Analysis of Algorithms
Project 2 Pathfinder Report

Group 8 Members: Maggie Liott, Vatsal Kapoor, Jason Vo

Questions:

1. **How can you break down the problem of finding the minimum distance to one of the cells on the bottom row of the matrix into one or more smaller subproblems? Your answer should include how the solution to the original problem is constructed from the subproblems and why this breakdown makes sense.**
    a. The problem of finding the minimum distance to one of the cells on the bottom row of the matrix can be broken down into smaller subproblems of choosing the smallest of the 3 options at each level. We traversed the matrix from top to bottom and found the optimal path to each cell on the bottom row. Each level is calculated and the final result is returned at the bottom row.

2. **What recurrence can you use to model the minimum distance problem using dynamic programming?**
    a. We can use a 2D array of our defined struct (length, start cell, and path) with indexes i and j iterate through the matrix and recursively find the best choice at each row. Once at the bottom row of the matrix, the total length, start column, and path can be returned as the output. The pseudocode below shows how this works in more detail.

3. **What are the base cases of this recurrence?**
    a. The base case of this recurrence is at the top row. For example, with only one row, (R = 1, C = n) the most efficient path across the terrain is the value in each column.
4. **Give pseudocode for a memoized algorithm that computes the solution to this problem.**

Minimum Distance Algorithm
INPUT: 2d array arr, number of rows m, number of columns n
OUTPUT: minimum distance value
Global Matrix[m+1][n+1] initialized with -1 values

Function graphTraversal(arr,m,n):

```
If matrix[m][n] is not -1:
        Return matrix[m][n]
if n < 0 or n > number of columns in array arr:
    matrix[m][n]= positive infinity
Else if at top most row (m==0):
    matrix[m][n]= arr[m][n]
else:
```

```
    a=arr[m][n]+graphTraversal(arr,m-1,n)
    b=1.4*arr[m][n]+graphTraversal(arr,m-1,n+1)
    c=1.4*arr[m][n]+graphTraversal(arr,m-1,n-1)
    matrix[m][n]= min(a,b,c)
  Return matrix[m][n]
```

5. **What is the time complexity of your memoized algorithm?**
    a. The time complexity for our memoized algorithm is O(m*n) because all the possible recursive calls are completely executed once and then the next time the function is called with the same arguments, it takes O(1) time since we already have it memoized and have a memoization check at the beginning of the function.

6. **Give pseudocode for an iterative dynamic programming algorithm for this problem.**

Iterative Minimum Distance Algorithm
INPUT: 2d array arr, number of rows m, number of columns n
OUTPUT: minimum distance value

```
Function iterativeGraphTraversal(arr, m, n):
    2d array result[m+1][n+1]
    for j from 0 to n:
       result[0][j] = arr[0][j]

    for i from 1 to m:
        for j from 0 to n:
           temp = matrix[i][j]
           a=temp + result[i - 1][j]
           b=(1.4 * temp) + result[i - 1][j - 1] if j-1>=0 else infinity
           c=(1.4 * temp) + result[i - 1][j + 1] if j + 1 < n else infinity
           result[i][j]=min(a,b,c)

     return result
```

7. **Give pseudocode for an algorithm that computes the optimal route that ends at a given column on the first row of the matrix based on the dynamic programming data structure computed by your algorithm. If it is helpful, you may choose to modify your dynamic programming algorithm to make this problem easier to solve; however, you must describe any modification that are necessary.**

Algorithm: shortestPath

Input: matrix
Output: length, start cell, and the optimal route to reach each cell in the bottom row

```
   // Create a new matrix for the output (same size with input matrix)
   output = new 2D vector with dimensions matrix.rows x matrix.cols of cell (double length, int start_cell, string path)

   // Initialize the first row with the same values as the input matrix
   for i from 0 to matrix.cols - 1:
      output[0][i].length = matrix[i]
      output[0][i].start_cell = i
      output[0][i].path = ""

   // Case when there is only one row:
   if matrix.rows == 1:
      return output

   // Compute the optimal path for each cell in the output matrix
   i = 1
   while i < matrix.rows:
      j = 0
      while j < matrix.cols:
         // Set up the south, south_east, and south_west values to be infinity
         south = infinity
         south_east = infinity
         south_west = infinity

         // Case going south
         if i - 1 >= 0:
            south = output[i - 1][j].length + matrix[i][j]

         // Case going south west
         if i - 1 >= 0 and j + 1 < matrix.cols:
            south_west = output[i - 1][j + 1].length + 1.4 * matrix[i][j]

         // Case going south east
         if i - 1 >= 0 and j - 1 >= 0:
            south_east = output[i - 1][j - 1].length + 1.4 * matrix[i][j]
```

```
        // Initialize the length of the cell with the minimum value of south, south east, and south
west
        output[i][j].length = min(south, min(south_east, south_west))

        // Case south is the optimal path
        if output[i][j].length == south:
            output[i][j].start_cell = output[i - 1][j].start_cell
            output[i][j].path = output[i - 1][j].path + " S"

        // Case south east is the optimal path
        else if output[i][j].length == south_east:
            output[i][j].start_cell = output[i - 1][j - 1].start_cell
            output[i][j].path = output[i - 1][j - 1].path + " SE"

        // Case south west is the optimal path
        else:
            output[i][j].start_cell = output[i - 1][j + 1].start_cell
            output[i][j].path = output[i - 1][j + 1].path + " SW"

        j = j + 1
    i = i + 1

  return output
```