TRAFFIC LANE PROBLEM

CAVIT CAKIR 23657

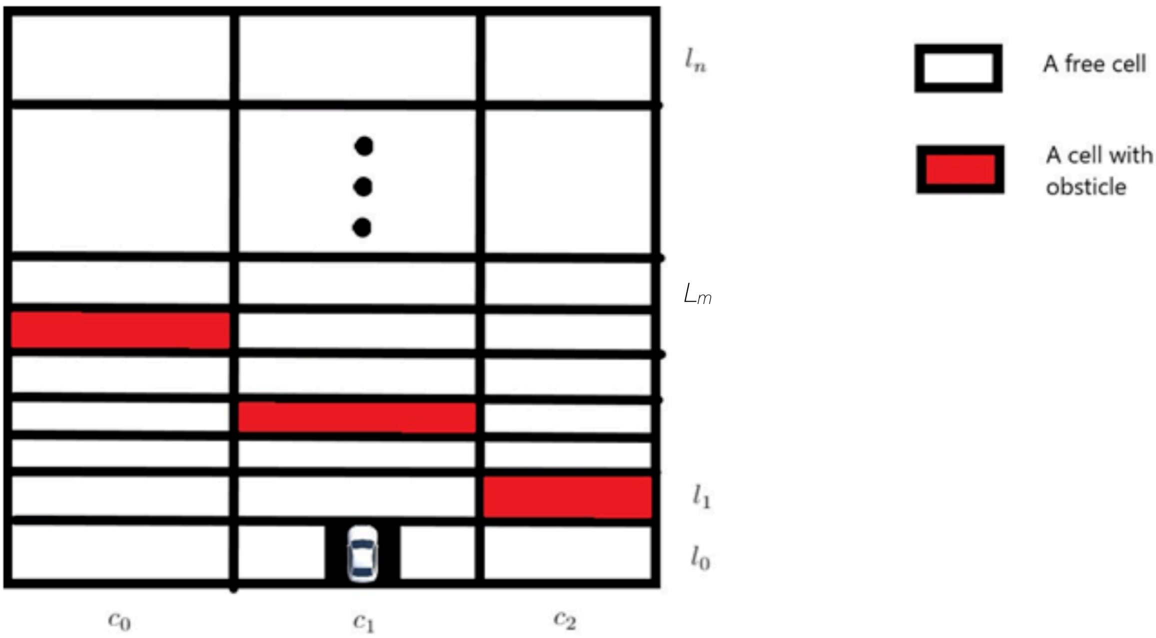


Figure A

a) Recursive formulation of the traffic lane problem:

i) **Identify the Subproblems** : To minimize the number of traffic lane changes to reach L_n which is the row at the top. We should consider minimum of the lane change values of c_0, c_1 and c_2 at n^{th} row. To calculate values of c's at n^{th} row, we should find the minimum lane change value of row below which is $(n-1)^{th}$ row. So our subproblem is the minimum value of the row below.

ii) **Optimal Substructure Property** : We always calculate the minimum lane change value while considering current row. Thus, every value at any cell is the optimal value after we fill the memorization table.

iii) **Overlapping Computations** : To find optimal value at m^{th} row, we should consider ==>
$\min((L_m,c_0), (L_m,c_1), (L_m,c_2))$.

(L_m,c_0) is $\min((L_{m-1},c_0), (L_{m-1},c_1))$
(L_m,c_1) is $\min((L_{m-1},c_0), (L_{m-1},c_1), (L_{m-1},c_2))$
(L_m,c_2) is $\min((L_{m-1},c_1), (L_{m-1},c_2))$

As clearly seen above, we are calculated (L_{m-1},c_0) for 2 times, (L_{m-1},c_1) for 3 times and (L_{m-1},c_2) for 2 times. Each step we should do these unnecessary calculations.

iv) **Define the Problem Recursively** : To find minimum value at the top we should call the recursive function as $R(L_n, c)$. It will call numWays with L_{n-1} and this step will repeat until L_0.

$$R(L, c) = min[\ (R(L-1,c),\ (\ R(L-1,c-1) + 1)\ ,\ (\ R(L-1,c+1) + 1)\ ]$$

b) ***Naive recursive algorithm:***

```
naiveCall(l, c)
{
    IF l == 0
        IF c == 1
            return 0
        ELSE
            return n+1
        END IF
    END IF

    IF places[l][c] is Blocked
        return n+1
    END IF

    IF c == 0
        return min(naiveCall(l-1, c), (naiveCall(l-1, c+1) + 1))
    ELSE IF c == 2
        return min(naiveCall(l-1, c), (naiveCall(l-1, c-1) + 1))
    ELSE
        return min(naiveCall(l-1, c), (naiveCall(l-1, c-1) + 1) , (naiveCall(l-1, c+1) + 1))
    END IF
}
```

—> Asymtotic Time Analysis : We start at L_n and call recursive function with L_{n-1} and repeat this step until L_0 and in each step we do at most 3 recursive calls which costs $O(3^n)$. Other comparisons done in $O(1)$. So, totally asymtotic time is $O(3^n)$.

—> Space Complexity Analysis : To determine the space complexity, we can consider a recursion tree. We store recursive calls in run-time stack memory, the total number of recursive calls that will be stored in a stack at max is the height of our recursive tree which is O(n). Therefore, our total space complexity is O(n)

c) ***Recursive algorithm, top down with memoization:***

Pseudo

```
checkMemo(l, c)
{
    IF myMemo[l][c] is Calculated Before
        return myMemo[l][c]
    ELSE
        myMemo[l][c] = topBottom(l, c)
        return myMemo[l][c]
    END IF
}

topBottom(l, c)
{
    IF l == 0: // if reached bottom
        IF c == 1
            return 0
        ELSE
            return n+1
        END IF
    END IF

    IF places[l][c] is BLOCKED
        return n+1
    END IF

    IF c == 0
        return min(checkMemo(l-1, c), (checkMemo(l-1, c+1) + 1))
    ELSE c == 2
        return min(checkMemo(l - 1, c), (checkMemo(l-1, c-1) + 1))
    ELSE
        return min(checkMemo(l-1, c), (checkMemo(l-1, c-1) + 1), (checkMemo(l-1, c+1) + 1))
    END IF
}
```

—> <u>Asymtotic Time Analysis</u> : We start at L_n and call recursive function with L_{n-1} and repeat this step until L_0 and in each step we write down optimal solution to additional array. So we calculate each cell in matrix for once which cost at most 3 * n and it is O(n). Other comparisons costs O(1).
So, totally asymtotic time is O(n).

—> <u>Space Complexity Analysis</u> : We used one additional memory matrix which costs 3 * n.
Therefore, our space complexity is O(n).

d) Iterative algorithm, bottom up with memoization:

Pseudo

```
FOR i = 1 to length of myMemo
    FOR j = 0 to 2
        IF myMemo[i][j] is not Blocked
            IF j==0
                myMemo[i][j] = min(myMemo[i-1][0],myMemo[i-1][1] + 1 )
            ELSE IF j == 2:
                myMemo[i][j] = min(myMemo[i - 1][2], myMemo[i - 1][1] + 1)
            ELSE
                myMemo[i][j] = min(myMemo[i - 1][1], myMemo[i - 1][2] + 1, myMemo[i - 1][0] + 1)
            END IF
        END IF
    END FOR
END FOR
```

—> <u>Asymtotic Time Analysis</u> : We start at L_1 and compare values at L_{n+1} and choose minimum of them then repeat this step until L_n and in each step we write down optimal solution to additional array. So we calculate each cell in matrix for once which cost at most 3 * n and it is O(n). Other comparisons costs O(1).

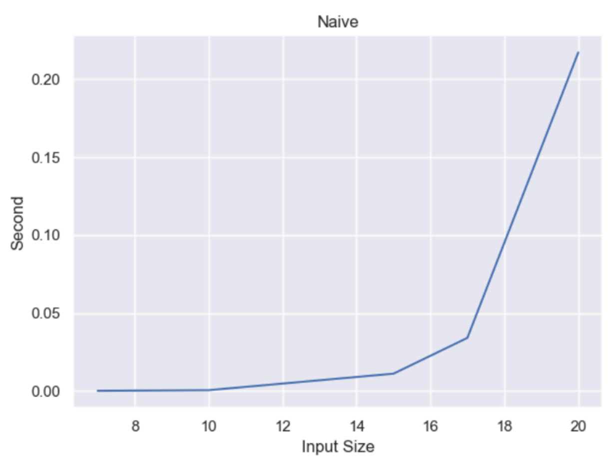
So, totally asymtotic time is O(n).

—> <u>Space Complexity Analysis</u> : We used one additional memory matrix which costs 3 * n. Therefore, our space complexity is O(n).

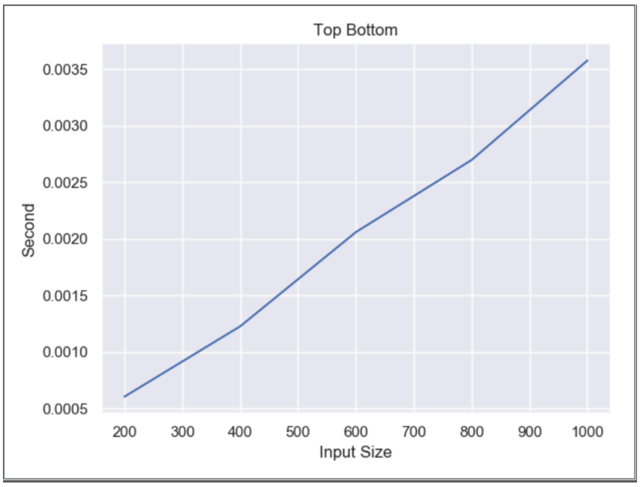
PART E FOLLOWS

```
 |   |          |   |          |   |          |   |
 |   |          |   |          |   |          |   |
 |   |          |   |          |   |          |   |
 |   |          |   |          |   |          |   |
\     /        \     /        \     /        \     /
 \   /          \   /          \   /          \   /
   V              V              V              V
```

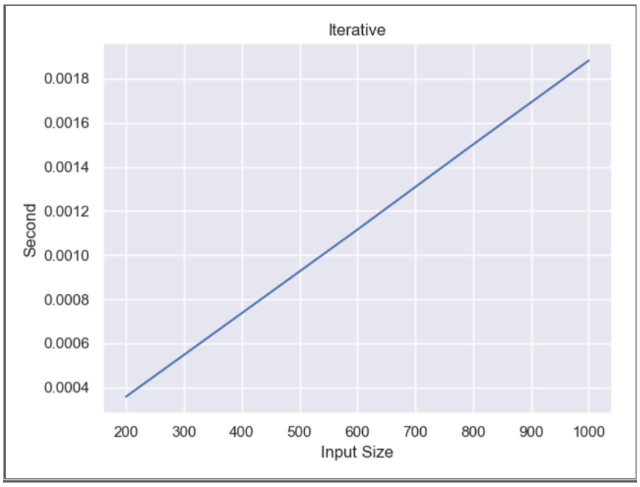
e) <u>Experimental evaluations:</u>



Result of **<u>naive algorithm</u>** proves the asymtotic complexity analysis. It is exponential function.



Result of **<u>top bottom algorithm</u>** proves the asymtotic complexity analysis. It is linear function.



Result of **<u>iterative algorithm</u>** proves the asymtotic complexity analysis. It is linear function.

TEST CASES FOLLOWS

```
 |   |        |   |        |   |        |   |
 |   |        |   |        |   |        |   |
\     /      \     /      \     /      \     /
 \   /        \   /        \   /        \   /
   V            V            V            V
```

<u>Test Cases:</u> In following screenshots, 'Naive' wrongly typed as 'Native'

1) EMPTY LANE

```
[1, 0, 1]
[0, 0, 0]
[1, 0, 0]
[0, 0, 0]
[0, 0, 0]
[0, 0, 0]
[1, 0, 0]
[0, 0, 1]
[0, 0, 1]
[1, 0, 0]
Native time = 0.0011082079999999994 Optimal= 0

 TopBottom time= 3.391099999999758e-05 Optimal= 0

 Iterative time= 2.058799999998806e-05 Optimal= 0
```

2) ZigZag

```
[1, 0, 1]
[1, 0, 0]
[0, 1, 0]
[0, 0, 1]
[0, 1, 0]
[1, 0, 0]
[0, 1, 0]
[0, 0, 1]
[0, 1, 0]
[1, 0, 0]

Native time = 0.00016027400000000158 Optimal= 4

TopBottom time= 2.771899999998162e-05 Optimal= 4

Iterative time= 1.8860999999981143e-05 Optimal= 4
```

3) Straight Obstacles

```
[1, 0, 1]
[0, 1, 0]
[0, 1, 0]
[0, 1, 0]
[0, 1, 0]
[0, 1, 0]
[0, 1, 0]
[0, 1, 0]
[0, 1, 0]
[0, 1, 0]

Native time = 1.8408000000025293e-05 Optimal= 1

TopBottom time= 2.3482999999990817e-05 Optimal= 1

Iterative time= 1.6119000000036632e-05 Optimal= 1
```

4) Single road

```
[1, 0, 1]

Native time = 2.7039999999645126e-06 Optimal= 0

TopBottom time= 1.5090000000106407e-06 Optimal= 0

Iterative time= 1.4469999999877636e-06 Optimal= 0
```

5) Empty Road

```
Native time = 0 Optimal= 0

TopBottom time= 0 Optimal= 0

Iterative time= 0 Optimal= 0
```

6) Correctness: We can see the correctness of algorithm by doing some trials.