Práctica 4

Programación dinámica

Algorítmica

segfault

Celia Arias Martínez

Miguel Ángel Fernández Gutiérrez

Sergio Quijano Rey

Lucía Salamanca López

UNIVERSIDAD DE GRANADA

Práctica 4

Programación dinámica

Algorítmica

segfault

Celia Arias Martínez

Miguel Ángel Fernández Gutiérrez

Sergio Quijano Rey

Lucía Salamanca López

UNIVERSIDAD
DE GRANADA

Índice

I | Introducción

En la **práctica 4**, de programación dinámica, teníamos que elegir entre los dos problemas propuestos:

- **Viajante de comercio:** dado un conjunto de ciudades y una matriz con las distancias entre ellas encontrar el camino más corto que las recorra todas una vez.
- **Subsecuencia de caracteres más larga:** encontrar la máxima subsecuencia de caracteres comunes.

Nosotros hemos decidido estudiar el segundo problema.

Objetivo de esta práctica

En esta práctica, pretenderemos apreciar la utilidad de los algoritmos de programación dinámica para encontrar la solución óptima en problemas que podemos dividir en subproblemas superpuestos.

Para ello, daremos una solución al problema de la *subsecuencia de caracteres más larga* y compararemos la eficiencia de este algoritmo respecto al algoritmo de *fuerza bruta* y *recurrencias*.

II | Desarrollo

Máxima subsecuencia de caracteres (LCS)

> Dadas dos secuencias de caracteres, encontrar la máxima subsecuencia de caracteres común que aparecen en ambas cadenas de izquierda a derecha (no necesariamente de forma contigua).

1 Enfoque por programación dinámica

Aplicamos programación dinámica en cuatro fases:

1. Verificación de la naturaleza n-etápica del problema.
2. Verificación del principio de optimalidad de Bellman.
3. Planteamiento de una recurrencia.
4. Cálculo de la solución.

1.1 Naturaleza n-etápica

El resultado es consecuencia de una sucesión de decisiones: en la etapa n debemos elegir qué cadena de caracteres dejamos fija y qué puntero movemos hacia la derecha.

Una solución optimal de decisiones será aquella que maximice la función objetivo, es decir, aquella que proporcione la subsecuencia de caracteres más larga.

1.2 Verificación del principio de optimalidad de Bellman

El **principio de optimalidad de Bellman (POB)**, es una condición necesaria para la optimalidad en programación dinámica.

Tenemos que escribir el valor del problema en la etapa n en función de los valores en la etapa 1 y las decisiones óptimas tomadas en las etapas anteriores. De esta forma conseguimos obtener subproblemas más simples, ya que no tendremos que volver a calcular todas las soluciones, pues tenemos asegurado que utilizamos solo las óptimas.

En nuestro problema la función a maximizar es el número de caracteres en común en las dos subsecuencias. Tendremos, por tanto, en la etapa n:

$$f_n(i,j) = \text{máx}\{f(i-1,j), f(i,j-1)\} + \delta_{ij}$$

donde δ_{ij} es la función que dados dos índices, cada uno respectivo a una subcadena, devuelve uno si coinciden y cero en caso contrario.

El caso base es:

$$f_1(i,j) = \text{máx}\{\delta_{ij}\} = \delta_{ij}$$

Es obvio que la solución proporcionada en la etapa n es optimal, pues estamos calculando el máximo de dos funciones optimales. De igual forma, si $f_{n-1}(i-1,j)$ ó $f_{n-1}(i,j-1)$ no son optimales en el paso anterior hubiéramos calculado $f'_{n-1}(i-1,j)$ ó $f'_{n-1}(i,j-1)$.

Por tanto, podemos decir que el problema cumple el principio de optimalidad de Bellman.

1.3 Planteamiento de recurrencia

La recurrencia planteada ha sido la siguiente:

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1,j-1] \\ \text{máx}\{LCS[i-1,j], LCS[i,j-1]\}) \end{cases}$$

Si encontramos dos caracteres que coincidan aumentamos en uno el tamaño de la solución parcial que tenemos y llamamos a la función moviendo a la izquierda los dos punteros. Si no coinciden volvemos a llamar a la función moviendo el puntero una posición a la izquierda sobre la cadena primera, y después volvemos a llamarla moviendo el puntero de la segunda cadena. De los valores obtenidos cogemos siempre el máximo.

De esta forma conseguimos asegurar el principio de optimalidad: si estamos en la etapa n, las decisiones tomadas hasta la etapa $n-1$ son óptimas.

1.4 Cálculo de la solución

Para ello, hemos planteado el siguiente algoritmo:

```
vector<vector<int> > constructLCSMat(string str1, string str2) {
    vector<vector<int> > lcs_mat(str1.size() + 1, vector<int>(str2.size() + 1, 0));

    for ( int i = 1; i <= str1.size(); i++ ) {
        for ( int j = 1; j <= str2.size(); j++ ) {
            if ( str1[i-1] == str2[j-1] )
                lcs_mat[i][j] = 1 + lcs_mat[i-1][j-1];
            else
                lcs_mat[i][j] = max(lcs_mat[i-1][j], lcs_mat[i][j-1]);
        }
    }

    return lcs_mat;
}
```

```
string getLCS(string str1, string str2) {
    vector<vector<int> > lcs_mat = constructLCSMat(str1, str2);
    string word;

    int i = str1.size(), j = str2.size();

    while ( j > 0 ) {
        j--;
        if ( lcs_mat[i][j] != lcs_mat[i][j+1] ) {
            word = str2[j] + word;
            i--;
        }
    }

    return word;
}
```

En el que utilizamos las siguientes funciones y estructuras de datos:

- Una función **`constructLCSMat`**, donde creamos la matriz que utilizaremos para encontrar el número de caracteres y las letras que conforman la máxima subsecuencia en común. Con dos bucles `for` anidados recorremos las dos palabras, comprobando si coinciden las letras y consultando los datos de la matriz ya escritos anteriormente, como podemos ver en la recurrencia especificada antes.

  Cuando la matriz esté calculada, la devolvemos. Esta matriz será usada por la función `getLCS`.

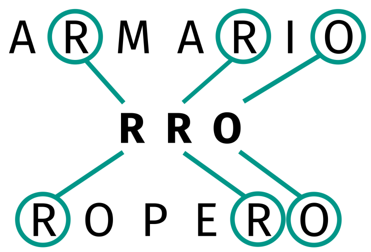
- La función **`getLCS`**, que recibe como argumentos las dos palabras estudiadas. Dentro llamamos a la función `constructLCSMat` para construir la matriz. Con el bucle `while` recorremos la matriz de derecha a izquierda por filas, empezando por la última fila y la última columna: cuando veamos que el valor respecto a la columna anterior cambia, subimos de fila y guardamos la letra que corresponda a esa posición, ya que significará que hemos sumado uno, y por tanto que los valores en la palabra de i y j coinciden.

1.5 Un ejemplo ilustrativo

Para ilustrar lo que hacen nuestras funciones, tomamos como ejemplo las secuencias de caracteres:

`str1`="ropero" `str2`="armario"

Evidentemente, la solución es **"rro"**.



La función `getLCS` llamaría en primer lugar, a `constructLCSMat`, que generaría la siguiente matriz:

$$\begin{array}{cc}
 & \begin{array}{cccccccc} & a & r & m & a & r & i & o \end{array} \\
\begin{array}{c} \\ r \\ o \\ p \\ e \\ r \\ o \end{array} & \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 2 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 2 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 2 \\
0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\
0 & 0 & 1 & 1 & 1 & 2 & 2 & 3
\end{pmatrix}
\end{array}$$

Esta matriz, será procesada en `getLCS`. En su recorrido, obtendremos la subsecuencia más corta:

$$\begin{array}{cc}
 & \begin{array}{cccccccc} & a & r & m & a & r & i & o \end{array} \\
\begin{array}{c} \\ r \\ o \\ p \\ e \\ r \\ o \end{array} & \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{1} & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & \mathbf{1} & 1 & 1 & 1 & 1 & 2 \\
0 & 0 & \mathbf{1} & 1 & 1 & 1 & 1 & 2 \\
0 & 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 & 1 & 2 \\
0 & 0 & 1 & 1 & 1 & \mathbf{2} & \mathbf{2} & 2 \\
0 & 0 & 1 & 1 & 1 & 2 & 2 & \mathbf{3}
\end{pmatrix}
\end{array}$$

Teniendo como resultado la subsecuencia "rro" (rope**ro**, a**r**ma**r**i**o**).

2 Análisis empírico

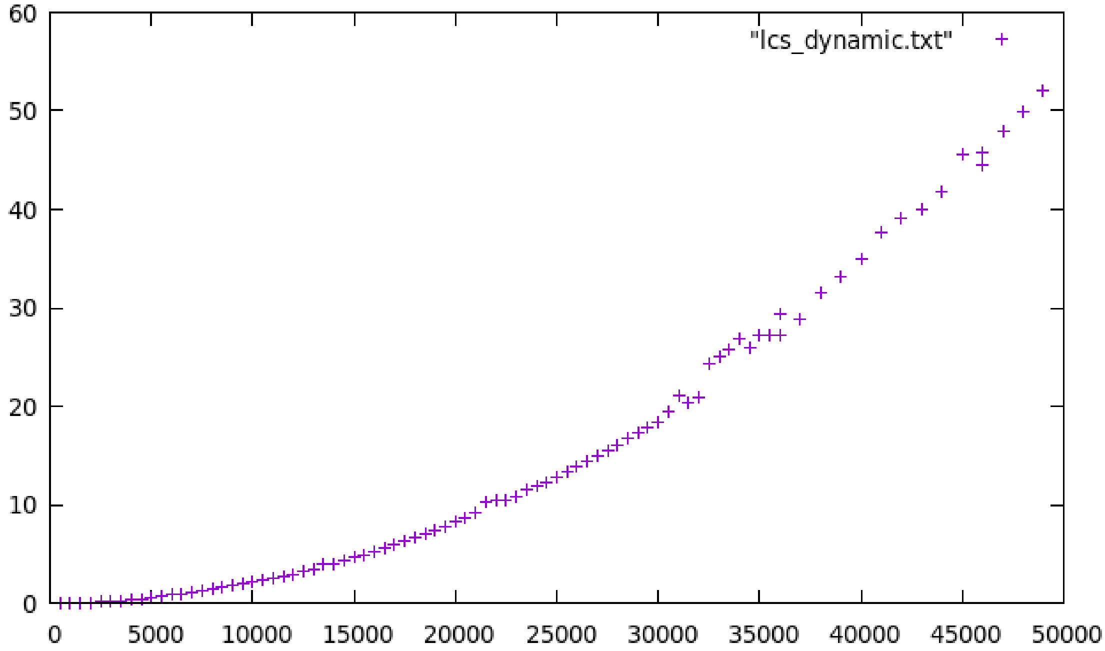
Para el análisis de tiempo, hemos creado un código que ejecute, sobre un mismo conjunto de datos aleatorio, dos algoritmos: `dynamic`, el explicado anteriormente (que hace uso de programación dinámica) y `brute`, por fuerza bruta usando la recursividad, directamente.

Los tamaños de prueba para ejecutar el algoritmo han sido:

- desde 500 hasta 3600 en saltos de 500
- desde 3600 hasta 56000 en saltos de 10000.
- hasta 49000 en saltos de 1000.

A su vez cada iteración la hemos hecho 100 veces y hemos calculado la media, con el fin de eliminar los mejores y peores casos.

Gráfico 2.1. Datos empíricos: algoritmo dinámico



Los datos de las gráficas se encuentran en la ***sección IV: Anexo I*** de este documento.

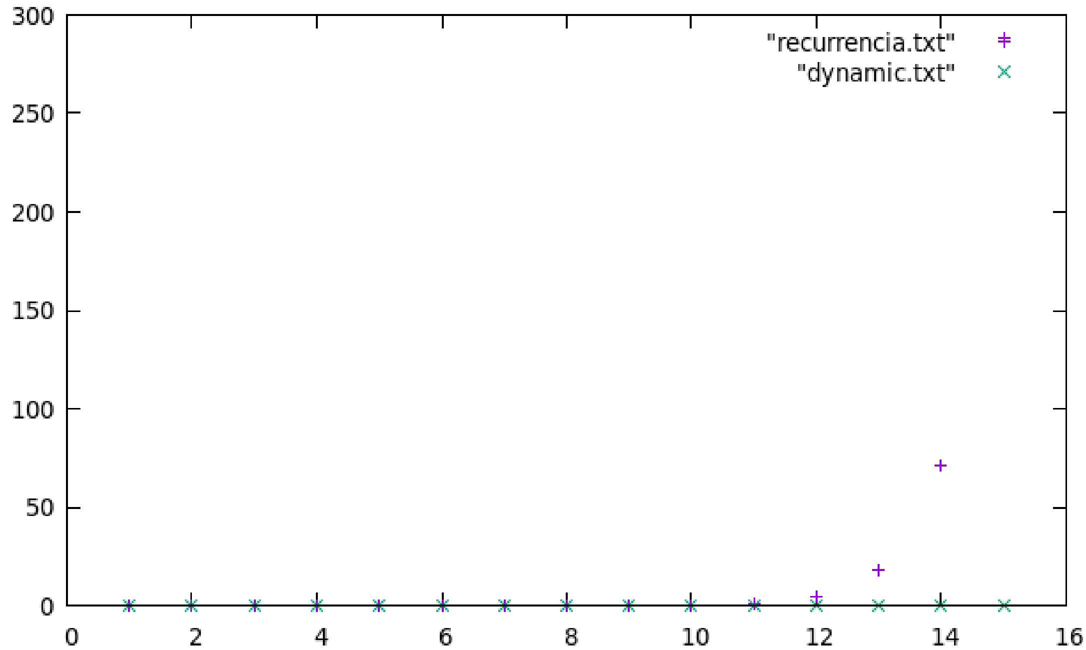
Podemos decir que la gráfica crece de forma cuadrática, por lo que podemos ejecutarlo para valores suficientemente grandes. Esto se debe que tenemos que generar la matriz, que es de complejidad O(n^2). Puede parecer una complejidad alta, pero comparada con el coste de llamar a la función recursivamente es aceptable. Probaremos lo dicho empíricamente en la sección siguiente.

3 Comparación de enfoques

Hemos comparado nuestro algoritmo con un algoritmo de *recurrencia* simple. Como *recurrencia* solo funciona para tamaños muy pequeños hemos recogido datos con $n = 17$.

Podemos observar en el gráfico 6.2 que a partir de $n = 13$ se aprecia una diferencia notable en los tiempos de ejecución: con estos valores nuestro algoritmo parece que se mantiene constante mientras que el algoritmo de fuerza bruta, por recurrencia, crece exponencialmente.

Gráfico 3.1. Contraste de datos empíricos: algoritmo dinámico vs. fuerza bruta



III Conclusiones

Con esta práctica, hemos aprendido a crear algoritmos de programación dinámica para resolver problemas encadenados que podemos optimizar.

Hemos comprobado las cuatro condiciones que tiene que reunir un problema para poder resolverse con programación dinámica.

Así mismo, hemos observado la utilidad de este tipo de algoritmos: al contrario de algoritmos simples de recurrencia en cada iteración no hay que volver a comprobar todas las soluciones posibles, ya que hemos guardado las mejores soluciones anteriores (en este caso, en una tabla o matriz).

Esto es de especial relevancia en los que queremos encontrar la solución óptima tomando una serie de decisiones: el tiempo será peor que en algoritmos *greedy*, sin embargo nos aseguramos de que la solución proporcionada es la mejor.

IV | Anexos

Anexo I. Códigos

Programación dinámica: `lcs_dynamic.hpp`

```
/**
 * @brief Longest Common Subsequence: encontrar subcadena mas larga - Programacion Dinamica
 * @file lcs_dynamic.cpp
 * @author segfault
 */
#include <vector>
#include <string>
#include <iostream>

using namespace std;


namespace dynamic {

    /**
     * @brief Calcula el maximo de dos enteros
     * @param a: uno
     * @param b: el otro
     * @return el maximo de ambos
     */
    int max(int a, int b) {
        if (a > b)
            return a;
        else
            return b;
    }

    /**
     * @brief Funcion para imprimir la matriz LCS, incluyendo caracteres
     * @param mat: matriz LCS
     * @param str1: primera palabra
     * @param str2: segunda palabra
     */
    void printmat(vector<vector<int> > mat, string str1, string str2) {
        // cabecera: segunda palabra
        cout << "     ";
        for ( int i = 0; i < str2.size(); i++ )
            cout << str2[i] << ' ';
        cout << endl;

        for ( int i = 0; i < mat.size(); i++ ) {
            // primera palabra
            if ( i == 0 )
                cout << "  ";
```

```
        else
            cout << str1[i-1] << ' ';

        // imprimir elementos de la matriz
        for ( int j = 0; j < mat[0].size(); j++ )
            cout << mat[i][j] << " ";

        cout << endl;
    }
}

/**
 * @brief Crea la matriz LCS
 * @param str1: primera palabra
 * @param str2: segunda palabra
 * @return matriz LCS de las palabras str1, str2
 */
vector<vector<int> > constructLCSMat(string str1, string str2) {
    vector<vector<int> > lcs_mat(str1.size() + 1, vector<int>(str2.size() + 1, 0));

    for ( int i = 1; i <= str1.size(); i++ ) {
        for ( int j = 1; j <= str2.size(); j++ ) {
            if ( str1[i-1] == str2[j-1] )
                lcs_mat[i][j] = 1 + lcs_mat[i-1][j-1];
            else
                lcs_mat[i][j] = max(lcs_mat[i-1][j], lcs_mat[i][j-1]);
        }
    }

    //printmat(lcs_mat, str1, str2);

    return lcs_mat;
}

/**
 * @brief Obtiene la LCS (longest common subsequence)
 * @param str1: primera palabra
 * @param str2: segunda palabra
 * @return la subcadena comun mas larga
 */
string getLCS(string str1, string str2) {
    vector<vector<int> > lcs_mat = constructLCSMat(str1, str2);
    string word;

    int i = str1.size(), j = str2.size();

    while ( j > 0 ) {
        j--;
        if ( lcs_mat[i][j] != lcs_mat[i][j+1] ) {
            word = str2[j] + word;
            i--;
        }
    }

    return word;
```

```
    }

}
```

Fuerza bruta: `lcs_brute.hpp`

```
/**
 * @brief Longest Common Subsequence: encontrar cadena mas larga - version Fuerza Bruta
 * @file lcs_brute.cpp
 * @author segfault
 * @note Tener mucho cuidado al ejecutar, para valores muy grandes usar valgrind
 *       para que la recursividad no acabe con vuestro ordenador
 */
#include <vector>
#include <string>

using namespace std;


namespace brute {

    /**
     * @brief Devuelve todas las subsecuencias de un texto
     * @param text: el texto del que queremos obtener todas las subsecuencias
     * @return un vector con todas las subsecuencias
     */
    vector<string> getSubsequences(string text) {
        if ( text.size() > 1 ) {
            vector<string> subsequences;

            string slice = string(text.begin() + 1, text.end());

            for ( auto sub : getSubsequences(slice) ) {
                string current1 = text[0] + sub;
                string current2 = "" + sub;
                subsequences.push_back(current1);
                subsequences.push_back(current2);
            }

            return subsequences;
        } else {
            vector<string> subsequences;

            subsequences.push_back(text);
            subsequences.push_back("");

            return subsequences;
        }
    }

    /**
     * @brief Calcula una porcion de un string
     * @param text: el texto del que quiero obtener una porcion suya
     * @param start: la posicion de inicio
     * @param end: la posicion final
```

```
     * @return text[start:end]
     * */
    string slice(string text, int start, int end) {
        string current;

        for(int i = start; i <= end; i++){
            current.push_back(text[i]);
        }

        return current;
    }

    /**
     * @brief Encuentra por fuerza bruta la subsecuencia comun mas larga
     * @param text1: el primer texto
     * @param text2: el segundo texto
     * @return la subsecuencia comun mas larga,
     *          "" si no hay ninguna letra en comun
     * */
    string getLCS(string text1, string text2) {
        string largest = "";

        for(auto sub1 : getSubsequences(text1)){
            for(auto sub2 : getSubsequences(text2)){
                if(sub1 == sub2 && sub1.size() > largest.size()){
                    largest = sub1;
                }
            }
        }

        return largest;
    }

}
```

Programa para ejecutar: `lcs.cpp`

```
/**
 * @brief Interfaz para LCS
 * @file lcs.cpp
 * @author segfault
 */

#include "lcs_dynamic.hpp"
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char** argv) {
    if ( argc != 3 ) {
        cerr << "[ERROR] - Debe usar este programa como: [nombre-programa] <str1> <str2>\n";
        exit(EXIT_FAILURE);
    }

```

```
    string str1 = argv[1], str2 = argv[2];

    vector<vector<int> > mat = dynamic::constructLCSMat(str1, str2);
    dynamic::printmat(mat, str1, str2);

    string result = dynamic::getLCS(str1, str2);
    cout << "LCS: " << result << endl;

    exit(EXIT_SUCCESS);
}
```

Medición de tiempos: `measure_separate.cpp`

```
/**
 * @brief Medicion de tiempos para LCS
 * @filename measure.cpp
 * @author segfault
 */
#include "lcs_dynamic.hpp"
#include "lcs_brute.hpp"
#include <iostream>
#include <fstream>
#include <chrono>
#include <stdlib.h>
#include <time.h>
#include <cstdlib>
#include <vector>

using namespace std;


/**
 * @brief Inicia generador aleatorio
 */
void startRandom() {
    std::srand(time(NULL));
}

/**
 * @brief Genera un numero aleatorio en un rango especifico
 * @param min: minimo en rango [min..max]
 * @param max: maximo en rango [min..max]
 * @return Numero aleatorio en rango [min..max]
 */
int randomInt(int min, int max) {
    int value = min + std::rand() % (max +1 - min) ;
    return value;
}

/**
 * @brief Genera un string aleatorio de cierto size
 * @param size: tamanio del string aleatorio que queremos
 * @return el string especificado
 **/
string generateRandomString(int size){
```

```
    string random_string;

    for ( int i = 0; i < size; i++ ) {
        char random_char = randomInt(97, 122);
        random_string.push_back(random_char);
    }

    return random_string;
}

int main(int argc, char** argv) {
    int n;

    if ( argc < 2 ) {
        cerr << "Error, parametros incorrectos" << endl;
        cerr << "Modo de uso: [nombre-programa] <size>" << endl;
        return 1;
    } else {
        n = atoi(argv[1]);
    }

    string t1 = generateRandomString(n);
    string t2 = generateRandomString(n);

    ofstream fb, fd;
    fb.open("brute_output.txt", ios::app);
    fd.open("dynamic_output.txt", ios::app);

    // Dynamic
    // ==========================================================================

    // Empiezo a cronometrar
    auto start = chrono::high_resolution_clock::now();

    // Se calcula y muestra la solucion
    string common = dynamic::getLCS(t1, t2);

    // Termino de cronometrar
    auto end = chrono::high_resolution_clock::now();

    // Tomo el tiempo
    auto duration_microseconds = chrono::duration_cast<chrono::microseconds>(end - start).
    count();
    auto duration_seconds = duration_microseconds / 1000000.0;

    fd << n << ", " << (double)duration_seconds << endl;

    // Brute
    // ==========================================================================

    // Idem anterior
    start = chrono::high_resolution_clock::now();
    common = brute::getLCS(t1, t2);
    end = chrono::high_resolution_clock::now();
    duration_microseconds = chrono::duration_cast<chrono::microseconds>(end - start).count();
```

```
    duration_seconds = duration_microseconds / 1000000.0;

    fb << n << ", " << (double)duration_seconds << endl;

    fd.close();
    fb.close();

    // Todo ha salido OK
    return 0;
}
```

En los archivos adjuntos hay una versión `measure.cpp`, que escribe los datos en un mismo archivo.

Anexo II. Tablas

Datos 1. Tiempos de ejecución de algoritmo dinámico

n	**Tiempo** (ms)
1000	0.026903
1500	0.054192
2000	0.087033
2500	0.131351
3000	0.19033
3500	0.258181
6500	0.97556
7000	1.06814
7500	1.21023
8000	1.39687
8500	1.62728
9000	1.8008
9500	1.94643
10000	2.15833
10500	2.28479
11000	2.47876
11500	2.74841
12000	2.95987
12500	3.26362
14500	4.33622
15000	4.61496

n	**Tiempo** (ms)
15500	4.93255
17500	6.27291
18000	6.65608
18500	7.05251
21500	10.2461
23500	11.5364
27000	15.0008
32000	20.9819
32500	24.302
33000	25.0637
34500	26.0168
39000	33.0729
40000	34.885
41000	37.5861
42000	39.1102
43000	40.0442
44000	41.8295
48000	49.8505
49000	52.0673
56000	405.349

Datos 2. Comparación de tiempos: dinámico vs. fuerza bruta

Algoritmo dinámico	
n	**Tiempo** (ms)
1	7e-06
2	9e-06
3	8e-06
4	7e-06
5	8e-06
6	8e-06
7	1e-05
8	1e-05
9	1e-05
10	1.1e-05
11	1.1e-05
12	1.2e-05
13	1.2e-05
14	1.4e-05
15	1.4e-05

Fuerza bruta	
n	**Tiempo** (ms)
1	1.4e-05
2	2.9e-05
3	7e-05
4	0.000208
5	0.000538
6	0.001815
7	0.006395
8	0.021562
9	0.076892
10	0.276634
11	1.11327
12	4.51193
13	17.9169
14	71.6379
15	288.533