Maximum Amount of Gold Collected

Ishneet Sethi
(IIT2019237)

Chandramani Kumar
(IIT2019238)

Mrityunjaya Tiwari
(IIT2019239)

IV Semester B.Tech in Information Technology
Indian Institute of Information Technology Allahabad, Prayagraj(U.P.), India

***Abstract*—In this paper, we have devised an algorithm to find the maximum amount of gold collected by a miner while starting from any cell of the first column and traversing a matrix of N X M size whose each cell contains the amount of gold.**

I. Introduction

Bottom up (tabulation) dynamic programming approach has been used to implement this algorithm.
Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.
Bottom approach is so called because one starts with the base result and gets to the topmost desired result.

This report further contains -
II. Algorithm Design
III Algorithm and Illustration
IV. Algorithm Analysis
V. Conclusion
VI. References

II. ALGORITHM DESIGN

A. Algorithm-1(Recursion):

Here the idea is to recursively get the maximum amount of gold collected by the miner while starting from each cell of the first column and traversing the matrix in all possible ways thereby updating the maximum amount of gold collected by recurring in the three directions as mentioned in the question i.e right,right_up and right_down.
Finally, the value the variable max_gold_amt will contain gives us the required answer.

B. Algorithm-2(Dynamic Programming) :

The idea is to store the amount of gold that can be collected in each cell if the miner starts collecting from that cell.

To do this we are implementing the bottom up approach of dynamic programming. We are creating a 2D matrix of the same size as the given matrix to store what we discussed before. For each row of the last column of the created matrix, value will be equal to that of the given matrix as no move can be possible from those cells. For the rest of the columns, we will store the sum of the value of the present cell in the given matrix and maximum of values of the cells in the created matrix which are feasible to be reached on moving from that cell. Finally the required result will be found on traversing through each row and finding the maximum of the first column.

III. ALGORITHM AND ILLUSTARTION

A. Algorithm 1 Pseudo Code

Code 1. Algorithm 1

```
//====================================
            //--PSEUDO CODE--//
//====================================

function main()
        srand(time(0))
        get R and C
            for  i     1 to R do
                 for  j     1 to C do
      matrix[i][j]     rand
           for  i     1 to R do
                ans     max(ans,
                maxm_Gold_amnt(R,C,i,1,matrix))

       print ans


function maxm_Gold_amnt(R, C, row, col, matrix)
         if (col == C+1 ||
         row == 0 || row == R+1) then
                return 0;

      long long int right
maxm_Gold_amnt(row,col+1)
      long long int
right_up      maxm_Gold_amnt(row-1,col+1)
        long long int
right_down     maxm_Gold_amnt(row+1,col+1)

     return matrix[row][col] +
                      max( right,
```

```
                    max(right_up, right_down))



function max(a,b)
        if (a>b) then
                return a
        return a
```

B. Algorithm 2 Pseudo Code

Code 2. Algorithm 2

```
//====================================
            //--PSEUDO CODE--//
//====================================

function main( )
srand(time(0))
get R and C
for  i     1 to R do
        for  j     1 to C do
                matrix[i][j]     rand

print maxm_Gold_amnt(R, C,matrix)




function maxm_Gold_amnt(R, C,matrix)
        long long int dp[R][C]      {0}

      for  i     1 to R do
                dp[i][C]      matrix[i][C]

        for i     C-1 to 1 do
            for j     1 to R  do
     if  (i      1) then
          if ( i+1 <= R ) then
        dp[i][j]     matrix[i][j] +
max(dp[i][j+1],dp[i+1][j+1])
                          else
         dp[i][j]     matrix[i][j] + dp[i][j+1]

      else if ( i     R) then
            dp[i][j]      matrix[i][j] +
max(dp[i][j+1],dp[i-1][j+1])

      else
            long long int
m
max(dp[i+1][j+1],dp[i-1][j+1])

            dp[i][j]
matrix[i][j] +
max(dp[i][j+1],m);

      long long int maxm     -1

        for i     1 to R do
                if ( dp[i][1] > maxm) then
                      maxm      dp[i][1]


      return maxm
```

```
function max(a,b)
        if (a>b) then
                return a
        return a
```

C. ILLUSTRATION

- INPUT: R = 3, C = 3
  m[3][3] = $\begin{Bmatrix} 4 & 5 & 6 \\ 7 & 10 & 20 \\ 10 & 20 & 12 \end{Bmatrix}$
- Avoid combining SI and CGS units, such as current in amperes and magnetic field in oersteds. This often leads to confusion because equations do not balance dimensionally. If you must use mixed units, clearly state the units for each quantity that you use in an equation.
- First, we will create a 2d matrix dp[3][3] with each cell initialised with 0.

  Initially dp[][] matrix be :

  dp[3][3] = $\begin{Bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{Bmatrix}$
- Now, using algorithm 2 for last column cells the updated matrix will be :
  dp[3][3] = $\begin{Bmatrix} 0 & 0 & 6 \\ 0 & 0 & 20 \\ 0 & 0 & 12 \end{Bmatrix}$
- Now, for all other column cells algorithm 2 implementation is demonstrated as :

  dp[0][1] = matrix[0][1] + max(6,20)
  = matrix[0][1] + 20
  = 5 + 20 = 25

  dp[1][1] = matrix[1][1] + max(6,max(20,12))
  = matrix[1][1] + 20
  = 10 + 20 = 30

  dp[2][1] = matrix[2][1] + max(20,12)
  = matrix[2][1] + 20
  = 20 + 20 = 40

  Updated matrix dp[][] will be :
  dp[3][3] = $\begin{Bmatrix} 0 & 25 & 6 \\ 0 & 30 & 20 \\ 0 & 40 & 12 \end{Bmatrix}$
- dp[0][0] = matrix[0][0] + max(25,30)
  = matrix[0][0] + 30
  = 4 + 30 = 34

dp[1][0] = matrix[1][0] + max(25,max(30,40))
= matrix[1][0] + 40
= 7 + 40 = 47

dp[2][0] = matrix[2][0] + max(30,40)
= matrix[2][0] + 40
= 10 + 40= 50

Finally the dp[][] matrix will be :

dp[3][3] = $\begin{Bmatrix} 34 & 25 & 6 \\ 47 & 30 & 20 \\ 50 & 40 & 12 \end{Bmatrix}$

- Finally, the required maximum amount of gold collected will be maximum of 34, 47 and 50, i.e 50.
- Output: 50.

IV. ALGORITHM ANALYSIS

A. Time Complexity

RECURSION

In this approach, we are following some steps :
1) Traversing each row for updating maxm.
In each iteration of the loop, we are recurring thrice through m-1 columns to get the maximum amount of gold collected if started with that cell of the first column.

Clearly, time complexity gets affected by the above two steps.
Let's suppose step 1 time complexity be T1

So, the Time complexity of the whole program:

$$(T) = T1$$

Assuming 1 unit time for comparison and assignment.

Now, calculating T1 :
No of rows traversed = N
So, T1 = N*(T')
where , T' = Time complexity for recursive function
= width of the recursive tree
= $3 \wedge (m-1)$
Therefore the time complexity would be:

$$O(N * 3 \wedge (M-1))$$

DYNAMIC PROGRAMMING

In this approach, we are following some steps :
1) Traversing each row for the last column cells' value.
2) Traversing each row and column except the last one to store the maximum gold value in each cell of the dp[][] matrix.
3) Traversing each row of the first column to find the maximum among them.

Clearly, time complexity gets affected by the above three steps.
Let's suppose step 1 time complexity be T1, step 2 time complexity be T2 and that of step 3 be T3
So, the Time complexity of the whole program

(T)=T1+T2+T3

Now, calculating T1 :

No of rows traversed = N
So, T1 = N*(T')
where , T' = 1 unit for assigning operation

$$T1 = O(N)$$

Now, calculating T2 :
No of rows traversed = N
No of columns traversed = M-1
So, T2 = O((N)*(M-1))*(T")
where T" = 1 unit time for each comparisons made and for assignment operation

$$T2 = O(N * (M-1)))$$

Now, calculating T3 :
No of rows traversed = N
So, T3 = N*(T"')
where , T"' = 1 unit time for each comparisons
made and for assignment operation

$$T3 = O(N)$$

Hence, T = O(N) + O(N*(M-1)) + O(N)
' T = O(N*M)

Best Case Complexity: O(N) [when M = 1]

Worst Case Complexity: O(N*M)

B. Space Complexity

Recursion algorithm

Our memory complexity is determined by the number of return statements because each function call will be stored on the program stack. To generalize, a recursive function's memory complexity is O(recursion depth).Hence in this case space complexity would be O(M).

Dynamic Programming

Since we are creating a 2D matrix of the same size as of the original matrix, i.e N X M, so we are using N*M space .Hence, the space complexity of the algorithm 2 will be O(N*M).

For the analysis of the algorithm, we break the code snippet into several segments and count the number of addition, subtraction, multiplication, division, assign, comparison and by considering all those operations as 1 unit of operation, we will try to find the overall time complexity of the algorithm by multiplying the total number of operations into the number of times the code segment is running in the algorithm. For the space complexity, we tend to count the total amount of space taken by that algorithm to implement and then find a relation between the amount of space required and the input. The time and space complexity analysis for various steps is as shown below .

C. Aposteriori analysis

Row size	Column size	Time(in milliseconds)
10	100	10.270
10	500	11.804
10	1000	10.796
10	3000	11.842
10	5000	10.618
10	8000	9.827
10	10000	10.369
10	15000	9.775
100	10	9.816
500	10	10.551
1000	10	11.834
5000	10	10.593
8000	10	11.899
10000	10	10.379
15000	10	18.520

FIGURE1: TIME COMPLEXITY GRAPH FOR APOSTERIORI ANALYSIS. (KEEPING N CONSTANT AND M ON X-AXIS)

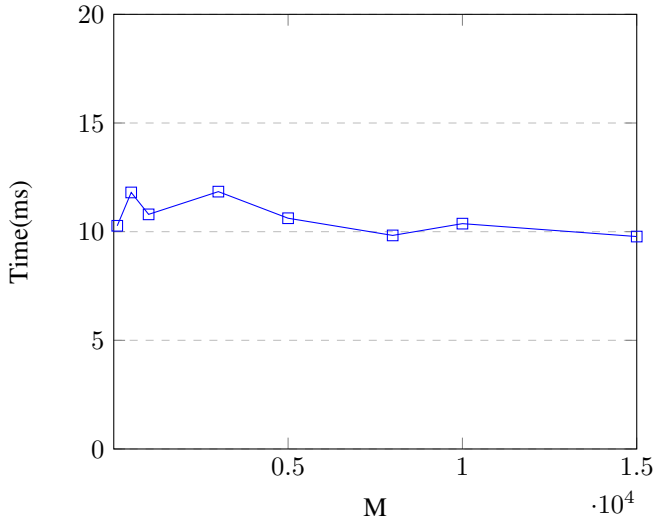
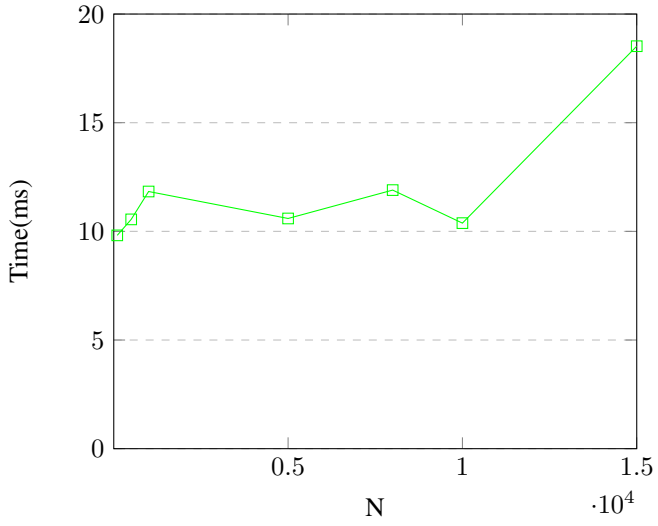


FIGURE2: TIME COMPLEXITY GRAPH FOR APOSTERIORI ANALYSIS. (KEEPING M CONSTANT AND N ON X-AXIS)



In the above graphs, the value 'N' and 'M' i.e., row and columns (randomly generated in their graph) are considered along the x-axis (M in Fig:1 and N in Fig:2) and time along the y-axis. The graph represents Time v/s M (or N).

V. Conclusion

The algorithms discussed in this paper can be used to find the maximum amount of gold collected by a miner while starting from any cell of the first column. traversing a matrix of N X M size whose each cell contains the amount of gold.In the matter of comparison between these two algorithms both approaches had the same space complexity. But, in Approach 1, time complexity would be higher as compared to Algorithm 2. So we can conclude that Dynamic Programming Solution i.e the Algorithm 2 is optimal and efficient.

VI. REFERENCES

[1] https://en.wikipedia.org/wiki/Dynamic$_{p}rogramming$

[2] https://www.geeksforgeeks.org/tabulation-vs-memoization/

[3] https://www.ideserve.co.in/learn/time-and-space-complexity-of-recursive-algorithms

[4] https://www.geeksforgeeks.org/auxiliary-space-recursive-functions/