

Instituto Politécnico Nacional

Escuela Superior de Cómputo

* **UNIDAD DE APRENDIZAJE:** ANÁLISIS DE ALGORITMOS
* **PROFESOR:** EDGARDO ADRIÁN FRANCO MARTÍNEZ
* **ALUMNO:**

BARRERA PÉREZ CARLOS TONATIHU



## **Ejercicio 7: Diseño de soluciones con programación dinámica**

## 

* **GRUPO:** 3CM3

# Subsecuencia común más larga (LCS)

## Redacción

Este problema consiste en que 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.

# **Límites**

* 1 ≤ |A| ≤ 1000
* 1 ≤ |B| ≤ 1000

## Análisis

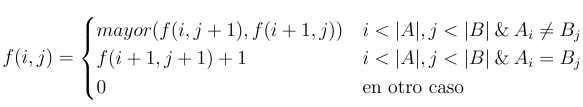
Las opciones que pueden existir en este problema son tres, la más sencilla de ellas es cuando las cadenas son de longitud cero debido que en ese caso la longitud de la subsecuencia más larga es cero debido a que no hay caracteres que comparar.

Después, tenemos el caso en el cual comenzamos a comparar las cadenas y nos encontramos con que el primer carácter de cada cadena es el mismo por lo que en este caso tenemos una subsecuencia de uno, ahora lo único que tenemos que hacer es seguir buscando más coincidencias en las dos cadenas restantes que tendrán una longitud una unidad menor que la anterior.

Existe otra posibilidad, en la que no hay coincidencia al inicio de las cadenas por lo que esto produce dos opciones

* Tomar la primera cadena y remover el primer carácter y con ello generar una nueva cadena que comparar con la anterior.
* Hacer lo mismo con la segunda cadena y comparar con la otra cadena.

Además, hay que considerar que estas dos opciones pueden generar cadenas de longitud diferente por lo que hay que obtener el mayor entre estas dos opciones. Finalmente, para obtener la longitud final estos pasos se tienen que realizar de forma recursiva por lo que podemos expresar estas opciones de forma recursiva como en la siguiente formula.



Esta función se implementó utilizando el modelo Bottom up con ayuda de una tabla para guardar los valores base que sirven para calcular los valores de los casos finales. Esto se realiza mediante dos ciclos for donde se empiezan a comparar las cadenas. Solo se utilizó una tabla de 2xN, donde N es la longitud de una de las cadenas, solamente son necesarias dos filas debido a que solo se utilizan a la vez la fila actual y la fila anterior.

## Código

Para compilar el código se utiliza el siguiente comando

|  |
| --- |
| $ g++ secuencia.cpp -o ./secuencia |

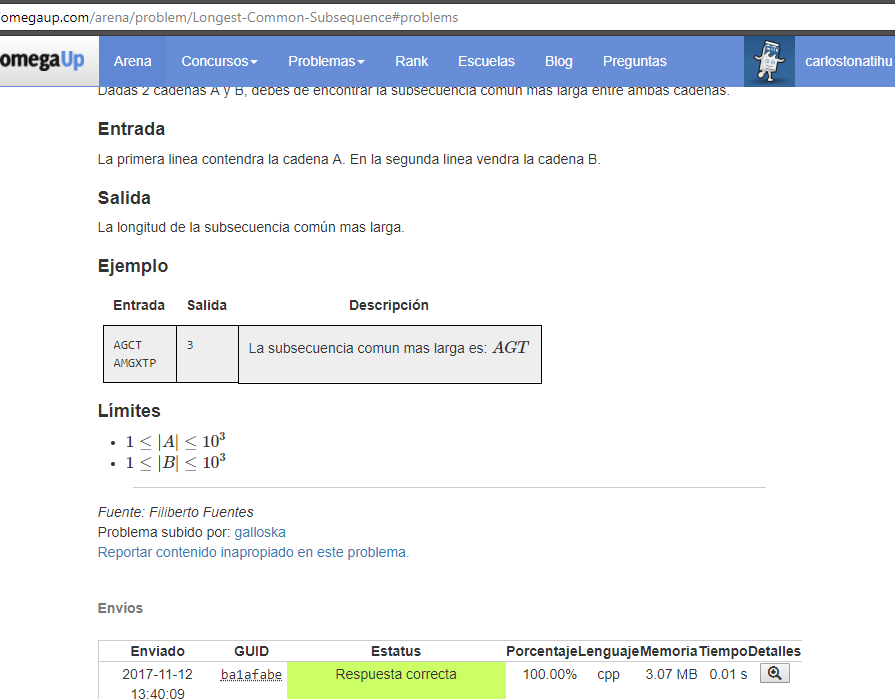
Para ejecutar el archivo resultante utilizamos el siguiente comando

|  |
| --- |
| $ ./secuencia |

secuencia.cpp

|  |
| --- |
| #include <iostream>  #include <string.h>  **using** **namespace** std;  int LCS(char \*, char \*);  int main(int argc, char **const** \*argv[]){  *// Captura de los datos*  char cadena1[1000];  char cadena2[1000];  cin >> cadena1;  cin >> cadena2;  int longitud = LCS(cadena1, cadena2);  cout << longitud;  **return** 0;  }  *// Encuentra la longitud de la secuencia comun mas larga entre dos cadenas*  int LCS(char \*cadena1, char \*cadena2) {  *// Longitud de la cadena*  int longitud1 = strlen(cadena1);  int longitud2 = strlen(cadena2);  *// Tabla para guardar datos ya calculados*  int tabla[2][longitud2+1];  *// Alternar entre las dos filas de la tabla*  int actual = 1;  int anterior = 0;  *// Inicializamos nuestra tabla*  **for** (int i = 0; i < 2; i++)  **for**(int j = 0; j<=longitud2; j++)  tabla[i][j] = 0;  *// Implementacion bottom up*  **for** (int i = 1; i <= longitud1; ++i){  **for** (int j = 1; j <= longitud2; j++){  **if** (cadena1[i-1] == cadena2[j-1])  tabla[actual][j] = tabla[anterior][j-1] + 1;  **else** **if** (tabla[anterior][j] >= tabla[actual][j-1])  tabla[actual][j] = tabla[anterior][j];  **else**  tabla[actual][j] = tabla[actual][j-1];  }  actual = !actual;  anterior = !anterior;  }  **return** tabla[anterior][longitud2];  } |

## Captura de pantalla del juez



# Bacterias

## Redacción

**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 pueden 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 & 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.

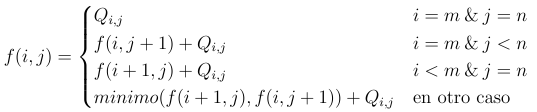
**Salida**

Un entero c el cual representa la cantidad mínima del compuesto químico al que puede estar expuesto la colonia durante su recorrido.

## Análisis

Es fácil darse cuenta de que la colonia de bacterias solo puede llegar a la casilla final desde dos casillas diferentes por lo que debemos de considerar cuál de estas casillas tiene el camino que recorre menos químico, a su vez estas dos casillas solo pueden ser accedidas por otras dos casillas diferentes donde una de ellas debe de ser la que tenga menor químico por lo que al final esto nos generara un camino donde se recorre la menor cantidad de químico.

Esto se puede modelar con una función recursiva considerando los casos en los que ya no te puedas mover más hacia la derecha por lo que la única opción es moverte hacia abajo y el caso opuesto donde ya no te puedas mover hacia abajo y solo nos podamos mover hacia la derecha, finalmente tenemos el caso donde ya habremos llegado al final de la matriz.



En la formula Qij es el químico que hay en una casilla de la matriz, y mínimo nos devuelve el mínimo de tomar el camino de la derecha o el camino hacia abajo, esto considerando que empezamos en la casilla (0, 0). Es importante siempre sumar el químico Qij debido a que con esto consideramos la casilla en la que nos encontramos.

Para programar esta función recursiva se utilizó el modelo Top-down junto a una tabla para almacenar los valores que ya hemos calculado.

## Código

Para compilar el código se utiliza el siguiente comando

|  |
| --- |
| $ g++ bacterias.cpp -o ./bacterias |

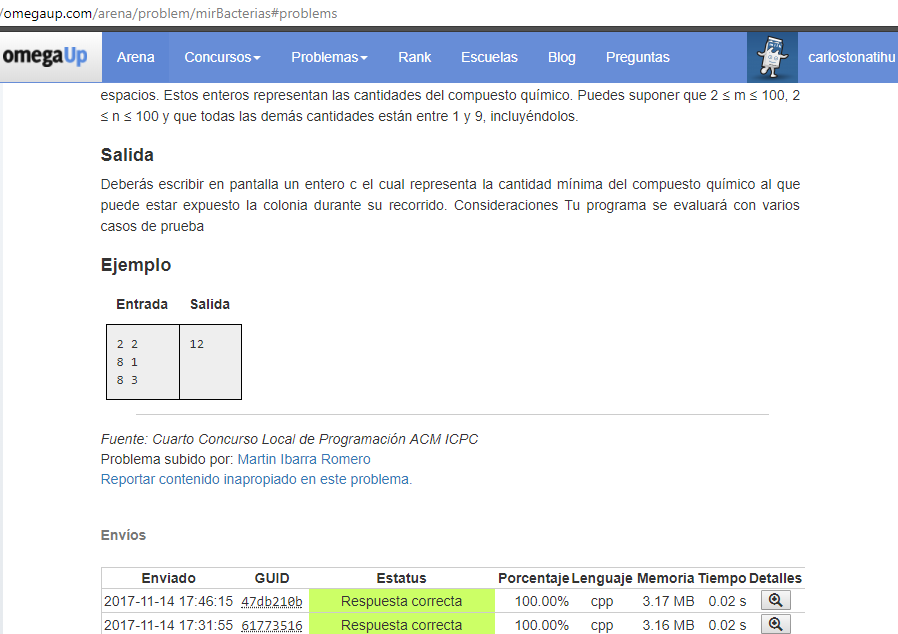
Para ejecutar el archivo resultante utilizamos el siguiente comando

|  |
| --- |
| $ ./bacterias |

bacterias.cpp

|  |
| --- |
| #include <iostream>  #include <stdlib.h>  **using** **namespace** std;  int minimo(int, int);  int minimo\_quimico(int, int, int \*\*, int \*\*);  *// Dimensiones de la matriz*  int fila;  int col;  int main(int argc, char **const** \*argv[]) {  cin >> fila >> col;  *// matriz de los datos de entrada*  int \*\*quimicos = (int\*\*) malloc(fila \* **sizeof**(int\*));  *// Tabla para guardar datos ya calculados*  int \*\*tabla = (int\*\*) malloc(fila \* **sizeof**(int\*));  **for** (int i = 0; i < fila; i++){  quimicos[i] = (int\*) malloc(col \* **sizeof**(int));  tabla[i] = (int\*) malloc(col \* **sizeof**(int));  }  *// Entrada de datos*  **for** (int i = 0; i < fila; i++)  **for** (int j = 0; j < col; j++){  cin >> quimicos[i][j];  tabla[i][j] = -1;  }  *// Inicializacion de la tabla auxiliar*  tabla[fila-1][col-1] = quimicos[fila-1][col-1];  **for** (int j = col-2; j > -1; j--)  tabla[fila-1][j] = tabla[fila-1][j+1] + quimicos[fila-1][j];  **for** (int i = fila-2; i > -1; i--)  tabla[i][col-1] = tabla[i+1][col-1] + quimicos[i][col-1];  *// Imprimimos el resultado*  cout << minimo\_quimico(0, 0, quimicos, tabla);  **return** 0;  }  *// BUscamos el menor quimico por top-down*  int minimo\_quimico(int i, int j, int \*\*quimicos, int \*\*tabla) {  **if** (tabla[i][j] != -1)  **return** tabla[i][j];  **if** (i == fila-1 && j == col-1) {  **if** (tabla[i][j] != -1)  **return** tabla[i][j];  tabla[i][j] = quimicos[i][j];  **return** tabla[i][j];  }  **else** **if** (i == fila-1 && j < col) {  **if** (tabla[i][j] != -1)  **return** tabla[i][j];  tabla[i][j] = minimo\_quimico(i, j+1, quimicos, tabla) + quimicos[i][j];  **return** tabla[i][j];  }  **else** **if** (i < fila && j == col-1) {  **if** (tabla[i][j] != -1)  **return** tabla[i][j];  tabla[i][j] = minimo\_quimico (i+1, j, quimicos, tabla) + quimicos[i][j];  **return** tabla[i][j];  }  **else**{  **if** (tabla[i][j] != -1)  **return** tabla[i][j];  tabla[i][j] = minimo(minimo\_quimico(i+1, j, quimicos, tabla), minimo\_quimico(i, j+1, quimicos, tabla)) + quimicos[i][j];  **return** tabla[i][j];  }  }  *// Obtiene el minimo de dos numeros*  int minimo(int a, int b) {  **if** (a < b)  **return** a;  **else** **return** b;  } |

## Captura de pantalla del juez



# BYTESM 2 – Philosophers Stone

## Redacción

La cámara está cubierta por h x w cuadrados donde hay h filas y w columnas por fila. Cada azulejo tiene una a cien piedras. Harry tiene que tomar tantas piedras como sea posible, sujeto a las siguientes restricciones.

* Se empieza por escoger cualquier azulejo en la primera fila y toma todas las piedras en ese azulejo. Entonces se mueve al siguiente azulejo y toma todas las piedras y así hasta que llega a la última fila.
* Cuando se mueve de una fila a otra solo lo puede hacer en diagonal a la izquierda o a la derecha o hacia enfrente.

Dados los valores de H y W y el número de piedras filosofales en cada mosaico escribe un programa que encuentre el máximo valor posible de piedras filosofales que se pueden tomar desde la primera fila a la última.

**Entrada**

La primera línea contiene un numero T que son el número de casos de prueba. En la siguiente línea están las dimensiones H & W de la matriz y las siguientes H líneas contienen W números que son la cantidad de piedras filosofales.

**Salida**

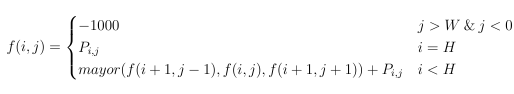
La salida consiste de T líneas de resultados del máximo valor posible de piedras filosofales que se pueden tomar.

## Análisis

Para este programa primero se debe de considerar la posibilidad de que se empiece en una casilla de la primera fila en específico y a partir de ahí realizar todos los movimientos posibles para llegar a la última fila tratando de maximizar el número de piedras que se pueden recoger. Para esto hay que considerar todas las condiciones que involucra moverse de una fila a otra.

* El máximo de poder moverse en diagonal a la izquierda o derecha o en forma recta siempre y cuando haya casillas para moverse a los lados y hacia abajo.
* Si no nos podemos mover hacia los lados entonces el número de piedras que podemos recoger es cero, para simplificar las cosas se utilizó un numero negativo.
* Si ya no nos podemos mover hacia abajo el número de piedras que podemos recoger es el de esa celda.

Por estas condiciones podemos modelar la siguiente función recursiva.



Donde W es el número de columnas de la matriz y H es el número de filas Pij es el número de piedras en una casilla. Ahora, ya que debemos de considerar que se puede iniciar en cualquier casilla de la primera fila debemos de obtener el máximo de piedras para cada fila y después obtener el mayor de todos estos máximos.

Es por esto por lo que se implementó programación dinámica para guardar valores ya calculados, en este caso se utilizó el modelo bottom up ya que es más fácil obtener los caminos mayores para cada fila empezando desde la última fila.

Y en lugar de implementar una tabla de HxW se utilizó una de 2xW debido que al estar en una iteración solo utilizamos la tabla actual y la anterior, al hacer esto podemos reducir la memoria que se necesita.

## Código

Para compilar el código se utiliza el siguiente comando

|  |
| --- |
| $ g++ piedras.cpp -o ./piedras |

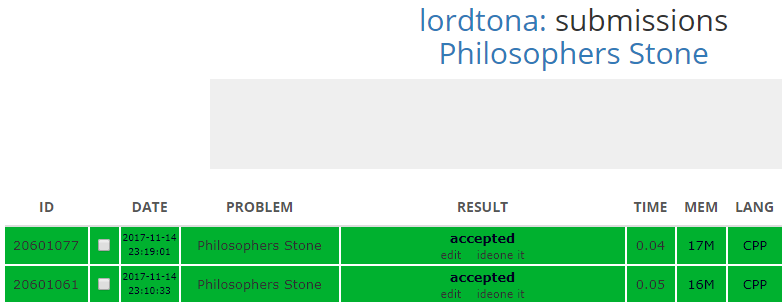
Para ejecutar el archivo resultante utilizamos el siguiente comando

|  |
| --- |
| $ ./piedras |

piedras.cpp

|  |
| --- |
| #include <iostream>  #include <stdlib.h>  #include <stdio.h>  **using** **namespace** std;  int mayor(int, int, int);  void obtener\_max\_piedras();  *// Numero de casos a ejecutar*  int main(int argc, char **const** \*argv[]) {  int T;  cin >> T;  **while** (T--) {  obtener\_max\_piedras();  }  **return** 0;  }  *// Funcion que obtiene el maximo de piedras*  void obtener\_max\_piedras() {  *// Dimensiones de la matriz*  int H;  int W;  cin >> H >> W;  *// Matriz para guardar los datos de entrada*  int \*\*matriz = (int\*\*) malloc(H \* **sizeof**(int\*));  *// Tabla para guardar valores ya calculados*  int \*\*tabla = (int\*\*) malloc(2 \* **sizeof**(int\*));  tabla[0] = (int\*) malloc(W \* **sizeof**(int));  tabla[1] = (int\*) malloc(W \* **sizeof**(int));  **for** (int i = 0; i < H; i++)  matriz[i] = (int\*) malloc(W \* **sizeof**(int));    *// Entrada de datos*  **for** (int i = 0; i < H; i++)  **for** (int j = 0; j < W; j++)  cin >> matriz[i][j];  **for** (int j = 0; j < W; j++)  tabla[0][j] = matriz[H-1][j];  *// Implementacion de programacion dinamica por bottom-up*  short actual = 1;  short pre = 0;  **for** (int i = H-2; i >= 0; i--) {  **for** (int j = 0; j < W; j++) {  **if** (j == 0)  tabla[actual][j] = mayor(tabla[pre][j], tabla[pre][j+1], -10) + matriz[i][j];  **else** **if** (j == W-1)  tabla[actual][j] = mayor(tabla[pre][j], tabla[pre][j-1], -10) + matriz[i][j];  **else**  tabla[actual][j] = mayor(tabla[pre][j-1], tabla[pre][j], tabla[pre][j+1]) + matriz[i][j];  }  actual = !actual;  pre =!pre;  }  *// Buscamos el maximo valor en la primera fila*  int maximo = tabla[pre][0];  int aux = -1;  **for** (int j = 1; j < W; j++) {  aux = tabla[pre][j];  **if** (maximo < aux)  maximo = aux;  }  *// Imprimimos el resultado*  cout << maximo << endl;  }  *// Retorna el maximo de tres numeros*  int mayor(int a, int b, int c) {  **if** (a > b)  **if** (a > c)  **return** a;  **else**  **return** c;  **else** **if** (b > c)  **return** b;  **else**  **return** c;  } |

## Captura de pantalla del juez



## KNAPSACK - The Knapsack Problem

## Redacción

Estas empacando para vacacionar en el mar y solo vas a cargar una bolsa de capacidad (1 <= S <=2000). También tienes N (1<=N<=2000) artículos que podrías querer llevar contigo. Desafortunadamente no puedes empacar todos estos artículos así que tendrás que escoger. Por cada cosa te dan su tamaño y su valor. Quieres maximizar el valor total de todos los elementos que quieres llevar. ¿Cuál es el valor total máximo?

**Entrada**

En la primera línea te dan S y N. Las siguientes N líneas tienen dos enteros en cada línea describiendo los ítems. EL primer número es el tamaño y el segundo su valor.

**Salida**

Un simple entero que es el máximo valor de los ítems escogidos.

## Análisis

En este problema tenemos tres posibilidades para llenar la mochila, la primera y la más evidente es que aun haya espacio en la mochila y que tengamos elementos que meter. Para este caso debemos de elegir entre que nos produce el máximo valor posible si meter un objeto en la mochila e intentar llenar la mochila con una capacidad reducida por el peso del objeto que se metió y utilizar el siguiente objeto, la otra opción que podría maximizar el valor es no meter el objeto y tratar de meter el siguiente objeto.

La siguiente posibilidad es que ya no tengamos objetos que meter en la mochila o que ya no tengamos espacio en la mochila por lo que el valor que se tiene en este caso es 0.

El último caso es que el espacio que tengas para meter un objeto sea menor al peso del objeto por lo que intentaríamos meter el siguiente objeto disponible en este caso el valor de que se obtiene es el máximo posible valor que generaría llenar la mochila con la capacidad actual, pero intentando meter el siguiente objeto.

Considerando esto podemos expresar la solución a este problema como la siguiente función recurrente.

Donde Vi es el valor del objeto i, Pi es el peso del objeto i, S es el peso que queda en la mochila. La función se comienza con f (S, 0) debido a que intentamos meter el objeto 0 con una capacidad S.

Se utilizo el modelo Top-down para este problema debido a que es fácil utilizar esta función recursiva y una tabla extra para poder guardar los datos que se repitan y con esto reducir el tiempo de ejecución.

## Código

Para compilar el código se utiliza el siguiente comando

|  |
| --- |
| $ g++ mochila.cpp -o ./mochila |

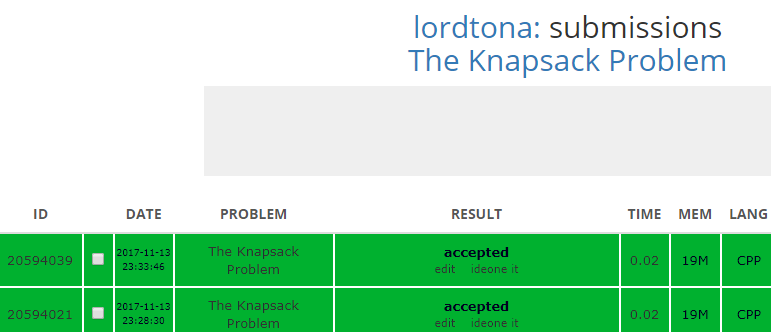
Para ejecutar el archivo resultante utilizamos el siguiente comando

|  |
| --- |
| $ ./mochila |

mochila.cpp

|  |
| --- |
| #include <iostream>  **using** **namespace** std;  int objetos[2][2000];  int cantidad;  int capacidad;  int max\_valor(int, int, int \*\*);  int mayor(int, int);  int main(int argc, char **const** \*argv[]) {  cin >> capacidad >> cantidad;    **for** (int i = 0; i < cantidad; i++)  cin >> objetos[0][i] >> objetos[1][i];  int \*\*tabla = (int\*\*) malloc((capacidad+1) \* **sizeof**(int\*));  **for** (int i = 0; i < capacidad+1; i++){  tabla[i] = (int\*)malloc((cantidad+1) \* **sizeof**(int));  **for** (int j =0; j < cantidad+1; j++)  tabla[i][j] = -1;  }  cout << max\_valor(capacidad, 0, tabla);  **return** 0;  }  int max\_valor(int j, int i, int \*\*tabla) {  **if** (tabla[j][i] != -1)  **return** tabla[j][i];  **if** (j < objetos[0][i]){  **if** (tabla[j][i] == -1)  tabla[j][i] = max\_valor(j, i+1, tabla);  **return** tabla[j][i];  } **else** **if** (j <= 0 || i >=cantidad){  **if** (tabla[j][i] == -1)  tabla[j][i] = 0;  **return** 0;  } **else**{  **if** (tabla[j][i] == -1)  tabla[j][i] = mayor(max\_valor(j-objetos[0][i], i+1, tabla)+objetos[1][i], max\_valor(j, i+1, tabla));  **return** tabla[j][i];  }  }  int mayor(int a, int b) {  **if** (a > b)  **return** a;  **return** b;  } |

## Captura de pantalla del juez



# AIBOHP – Aibohphobia

## Redacción

BuggyD sufre de AIBOHPHOBIA (el miedo a los palíndromos). Para curarlo los doctores deciden exponerlo a un gran número de palíndromos. Por lo que deciden jugar un juego con BuggyD, las reglas son las siguientes.

BuggyD les da una cadena S. Los doctores tienen que insertar caracteres en S para volverlo un palíndromo (se pueden insertar caracteres en cualquier posición de la cadena).

Actualmente BuggyD está curado de su enfermedad, pero quiere seguir jugando el juego con sus propias reglas. Ahora le pide a los doctores que inserten el mínimo numero de caracteres necesarios para hacer a la cadena un palíndromo. Ayuda a los doctores a alcanzar su meta.

**Entrada**

La primera línea de la entrada contiene el número de casos T, después siguen T líneas en las cuales es dado una cadena de longitud que no va a ser mayor de 6100 y no va a tener espacios.

**Salida**

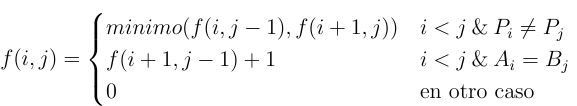
Por cada caso de entrada se imprime una línea con un entero que es el mínimo numero de caracteres que se tienen que insertar en la cadena para volverlo un palíndromo.

## Análisis

En este problema debemos de considerar tres condiciones.

* El caso más sencillo en el cual la cadena es de longitud cero por lo que ya es palíndromo y la cantidad de letras que se deben insertar es cero.
* El siguiente caso es en el cual los dos caracteres del extremo de la palabra son iguales por lo que ahora tenemos que buscar formar un palíndromo en la cadena que se forma al remover los caracteres de los extremos.
* El último caso es en el cual los caracteres de los extremos no coinciden se tienen dos posibles movimientos que hacer, o bien remover el carácter de inicio y buscar el mínimo de letras que se tienen que insertar para volver palíndromo la cadena que nos queda o remover el carácter del final y buscar los el mínimo.

Al tener en cuenta estos casos podemos aplicarlos de forma recursiva para obtener una función que nos permita saber el mínimo de letras que se necesitan para una cadena, por lo que la función recursiva que modela esto es.



Se utilizo el modelo Top-down en el cual junto a la función recursiva se utilizó una tabla para guardar los datos ya calculados y no tener que calcularlos de nuevo.

## Código

Para compilar el código se utiliza el siguiente comando

|  |
| --- |
| $ g++ palindromo.cpp -o ./palindromo |

Para ejecutar el archivo resultante utilizamos el siguiente comando

|  |
| --- |
| $ ./palindromo |

palindromo.cpp

|  |
| --- |
| #include <iostream>  #include <string.h>  **using** **namespace** std;  int palindromo(char \*, int, int, int\*\*);  int minimo(int, int);  int main(int argc, char **const** \*argv[]) {  *// Captura del numero de casos*  int t;  cin >> t;  **while** (t--) {  *// Captura de los datos*  char palabra[6100];  cin >> palabra;  int tam = strlen(palabra);  *// Tabla para utilizar el modelo top down*  int \*\*tabla = (int\*\*) malloc(tam\* **sizeof**(int\*));  **for** (int i = 0; i < tam; i++){  tabla[i] = (int\*) malloc(tam \* **sizeof**(int));  **for** (int j = 0; j < tam; j++)  tabla[i][j] = -1;  }  *// Impresion del resultado*  cout << palindromo(palabra, 0, tam-1, tabla) << endl;  }  **return** 0;  }  *// Encuentra el numero de cambios minimos para volver una palabra palindromo*  int palindromo(char \*palabra, int i, int j, int \*\*tabla) {  *// Los extremos de la palabra no coinciden*  **if** (i < j && (palabra[i] != palabra[j])){  **if** (tabla[i][j] != -1)  **return** tabla[i][j];  tabla[i][j] = minimo(palindromo(palabra, i, j-1, tabla),  palindromo(palabra, i+1, j, tabla)) + 1;  **return** tabla[i][j];  *// Coincidencia entre los extremos de la palabra*  } **else** **if** (palabra[i] == palabra[j] && (i < j)) {  **if** (tabla[i][j] != -1)  **return** tabla[i][j];  tabla[i][j] = palindromo(palabra, i+1, j-1, tabla);  **return** tabla[i][j];  }  *// Caso en el que no tenemos mas caracteres que analizar en la palabra*  **return** 0;  }  int minimo(int a, int b) {  **if** (a < b)  **return** a;  **return** b;  } |

## Captura de pantalla del juez

