(Traffic Lane Problem) **Efe Sencan**

(a) In the traffic lane problem, we are given a (n x 3) matrix as an input, where each element of the matrix is either 1 or 0. The road in the problem is represented as a matrix in the problem whereas obstacles are shown with the elements which are 1 in the matrix. Our goal in this problem is to find a path from a starting point which is denoted as (l,c), to the ending level which is defined as any cell at the last level of the matrix. Since our goal is to find a path which minimizes the number of traffic lane changes, our output would be an integer number which contains the number of minimum lane changes that solves the problem.

Sub-problems: If we assume that the last level index in the matrix that solves the problem is (l,c) which is given as an input. In order to determine the path that has the minimum number of lane changes from the starting point to (l,c), we have to solve the following recursion: T(l,c) = min((l-1,c),(l-1,c-1) +1,(l-1,c+1)+1). If we apply this formula for the other parts of the cells, we would obtain a recursion tree with height of n. Thus, every element of the tree would be our sub-problem.

Overlapping Sub-problems: Suppose the last level index in the matrix that solves the problem is (l,c). In order to find a path which contains the minimum number of lane changes and reaches to index (l,c) from the starting point, we have to determine the cell which has the minimum number lane changes that reaches to index(l,c), which is ***min((l-1,c),(l-1,c-1) +1,(l-1,c+1)+1)***.In order to calculate the cost of T(l-1,c-1), we also need to calculate the cost of T(l-2,c-1) and T(l-2,c), but we will also calculate these values while calculating the cost of T(l-1,c), such that (T(l-1,c) = min((l-2,c-1)+1,(l-2,c),(l-2,c+1)+1)). Moreover, reaching the any cell at the top the matrix is valid, we have to consider all the possible c values such as(0,1,2) to find minimum cost of the all problem. This will also cause to increase the amount of overlapping sub-problems that we calculate such that we will calculate T(l-1,c-1) +1 and T(l-1,c+1) +1 two times and T(l-1,c) three times.

Optimal Substructure: Since the sub-problems are dependent to each other, solving deepest level of the recursion tree (the sub-problem at the bottom) will lead to an optimal solution to the sub-problem at the one level above in the tree and that solution will lead to the one above. If we continue like that, we

will have an optimal solution for the whole problem which satisfies the optimal substructure property. Corollary, the optimal solution to the problem, leads to optimal solution to the subproblems.

Recurrence Relation: T(l,c) = min((l-1,c-1) + 1, (l-1,c), (l-1,c+1) + 1))

Algorithm 1 Naive Recursive Algorithm

(b)

```
1:  if matrix[l][c] != 1 then
2:      if l equals to 0 and c equals to 1 then
3:          return 0
4:      end if
5:      if c equals to 1 then
6:          return min(f(matrix,l-1,c),f(matrix,l-1,c-1)+1,f(matrix,l-1,c+1)+1)
7:      end if
8:      if c equals to 0 then
9:          return min(f(matrix,l-1,c),f(matrix,l-1,c+1) + 1,MAXINT)
10:     end if
11:     if c equals to 2 then
12:         return min(f(matrix,l-1,c),f(matrix,l-1,c-1)+ 1,MAXINT)
13:     end if
14: end if
15:
16: return MAXINT
```

Efe Sencan

Time and Space Complexity: In order to determine the space complexity for the naive algorithm, we can think of a recursion tree. Since we store each recursive calls memory on our run-time stack, the total number of recursive calls that will be stored in a stack at a given specific time is O(n), which is the height of our recursive tree. At each recursive call, we store our matrix which holds O(n) space complexity in the memory, where n denotes the total number of rows. Since our matrix is passed by referenced in the function, we only use O(n) amount of memory. Therefore, our total space complexity is O(n + n) = O(n). On the other hand, for the time complexity when we form our recursion tree, we can observe that from a single recursive call, we call either 2 or 3 recursion calls. Since at each recursive call, we proceed in our recursion tree by one step and our matrix have n elements, thus the height of our tree would be n. Therefore the number of elements in our recursion tree will be more than 2^n and less than 3^n. Therefore we have O(3^n) time complexity.

Algorithm 2 Recursive Algorithm with Dynamic Programming

```
        //matrix will be denoted as "m", additional memory would be "a"
(c)  1: if m[l][c] != 1 then
     2:     if a[l][c] equals to MAXINT then
     3:         if c equals to 1 then
     4:             a[l][c]=min(f(m,l-1,c,a),f(m,l-1,c-1,a)+1,f(m,l-1,c+1,a)+1)
     5:             return a[l][c]
     6:         end if
     7:         if c equals to 0 then
     8:             a[l][c] = min(f(m,l-1,c,a),f(m,l-1,c+1,a)+1,MAXINT)
     9:             return a[l][c]
    10:         end if
    11:         if c equals to 2 then
    12:             m[l][c] = min(f(m,l-1,c,a),f(m,l-1,c-1,a)+1,MAXINT)
    13:             return a[l][c]
    14:         end if
    15:     else
    16:         return a[l][c]
    17:     end if
    18: else
    19:     return a[l][c]
```

Efe Sencan

Time and Space Complexity: Since we calculate the sub-problems that we already calculate in the naive recursion version, we proceed with dynamic method (top-down) approach, which we create an additional matrix called "a". In this matrix, at each cell, we store the minimum cost (minimum number of lane changes) to reach that cell as an integer number. Initially, all the cells stores the MAXINT value, except our starting cell which the cost is 0. Thus, our space complexity would be 3*n where n is the total number of rows which results O(n) space complexity. At each recursive call, we try to fill the matrix "a" recursively, if we have not calculated the cost of that cell O(n). Otherwise, we simply return the cost of that cell which results in O(1) time complexity. Therefore, once we fill our matrix which takes O(n) time complexity, we do not need to calculate the cost for that cell again. Thus, our time complexity would be O(n*3) = O(n).

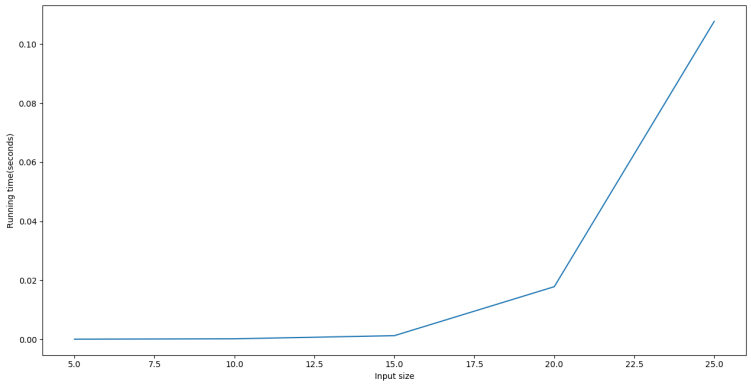
Algorithm 3 Iterative algorithm (bottom - up approach)

```
        //matrix will be denoted as "m", additional memory would be "a"
(d)  1: for <i in range(1,len(m))> do
     2:     for <x in range(1,3)> do
     3:         if m[i][x] is not equal to 1 then
     4:             if x equals to 0 then
     5:                 a[i][x] = min(a[i-1][x],a[i-1][x+1]+1,MAXINT)
     6:             end if
     7:             if x equals to 1 then
     8:                 a[i][x] = min(a[i-1][x],a[i-1][x-1]+1,a[i-1][x+1]+1)
     9:             end if
    10:             if x equals to 2 then
    11:                 a[i][x] = min(a[i-1][x],a[i-1][x-1]+1,MAXINT)
    12:             end if
    13:         end if
    14:     end for
    15: end for
    16: return m[l][c] // "= 0" should be deleted
         =0
```

Efe Sencan

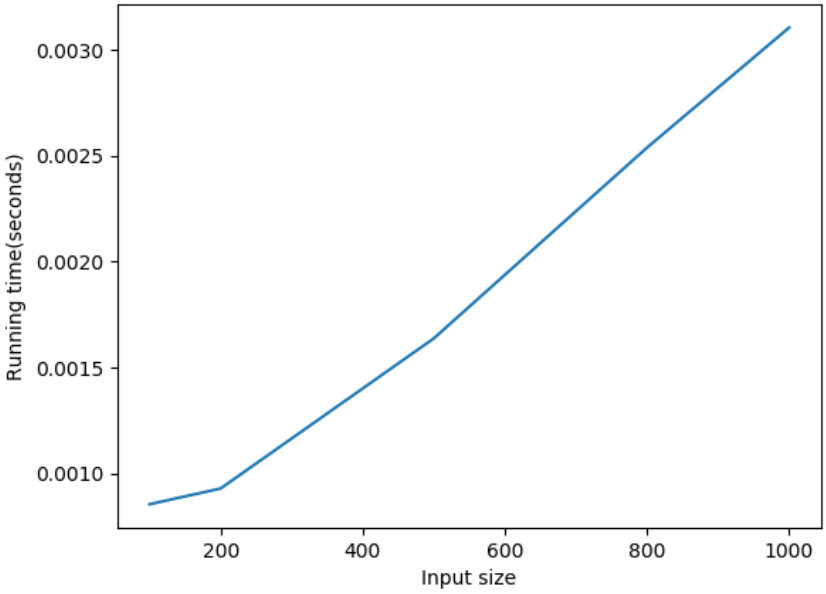
Time and Space Complexity: For the bottom - up approach algorithm, we also need a auxiliary matrix of size (n*3) in order not to solve the sub-problems that we already solve before, where n is the number of rows of our original input matrix. Therefore our space complexity would be O(n). As opposed to the top - bottom approach, we will fill the cells of the additional matrix, starting from the second row. Because initially, all the cells are filled with MAXINT value except for our starting point which the cost is 0, therefore we do not need to look for the first row. Since finding the min function takes constant time to compute and we visit each cell once in the matrix, the time complexity would be O(n).

Naive Recursive Algorithm:

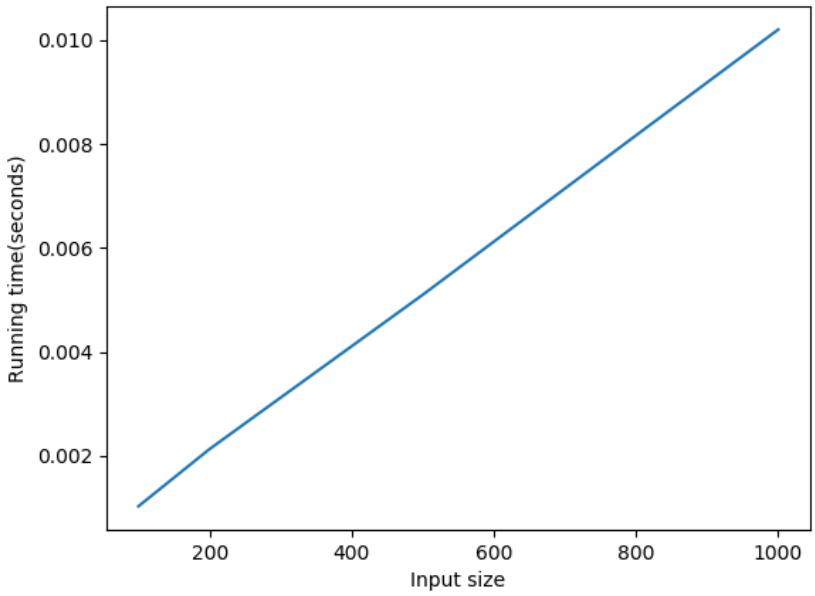


According to our analysis for the naive recursion algorithm, the time complexity would be exponential. We can clearly observe the exponential increase in time complexity from the graph as the input size increases.

Dynamic Recursive Algorithm (top-bottom):
For the recursive algorithm that builds the solution to top-bottom memoization, we found the the time complexity would be linear, which is O(n). We can also observe the linearity from the graph that we plotted. We can also conclude that the algorithm is quite faster than the naive recursion one.



Iterative Algorithm (bottom - up):
Finally, for the iterative approach that builds the solution to bottom - up approach, we found that the time complexity would be again linear, which is O(n). When we take a look at the graph, we can see that the graph is linear as expected. We can also observe that the iterative algorithm is a lot faster than the naive recursion one.


0.010
0.008
0.006
0.004
0.002
Running time(seconds)
200
400
600
800
1000
Input size