# ARK Task Round Documentation

## Basic Tasks

## Task 2.1 Optimise Me

### I. Introduction

The first task was related to decreasing the time complexity of the code. The code had to be changed to make it work for larger values of n. The first thing that I had tried to do was very basic things like decreasing the number of loops used in the program. Later on, I tried to use vectorisation, parallel programming and priority queue to decrease the time complexity of the code.

### II. Problem Statement

This task was divided into 3 smaller tasks. Basically, what we had to do was decreases the time that the program takes to run for higher order matrices. We had to make the program faster and more efficient.

The first part involved calculating the minimum cost required to go from a cell i,j to n,n. We had to declare a nxn matrix with random integers from 1 to 10. This integer represents the cost of landing on that particular cell. You had to find the total cost to reach the n,n cell and this cost had to be the minimum of all costs. Similarly, we had to declare another matrix but in this one we had to find the maximum cost to reach the n,n cell from any I,j start cell.(The only moves around are right and down).

| | | | |
|---|---|---|---|
| 1 | 7 | 5 | 8 |
| 8 | 6 | 9 | 4 |
| 4 | 4 | 5 | 2 |
| 9 | 1 | 2 | 3 |

For example, in the above 4X4 table the highlighted numbers represent the minimum cost path to go from the first cell to the nxnth cell. The minimum cost would be 23 in this case.

The second part of the question you had to multiply these two matrices, that is the minimum cost matrix A and the maximum cost matrix B. You had to multiply each element of matrix A with element of matrix B. The minimum cost matrix a is a matrix which has the minimum cost from that cell-to-cell n,n whereas the maximum cost matrix B has the maximum cost from that cell-to-cell n,n.

In the third part of the task, you had to apply a filter on the product matrix. The filter would be a matrix of dimensions 4xn initialised by random values between 0 and 2. On applying the filter on the product matrix you would final matrix of dimensions n/4. This matrix would be the sum of the dot product of the product matrix and filter matrix.

So, if the product matrix is [[2,7,5,2], [1,4,2,8]] and filter Matrix is [[1,1,2,1], [1,1,1,2]] the final matrix would be [44]

## III. Related Work

I had tried to use Dijkstra's algorithm for the first part of the problem and priority queue to decreases the time complexity. I had tried dynamic programming as well. For the second part I had decided to create another matrix that saves the cost from the nxn cell to visit that particular cell instead of calling the function again and again so as to save time. For the third part I had decided to use vectorisation and parallelisation to decrease the time that the code takes to run.

## IV. Initial Attempts

Firstly, I had just printed the original matrix and the cost of the matrix. It was not showing the correct cost. So, then I realised there was some mistake with the return statements that was creating an error and hence showing the wrong value for the minimum cost. To make things simpler I declared a 4x4 matrix instead of initialising it with random values and used that matrix to check if the code was running correctly or no. Also, I used debugging techniques to keep a track of how the variable values were changing thus making it easier to understand the code ad debug errors. I also had used a lot of print statements to understand till where the code was running alright and again to make it easier for debugging. Also, instead of running the entire code at once, i.e all the 3 parts of the codes at once I created smaller snippets of the codes. For vectorisation initially I had run a simple for loop that displays a multiplication table to get familiarized with the vectorisation techniques.

## V. Final Approach

To compute the first part of the task I have used Dijkstra's algorithm with Min Heap.

The algorithm finds the minimum cost to reach the n,n cell. We have defined moves (0,1) that is either to move down or (1,0) that is to move right. We have declared a Boolean function to check if the cell is within boundary. We have also defined a structure to store the coordinates of the cell and the cost of each cell. We have declared arrays to store the minimum cost of the path and to store if a cell has been visited or not. We have defined a reverse priority queue to find the minimum cost. We are using recursion to find the minimum cost path. In the main function we have initialised the cost matrix.

This code is to find the minimum cost of the path.

```
/* Minimum Cost Path using Dijkstra's shortest path

   algorithm with Min Heap  */

#include <stdio.h>

#include <queue>
```

```
#include <limits.h>

#include <bits/stdc++.h>

#include <iostream>

using namespace std;


/* define the number of rows and the number of columns */

const int sizen=4;




/* 2 possible moves */

int dx[] = {1, 0};

int dy[] = {0, 1};


/* The data structure to store the coordinates of \\

  the unit square and the cost of path from the top left. */

struct Cell{int x;int y;int cost;};


/* The compare class to be used by a Min Heap.

 * The greater than condition is used as this

   is for a Min Heap based on priority_queue.

 */

class mycomparison

{

public:

  bool operator() (const Cell &lhs, const Cell &rhs) const

  {

    return (lhs.cost > rhs.cost);

  }

};


/* To verify whether a move is within the boundary. */
```

```
bool isSafe(int x, int y)
{
    return x >= 0 && x < sizen && y >= 0 && y < sizen;
}

/* This solution is based on Dijkstra’s shortest path algorithm
 * For each unit square being visited, we examine all
    possible next moves in 2 directions,
 *    calculate the accumulated cost of path for each
     next move, adjust the cost of path of the adjacent
     units to the minimum as needed.
 *    then add the valid next moves into a Min Heap.
 * The Min Heap pops out the next move with the minimum
   accumulated cost of path.
 * Once the iteration reaches the last unit at the lower
   right corner, the minimum cost path will be returned.
 */
int minCost(int cost[sizen][sizen], int m, int n)
 {

    /* the array to store the accumulated cost
       of path from top left corner */
    int mincost[sizen][sizen];

    /* the array to record whether a unit
       square has been visited */
    bool visited[sizen][sizen];

    /* Initialize these two arrays, set path cost
      to maximum integer value, each unit as not visited */
    for(int i = 0; i < sizen; i++)
```

```
{
    for(int j = 0; j < sizen; j++)
     {
        mincost[i][j] = INT_MAX;
        visited[i][j] = false;
    }
}

/* Define a reverse priority queue.
 */
priority_queue<Cell, vector<Cell>, mycomparison> pq;

/* initialize the starting top left unit with the
  cost and add it to the queue as the first move. */
mincost[0][0] = cost[0][0];
pq.push({0, 0, cost[0][0]});

while(!pq.empty())
 {

    /* pop a move from the queue, ignore the units
       already visited */
    Cell cell = pq.top();
    pq.pop();
    int x = cell.x;
    int y = cell.y;
    if(visited[x][y]) continue;

    /* mark the current unit as visited */
    visited[x][y] = true;
```

```
        /* examine all non-visited adjacent units in 2 directions
         * calculate the accumulated cost of path for
           each next move from this unit,
         * adjust the cost of path for each next adjacent
           units to the minimum if possible.
         */
        for(int i = 0; i < 2; i++)
         {
            int next_x = x + dx[i];
            int next_y = y + dy[i];
            if(isSafe(next_x, next_y) && !visited[next_x][next_y]) {
                mincost[next_x][next_y] = min(mincost[next_x][next_y],
                   mincost[x][y] + cost[next_x][next_y]);
                pq.push({next_x, next_y, mincost[next_x][next_y]});
            }
        }
    }

    /* return the minimum cost path at the lower
       right corner */
    return mincost[m][n];
}

/* Driver program to test above functions */
int main()
{
  int cost[sizen][sizen];
   for (int i = 0; i < sizen; i++)
    {
        for (int j = 0; j < sizen; j++)
        {
            cost[i][j] = 1 + rand() % 10;



        }

    }

   printf(" %d ", minCost(cost, sizen-1, sizen-1));

   return 0;

}
```

The algorithm finds the maximum cost to reach the n,n cell. We have defined moves (0,1) that is either to move down or (1,0) that is to move right. We have declared a Boolean function to check if the cell is within boundary. We have also defined a structure to store the coordinates of the cell and the cost of each cell. We have declared arrays to store the cost of the path and to store if a cell has been visited or not. We have defined a reverse priority queue. We are using recursion to find the maximum cost path. In the main function we have initialised the cost matrix.

This code is to find the maximum cost of the path.

```
/* Maximum Cost Path using Dijkstra’s shortest path

   algorithm with Min Heap  */

#include <stdio.h>

#include <queue>

#include <limits.h>

#include <bits/stdc++.h>

#include <iostream>

using namespace std;


/* define the number of rows and the number of columns */
```

```
const int sizen=4;



/* 2 possible moves */
int dx[] = {1, 0};
int dy[] = {0, 1};

/* The data structure to store the coordinates of \\
  the unit square and the cost of path from the top left. */
struct Cell{int x;int y;int cost;};

/* The compare class to be used by a Min Heap.
 * The greater than condition is used as this
   is for a Min Heap based on priority_queue.
 */
class mycomparison
{
public:
  bool operator() (const Cell &lhs, const Cell &rhs) const
  {
    return (lhs.cost < rhs.cost);
  }
};

/* To verify whether a move is within the boundary. */
bool isSafe(int x, int y)
{
    return x >= 0 && x < sizen && y >= 0 && y < sizen;
}

/* This solution is based on Dijkstra's shortest path algorithm
```

```
 * For each unit square being visited, we examine all
    possible next moves in 2 directions,
 *    calculate the accumulated cost of path for each
     next move, adjust the cost of path of the adjacent
     units to the maximum as needed.
 *    then add the valid next moves into a Min Heap.
 * The Min Heap pops out the next move with the maximum
   accumulated cost of path.
 * Once the iteration reaches the last unit at the lower
   right corner, the minimum cost path will be returned.
 */
int maxCost(int cost[sizen][sizen], int m, int n)
 {

    /* the array to store the accumulated cost
       of path from top left corner */
    int maxcost[sizen][sizen];

    /* the array to record whether a unit
       square has been visited */
    bool visited[sizen][sizen];

    /* Initialize these two arrays, set path cost
      to maximum integer value, each unit as not visited */
    for(int i = 0; i < sizen; i++)
    {
        for(int j = 0; j < sizen; j++)
         {
            maxcost[i][j] = INT_MIN;
            visited[i][j] = false;
        }
```

```
}

/* Define a reverse priority queue.
 */
priority_queue<Cell, vector<Cell>, mycomparison> pq;

/* initialize the starting top left unit with the
  cost and add it to the queue as the first move. */
maxcost[0][0] = cost[0][0];
pq.push({0, 0, cost[0][0]});

while(!pq.empty())
 {

    /* pop a move from the queue, ignore the units
       already visited */
    Cell cell = pq.top();
    pq.pop();
    int x = cell.x;
    int y = cell.y;
    if(visited[x][y]) continue;

    /* mark the current unit as visited */
    visited[x][y] = true;

    /* examine all non-visited adjacent units in 2 directions
     * calculate the accumulated cost of path for
       each next move from this unit,
     * adjust the cost of path for each next adjacent
       units to the minimum if possible.
     */
```

```
        for(int i = 0; i < 2; i++)
         {
            int next_x = x + dx[i];
            int next_y = y + dy[i];
            if(isSafe(next_x, next_y) && !visited[next_x][next_y]) {
                maxcost[next_x][next_y] = max(maxcost[next_x][next_y],
                   maxcost[x][y] + cost[next_x][next_y]);
                pq.push({next_x, next_y, maxcost[next_x][next_y]});
            }
        }
    }

    /* return the minimum cost path at the lower
       right corner */
    return maxcost[m][n];
}

/* Driver program to test above functions */
int main()
{
  int cost[sizen][sizen];
   for (int i = 0; i < sizen; i++)
    {
        for (int j = 0; j < sizen; j++)
        {
            cost[i][j] = 1 + rand() % 10;



        }
    }
   printf(" %d ", maxCost(cost, 3, 3));
```

```
    return 0;
}
```

For the second part I have used dynamic implementation. So firstly, I have inverted the array, that is cell 0,0 is cell n,n. Then I have declared another array that stores the minimum cost of reaching the particular i,j cell from the start cell. After that I have inverted this array so as to find the minimum cost from that cell to the nxn cell. Lastly, I have tried to apply sequential element access to reduce the time taken.

```
#include <bits/stdc++.h>
#include <iostream>


using namespace std;


const int sizen = 4;


//Initialisng variables
long long CostMatrixAA[sizen][sizen];
long long CostMatrixBB[sizen][sizen];
long long CostMatrixA[sizen][sizen];
long long CostMatrixB[sizen][sizen];
long long A[sizen][sizen];
long long B[sizen][sizen];
long long AA[sizen][sizen];
long long BB[sizen][sizen];
long long productMat[sizen][sizen];
```

```
//Function to store the minimum cost of reaching cell n,n from that cell.
long long FindMinCostA(int i, int j, int n)
{
    A[0][0]=CostMatrixA[0][0];
    int count=0;
    for(int i=1;i<sizen;i++)
    {
        for(int j=1;j<sizen;j++)
      {
          //Initialise cells to zero
          A[i][j]=0;
          if(count<4)
          {
              //initialise  cells to 0
               A[0][j]=0;
               A[j][0]=0;
               //To store the cost values of first row and first column cells.
               A[j][0]=CostMatrixA[j][0]+A[j-1][0];
               A[0][j]=CostMatrixA[0][j]+A[0][j-1];
          }
          count++;
        //To find the minimum cost
          A[i][j]=min(A[i-1][j],A[i][j-1])+CostMatrixA[i][j];


      }
    }
    // To invert the matrix as the cost matrix has initially been inverted to
find the cost
    for(int i=0;i<sizen;i++)
    {
        for(int j=0;j<sizen;j++)
        {
            AA[i][j]=A[sizen-i-1][sizen-1-j];
        }
    }

    return A[3][3];
}
//Function to store the maximum cost of reaching cell n,n from that cell.
long long FindMaxCostB(int i, int j, int n)
{
    B[0][0]=CostMatrixB[0][0];
    int count=0;
    for(int i=1;i<sizen;i++)
    {
        for(int j=1;j<sizen;j++)
      {

          //Initialise cells to zero
          B[i][j]=0;
          if(count<4)
          {
              //initialise  cells to 0
               B[0][j]=0;
               B[j][0]=0;
               //To store the cost values of first row and first column cells.
               B[j][0]=CostMatrixB[j][0]+B[j-1][0];
               B[0][j]=CostMatrixB[0][j]+B[0][j-1];
          }
          count++;
           //To find the maximum cost
          B[i][j]=max(B[i-1][j],B[i][j-1])+CostMatrixB[i][j];
      }
    }
    // To invert the matrix as the cost matrix has initially been inverted to
find the cost
    for(int i=0;i<sizen;i++)
    {
        for(int j=0;j<sizen;j++)
        {
            BB[i][j]=B[sizen-i-1][sizen-1-j];
        }
    }

    return B[3][3];
}
int main()
{
    int t;
    int s;
    //Initialising cost matrices
     for (int i = 0; i < sizen; i++)
    {
        for (int j = 0; j < sizen; j++)
        {
            t=1 + rand() % 10;
            s=1 + rand() % 10;
            CostMatrixAA[i][j]=t;
            CostMatrixA[sizen-1-i][sizen-1-j]=t;
            CostMatrixBB[i][j]=s;
            CostMatrixB[sizen-1-i][sizen-1-j]=s;
        }
    }
    // Calling the function
   printf(" %lld\n", FindMinCostA(0, 0, sizen));
   printf(" %lld\n", FindMaxCostB(0, 0, sizen));
   long long int temp1,temp2;
   //Declaring product matrix
   for (int  k= 0; k < sizen; k++)
    {
        for (int j = 0; j < sizen; j++)
        {
            for (int i = 0; i < sizen; i++)
                {
                    temp1=AA[i][k];
                    temp2=BB[k][j];
                    productMat[i][j] += temp1*temp2;
                }
        }


    }
}
```

For the third part of the code, I have tried to use vectorisation and parallelisation. I have used openMD SIMD wherein you can have multiple threads running together. You can declare the number of threads that you wish to declare in the bash shell using 'export OMP_NUM_THREADS=n'. For parallelisation I have tried using **Intel's Threading Building Blocks (TBB)**. And the library used **'tbb/parallel_for.h'** but I was not able to proceed much due to compiler issues. The format to parallelise a loop is **'tbb::parallel_for( range, kernel );'**. Also, I have declared the kernel along with the cost matrices to reduce the number of loops.

```
#pragma omp simd
long long finalMat[sizen / 4];
    // applying the filter
    for (int i = 0; i < sizen - 4; i += 4)
    {
        long long sum = 0;
        // dot product of 4xn portion of productMat
        for (int j = 0; j < sizen; j++)
        {
            for (int filterRow = 0; filterRow < 4; filterRow++)
            {
                sum += productMat[i + filterRow][j];
            }
        }
        finalMat[i / 4] = sum;
    }


long long filterArray[4][sizen];
     for (int i = 0; i < sizen; i++)
    {
        for (int j = 0; j < sizen; j++)
        {
            t=1 + rand() % 10;
            s=1 + rand() % 10;
            CostMatrixAA[i][j]=t;
            CostMatrixA[sizen-1-i][sizen-1-j]=t;
            CostMatrixBB[i][j]=s;
            CostMatrixB[sizen-1-i][sizen-1-j]=s;
            filterArray[i][j] = rand() % 2;


        }

    }
```

The first problem that I had faced was that I was not able to implement priority queue in Dijkstra's algorithm and I was able to do so by reading online sources. Another error that I was facing was the cells were not getting properly initialised during dynamic programming. Like some of them were getting initialized after the cost matrix statements and hence the output was messed up. Debugging techniques helped me to rectify this. And while trying to do vectorisation it was not able to recognise openMp library so I ended up downloading a lot of libraries and other stuff.

## Result and Observations

Firstly, I noticed that using loops was quite expensive when it comes to time complexity. Also, by changing small things like repetitive initialisation or by changing the order of loops could help you increase the efficiency of your program. Another thing was that priority queues are way more efficient. Dynamic allocation also helps you to save a great deal of time.

A drawback of my algorithm is that there are still a lot of usage of loops. Like in dynamic allocation the inversion of matrices adds up extra time.

I picked up this approach as I found it more user friendly and it was efficient as well as easy to implement.

## Future Work

So, the areas I feel I have to work on this code is firstly implement parallelisation. Also I was not able to use dynamic allocation and priority queues in a single set of programs so I would want to work on that as well. I had read about locality of references but I was not able to implement it. And lastly I would want to use the Linux operating system cause windows has a lot of security and to do the simplest of tasks I ended up banging my head.

## Conclusion

So, the problem in short was about time complexity. I solved it using Dijkstra's algorithm, priority queues, sequential element access, dynamic programming and vectorisation. (And parallel programming too XD)

Umm the maze solving techniques could be used to find the shortest path or the path that takes the least time or the path that has the route with least obstacles. Reduction of time complexity would ensure faster and more efficient programming.

## References

https://www.quora.com/What-do-you-do-to-improve-the-time-complexity-of-an-algorithm

http://www.algorithmsandme.com/minimum-cost-path-matrix/

https://www.codingninjas.com/codestudio/problem-details/minimum-cost-to-destination_630411

https://www.geeksforgeeks.org/locality-of-reference-and-cache-operation-in-cache-memory/

https://chryswoods.com/parallel_c++/parallel_for.html

https://chryswoods.com/vector_c%2B%2B/limitations.html