Laboratory practice No. 5: Divide and Conquer Algorithms and Dynamic Programming

Sarah Henao Gallego
Universidad EAFIT
Medellín, Colombia
shenaog@eafit.edu.co

October 25, 2018

1) Code uploaded to GitHub

1.a. Given two string characters a and b determine the Levenshtein distance between them. (The Levenshtein distance is the minimal quantity of operations like insert, remove and changing letters. Use dynamic programming.

– The code is uploaded in GitHub along with a Main to test the functions. The file uploaded to GitHub is called **Levenshtein.java** includes a Recursive algorithm and a Dynamic Programming algorithm. These two approaches are used to compare and see if the result is the same as well as to evaluate the complexities of each. The complexity is optimized using dynamic programming.

1.b. Implement the Held-Karp algorithm.

—-

1.c. The longest common sub-sequence problem: Given two sequences, find the length of the longest sub-sequence that is present in both sequences.

– The code is uploaded in GitHub along with a Main to test the functions. The file uploaded to GitHub is called **longestComSubseq.java**. The algorithm is the same algorithm shown in Section 4, problem 2 of this lab. The algorithm is not cited because there are no authors stated in the PDF file.

2) Online Exercise

—-

3) *Project-type Questions*

3.a. *Explain the data structures and algorithm in problem 1.2.*

—-

3.b. *The most efficient way to solve the Traveling Salesman Problem (TSP) is through dynamic programming. Unfortunately, this method is not applicable to problems with large sets of vertices. Other than greedy algorithms, what other algorithms exist to solve this problem?*

– Another option to solving the TSP is using Brute Force, also known as Exhaustive Search, where the algorithm passes through the whole data set analyzing all the combinations for solutions and finally chooses the best one. This method is usually good for finding an initial solution to the problem, but is also not the most efficient way to solve the TSP.

3.c. *Explain the data structures and algorithm in problem 2.*

—-

3.d. *Calculate the Complexity of the Algorithm in problem 2.*

—-

3.e. *What is the meaning of the 'n' and 'm' in the previous question?*

—-

4) *Practice Test Problems*

4.a. *LEVENSHTEIN DISTANCE*

The answers are shown in Table 1 and Table 2.

		m	**a**	**d**	**r**	**e**
	0	*1*	*2*	*3*	*4*	*5*
m	*1*	0	1	2	3	4
a	*2*	1	0	2	3	4
m	*3*	2	2	1	2	3
a	*4*	3	2	2	2	**3**

Table 1: It takes 3 operations to convert *madre* to *mama*.

		c	**a**	**l**	**l**	**e**
	0	*1*	*2*	*3*	*4*	*5*
c	*1*	0	1	2	3	4
a	*2*	1	1	2	3	4
s	*3*	2	2	2	3	4
a	*4*	3	2	3	4	**4**

Table 2: It takes 4 operations to convert *calle* to *casa*.

4.b. LONGEST COMMON SUBSEQUENCE

4.2.1 Asymptotic computational complexity for the worst-case scenario

```
O(n * m) where n = leny and m = lenx
```

4.2.2 Line 31

```
return table[lenx][leny];
```

4.c. FIBONACCI 1

4.3.1 Complexity

```
O(n)
```

4.3.2 Reason

$T(n) = C_1 * n_1 + C_2$ where C_i are constants $\forall$ i

4.d. FIBONACCI 2

4.4.1 Complexity

```
O(2^n)
```

4.e. BINARY SEARCH

4.5.1 Asymptotic computational complexity for the worst-case scenario

```
T(n) = 2* T(n/2) + C * n which equals to O(n log n)
```

4.5.2 Line 8

```
return bus(a, iz, mitad-1, z);
```

4.5.3 Line 15

```
return bus(a, mitad+1, de, z);
```

4.f. LONGEST COMMON INCREASING SUBSEQUENCE

4.6.1 Line 7

```
/* Inicializar la tabla scm */
for ( i = 0; i < n; i++ )
{
    scm[i] = 0;      // Line 7
}
```

4.6.2 Line 12

```
/* Calcular usando la tabla scm*/
for ( i = 1; i < n; i++ )
{
    for ( j = 0; j < i; j++ )
    {
        if ( arr[i] > arr[j] && scm[i] < scm[j] + 1)
        {
            scm[i] = scm[j] + 1;      // Line 12
        }
    }
}
```

4.6.3 Line 16

```
/* El maximo en la tabla scm */
for ( i = 0; i < n; i++ )
{
    if ( max < scm[i] )
    {
         max = scm[i];      // Line 16
    }
}
```

4.6.4 Asymptotic computational complexity for the worst-case scenario

```
 O(n^2) where n is the size of input array arr
```

4.g. FLOYD-WARSHALL

4.7.1 Line 12

```
int ni = g[i][j];
```

4.7.2 Line 13

```
int nj = g[i][k];
```

4.7.3 Line 14

```
int nk = g[k][j];
```

4.7.4 Asymptotic computational complexity for the worst-case scenario

```
O(n^3)
```