |
| 4 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 5+1 | 9+5+1 | 8+9+5+1 | 7+8+9+5+1 |
| 9+1 |
| 1+9+1 |
| 4+1+9+1 |

Sumar de esta forma las casillas me asegura tener el mínimo al moverme con el comportamiento de las bacterias y las restricciones puestas. Sin embargo llega el problema de ya no estar en los bordes del recipiente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 5 | 9 | 8 | 7 |
| 9 | 3 | 2 | 6 | 6 |
| 1 | 1 | 7 | 8 | 9 |
| 4 | 5 | 2 | 7 | 1 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 6 | 15 | 23 | 30 |
| 10 | ¿? | ¿? | ¿? | ¿? |
| 11 | ¿? | ¿? | ¿? | ¿? |
| 15 | ¿? | ¿? | ¿? | ¿? |

Como la bacteria solo tiene dos movimientos (derecha o abajo) eso me reduce a dos opciones las comparaciones que tengo que hacer para tomar la decisión de a donde moverme. Aquí el truco que encontré es que en lugar de empezar de un punto y evaluar a donde me puedo mover es mejor posicionarme en el destino y ver de dónde pude haber venido (de la casilla de arriba o de la izquierda). Siguiendo con el ejemplo, para la casilla roja:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 6 | 15 | 23 | 30 |
| 10 | ¿? | ¿? | ¿? | ¿? |
| 11 | ¿? | ¿? | ¿? | ¿? |
| 15 | ¿? | ¿? | ¿? | ¿? |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 5 | 9 | 8 | 7 |
| 9 | 3 | 2 | 6 | 6 |
| 1 | 1 | 7 | 8 | 9 |
| 4 | 5 | 2 | 7 | 1 |

Para la casilla roja (que es la casilla destino) comparamos las casillas naranja (que son las casillas de donde pude haber llegado) y tomo la que sea menor para tener el menor costo, en este caso 6<10 por lo que tomamos 6 (es decir nos movemos a la derecha) y le sumamos la casilla destino 6+3 (la roja).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 6 | 15 | 23 | 30 |
| 10 | 9 | ¿? | ¿? | ¿? |
| 11 | ¿? | ¿? | ¿? | ¿? |
| 15 | ¿? | ¿? | ¿? | ¿? |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 5 | 9 | 8 | 7 |
| 9 | 3 | 2 | 6 | 6 |
| 1 | 1 | 7 | 8 | 9 |
| 4 | 5 | 2 | 7 | 1 |

Y repetimos el proceso para llenar toda la tabla, de esta forma estamos asegurando tomar el menor de las dos casillas posibles para llegar a una casilla destino y por tanto minimizamos la cantidad de valores.

Al final la tabla del ejemplo se ve así, y el valor en [m-1,n-1] tiene el valor resultante de escoger las casillas con el valor mínimo, y esa es nuestra respuesta.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 6 | 15 | 23 | 30 |
| 10 | 9 | 11 | 17 | 23 |
| 11 | 10 | 17 | 25 | 32 |
| 15 | 15 | 17 | 24 | 25 |

## Complejidad:

Al tener que llenar una tabla de m\*n con todos los valores mínimos posibles y teniendo en cuenta que la comparación, suma y asignar a la matriz de valores mínimos es constante la complejidad sería: O(mn) que podría llegar a ser O(n2) si m=n.

## Código:

#include <bits/stdc++.h>

using namespace std;

int main(){

  int m, n;

  cin>>m>>n;

  int recipiente[m][n];

  int valores[m][n];

  for(int i=0;i<m;i++){

    for(int j=0;j<n;j++){

      cin>>recipiente[i][j];

      valores[i][j]=0;

    }

  }

  for(int i=0;i<m;i++){

    for(int j=0;j<n;j++){

      if(i==0 && j==0){

        valores[i][j]=recipiente[i][j];

      }else if(i==0){

        valores[i][j]=valores[i][j-1]+recipiente[i][j];

      }else if(j==0){

        valores[i][j]=valores[i-1][j] + recipiente[i][j];

      }else{

        valores[i][j]=min(valores[i][j-1], valores[i-1][j])+recipiente[i][j];

      }

    }

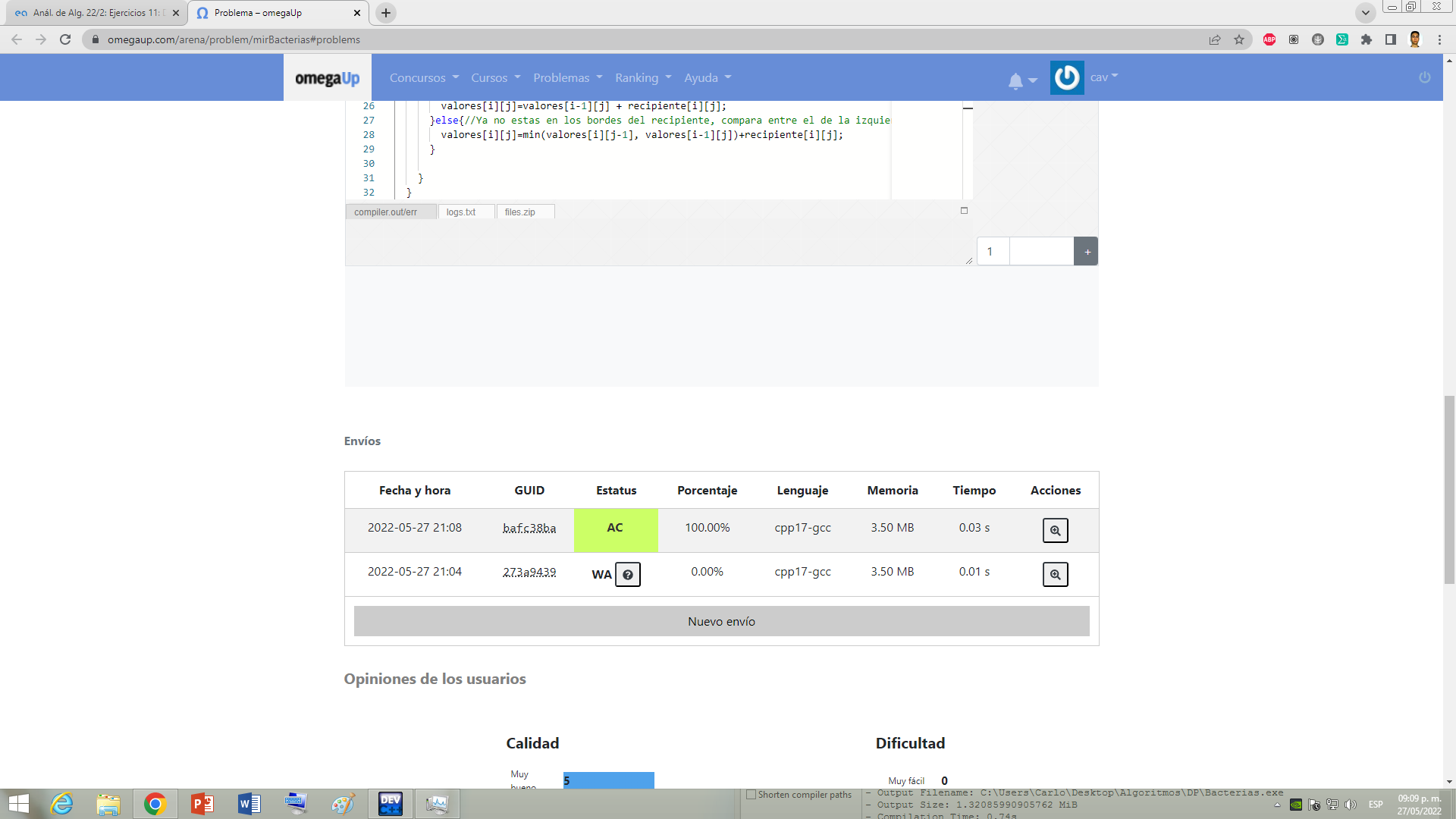
  }

  cout<<valores[m-1][n-1];

  return 0;

}

## Captura:



# **Problema 07: KNAPSACK - The Knapsack Problem. [4]**

## Descripción:

Vas de vacaciones y tienes una mochila con capacidad S (1 <= S <= 2000). También tienes N (1<= N <= 2000) objetos que te quieres llevar.

Entrada:

En la primera línea se da S y N. En las siguientes N líneas se dan dos números, el primero el tamaño del objeto y el siguiente el valor.

Salida:

Un solo número que sea el valor total máximo de los mejores objetos escogidos.

## Explicación:}

Teniendo el ejemplo:

|  |  |  |
| --- | --- | --- |
| Objeto | Peso | Valor |
| 1 | 2 | 12 |
| 2 | 1 | 10 |
| 3 | 3 | 20 |
| 4 | 2 | 15 |

Tenemos que la ganancia está dada por:

0 si i=j=0

V[i,j]= V[i-1,j] si wi>j

Max(V[i-1,j],V[i-1,j-wi]+vi) si wi<=j

Resolviendo con esa regla obtenemos la tabla:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 12 | 12 | 12 | 12 |
| 2 | 0 | 10 | 12 | 22 | 22 | 22 |
| 3 | 0 | 10 | 12 | 22 | 30 | 32 |
| 4 | 0 | 10 | 15 | 25 | 30 | 37 |

Siendo la casilla azul la solución óptima. Quería implementar la solución top down ya que no entendí del todo la bottom up pero por alguna razón mi código no funciona y no sé dónde está el error.

#include <bits/stdc++.h>

using namespace std;

int dp[2000][2000];

int KS(int wt[], int profit[], int W, int N){

  int result;

  if(W==0 || N==0){

    return 0;

  }

  if(dp[W][N]!=-1){

    return dp[W][N];

  }

  if(wt[N]>W){

    result=KS(wt,profit,W,N-1);//Descartamos el elemento que sipera la capacidad

  }else{

    result= max(KS(wt,profit,W,N-1),profit[N]+KS(wt,profit,W-wt[N],N-1));

  }

  return dp[W][N]=result;

}

int main(){

    int N,S;//N los objetos, S la capacidad de la mochila

    cin>>S>>N;

    int tamano[N],valor[N];

    for(int i=0;i<N;i++){

      cin>>tamano[i]>>valor[i];

  }

  for(int i=0;i<S;i++){

      for(int j=0;j<N;j++){

        dp[i][j]=-1;

    }

  }

    cout << KS(tamano, valor, S, N);

    return 0;

}

## Complejidad:

Al llenar una tabla la complejidad está dada por las dimensiones de la tabla en este caso O(SN).

## Código:

#include <bits/stdc++.h>

using namespace std;

int knapsack(int tamano[], int valor[], int capmoc/\*capacidadmochila\*/, int numel /\*numero de elementos\*/){

    ……

}

int main(){

    int N,S;//N los objetos, S la capacidad de la mochila

    cin>>S>>N;

    int tamano[N],valor[N];

    for(int i=0;i<N;i++){

      cin>>tamano[i]>>valor[i];

  }

    cout << knapsack(tamano, valor, S, N);

    return 0;

}

int knapsack(int tamano[], int valor[], int capmoc/\*capacidadmochila\*/, int numel /\*numero de elementos\*/){

    int memoizador[numel+1][capmoc+1];

  //En este caso se hizo la bottom down para hacer algo diferente a lo de la clase

    for (int i=0;i<=numel;i++){

        for (int j=0;j<=capmoc;j++){

            if (i==0 || j==0){//Caso base

              memoizador[i][j] = 0;

      }else if (tamano[i-1]<=j){

        memoizador[i][j] = max(valor[i-1]+memoizador[i-1][j-tamano[i-1]], memoizador[i-1][j]);

      }else{

        memoizador[i][j]=memoizador[i-1][j];

      }

        }

    }

## Captura:



# **Referencias:**

[1] F. Filiberto, (2017, Mar 30). Longest Common Subsequence. <https://omegaup.com/arena/problem/Longest-Common-Subsequence/#problems>

[2] Omar ElAzazy, (2012, Mar 17). ELIS - Easy Longest Increasing Subsequence. <https://www.spoj.com/problems/ELIS/>

[3] I. Martin, (2015, Mar 30). Bacterias. <https://omegaup.com/arena/problem/mirBacterias#problems>

[4] B. Nikola, (2008, Nov 10). KNAPSACK - The Knapsack Problem. <https://www.spoj.com/problems/KNAPSACK/>