Summer Orienteering

Assumptions:

```
# speedMap for each type of terrain
# 6 being fastest and 0 being the slowest
speedMap = {
    "openLand": 6,
    "roughMeadow": 0.5,
    "easyMovementForest": 2,
    "slowRunForest": 3,
    "walkForest": 4,
    "impassibleVegetation": 0.5,
    "lakeSwampMarsh": 0.5,
    "pavedRoad": 5,
    "footpath": 3,
    "outOfBounds": 0
}
```

For speed, on the scale of 0 – 5, 5 is considered to be the fastest while 0 the lowest.

1. All the constant speed values are considered after seeing the respective images for each of the given terrain type.

2. OUT_OF_BOUND is considered not reachable. So, its speed is taken 0.

3. IMPASSIBLE_VEGETATION, ROUGH_MEADOW and LAKE_SWAMP are considered as hurdles which the participants will just go by from side, dodging them.
   It's actually not feasible for orienteering and hence they are just given 0.5 as speed.

4. The input terrain image only given geographical representation and their colours are the same as mentioned in the question of this assignment.

Executing the program:

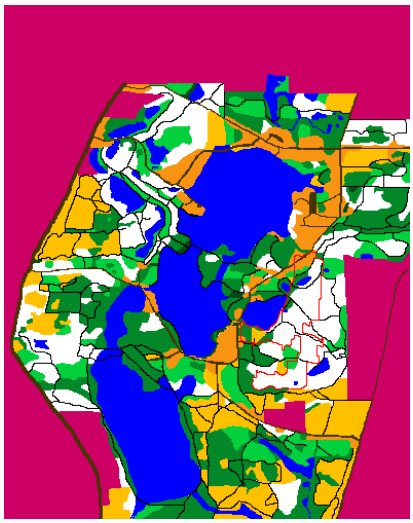
For executing the program, run **lab1.py** on your system with 4 arguments, namely inputImageName, elevationMap, courseFile and outputImageName.

The command in the terminal should be something like this,

```
>python lab1.py terrain.png mpp.txt red.txt redOut.png
```

For the output:

I'm using RED (255, 0, 0) colour to denote the optimal path drawn in this program.
The output image will be stored in the same directory the file, lab1.py will be executed from.



Files:

Lab1.py

1. I've used three python libraries – **PIL**, **sys** and **math**.

   PIL is used to work with the input and output image.
   sys is used to take arguments from the user.
   math is used to perform arithmetic operations for calculating the distance.

2. The program contains a class **Node**, which represents a pixel.

   The class handles attributes as x coordinate, y coordinate, type, elevation, parent and score of a particular Node. Pixel in our case.

   Concept of Node is used as we needed to find the neighbours and go back and forth for a particular pixel. Therefore, I decided to use Node to represent a pixel.

3. We then three functions - **calculateG**, **calculateH** and **calculateF.**

   - **calculateG** calculates the cost between two nodes/pixels.
   - Real world pixel size is used. We use all the three coordinates x,y and z to calculate the Euclidean distance between two nodes.
   - Cost, which is nothing but time in our case is nothing but distance/speed.

- Distance is calculated using the x, y and z coordinates, the elevations and are multiplied by the appropriate longitude and latitude factor.
- Speed (SpeedMap) is already defined based on the data given in the assignment.

- **calculateH** calculates the heuristics for a node.
- Again the Euclidean distance between the two nodes is calculated by using $sqrt((x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2)$.
- We calculated the values of z1 and z2 by using making use of the elevation dataset.
- We directly mapped elevation data points on the x and y coordinates and the matched coordinates were nothing but the z coordinates.

- **calculateF** is nothing but calculateG + calculateH that is,
  $f(n) = g(n) + h(n)$ where f(n) is the score, g(n) is the cost function and h(n) is the heuristics.
- Based on the axis we have, that is either x axis(horizontal) or y axis(vertical) we calculate the score for node.
- We then return the total score calculated for the same.

4. We then define a function **possibleNeighbors** for exploring and storing all the possible and valid neighbors for the node.

We keep track of 4 neighbors (4 pixels) as in left, right, top and bottom along with handling all the other edge cases.

We check for the node to be actually in bound that is inside of our image which is 395*400 in dimensions. So, 395 and 400 are considered as edges and all the edge cases are considered by exploring the extreme values of x and y coordinates.

A list of valid neighbors is then returned.

5. From the list of valid neighbors, we actually need to select the neighbor with the lowest f-score and then keep continuing the process till we reach our destination node and that's what the **nodeWithMinScore** function do.

6. Now we have everything necessary to feed to our A* algorithm. We implement the A* search algorithm by giving all the necessary parameters and the algorithm starts to find the optimal path. Function **aStarSearch** is implemented.

If the path is not found, it simply prints, No path!
Whereas, if found, it returns the finalPath stored in a list.

7. Now we actually kind of build of own terrain by using the Image data and elevations file. Both, terrain data from image and elevation data combines together to form a single data structure.

8. We then define the different types of terrains and their respective speeds.

9. The main() function. Program execution starts from main().

   In the main function,

   - We define four arguments for which values will be given by the user.
   - We store the image data in a array.
   - We split the course files and store them as list of list of x and y coordinates.
   - We also, take care of our elevation dataset. We skip the last 5 lines of every row and store each line as a list of list of these elevation dataset points.
   - We then start the traversing by calling the A*.
   - The A* runs till the length of control points is exhausted.
   - The control points are where our startNode and destinationNode will reside.
   - After the get the optimal path, we have the path stored in the finalPath list that e defined earlier.
   - We just iterate over the coordinates in the finalPath and draw a line in RED color to display the path that our A* found.
   - We need save the image and output the length of the path.