



**INSTITUTO POLITÉCNICO NACIONAL**  
**ESCUELA SUPERIOR DE CÓMPUTO**



-----**ANÁLISIS DE ALGORITMOS**-----

**ACTIVIDAD**

Diseño de soluciones mediante DP

**PROFESOR:**

Franco Martínez Edgardo Adrián

**ALUMNO:**

Meza Vargas Brandon David – 2020630288

**GRUPO:**

3CM13



## índice

<b>Problema: Longest Common Subsequence.....</b>	<b>3</b>
Redacción .....	3
Captura de aceptación por juez .....	3
Explicación Algoritmo .....	3
Análisis de complejidad en cota $O()$ .....	5
Código de solución completo .....	6
<b>Problema: ELIS – Easy Longest Increasing Subsequence .....</b>	<b>7</b>
Redacción .....	7
Captura de aceptación por juez .....	7
Explicación Algoritmo .....	7
Análisis de complejidad en cota $O()$ .....	8
Código de solución completo .....	9
<b>Problema: The Knapsack Problem .....</b>	<b>10</b>
Redacción .....	10
Captura de aceptación por juez .....	10
Explicación Algoritmo .....	10
Análisis de complejidad en cota $O()$ .....	12
Código de solución completo .....	13
<b>Problema: AIBOHP - Aibohphobia.....</b>	<b>14</b>
Redacción .....	14
Captura de aceptación por juez .....	14
Explicación Algoritmo .....	15
Análisis de complejidad en cota $O()$ .....	15
Código de solución completo .....	17

## Problema: Longest Common Subsequence

### Redacción

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:

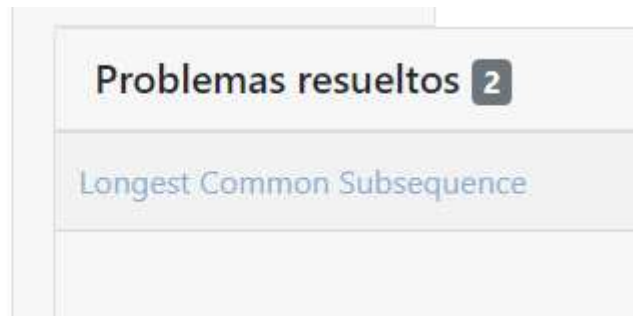
Dadas 2 cadenas A y B, debes de encontrar la subsecuencia común más larga entre ambas cadenas.

### Captura de aceptación por juez

Enviado	GUID	Estatus	Porcentaje	Lenguaje	Memoria	Tiempo	Acciones
2021-11-14 12:54:46	1788c883	AC	100.00%	c11-gcc	3.71 MB	0.02 s	



brandonmv2001



### Explicación Algoritmo

A continuación se presenta la captura del algoritmo:

```
int longestSub(int longA, int longB, char A[], char B[]){
    int i, j;
    int longitud[longA+1][longB+1];
    /*
     *El siguiente for es para la solución bottom up, recorre las longitudes
     *de la cadena A y la cadena B
     */
    for(i=0; i<=longA; i++){
        for(j=0; j<=longB; j++){
            if(i == 0 || j == 0)
                longitud[i][j] = 0;
            else if(A[i-1] == B[j-1])
                longitud[i][j] = longitud[i-1][j-1] + 1;
            else
                longitud[i][j] = MAX(longitud[i-1][j], longitud[i][j-1]);
        }
    }
}
```

Primeramente se usa un arreglo bidimensional para ir guardando la longitud de la subsecuencia común mas larga de la cadena A y la cadena de B.

Posteriormente van los for anidados que recorren ambas longitudes. Si nos encontramos con que  $i$  y  $j$  no avanzan, es decir, se quedan en 0, es por que las cadenas son vacías y no existe una lcs.

Pero si el último carácter de las cadenas coincide se retira ese carácter y se encuentra el lcs de los caracteres restantes, por ejemplo:

cadena A : ABCDE

cadena B: XYZE

La E coincide, se retira y se calcula el lcs de las anteriores, al no tener ninguna subsecuencia común queda el resultado final con **1**.

Si el último carácter no coincide se encuentra la cadena más larga entre las subcadenas de todos los elementos anteriores de la cadena A o las subcadenas de todos los elementos anteriores a la cadena B, por ejemplo:

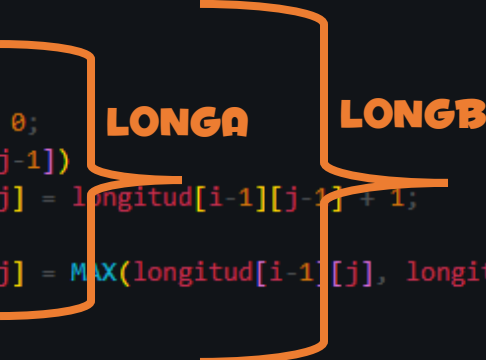
A: ABCDE

B: ABCDEA

El último carácter no coincide, si se retira la E de la cadena A, queda una subcadena de 4 y si se retira la A de la cadena B queda una subcadena de 5, por lo tanto se almacena la cadena B.

### Análisis de complejidad en cota $O()$

```
int longestSub(int longA, int longB, char A[], char B[]){
    int i, j;
    int longitud[longA+1][longB+1];
    /*
     *El siguiente for es para la solución bottom up, recorre las longitudes
     *de la cadena A y la cadena B
     */
    for(i=0; i<=longA; i++){
        for(j=0; j<=longB; j++){
            if(i == 0 || j == 0)
                longitud[i][j] = 0;
            else if(A[i-1] == B[j-1])
                longitud[i][j] = longitud[i-1][j-1] + 1;
            else
                longitud[i][j] = MAX(longitud[i-1][j], longitud[i][j-1]);
        }
    }
}
```



En el algoritmo se presentan dos loops, uno que recorre toda la longitud de la cadena A y otro que recorre la cadena B, por lo tanto la complejidad será:

$$O(\text{longA} * \text{longB})$$

## Código de solución completo

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX(a,b) (a > b) ? a : b;    /* Macro para encontrar el mayo de dos valores */

int longestSub(int longA, int longB, char A[], char B[]) {
    int i, j;                        /* Variable para loops */
    int longitud[longA+1][longB+1]; /* Arreglo bidimensional que guarda las longitudes de ambas cadenas. */

    /* El siguiente for es para la solución bottom up, recorre las longitudes de ambas cadenas
    * de la cadena A y la cadena B */
    //
    for(i=0; i<longA; i++){
        for(j=0; j<longB; j++){
            if(i == 0 || j == 0) {
                longitud[i][j] = 0; /* En caso de que las longitudes sean 0 las cadenas no tendrán lcs */
            } else if(A[i-1] == B[j-1]) {
                longitud[i][j] = longitud[i-1][j-1] + 1; /* Si los últimos caracteres de las cadenas son iguales, se encuentra lcs de
                /* las cadenas anteriores de A y B */
            } else {
                longitud[i][j] = MAX(longitud[i-1][j], longitud[i][j-1]); /* Retornamos el mayor de las subcadenas de los elementos de A o de los elementos de B */
            }
        }
    }

    return longitud[longA][longB]; /* Retornamos el valor final que es el valor de la subsecuencia común mas larga */
}

int main()
{
    char A[10000]; /* Primer cadena */
    char B[10000]; /* Segunda cadena */

    scanf("%s", A); /* Leemos la primera cadena */
    scanf("%s", B); /* Leemos la segunda cadena */

    int longA = strlen(A); /* Obtenemos la longitud de la cadena A */
    int longB = strlen(B); /* Obtenemos la longitud de la cadena B */

    printf("%d", longestSub(longA, longB, A, B));

    return 0;
}

```

## Problema: ELIS – Easy Longest Increasing Subsequence

### Redacción

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.

### Captura de aceptación por juez

ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
28738556	2021-11-14 20:32:46	Brandon Meza	Easy Longest Increasing Subsequence	accepted edit ideone it	0.04	5.4M	C

### Explicación Algoritmo

```
int elis(int A[], int N, int auxNums[]){
    qsort(A,N,sizeof(int),cmpfunc);    /* Ordenamos nuestro arreglo origi
    int m = N;                          /* m será la longitud del arreglo
    int longitud[N+1][m+1];             /* Arreglo bidimensional para guar
    int i,j;                            /* Variables para loops
    /*
    /*Ordenando el arreglo original podemos pasar este problema comparandolo
    /*
    /*
    /*El siguiente for es para la solución bottom up, recorre la longitud de
    /*
    for(i=0; i<=N; i++){
        for(j=0; j<=m; j++){
            if(i == 0 || j == 0)
                longitud[i][j] = 0;
            else if(A[i-1] == auxNums[j-1])
                longitud[i][j] = longitud[i-1][j-1] + 1;
            else
                longitud[i][j] = MAX(longitud[i-1][j], longitud[i][j-1]);
        }
    }
    return longitud[N][m];
}
```

Este problema nos pide encontrar la subsecuencia creciente más larga, por ejemplo, si tenemos:

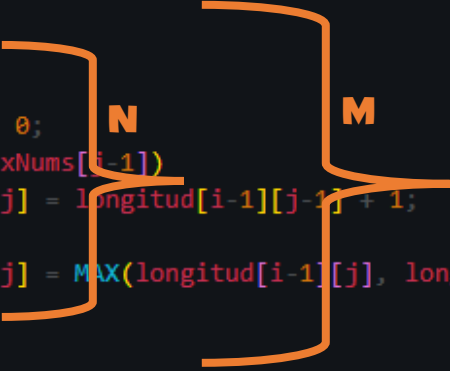
**[10,9,2,5,3,7,101,18]**

Podemos ver fácilmente que a partir del 2 hasta el 101 hay una subsecuencia creciente, sin embargo el 3 nos está estorbando ahí, al quitarlo obtenemos la LIS, siendo esta de 4: 2,5,7,101.

Para lograr lo anterior podemos simplificar este problema al primer problema mostrado que obtiene la LCS, para esto debemos tener dos arreglos, uno que será el original y el otro que será el original pero ordenado, de esta forma si obtenemos la subsecuencia común más larga, nos dará automáticamente la subsecuencia creciente más larga. Haciéndose el mismo procedimiento que el primero problema mostrado en el presente documento.

### Análisis de complejidad en cota $O()$

```
int elis(int A[], int N, int auxNums[]){
    qsort(A,N,sizeof(int),cmpfunc);           /* Ordenamos nuestro arreglo origi
    int m = N;                                /* m será la longitud del arreglo
    int longitud[N+1][m+1];                    /* Arreglo bidimensional para guar
    int i,j;                                   /* Variables para loops
    /*
    /*Ordenando el arreglo original podemos pasar este problema comparandolo
    /*
    /*
    /*El siguiente for es para la solución bottom up, recorre la longitud de
    /*
    for(i=0; i<=N; i++){
        for(j=0; j<=m; j++){
            if(i == 0 || j == 0)
                longitud[i][j] = 0;
            else if(A[i-1] == auxNums[j-1])
                longitud[i][j] = longitud[i-1][j-1] + 1;
            else
                longitud[i][j] = MAX(longitud[i-1][j], longitud[i][j-1]);
        }
    }
    return longitud[N][m];
}
```



Como podemos ver se realizan dos ciclos que van recorriendo la longitud del arreglo original, por lo tanto la complejidad será:

$$O(N^2)$$



## Código de solución completo

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX(a,b) (a > b) ? a : b;    /* Macro para encontrar el mayor de dos valores

/** Función para qsort
int cmpfunc (const void * a, const void * b) {
    return ( *(int*)a - *(int*)b );
}

intellis(int A[], int N, int auxnums[]){
    qsort(A,N,sizeof(int),cmpfunc);    /* Ordenamos nuestro arreglo original
    int m = N;                        /* m será la longitud del arreglo original
    int longitud[N+1][m+1];           /* Arreglo bidimensional para guardar las longitudes
    int i,j;                          /* Variables para loops

    *Ordenando el arreglo original podemos pasar este problema comparandolo con el mismo arreglo pero sin ordenar
    *El siguiente for es para la solución bottom up, recorre la longitud del arreglo

    for(i=0; i<=N; i++){
        for(j=0; j<=m; j++){
            if(i == 0 || j == 0)
                longitud[i][j] = 0;    /* En caso de que las longitudes sean 0 las cadenas no tendrán LCS
            else if(A[i-1] == auxnums[j-1])
                longitud[i][j] = longitud[i-1][j-1] + 1;    /* Si los últimos caracteres de las cadenas son iguales, se encuentra LCS de
            else
                longitud[i][j] = MAX(longitud[i-1][j], longitud[i][j-1]);    /* Las cadenas anteriores de A y B
            /* Retornamos el mayor de las subcadenas de los elementos de A o de los elementos de B
        }
    }
    return longitud[N][m];
}

int main()
{
    int N, i;
    scanf("%d", &N);    /* Variable para loop y N para longitud del arreglo
                        /* Leemos la longitud

    int A[N];
    int auxnums[N];
    for(i=0; i<N; i++){
        scanf("%d", &A[i]);    /* Arreglo con los números
        auxnums[i] = A[i];    /* Auxiliar para resolver el problema
        /* Leemos los números
        /* Se va copiando en el auxiliar
    }
    printf("%d",ellis(A, N, auxnums));
    return 0;
}

```

## Problema: The Knapsack Problem

### Redacción

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.

### Captura de aceptación por juez

ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
28738733	2021-11-18 22:12:07	Brandon Meza	The Knapsack Problem	accepted edit ideone it	0.04	5.4M	C

### Explicación Algoritmo

A continuación se presenta la captura del algoritmo usado para resolver este problema:

```
int knapsack(Item elementos[], int N, int S){
    int i, j;                               /* Varibales para loops
    int mochila[N+1][S+1];                  /* Variable que guarda La longitud de los items y la capacidad, la t

    /* Ciclos para construir la tabla en la manera bottom up
    for(i=0; i<=N; i++){
        for(j=0; j<=S; j++){
            if(i == 0 || j == 0)             /* Si no hay elementos será 0
                mochila[i][j] = 0;

            /*
            *Si el tamaño del articulo es de volumen menor o igual al que permite la mochila
            *entonces vamos a sumar el beneficio del articulo actual mas el beneficio que ya teniamos
            *recorriendo la tabla de acuerdo al tamaño del articulo actual
            *Se almacena en la casilla actual [i,j] el que maximice el beneficio de meter el articulo anterior o e

            */
            else if(elementos[i-1].tam <= j){
                mochila[i][j] = MAX(elementos[i-1].valor + mochila[i-1][j-elementos[i-1].tam], mochila[i-1][j]);
            } else{
                /* Si no cabe en la mochila el articulo el beneficio será el mismo para las demás casillas
                mochila[i][j] = mochila[i-1][j];
            }
        }
    }

    /* Retornamos el mayor beneficio resultante
    return mochila[N][S];
}
```

En primer lugar establecemos un arreglo bidimensional que nos servirá para representar la tabla, este será del tamaño de los artículos y de la capacidad que tiene la mochila más uno.

Posteriormente viene un ciclo for anidado, el primer for recorre cada ítem de la mochila (N) y el segundo la capacidad que puede tener la mochila (S).

Cabe aclarar que se usó la siguiente estructura para representar al artículo:

```
typedef struct item{  
    int tam;    //  
    int valor;  //  
} Item;
```

Dentro del segundo for se pregunta si  $i$  o  $j$  son 0, en caso de ser 0 el beneficio será 0, pues no habrá ningún artículo en la mochila.

En caso contrario y si el artículo que se quiere meter a la mochila tiene un tamaño menor al volumen que puede contener la mochila vamos a poner en la casilla actual  $[i,j]$  el valor del beneficio que maximice este beneficio de meter o no cierto artículo. Para esto se encuentra el máximo de dos beneficios; el primer beneficio será la suma del beneficio del artículo actual más el beneficio obtenido en la casilla al recorrer hacia la izquierda el tamaño o volumen del artículo actual, el segundo beneficio será el de meter el artículo anterior. El mayor beneficio de estos dos es el que se almacena.

En caso de que el tamaño del elemento sea mayor al soportado por la mochila se coloca el mismo beneficio en las casillas restantes de la tabla.

## Análisis de complejidad en cota $O()$

```
/* Ciclos para construir la tabla en la manera bottom up
for(i=0;i<=N;i++){
    for(j=0;j<=S;j++){
        if(i == 0 || j == 0)                /* Si no hay elementos será 0
            mochila[i][j] = 0;
        /*
        *Si el tamaño del articulo es de volumen menor o igual al que permite la mochila
        *entonces vamos a sumar el beneficio del articulo actual mas el beneficio que ya teniamos
        *recorriendo la tabla de acuerdo al tamaño del articulo actual
        *Se almacena en la casilla actual [i,j] el que maximice el beneficio de meter el articulo anterior o e
        */
        else if(elementos[i-1].tam <= j){
            mochila[i][j] = MAX(elementos[i-1].valor + mochila[i-1][j-elementos[i-1].tam], mochila[i-1][j]);
        } else{
            /* Si no cabe en la mochila el articulo el beneficio será el mismo para las demás casillas
            mochila[i][j] = mochila[i-1][j];
        }
    }
}

/* Retornamos el mayor beneficio resultante
return mochila[N][S];
```

Podemos ver que la complejidad radica en el llenado de la tabla, haciéndose dos ciclos for, uno recorriendo la cantidad de elementos y otro que recorre la capacidad de la mochila, siendo la complejidad:

$$O(N \cdot S)$$

## Código de solución completo

```
#include <stdio.h>
#include <stdlib.h>
#define MAX(a,b) (a > b)? a : b;    /* Macro para encontrar el mayo de dos valores
/* Estructura para el item de la mochila con un tamaño y valor
typedef struct item{
    int tam;        /* Tamaño del articulo
    int valor;      /* Beneficio del articulo
} Item;
int knapsack(Item elementos[], int N, int S){
    int i, j;                /* Varibales para loops
    int mochila[N+1][S+1];   /* Variable que guarda La Longitud de Los items y La capacidad, La tabla
    /* Ciclos para construir la tabla en la manera bottom up
    for(i=0;i<=N;i++){
        for(j=0;j<=S;j++){
            if(i == 0 || j == 0)                /* Si no hay elementos será 0
                mochila[i][j] = 0;
            /*
            *Si el tamaño del articulo es de volumen menor o igual al que permite la mochila
            *entonces vamos a sumar el beneficio del articulo actual mas el beneficio que ya teniamos
            *recorriendo la tabla de acuerdo al tamaño del articulo actual
            *Se almacena en la casilla actual [i,j] el que maximice el beneficio de meter el articulo anterior o el actual
            */
            else if(elementos[i-1].tam <= j){
                mochila[i][j] = MAX(elementos[i-1].valor + mochila[i-1][j-elementos[i-1].tam], mochila[i-1][j]);
            } else{
                /* Si no cabe en la mochila el articulo el beneficio será el mismo para las demás casillas
                mochila[i][j] = mochila[i-1][j];
            }
        }
    }
    /* Retornamos el mayor beneficio resultante
    return mochila[N][S];
}
int main()
{
    int i;                /* Variable para loops
    int S;                /* Variable para la capacidad
    int N;                /* Variable para la cantidad de items
    scanf("%d", &S);      /* Leemos la capacidad
    scanf("%d", &N);      /* Leemos la cantidad de items
    Item elementos[N];     /* Arreglo de items de longitud N
    /* Leemos Los elementos
    for(i=0;i<N;i++){
        scanf("%d",&elementos[i].tam);
        scanf("%d",&elementos[i].valor);
    }
    printf("%d", knapsack(elementos, N, S));
    return 0;
}
```

## Problema: AIBOHP - Aibohphobia

### Redacción

BuggyD suffers from AIBOHPHOBIA - the fear of Palindromes. A palindrome is a string that reads the same forward and backward.

To cure him of this fatal disease, doctors from all over the world discussed his fear and decided to expose him to large number of palindromes. To do this, they decided to play a game with BuggyD. The rules of the game are as follows:

BuggyD has to supply a string **S**. The doctors have to add or insert characters to the string to make it a palindrome. Characters can be inserted anywhere in the string.

The doctors took this game very lightly and just appended the reverse of **S** to the end of **S**, thus making it a palindrome. For example, if **S** = "fft", the doctors change the string to "fftfff".

Nowadays, BuggyD is cured of the disease (having been exposed to a large number of palindromes), but he still wants to continue the game by his rules. He now asks the doctors to insert the minimum number of characters needed to make **S** a palindrome. Help the doctors accomplish this task.

For instance, if **S** = "fft", the doctors should change the string to "tfft", adding only 1 character.

### Input

The first line of the input contains an integer **t**, the number of test cases. **t** test cases follow.

Each test case consists of one line, the string **S**. The length of **S** will be no more than 6100 characters, and **S** will contain no whitespace characters.

### Output

For each test case output one line containing a single integer denoting the minimum number of characters that must be inserted into **S** to make it a palindrome.

### Captura de aceptación por juez

ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
28753691	2021-11-18 00:16:08	Brandon Meza	Aibohphobia	accepted edit ideone it	0.48	97M	C

### Explicación Algoritmo

El algoritmo empleado es el mismo que el algoritmo para obtener la subsecuencia común más larga con una pequeña modificación al inicio. Esta modificación radica en tener otra cadena igual a la original ingresada, con la característica de estar invertida, de esta manera podemos obtener fácilmente el lcs que crea un palíndromo, al final hacemos una resta de la longitud de la cadena original menos la longitud del lcs de esta manera el resultado nos dará los caracteres necesarios para formar el palíndromo. El código del algoritmo se presenta a continuación.

```
int lcs(int longA, int longB, char A[], char B[]){
    int i, j;
    int longitud[longA+1][longB+1];
    /*
     *El siguiente for es para la solución bottom up, recorre las longitudes
     *de la cadena A y la cadena B
     */
    for(i=0; i<=longA; i++){
        for(j=0; j<=longB; j++){
            if(i == 0 || j == 0)
                longitud[i][j] = 0;
            else if(A[i-1] == B[j-1])
                longitud[i][j] = longitud[i-1][j-1] + 1;
            else
                longitud[i][j] = MAX(longitud[i-1][j], longitud[i][j-1]);
        }
    }

    return longA - longitud[longA][longB];
}
```

### Análisis de complejidad en cota $O()$

```
int longestSub(int longA, int longB, char A[], char B[]){
    int i, j;
    int longitud[longA+1][longB+1];
    /*
     *El siguiente for es para la solución bottom up, recorre las longitudes
     *de la cadena A y la cadena B
     */
    for(i=0; i<=longA; i++){
        for(j=0; j<=longB; j++){
            if(i == 0 || j == 0)
                longitud[i][j] = 0;
            else if(A[i-1] == B[j-1])
                longitud[i][j] = longitud[i-1][j-1] + 1;
            else
                longitud[i][j] = MAX(longitud[i-1][j], longitud[i][j-1]);
        }
    }
}
```

**LONGA** **LONGB**

En el algoritmo se presentan dos loops, uno que recorre toda la longitud de la cadena A y otro que recorre la cadena B, por lo tanto la complejidad será:

$$O(\text{longA} * \text{longB})$$

Podemos darnos cuenta de que al ser el mismo algoritmo que usamos para obtener el lcs la complejidad es la misma, sin embargo, al tener ambas cadenas la misma longitud la complejidad termina siendo:

$$O(\text{lonS}^2)$$



## Código de solución completo

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX(a,b) (a > b)? a : b;    /** Macro para encontrar el max de dos valores
/**Función para revertir una cadena usando punteros
char *rev(char *str)
{
    char *p1, *p2;

    if (! str || ! *str)
        return str;
    for (p1 = str, p2 = str + strlen(str) - 1; p2 > p1; ++p1, --p2){
        *p1 ^= *p2;
        *p2 ^= *p1;
        *p1 ^= *p2;
    }
    return str;
}

int lcs(int longA, int longB, char A[], char B[]){
    int i, j;                                /** Variable para loops
    int longitud[longA+1][longB+1];          /** Arreglo bidimensional que guarda las longitudes de ambas cadenas

    /** El siguiente for es para la solución bottom up, recorre las longitudes de ambas cadenas
    /** de la cadena A y la cadena B
    for(i=0; i<=longA; i++){
        for(j=0; j<=longB; j++){
            if(i==0 || j==0)
                longitud[i][j] = 0;          /** En caso de que las longitudes sean 0 las cadenas no tendrán lcs
            else if(A[i-1] == B[j-1])
                longitud[i][j] = longitud[i-1][j-1] + 1;    /** Si los últimos caracteres de las cadenas son iguales, se encuentra lcs de
                                                                /** las cadenas anteriores de A y B
            else
                longitud[i][j] = MAX(longitud[i-1][j], longitud[i][j-1]);    /** Retornamos el mayor de las subcadenas de los elementos de A o de los elementos de B
        }
    }

    return longA - longitud[longA][longB];    /** Retornamos el valor final que es el valor de la subsecuencia común mas larga
}

int main(int argc, char const *argv[])
{
    int i;                                /** Variable para loop
    int t;                                /** Variable para los casos de prueba
    scanf("%d", &t);                        /** Leemos los casos de prueba
    char S[6099];                          /** Variable para la cadena
    char saux[6099];                       /** Variable para cadena auxiliar

    /** Ciclo que va leyendo las cadenas de acuerdo al número de casos de prueba
    for(i=0; i<t; i++){
        scanf("%s", S);                    /** Leemos la cadena
        strcpy(saux, S);                   /** Copiamos la cadena leída en un auxiliar
        rev(saux);                         /** Revertimos la cadena auxiliar
        int longS = strlen(S);              /** Obtenemos la longitud de la cadena S
        int longSaux = strlen(saux);        /** Obtenemos la longitud de la cadena S

        printf("%d\n", lcs(longS, longSaux, saux, S));
    }

    return 0;
}

```