- CS301 Homework 3 -

- Traffic Lane Problem -

a. Recursive Formulation of the "Traffic Lane Problem":

In the "Traffic Lane Problem" we need to understand the problem first. While going straight does not cost extra, going left or right have a cost of +1. To get the minimal cost or to find the minimum path from beginning to end, we need to go straight as much as possible, otherwise we need to turn left or right knowing that one of them has straighter path than the other. This would be the strategy to find the minimum path.

Recursion we consider in this problem is that in each step we move on we need to know which path is minimum. So, we if we are the n^{th} step of the road so far, we need to know the rest of the roads minimum path. The main problem is calculating the minimum path's cost for the whole road and the subproblems would be calculating the minimum path's cost for the remaining roads and the stopping point will be the end of the road or the beginning of the road. -depending on how you are going to implement the function- By following this procedure we will create the optimal substructure to calculate the optimal route.

In the end our recursion formula will be:

$$T(l,c) = \min[\, T(l-1,c-1)+1\,, T(l-1,c)\,, T(l-1,c+1)+1\,]$$

Where l denotes the length, the car travelled so far, and c denotes the traffic lane we are in. Then T(l,c) stands for the minimum path's cost. For example, if we want to calculate the minimum path for length 10 and traffic lane 1, the recursion formula will be like this:

$$T(10,1) = \min[\, T(9,0)+1, T(9,1)\,, T(9,2)+1\,]$$

And for than we need to find the minimum paths for T(9,c) where c can be (0,1,2). This time our recurrence relation will be continue like this. The middle lane calls 3 recursive calls, but left or right lanes call 2 recursive calls since we cannot jump left to right or right to left. Let's take c as 1 here (we are at the middle lane):

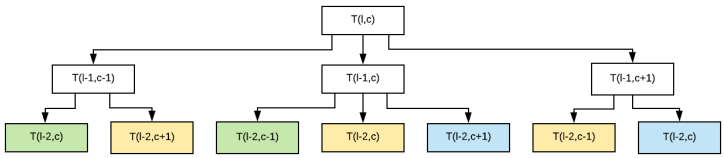
$$T(9,1) = \min[T(8,0)+1\,, T(8,1), T(8,2)+1]$$

If c is 0 (we are at the left lane):

$$T(9,0) = \min[T(8,0)\,, T(8,1)+1]$$

To sum up the relation will repeat like this. So, to solve the problem we divide it into smaller problems and solve the small problems to find the real problem. This means that we have an optimal substructure property.

Our recurrence relation's recurrence tree is given below. As we can see there are multiple calculations for the same solution. The colors indicate that this solution is the same as the other solutions. For example, yellow nodes are all calculating the same solution for T(l-2,1) relation. Thus, we are calculating the same solutions repeatedly. This is called overlapping computations and we can get rid of this by using dynamic programming. (with memoization)



...

We are going to solve the same problem with different approaches such as naïve recursive algorithm, top-down dynamic programming algorithm and bottom-up dynamic programming.

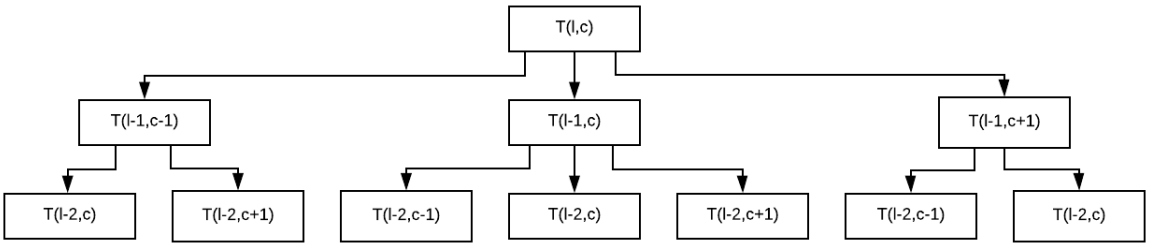
b. Pseudocode of a Naive Recursive Algorithm:

```
#recursive naive algorithm
FUNCTION minPath(road,end,start,lane):
    sum=0
    IF (end = start):
        RETURN sum
    ELSE:
        IF(lane = 1):
            IF(road[start+1][1] = 1):
                sum += ( min ( 1 + ( minPath(road,end,start+1,0) ), ( 1 + minPath(road,end,start+1,2) ) ) )
            ELSE:
                IF( road[start+1][0] = 1 ):
                    sum += ( min( minPath(road,end,start+1,1) , ( 1 + minPath(road,end,start+1,2) ) ) )
                ELSEIF( road[start+1][2] = 1):
                    sum += ( min( minPath(road,end,start+1,1) , ( 1 + minPath(road,end,start+1,0) ) ) )
                ELSE:
                    sum += ( min( minPath(road,end,start+1,1) , ( 1 + minPath(road,end,start+1,0) ) , ( 1 + minPath(road,end,start+1,2) ) ) )
            ENDIF
        ENDIF
        ELSEIF(lane = 0):
            IF(road[start+1][0] = 1):
                sum += ( 1 + minPath(road,end,start+1,1) )
            ELSE:
                IF(road[start+1][1] = 1):
                    sum += ( minPath(road,end,start+1,0) )
                ELSE:
                    sum += ( min ( minPath(road,end,start+1,0) , ( 1 + minPath(road,end,start+1,1) ) ) )
                ENDIF
            ENDIF
        ELSE:
            IF(road[start+1][2] = 1):
                sum += ( 1 + minPath(road,end,start+1,1) )
            ELSE:
                IF(road[start+1][1] = 1):
                    sum += ( minPath(road,end,start+1,2) )
                ELSE:
                    sum += ( min ( minPath(road,end,start+1,2) , ( 1 + minPath(road,end,start+1,1) ) ) )
                ENDIF
            ENDIF
        ENDIF
        RETURN sum
ENDFUNCTION
```

The recurrence relation of this pseudocode is given like this:

$$T(l,c) = \min[\, T(l-1,c-1)+1\,, T(l-1,c)\,, T(l-1,c+1)+1\,]$$

If we are to show this relation as a recurrence tree we will end up with a tree looking like this:



…

For the middle lane we need to call 3 recurrences whereas for the left and right lane we need to call 2 recurrences. So, in the end, the number of leaves in the tree will be between 2^n and 3^n since the branching will be done like shown in the tree. Then,

$$2^n < T(n) < 3^n$$

So, time complexity will be for the best-case $O(2^n)$ and for the average and the worst-case $O(3^n)$.

The space complexity will depend on the number of recurrences we call, since for each call we allocate a memory for the variable “sum” (see the pseudocode). All in all, space complexity will be the same as time complexity. for the best-case $O(2^n)$ and for the average and the worst-case $O(3^n)$.

c. Pseudocode of the Recursive Algorithm with a Top-Down Dynamic Programming Approach:

```
#recursive TopBottom dynamic programming
FUNCTION minPathTopBottom(road,end,start,lane,memo):
    IF(memo[start][lane] != -1):
        RETURN memo[start][lane]
    ELSE:
        IF(end = start):
            IF(road[end][lane] = 1):
                RETURN 1
            ENDIF
            RETURN 0
        ENDIF
        IF(road[start][lane] = 1):
            RETURN sys.maxsize
        ENDIF
        IF(lane = 1):
            memo[start][1] <- min( minPathTopBottom(road,end,start+1,1,memo) , 1 + minPathTopBottom(road,end,start+1,0,memo) , 1 + minPathTopBottom(road,end,start+1,2,memo))
            RETURN memo[start][lane]
        ELSEIF(lane = 0):
            memo[start][0] <- min ( minPathTopBottom(road,end,start+1,0,memo) , ( 1 + minPathTopBottom(road,end,start+1,1,memo) ) )
            RETURN memo[start][0]
        ELSEIF(lane = 2):
            memo[start][2] <- min ( minPathTopBottom(road,end,start+1,2,memo) , ( 1 + minPathTopBottom(road,end,start+1,1,memo) ) )
            RETURN memo[start][2]
        ENDIF
    ENDIF
```

memo = [[0 for i in range(3)] for j in range(length+1)

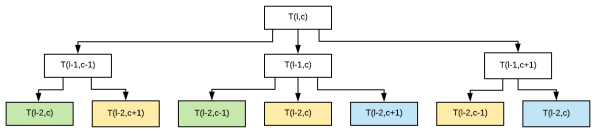
The pseudocode of the recursive top-down dynamic programming approach is given above. For this pseudocode we also have the same recurrence relation.

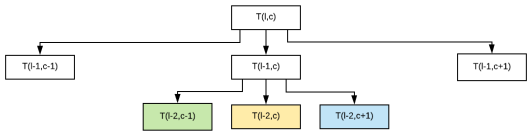
$$T(l,c) = \min[\, T(l-1,c-1)+1\,, T(l-1,c)\,, T(l-1,c+1)+1\,]$$

However, with this approach we store a memory for each calculations we made. So, instead of calculating the same recurrence repeatedly, we calculate it once and store it in our memory matrix and if we face of the same recurrence again we simply retrieve it from our memory matrix. With this strategy we get rid of the exponential bound and unnecessary calculations.

Memory matrix is matrix that stores the minimum values for the given path. For example, memo[length][lane] stores the minimum path till that length and traffic lane. The matrix size is 3xn, where n stands for the length of the road, and it stores integer values. All in all, the space complexity will be O(3n) which we can denote as O(n). -we also need to consider the space that is required to store the recursions but when n is too big it does not matter that much-

The recurrence above seems same with the naive recurrence algorithm but here the trick is that we are not always calculating the repeating recurrences. For example, in the recurrence tree below we can see that the nodes that has the same coloring are the same solution to the same recurrence. Thus, by using memoization we could get rid of the unnecessary calculations and the resulting tree will look like the right tree given below.





The time complexity will decrease rapidly since returning a value from a matrix with the given indices is constant time O(1). We will break the exponential bound and have a linear bound O(3n) which is O(n), where n stands for the length of the road. To be more specific, since we are not calculating the same problem again, all we must do is calculating the all roads just once and fill up the memory matrix. Memory matrix has 3xn entries so, the algorithm needs the calculate 3n problems only. Returning a value takes constant time, in the end the time complexity will be O(n).

d. Pseudocode of the Iterative Algorithm with a Bottom-Up Dynamic Programming Approach:

```
#iterative bottomTop dynamic programming
FUNCTION minPathBottomTop(road,end,memo2):
    x=1
    while(x<end):
        IF(x!=1):
            IF(road[x][0] != 1):
                memo2[x+1][0] <- min( memo2[x][0], memo2[x][1] + 1)
            ELSE:
                memo2[x+1][0] <- sys.maxsize
            ENDIF
            IF(road[x][1] != 1):
               memo2[x+1][1] <- min( memo2[x][0]+1, memo2[x][1], memo2[x][2]+1)
            ELSE:
                memo2[x+1][1] <- sys.maxsize
            ENDIF
            IF(road[x][2] != 1):
                memo2[x+1][2] <- min( memo2[x][1]+1, memo2[x][2])
            ELSE:
                memo2[x+1][2] <- sys.maxsize
            ENDIF
        ELSE:
            IF(road[x][0] != 1):
                memo2[x+1][0] <- 1
            ELSE:
                memo2[x+1][0] <- sys.maxsize
            ENDIF
            IF(road[x][1] != 1):
                memo2[x+1][1] <- 0
            ELSE:
                memo2[x+1][1] <- sys.maxsize
            ENDIF
            IF(road[x][2] != 1):
                memo2[x+1][2] <- 1
            ELSE:
                memo2[x+1][2] <- sys.maxsize
            ENDIF
        ENDIF
        x+=1
    ENDWHILE
    RETURN min( memo2[end] )
ENDFUNCTION
```

Just like the top-down dynamic programming approach, bottom-up approach is also using memoization but this time instead of calculating the minimum paths, it adds up the paths one after the other to solve the problem. Thus, this algorithm is an iterative algorithm and it builds up to the solution one by one. The rules are simple. Going straight does not cost any extra whereas going left or right cost extra +1 to the minimum path. The only condition check is the if the road is blocked or not. If the road is blocked the index that represent the road's cell will be the maximum value, otherwise the value will be calculated by selecting the minimum value of the previous indices in the memory matrix. The table below gives a visual understanding of the algorithm. Finally, the minimum value stored in the last row will be the shortest path.

In the end, the algorithm would iterate maximum n times where n is the length of the road. At each iteration, the traffic lanes will be checked for the availability, and then it will be calculated from the previous indices. Setting the maximum value for the blocked cells in the memory will take constant time O(1). Finding the minimum in the given list would take O(n) time but since at most we are taking the minimum of 3 elements we can say that taking minimum will take constant time also O(1). All in all, the whole algorithm has a time complexity of O(n), because of the loop that iterates n many times.

Space complexity will be the same as top-down approach since we are allocating a 3xn matrix to store the results. Thus, the space complexity will be O(3n) which is the same as O(n).

Example: (if the road is blocked (cell with a 1) INF is the minimum path to that cell)

Road to solve

0	0	0
1	0	0
0	0	1
0	1	0
0	0	1
0	1	0

Memory Matrix denoted as "m"

0	0	0
INF	min(m[0][0]+1, m[0][1], m[0][2]+1)	min(m[0][1] + 1, m[0][2])
min(m[1][1] + 1, m[1][0])	min(m[1][0]+1, m[1][1], m[1][2]+1)	INF
min(m[2][1] + 1, m[2][0])	INF	min(m[2][1] + 1, m[2][2])
min(m[3][1] + 1, m[3][0])	min(m[3][0]+1, m[3][1], m[3][2]+1)	INF
min(m[4][1] + 1, m[4][0])	INF	min(m[4][1] + 1, m[4][2])

→The solution to the road given as an example will be the smallest value of the last row of matrix "m"

e. Experimental evaluations of these three algorithms:

Since we know the theoretical results for these 3 algorithm we can conduct some experiments to see how they experimentally works.

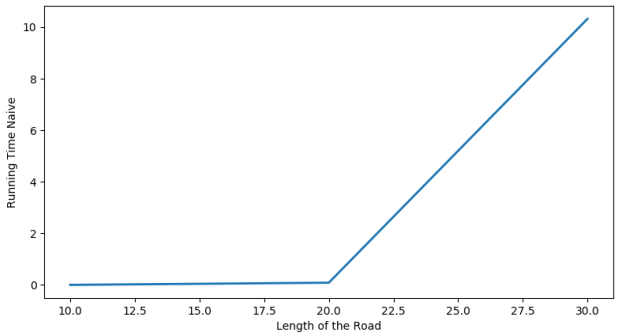
Theoretical Results:

Naive Algorithm → $O(3^n)$ Top-Down Approach→O(n) Bottom-Up Approach→O(n)

Experimental Results:

Experiment 1 conducted is the average time required in seconds to find the minimum path in the given road of length 10,20,30. (since the naive recursive algorithm takes too much time to complete, the longest path for comparing these 3 algorithms will be 30)

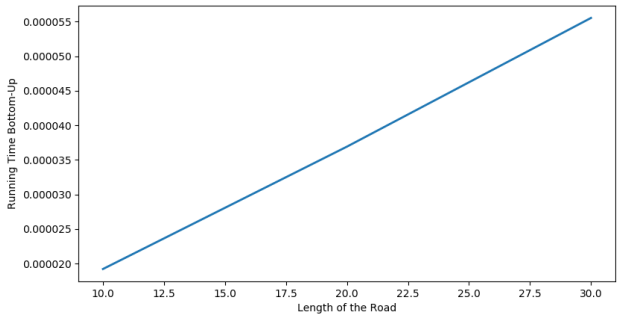
Naive Algorithm



The expected result should look like an exponential function and the result indeed looks like it.

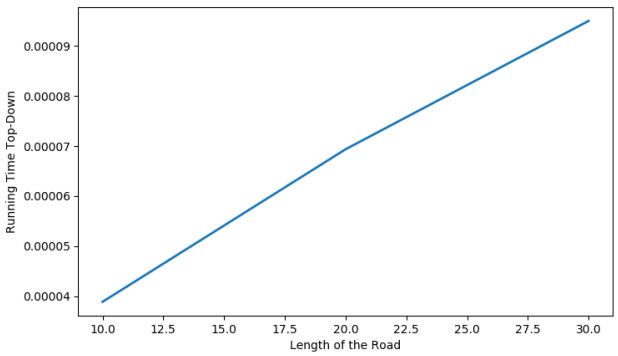
However, the down part is that it takes too much time only to solve the road of length 30.

Bottom-Up Approach



The expected result and the experimental results are matching, and they are linear.

Top-Down Approach

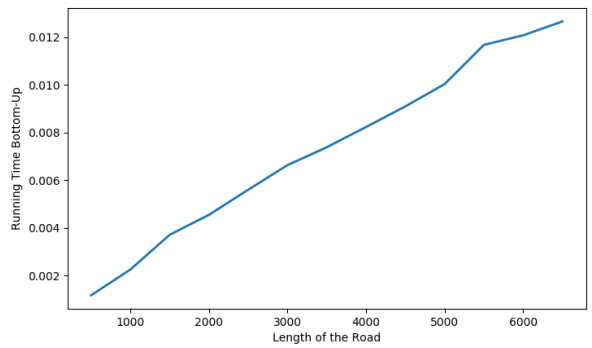


The expected result and the experimental results are matching, and they are linear.

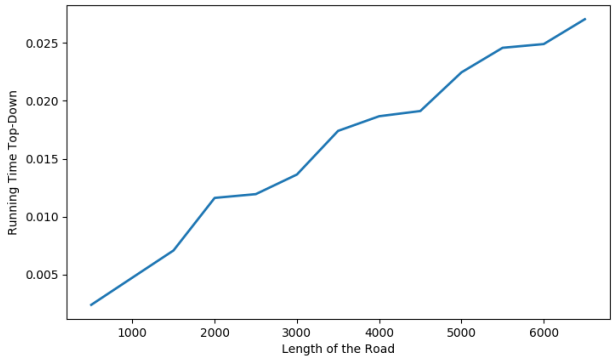
We can extend the comparison for the dynamic programming algorithms which are top-down approach and bottom-up approach since they both work fast compared to naive algorithm.

Experiment 2 is conducted by having a variety of roads with variety of length from 500 to 6500. (6500+ gives the error stack overflown) The graphs show the average time vs. road length.

Bottom-Up Approach



Top-Down Approach



They both seem to do well against large number of data. They are on average looks linear.

Results:

Final verdict is that naive recursive algorithm is not efficient at all, because we calculate the same recursion repeatedly which increase the running time rapidly. The dynamic programming approach take down this running time well enough that it does not even take 1 second for very large data. Bottom-Up works a faster than the Top-Down approach, maybe it is because bottom-up is an iterative function and top-down is a recursive function.