CS 301 – Algorithms

Homework 3 – 21/04/2019

Atilla Alpay Nalcaci - 19510

Traffic Lane Problem

(a) Along the lines of manifested rules of the *Traffic Lane Problem*, subproblems can be listed as follows;

- Having a roadmap with lane value (l_i) being zero, in which case program should terminate.
- Having a fully occupied level (where each lane c_i in the given level is one).
- Determining the unoccupied lanes of the next level at each stage.

By observing the optimal substructure properties and the overlapping computations of this problem, the recursive definition of traffic lane problem by means of finding the path from (l_0, c_1) to (l_n, c_i) for a given n and i € {1, 2, 3} with the minimum amount of traffic lane changes is provided below;

$$C(n) = \begin{cases} 0, & \text{if } l_0 \\ 1, & \text{if } (l_n, c_i) = 1 \\ min(minPath(c_i - 1, l_j - 1), minPath(c_i - 1, l_j) + 1), & \text{if } i = 0 \\ min(minPath(c_i - 1, l_j - 1) + 1, minPath(c_i - 1, l_j), minPath(ci - 1, l_j + 1)), & \text{if } i = 1 \\ min(minPath(c_i - 1, l_j) + 1, minPath(ci - 1, l_j + 1)), & \text{if } i = 2 \end{cases}$$

Initially $(j, i) = (0, 1)$ and i € {1, 2, 3}, j € {1, 2,.., n} for n being the number of rows

Recursive algorithm has been implemented to cover the following conditions of the problem;

- If there exists no level which means there exists no roadmap (matrix) to work with thus program terminating immediately.
- If there exists a level where every lane is occupied (e.g. values of each element being one) which contradicts the *Rule 3* where the car cannot jump over cells thus preventing the recursive algorithm to complete.

- If the car is currently (as in at the given moment) located at the left lane in which case the algorithm should check for the lane ahead and the lane that is diagonal to it (which there is only one that is to its right).
- If the car is currently (as in at the given moment) located at the middle lane in which case the algorithm should check the row ahead and the diagonal rows (basically all 3 lanes of the next row).
- If the car is currently (as in at the given moment) located at the right lane in which case the algorithm should check for the lane ahead and the lane that is diagonal to it (which there is only one that is to its left).

(b) Properties of random number generators have been used during the implementation process of this homework by means of generating a road map where obstacles are generated arbitrarily at each execution. Pseudocode provided below;

```
n = int(input("Enter a row number: "))  # so that we can test various cases
map = [[0 for i in range(3)] for j in range(n)]  # matrix of size 'map[n][3]'

# Roadmap matrix creation
#--------------------------BEGIN--------------------------#
for i in range(len(map)):
    x = random.randint(0, 2)  # defining here so that at each loop a random variable will be generated

    while x == laneloc and i == rowloc:  # to ensure that there are no obstacles at the location of the car
        x = random.randint(0, 2)

    map[i][x] = 1

    for a in range(3):
        if a != x:
            map[i][a] = 0
#---------------------------END---------------------------#
```

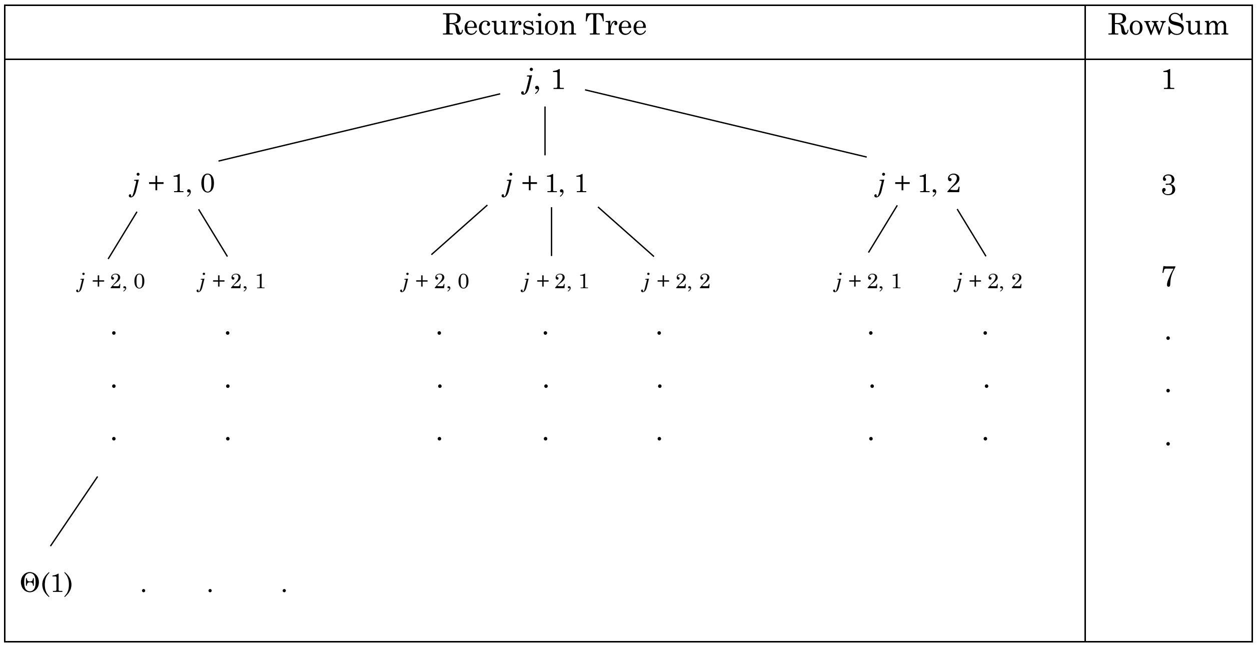
Figure 1. *Roadmap Matrix Creator*

Followingly, pseudocode for a naive recursive algorithm implementation is provided below (note that the full python code is included in the .zip file of this homework under the name of `naive_recursive_algorithm.py`);

```
# Main function (Naive Recursive Algorithm)
#---------------------------BEGIN---------------------------#
def TrafficLaneRecursive(row, lane):  # naive recursive function for the traffic lane problem
    if row == n:  # meaning that our initial row is the final row in which case program terminates
        return 0

    elif map[row][lane] == 1:  # if the location (namely [0][1]) is occupied in which case the car cannot be inserted
        return 1

    else:
        if lane + 1 < len(map[0]) and lane - 1 > 0:
            return min(TrafficLaneRecursive(row + 1, lane - 1) + 1,
TrafficLaneRecursive(row + 1, lane), TrafficLaneRecursive(row + 1, lane + 1) + 1)

        elif lane == 0:
            return min(TrafficLaneRecursive(row + 1, lane),
TrafficLaneRecursive(row + 1, lane + 1) + 1)

        else:
            return min(TrafficLaneRecursive(row + 1, lane - 1) + 1,
TrafficLaneRecursive(row + 1, lane))
#----------------------------END----------------------------#
```

Figure 2. *Traffic Lane Problem – Naive Recursive Algorithm*

Finally, asymptotic time complexity analysis of the naive recursive algorithm for the traffic lane problem is provided below through the representation of recursion tree method;



By observing the recursion tree, we can state that the recursion increases as; 1+3+7+17+..

→ Recursion will reach to the point; 1+3+..+($2^{(\# \text{ of rows})} - 1$)

→ $\sum_{k=1}^{n} 2^k - 1 = \frac{1-2^{n+1}}{1-2} + 2^0 - n = 2^{n+1} - n = O(2^{n+1} - n) = O(2^n)$

(c) Additionally to the Roadmap Matrix Creator that is provided in part (b), the Memoization Matrix Creator that is used during the implementation of this algorithm is provided below;

```
# Memoization matrix creation
#---------------------------BEGIN---------------------------#
def memoization(n):
    data = [[0 for i in range(3)] for j in range(n)]

    for i in range(len(data)):
        for k in range(len(data[0])):
            data[i][k] = -1

    return data

#----------------------------END----------------------------#
```

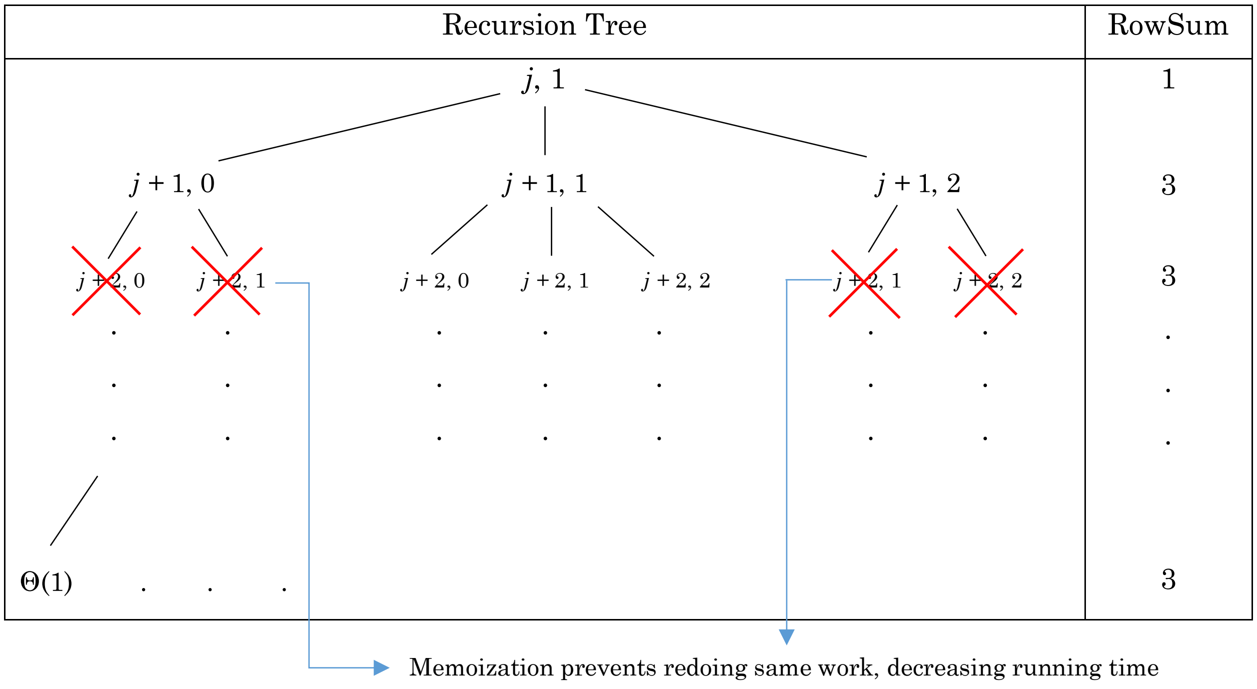
Figure 3. *Memoization Matrix Creator*

Followingly, pseudocode of a recursive algorithm that builds solutions to the recurrence from top down with memorization is provided below (note that the full python code is included in the .zip file of this homework under the name of `recursive_topdown_memoization.py`);

```
# Main function (Recursive Algorithm with Memoization)
#---------------------------BEGIN---------------------------#
def TrafficLaneMemoization(row, lane, map, table):
    if row == len(map):
        return 0

    elif table[row][lane] != -1:
        return table[row][lane]

    elif map[row][lane] == 1:
        return 1

    else:
        if lane - 1 > 0 and lane + 1 < len(map[0]):
            table[row][lane] = min(TrafficLaneMemoization(row + 1, lane - 1, map,
table) + 1, TrafficLaneMemoization(row + 1, lane, map, table),
TrafficLaneMemoization(row + 1, lane + 1, map, table) + 1)
        elif lane - 1 < 0:
            table[row][lane] = min(TrafficLaneMemoization(row + 1, lane, map,
table), TrafficLaneMemoization(row + 1, lane + 1, map, table) + 1)
        else:
            table[row][lane] = min(TrafficLaneMemoization(row + 1, lane - 1, map,
table) + 1, TrafficLaneMemoization(row + 1, lane, map, table))
    return table[row][lane]
#----------------------------END----------------------------#
```

Figure 4. *Traffic Lane Problem – Recursive Algorithm with Memoization*

Finally, asymptotic time complexity analysis of the naive recursive algorithm for the traffic lane problem is provided below through the representation of recursion tree method;

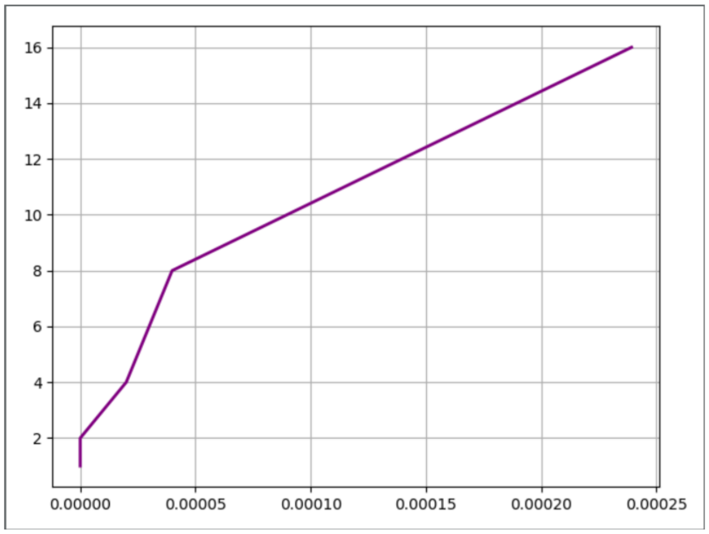
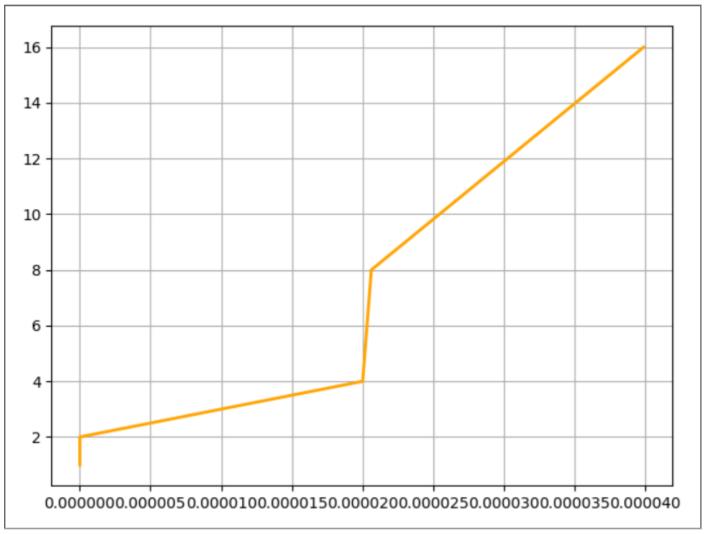


As the core principle of memorization algorithm suggests, each subproblem is called at most once, meaning that there will be only 3 subproblems at each stage. Thus, complexity of the algorithm will be the number of rows multiplied by 3;

$$O(3.n) = O(n)$$

(d) Pseudocode of an iterative algorithm that builds solutions to the recurrence from bottom up is not provided due to being unable to implement such algorithm in virtue of the traffic lane problem.

(e) Experimental evaluations with graphical representations are provided below;

Naive Recursive Algorithm	Top Down Recursive Alg. with Memoization
	

x-axis: Running Time (in seconds), y-axis: Input Size

Considering base principles of a naive recursive and memoization algorithm, experimental results are more or less indicate an expected distribution and growth by means of the increasing input size and their corresponding running times.

For naive recursive algorithm, the running time is expected to increase as we increase the input size since, as it is also depicted in the recursion tree in *part (b)*, algorithm tends to increase in linear time due to the fact that common subproblems being processed multiple times.

For the top down recursive algorithm with memoization, we can observe a rather reduced increase in running time as we increase the input size since memoization algorithm, in principle, prevents the execution of same work for a given recursion (through the implementation of memoization matrix, depicted in *part (c)*), meaning that the increasing input size will increase the running time in a much less manner (e.g. compared to naïve recursive algorithm).

For the bottom up iterative algorithm, there is no experimental data due to being unable to implement such algorithm. However, the expected asymptotic time growth would have been more or less the same with naive recursive algorithm with the difference of an increased space complexity since iteration would have occupied space in a continual manner.

Computer specifications provided below (for compare & contrast of experimental results);

Computer Specifications
CPU – Intel® Core™ i7-6700HQ (6M Cache, up to 3.50 GHz) **Chipset** – Intel® HM170 **Memory** – 32 GB DDR4 **Storage** – 512 GB SSD + 1 TB SATA HDD **Operating System** – Windows 10 Home