Instituto Politécnico Nacional

Escuela Superior de Cómputo

DISEÑO DE SOLUCIONES CON PROGRAMACIÓN DINÁMICA

Ejercicio 06

Análisis de Algoritmos

M. en C. Edgardo Adrián Franco Martínez

Grupo: 3CM3

Fecha: 12 / Mayo / 2018

Alumno:

Calva Hernández José Manuel 2017630201

Índice

Longest Common Subsequence

Redacción del ejercicio

Longest Common Subsequence

Points	16.21	Memory limit	32MB
Time limit (case)	0.5s	Time limit (total)	7.5s

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

Entrada	Salida	Descripción
AGCT AMGXTP	3	La subsecuencia comun mas larga es: AGT

Límites

- $1 \leq |A| \leq 10^3$
- $1 \leq |B| \leq 10^3$

Captura de aceptación

Submitted	GUID	Status	Percentage	Language	Memory	Runtime	Details
2018-06-10 12:29:36	4af6b63d	Accepted	100.00%	cpp11	5.29 MB	0.02 s	

Explicación de la solución

Usaremos una implementación Botton Up en este problema, ya que iniciaremos una matriz para mantener las coincidencias que haya entre las dos cadenas. El primer borde lo rellenaremos con 0's debido a que no llevaremos ninguna coincidencia al inicio, a continuación haremos un recorrido sobre ambas cadenas, si existe alguna coincidencia le sumaremos una unidad al acumulado en la diagonal anterior; sin embargo, si no existe coincidencia vamos a revisar los lados anteriores y anotaremos la mayor cantidad entre ellos, esto para mantener memoria de la mayor coincidencia que haya hasta el momento.

Código

```
#include < bits / stdc++.h >

using namespace std;

vector < vector < int > > vOptimize;

int LCS(string & s1, string & s2, int m, int n) {
    int nRow, nColumn;
    for (nRow = 0; nRow <= m; nRow++) {
        for (nColumn = 0; nColumn <= n; nColumn++) {
            if (nRow == 0 || nColumn == 0) {
                vOptimize[nRow][nColumn] = 0;
            } else if (s1[nRow - 1] == s2[nColumn - 1]) {
                vOptimize[nRow][nColumn] = vOptimize[nRow - 1][nColumn - 1] + 1;
            } else {
                vOptimize[nRow][nColumn] = max(vOptimize[nRow - 1][nColumn], vOptimize[nRow][nColumn - 1]);
            }
        }
    }
    return vOptimize[m][n];
}

int main(int argc, char const * argv[]) {
    string s1, s2;
    cin >> s1 >> s2;
    vOptimize.resize(s1.size() + 1, vector < int > (s2.size() + 1));
    cout << LCS(s1, s2, s1.size(), s2.size()) << endl;
    return 0;
}
```

Análisis de complejidad

Omitiendo la parte del main debido a que sólo es entrada y salida de datos, analizaremos la función LCS, podemos observar que son dos for anidados que en su interior únicamente tienen asignaciones y comparaciones, incluída la función max, lo que nos lleva a considerarlas constanes. Debido a que ambos ciclos recorren las cadenas de texto de principio a fin, su complejidad vendría dada por:

$$f_t(n,m) = O(nm)$$

Donde n es la longitud de la cadena 1 y m es la longitud de la cadena m.

ELIS - Easy Longest Increasing Subsequence

Redacción del ejercicio

ELIS - Easy Longest Increasing Subsequence

no tags

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
```

Captura de aceptación

ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
21813826	2018-06-10 19:53:31	akotadi	Easy Longest Increasing Subsequence	**accepted** edit ideone it	0.00	16M	CPP14

Explicación de la solución

LIS es un algoritmo ya bien conocido que está implementado en esta versión de una forma Top Down, esto debido a que recorrerá toda la cadena de números manteniendo memoria de las máximas secuencias de incremento hasta el momento, esto lo logra revisando los números que se encuentran detrás del actual, checando si es menor al actual y en caso de serlo lo que hará es compararlo con el resultado que llevamos guardado en la memoria que nos indica la cantidad de números menores al actual hasta ese momento, en caso de ser mayor, actualizaremos esa cantidad en la memoria.

Código

```
1.  #include < bits / stdc++.h >
2.
3.  using namespace std;
4.
5.  int LIS(vector < int > & vNumbers) {
6.      if (vNumbers.size() < 2) return vNumbers.size();
7.      int result = 0;
8.      vector < int > vOptimize(vNumbers.size(), 1);
9.      for (int i = 1; i < vNumbers.size(); ++i) {
10.         int aux = 1;
11.         for (int j = 0; j < i; ++j) {
12.             if (vNumbers[j] < vNumbers[i])
13.                 aux = max(aux, vOptimize[j] + 1);
14.         }
15.         vOptimize[i] = aux;
16.         result = max(result, aux);
17.     }
18.     return result;
19. }
20.
21. int main(int argc, char const * argv[]) {
22.     int n;
23.     cin >> n;
24.     vector < int > vNumbers(n);
25.     for (int i = 0; i < n; ++i) {
26.         cin >> vNumbers[i];
27.     }
28.     cout << LIS(vNumbers) << endl;
29.     return 0;
30. }
```

Análisis de complejidad

Omitiremos la función main debido a que es una entrada y salida de datos, irrelevante para el cálculo, así que enfocándonos en la función LIS, podemos observar que tenemos dos for anidados que son la parte medular de ello debido a que las demás operaciones son simples comparaciones y asignaciones. El primer for recorrerá casi por completo la secuencia de números y el segundo dependerá del primero, pero en el peor de los casos hará un recorrido por toda la secuencia de número menos un elemento, pero haciendo cota de ello nos resulta en la siguiente función de complejidad:

$$f_t(n) = O(n^2)$$

KNAPSACK - The Knapsack Problem

Redacción del ejercicio

KNAPSACK - The Knapsack Problem

no tags

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 <= S <= 2000). You also have N (1<= N <= 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 like - the total maximum value from the best choice of items for your trip.

Example

```
Input:
4 5
1 8
2 4
3 0
2 5
2 3


Output:
13
```

Captura de aceptación

ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
21813981	☐	2018-06-10 20:35:18	The Knapsack Problem	**accepted** edit ideone it	0.00	19M	CPP14

Explicación de la solución

La solución es de la forma Top Down, básicamente viene dada de una fuerza bruta ya que probaremos todas las posibilidades dadas por tomar o no tomar cada uno de los posibles objetos que nos den, sin embargo, nos encontraremos con que varias de las posibilidades se repiten y ello nos permite memorizar parte de las soluciones para optimizar el algoritmo y evitar que calcule opciones repetidas. Entonces la solución se reduce a calcular todos los caminos tomando o no tomando cada uno de los objetos y memorizando situaciones similares donde nos quede la misma cantidad de espacio con el mismo objeto.

Esta recursividad la detendremos cuando ya no haya más objetos que valorar, o bien, ya hayamos calculado el valor previamente; en caso contrario procederemos a calcularlo y memorizarlo para futuras referencias.

Código

```
1.  #include < bits / stdc++.h >
2.
3.  using namespace std;
4.
5.  const int INF = -(1 << 30);
6.
7.  vector < vector < int > > vOptimize;
8.  vector < pair < int, int > > vItems;
9.
10. int Knapsack(int pos, int avaliable) {
11.     if (pos == -1) {
12.         return 0;
13.     }
14.     if (vOptimize[pos][avaliable] != -1) {
15.         return vOptimize[pos][avaliable];
16.     }
17.     int aux;
18.     (vItems[pos].first <= avaliable) ? (aux = Knapsack(pos - 1, avaliable - vItems[pos].first) + vItems[pos].
    second) : (aux = INF);
19.     return vOptimize[pos][avaliable] = max(Knapsack(pos - 1, avaliable), aux);
20. }
21.
22. int main(int argc, char const * argv[]) {
23.     int s, n;
24.     cin >> s >> n;
25.     vOptimize.resize(n, vector < int > (s + 1, -1));
26.     vItems.resize(n);
27.     int weight, value;
28.     for (int i = 0; i < n; i++) {
29.         cin >> weight >> value;
30.         vItems[i] = make_pair(weight, value);
31.     }
32.     cout << Knapsack(n - 1, s) << endl;
33.     return 0;
34. }
```

Análisis de complejidad

Omitiremos la función main debido a que es una entrada y salida de datos, irrelevante para el cálculo, así que enfocándonos en la función Knapsack que es posible apreciar que es recursiva, por ello analizaremos directamente la parte donde es llamada recursivamente debido a que las demás operaciones las consideraremos constantes.

En el peor de los casos, supondremos que la función recursiva es llamada siempre dos veces en el algoritmo, una simulando que se toma el objeto y otra donde no se toma, sin embargo, en ambos casos continuamos la secuencia de búsqueda así que puede generalizarse a la misma llamada recursiva 2 veces.

La operación de query es un método recursivo que además tiene recurrencias no líneas al ir dividiendo la *n* en segmentos de mitades, sus operaciones quedarían distribuidas de la siguiente manera:

$$T(0) = 1$$

$$T(n) = 1 + 2T(n-1)$$

$$T(n) - 2T(n-1) = 1$$

$$(x-2)(x-1) = 0$$

$$x_1 = 2, x_2 = 1$$

$$T(n) = c_1 + c_2 2^n$$

Sistema de ecuaciones

$$T(0) = c_1 + c_2 = 1$$

$$T(1) = c_1 + 2c_2 = 3$$

$$\therefore c_1 = -1, c_2 = 2$$

$$\therefore T(n) = -1 + (2)(2^n) = -1 + 2^{n+1}$$

$$\therefore f_t(n) = O(2^n)$$