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

# PROBLEMAS DE PROGRAMACIÓN DINÁMICA

**LUIS FERNANDO RESENDIZ CHAVEZ**

**ANÁLISIS DE ALGORITMOS**
**PROFESOR EDGARDO ADRIÁN FRANCO MARTINEZ**

# Problema 1: Longest Common Subsequence

## Descripción

### Descripción

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

En esta ocasión tu trabajo es el siguiente:

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

### Entrada

La primera linea contendra la cadena A. En la segunda linea vendra la cadena B.

### Salida

La longitud de la subsecuencia común mas larga.

### Ejemplo

| Input | Output | Description |
|---|---|---|
| AGCT<br>AMGXTP | 3 | La subsecuencia comun mas larga es: $AGT$ |

## Explicación

El algoritmo funciona de la siguiente manera, primero hay que describir los casos base, cuando una cadena es vacía la respuesta es 0, cuando estamos en un estado de la dp donde el carácter actual de la primer cadena es igual a la segunda, podemos decir que ese carácter forma parte de la solución, y debemos seguir con el siguiente carácter, cuando son diferentes podemos decir que la solución óptima para ese estado es la mejor de las dos anteriores. La implementación que hice fue la bottom-up iterativa, donde cree una matriz de tamaño n x m donde n y m son las longitudes de la cadena a y b respectivamente, a cual sus valores inician en 0. Iteramos de izquierda a derecha, de arriba hacia abajo y en cada celda (o estado de la dp) evaluo si a[i-1] es igual a b[j-1], caso verdadero tomo la solución anterior, le sumo 1 a la respuesta y almaceno en la posición actual dp[i]j[j], como voy guardando las soluciones encontradas, estoy utilizando el concepto de memoizacion, ya que los resultados calculados me servirán para calcular los resultados posteriores. En caso de que los caracteres actuales sean distintos tendré que buscar en mis dos posibles soluciones anteriores cuál es la mejor y

almacenarla en mi estado actual, ya que hasta ese punto será la respuesta mas optima. Finalmente la respuesta se almacenará en la última posición de mi matriz, la cual contendrá la solución más optima para las dos cadenas dadas, después de revisar todos sus caracteres como subproblemas.

**Complejidad**

La complejidad temporal de este algoritmo es O(n*m) ya que se tiene que iterar sobre toda la matriz creada de tamaño n x m.
La complejidad espacial es O(n*m) por el espacio que ocupa la matriz de la dp.

**Captura del juez**

Username: lfrc0

| Submitted | GUID | Status | Percentage | Language | Memory | Runtime | Actions |
|---|---|---|---|---|---|---|---|
| 2021-01-17 23:45:52 | 17b5c64b | AC | 100.00% | cpp17-gcc | 5.36 MB | 0.06 s | |
| 2021-01-17 23:27:14 | f045643c | TLE | 20.00% | cpp17-gcc | 7.27 MB | >5.78 s | |

New submission

**Implementación**

```cpp
#include <bits/stdc++.h>
using namespace std;

#define len(s) int(s.size())

int main() {
  string a, b;
  cin >> a >> b;
  vector<vector<int>> dp(len(a)+1, vector<int> (len(b)+1,
0));
  for(int i = 1; i <= len(a); ++i) {
    for(int j = 1; j <= len(b); ++j) {
      if(a[i-1] == b[j-1])
        dp[i][j] = dp[i-1][j-1] + 1;
      else
        dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
    }
  }
  cout << dp[len(a)][len(b)] << '\n';
```

```
  return 0;
}
```

## Problema 2: ELIS - Easy Longest Increasing Subsequence

### Descripción

Given a list of numbers A output the length of the longest increasing subsequence. An increasing subsequence is defined as a set {i0 , i1 , i2 , i3 , ... , ik} such that 0 <= i0 < i1 < i2 < i3 < ... < ik < N and A[ i0 ] < A[ i1 ] < A[ i2 ] < ... < A[ ik ]. 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 <= N <= 10) the length of the list A.

Second line contains N numbers (1 <= each number <= 20), the numbers in the list A separated by spaces.

#### Output

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

#### Example

```
Input:
```

```
5
```

```
1 4 2 4 3
Output:
3
```

### Explicación

En este problema la solución funciona mediante sumas acumuladas de las soluciones encontradas en cada estado de la dp, es decir, iteramos sobre cada uno de los elementos en el arreglo dado y calculamos desde el inicio del arreglo hasta el elemento actual cual es la subsecuencia más larga de la siguiente manera: si el elemento actual es mayor a un elemento anterior (con índice más pequeño) significa que pueden ser parte de una solución, por lo tanto se suma en esa posicion la solucion anterior más 1, de esta forma estamos recordando la solución anterior y

agregando un elemento más cuando se cumpla dicha condición, finalmente obtendremos un arreglo con todos los valores de las subsecuencias del arreglo dado sin traslapes, únicamente hay que buscar cual es la mayor y retornarla, C++ proporciona una función llamada max_element(<contenedor>) el cual devuelve un apuntador al elemento mayor de un contenedor llámese arreglo, vector, matriz, pila, cola, etc.

**Complejidad**

La complejidad de este algoritmo es O(n*n) porque cada que avanzamos una posición a la derecha hay que comparar con todos los elementos anteriores y encontrar la solución más óptima.
La complejidad espacial de es O(n) puesto que necesitamos de un arreglo extra del mismo tamaño para almacenar las sumas acumuladas de los tamaños de las subsecuencias.

**Captura del juez**

Username: lfrc0

| ID | DATE | USER | PROBLEM | RESULT | TIME | MEM | LANG |
|---|---|---|---|---|---|---|---|
| 27294184 | 2021-01-18 07:09:14 | lfrc0 | Easy Longest Increasing Subsequence | accepted edit ideone it | 0.00 | 4.9M | CPP14 |

**Implementación**

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> a(n), dp(n, 1);
  for(int& val: a)
    cin >> val;
  for(int i = 1; i < n; i++)
    for(int j = 0; j < i; j++)
      if(dp[i] < (dp[j] + 1 ) && a[i] > a[j])
        dp[i] = dp[j] + 1;

  cout << *max_element(dp.begin(), dp.end()) << '\n';
  return 0;
}
```

## Problema 3: Bacterias

Not solved

## Problema 4: Cutting Sticks

### Descripción

You have to cut a wood stick into pieces. The most affordable company, The Analog Cutting Machinery, Inc. (ACM), charges money according to the length of the stick being cut. Their procedure of work requires that they only make one cut at a time.

It is easy to notice that different selections in the order of cutting can led to different prices. For example, consider a stick of length 10 meters that has to be cut at 2, 4 and 7 meters from one end. There are several choices. One can be cutting first at 2, then at 4, then at 7. This leads to a price of $10 + 8 + 6 = 24$ because the first stick was of 10 meters, the resulting of 8 and the last one of 6. Another choice could be cutting at 4, then at 2, then at 7. This would lead to a price of $10 + 4 + 6 = 20$, which is a better price.

Your boss trusts your computer abilities to find out the minimum cost for cutting a given stick.

#### Input

The input will consist of several input cases. The first line of each test case will contain a positive number $l$ that represents the length of the stick to be cut. You can assume $l < 1000$. The next line will contain the number $n$ ($n < 50$) of cuts to be made.

The next line consists of $n$ positive numbers $c_i$ ($0 < c_i < l$) representing the places where the cuts have to be done, given in strictly increasing order.

An input case with $l = 0$ will represent the end of the input.

#### Output

You have to print the cost of the optimal solution of the cutting problem, that is the minimum cost of cutting the given stick. Format the output as shown below.


UVa Online Judge

UVa Online Judge

uva.onlinejudge.org

### Explicación

Este algortimo resulta mas facil hacerlo de la forma top-down recursiva, veamos los casos base: en caso de que el estado actual no haya sido calculado, tenemos que realizar la busqueda binaria sobre el arreglo de cortes, despues calcular cual es el minimo de ellos y guardarlo en el estado actual de la dp, en caso de que el estado ya haya sido calculado simplemente lo retornamos, esto sucedera hats que se calculen todos los posibles cortes en el palo respecto al costo minimo.

### Complejidad

La complejidad temporal de este algoritmo es O(n*longn) ya que para calcular cada estado nos cuesta O(logn) y hay n estados.
La complejidad espacial es O(n*n) para crear la matriz de costos de cada estado y un O(n) acotado por la complejidad anterior para guardar los cortes actuales de cada estado.

**Captura del juez**

Username: lfrc0

My Submissions

| # | Problem | Verdict | Language | Run Time | Submission Date |
|---|---|---|---|---|---|
| 25963934 | 10003 Cutting Sticks | Accepted | C++11 | 0.190 | 2021-01-18 06:47:28 |
| 25963701 | 10003 Cutting Sticks | Accepted | C++11 | 0.200 | 2021-01-18 06:34:19 |

<< Start < Prev 1 Next > End >>

**Implementación**

```
#include <bits/stdc++.h>
using namespace std;

const int mx = 55;
int dp[mx][mx];

int cut(int left, int right, const vector<int> & cuts){
    int & current = dp[left][right];

    if (current == -1){
        current = 0;
        for (int i = left + 1; i < right; ++i){
            if (current == 0)
                current = cut(left, i, cuts) + cut(i, right,
cuts) + cuts[right] - cuts[left];
            else
                current = min(current, cut(left, i, cuts) +
cut(i, right, cuts) + cuts[right] - cuts[left]);
        }
    }
    return current;
}

int main(){
    int L;
    while (cin >> L, L) {
        int numCuts;
        cin >> numCuts;
```

```
        memset(dp, -1, sizeof dp);

        vector<int> cuts(numCuts + 2);
        cuts[0] = 0;
        for (int i = 1; i <= numCuts; ++i){
            cin >> cuts[i];
        }

        cuts[numCuts + 1] = L;

        cout << "The minimum cutting is " << cut(0,
cuts.size() - 1, cuts) << ".\n";
    }
}
```