Farthest and Nearest Neighbours of a Mobile Point in a Matrix

Mohd Mohtashim Nawaz∗ , Harsh Aryan†, Mayank Taksande‡ and Sneha Mishra§
Indian Institute of Information Technology, Allahabad
Allahabad-211012
Email: ∗itm2017005@iiita.ac.in, †itm2017003@iiita.ac.in, ‡ itm2017008@iiita.ac.in, § itm2017004@iiita.ac.in

***Abstract*—This Paper introduces an algorithm to find the k-nearest and k-farthest neighbours of a mobile point moving diagonally in a matrix, given the starting point and steps of unit size of the point. The point is to be moved diagonally in a matrix starting from the starting point provided and k-nearest and k-farthest neighbours are to be found at that point. The idea is to find an efficient algorithm in terms of time and space complexity.**

I. Introduction

We are to generate a 100 X 100 array and fill it with random numbers. Now from the starting point, we need to traverse diagonally. That is, if the starting point is in the top left corner, we will go till the bottom right corner, and if we are at the bottom left, we go to the top right corner and so on. At some cells of the matrix, we need to stop and find the k-nearest and k-farthest points.

k-nearest points mean the nearest points assuming one step we go is k units. Similarly, k-farthest points are the farthest points assuming one step is k units.

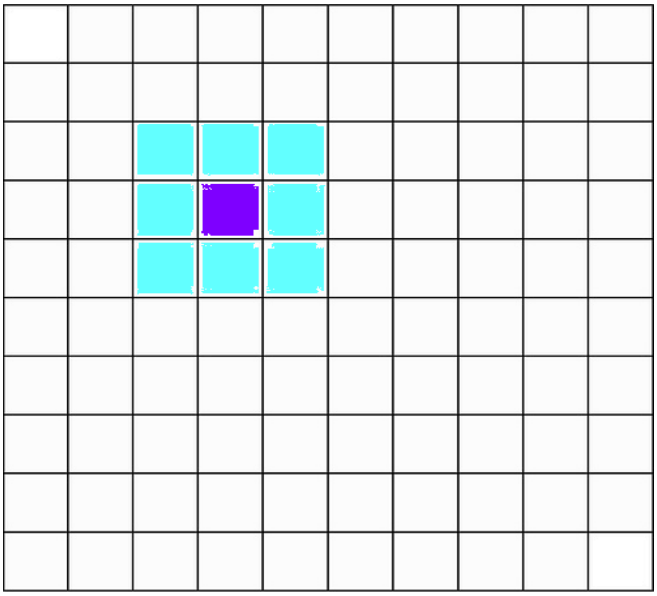

Fig 1.1: Figure depicting the nearest neighbours for k=1

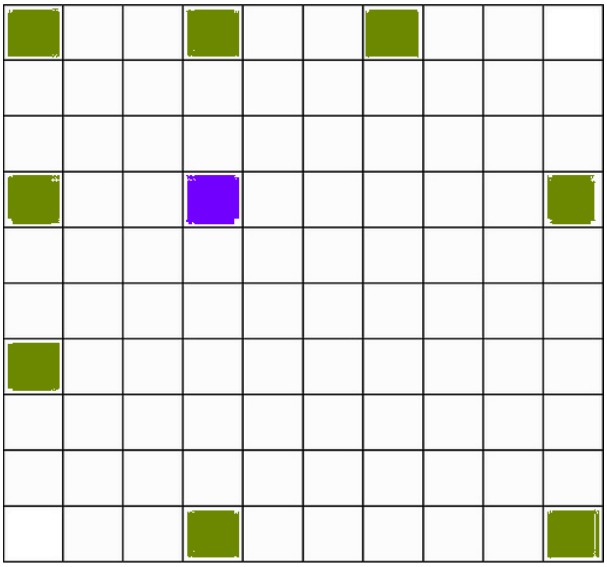

Fig 1.2: Figure depicting the farthest neighbours for k=1

Here we will try to develop an efficient algorithm that can move the point diagonally and find the k-nearest and k-farthest points in the matrix.

This algorithm is tested for different sample test cases for time complexity, space complexity and accuracy. In our case, we tend to focus on establishing a rule or a relationship between the time and input data.

This report further contains -

II. Algorithm Design and Analysis

III.Alternate Approach

III. Result

IV. Conclusion

V. References

II. Algorithm Design

An efficient algorithm can only be created by considering every possibility. So we have to consider every possibility while designing the algorithm.

Steps for designing the algorithm include:

1) Take k and coordinates of the starting point as input. Create a 100 X 100 matrix. Fill it with random numbers.

2) After that, traverse to the next diagonally located cell, starting from the start point as specified by the user. After every step movement, give a 2 seconds pause, so that the movement can be shown.

3) In the next step, we find the k-nearest points using k which has been taken as input from the user. The points are to be considered in every direction, that is 8 such neighbours are to be considered for all cases except the corners or edge cells.

4). Next, we find the k-farthest points using k as given already. Again the points are to be found along all possible directions. Then, move to the next iteration.

An algorithm to find the nearest neighbours:

```
int n ← 100 (as given in question)

def nearest_neighbours (int:matrix[][n],int:n,int:k,int:i,int:j)
        If (j+k) <= (n-1) then
                Print matrix[i][j+k]
        If (j-k)>=0 then
                Print matrix[i][j-k]
        If (i+k)<=(n-1) then
                Print matrix[i+k][j]
        If (i+k)<=(n-1)  and  (j+k)<=(n-1) then
                Print matrix[i+k][j+k]
        If (i+k)<=(n-1)  and (j-k)>=0 then
                Print matrix[i+k][j-k]
        If (i-k)>=0 then
                Print matrix[i-k][j]
        If (i-k)>=0 and (j-k)>=0 then
                Print matrix[i-k][j-k]
        If (i-k)>=0 and (j+k)<=(n-1)
                Print matrix[i-k][j+k]
end
```

Time Analysis:
Best case: Ω(26)
Worst case: O(52)

Algorithm to find farthest neighbours

```
def farthest_neighbours (int:matrix[][n],int:n,int:k,int:i,int:j)
{
        If (j+k)<=(n-1) then
                Print matrix[i][n-k]
        If (j-k)>=0 then
                Print matrix[i][k-1]
        If (i+k)<=(n-1) then
                Print matrix[n-k][j]
        If (i+k)<=(n-1) and (j+k)<=(n-1) then
                Print matrix[n-k][n-k]
        If (i+k)<=(n-1) and (j-k)>=0 then
                Print matrix[n-k][k-1]
        If (i-k)>=0 then
                Print matrix[k-1][j]
        If (i-k)>=0 and (j-k)>=0 then
                Print matrix[k-1][k-1]
        If (i-k)>=0 and (j+k)<=(n-1) then
                Print matrix[k-1][n-k]
```

Time Analysis:
Best case: Ω(26)
Worst case: O(52)

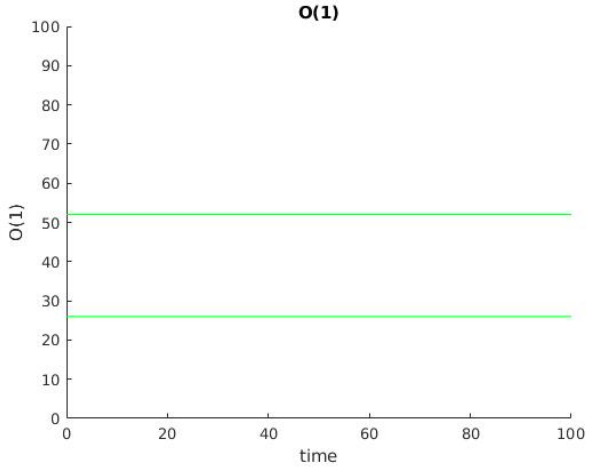

Fig 1.3: Graph depicting linear time complexity

Algorithm to generate a random matrix

```
def generatematrix (int:matrix[][n],int:n)
        srand (time(NULL))
        //standard libarary function
        int:i,j
        For i=0 to n-1 do
                For j=0 to n-1 do
                        Matrix[i][j] ← rand()%10
                        //generating random integers
```

Time Analysis:

Best case: Ω(5*n*n+2)
taking n=100, it gives 50002

Worst case: O(5*n*n+2)
taking n=100, it gives 50002

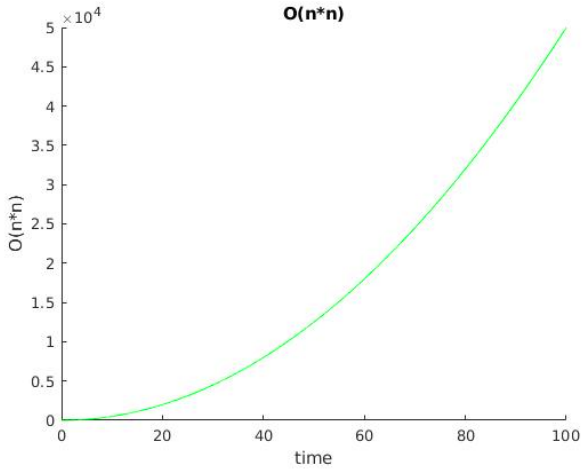

Fig 1.4: Graph depicting time complexity of function

Main Algorithm

```
def main()

        int:matrix[n][n]
        int:i,j
        generatematrix(matrix,n)

        For i=0 to n-1 do
                For j=0 to n-1
                        Print matrix[i][j]

        int:x,y,k
        Input → k,x,y
        If x=0 and y=0
                For i=x to n-1 do
                        For j=y to n-1 do
                            If i=j then
                                    Print i,j
                                    //position of point
                                    Print matrix[i][j]
                                    //value at that point
                                    nearest_neighbours
                                    (matrix,n,k,i,j)
                                    farthest_neighbours
                                    (matrix,n,k,i,j)
                                    sleep(2)
                        else if x=n-1 and y=n-1 then
                            For i=x to 0 do
                            For j=y to 0 do
                                    If i=j
                                     Print i,j
                                     Print matrix[i][j]

                                    nearest_neighbours
                                    (matrix,n,k,i,j)

                                    farthest_neighbours
                                    (matrix,n,k,i,j)

                                    sleep(2)
                else if x=0 and y=n-1 then
                        For i=x to n-1 do
                                For j=y to 0 do
                                        If i+j=n-1 then
                                            Print i,j
                                    //current point
                                    Print matrix[i][j]
                                    //value at current point
                                    nearest_neighbours
                                    (matrix,n,k,i,j)

                                    farthest_neighbours
                                    (matrix,n,k,i,j)

                                    sleep(2)

                        else if x=(n-1) and y=0 then
                            For i=x to 0 do
                            For j=y to n-1 do
                                If i+j = n-1 then
                                        Print i,j
                                        //current point
                                        Print matrix[i][j]
                                        //value at point
                                        nearest_neighbours
                                        (matrix,n,k,i,j)
                                        farthest_neighbours
                                        (matrix,n,k,i,j)
                                        sleep(2)
                            Else
                                Print “Invalid Point”
                                return 0
```

Time Analysis:
Best case: 6 + n*n*(5+26+26) = 57n*n + 6
Ω(n^2)
Worst case: 6 + n*n*(5+52+52) = 109n*n+6
O(n^2)

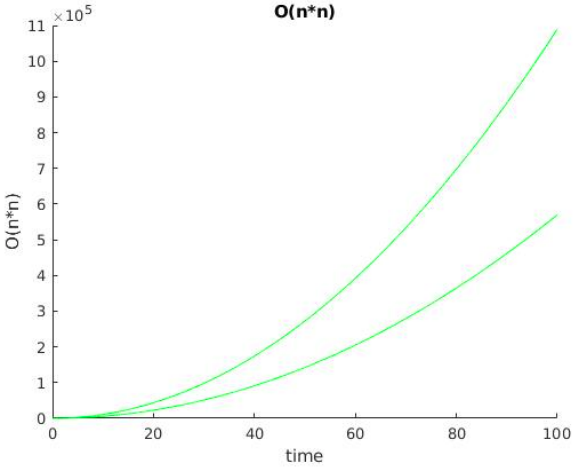

Fig 1.5: Graph depicting time complexity of code

III. Alternate Approaches

The above approach involves a number of conditional statements to ensure the boundary condition of the mobile point.
An alternate approach can be considered to avoid such comparisons by removing the need to check for boundary conditions for every point on the diagonal.
We can do so by constructing a matrix of 3*3 around the mobile point with it being in the middle of the matrix for all those cases where we do not have any element from original matrix we will put value 0 in those places

Thus a matrix of form :
3 1 4
1 5 9
2 6 5
can be visualised as :
0 0 0 0 0
0 3 1 4 0
0 1 5 9 0
0 2 6 5 0
0 0 0 0 0

Another Alternate Approach can be implemented by the following pseudo code

```
row_limit = count(array);
if(row_limit > 0){
  column_limit = count(array[0]);
  for(x = max(0, i-1); x <= min(i+1, row_limit); x++){
    for(y = max(0, j-1); y <= min(j+1, column_limit); y++){
      if(x != i || y != j){
        print array[x][y];}
    }
  }
}
```

IV. Result

Overall Time Complexity
Best case - Ω(n^2)
Worst case-O(n^2)

Overall Space Complexity
Best case -Ω(100*100)
Worst case-O(100*100)

V. Conclusion

We have developed an algorithm to solve the given problem, that is to find the k-farthest and k-nearest neighbours of a mobile point in a 100 X 100 matrix.

There could have been some other approaches however the one used is simple and both time and space efficient. Thus we have come up with the best possible approach of the given problem.

VI. References

[1] ”Sleep()”, Linux Man Page [Online] Available: https://linux.die.net/man/3/sleep [Accessed: March 9 2019]

[2] Thomas H. Corman, Introduction to Algorithms, 3rd Edition 2003

[3] Ellis Horowitz,Sartaj Sahni,Sanguthevar Rajasekaran,Fundamentals of Computer Algorithms, 2nd Edition