CS142, P.A. Late Submission **Due: 23:59, Tue 03/19/2023**

Name: Majd Kawak / NetID: mkawa025 / CodeForces ID: mkawa025 / Student ID: 862273310

Programming II (B. Transfer Money)

Submission#: 195494624

The main function first reads in the number of friends n, the number of edges m, and input all friends & edges and store them in an *adjacency_list* & *unordered_map* respectivley. Since routes are bidirectional I allocate 2 edges in my *edge_Map* representing forward and backward direction.

I declared *send_Money* function that implements **Dijkstra's Algorithm with Priority Queue** by initializing an array called *money* of size $num\_friends$ and sets all its values to INT_MAX, except for the start node, which is set to 100. It then initializes a priority queue called *pq*, which has a time complexity of $O(log\ n)$ for insertion and deletion operations, where n is the number of elements in the queue.

The *send_Money* function then enters a while loop that continues until *pq* is empty. In each iteration of the loop, it extracts the minimum value from *pq*, which has a time complexity of $O(log\ n)$. It then loops through all the neighbors of the extracted node and updates the money array and *pq* as necessary. This for loop has a time complexity of $O(m)$ because it iterates through each edge exactly once.

The time complexity of the *send_Money* function is $O((m + n)\ log\ n)$, where m is the number of edges and n is the number of friends. Therefore, the overall time complexity is $O((m + n)\ log\ n)$.

Programming II (C. Making New Friends)

Submission#: 196072093

****Note: I was previously given 0.3 points****

The main function first reads in the number of *rows* and *cols*, then I declare a structure called $UnionFind$ that takes $(rows * cols)$ as number of nodes and sets all nodes parent themself as well as their rank to 0. Then, I declare a 2d Matrix *grid_Map* to store shyness level as well as node number, lastly I declare an *edge_map* to represent edges. After scanning all "Shyness" levels, I compute all the edges in the grid using the *compute_Edges* function, which takes $O(N^2)$ time, where it scans left and down for all nodes in the grid and its the **most costly operation in my code**.

The edges are then sorted using the *sort* function, which takes $O(E\ log\ E)$ time. Finally, the code iterates over all the edges in the sorted order and performs union-find operations using the $UnionFind$ class, which takes O(E log E) time.

The $UnionFind$ class is used to keep track of connected components in the graph. The $find$ method of the class takes $O(log\ N)$ time, where N is the number of nodes in the graph, and the *unite* method takes $O(log\ N)$ time. The overall time complexity of the $UnionFind$ class in the given code is $O(E\ log\ N)$.

Therefore, the overall time complexity is $O(N^2 + E\ log\ E + E\ log\ N) = O(N^2)$.

Programming II (D. Stake Your Claim)

Submission#: 198062967

The main function first reads in the number of *rows* and *cols* and initializes the matrix to $(rows * 2) + 1$ by $(cols * 2) + 1$. Then i go and input all characters into matrix, my code initialize cell to 1 if its an area or pass through and 0 otherwise. Then I go and declare a function $color_matrix$ to colors different sections of my input matrix with different numbers, the function first initializes a visited matrix, which takes $O(((rows * 2) + 1) * ((cols * 2) + 1))$ time. Then it traverses through each cell in the input matrix and calls the DFS function if the cell is not visited and not equal to 0.

The DFS function can visit all the adjacent cells that are not visited and not equal to 0, which takes $O(1)$ time for each cell, assuming constant time operations. Therefore, the *color_matrix* function takes $O(((rows * 2) + 1) * ((cols * 2) + 1))$ time.

After all matrix have been colored with different numbers, I declare a function called $count_odds$ function also traverses through the input matrix, but only the odd-indexed rows and columns. It checks if the cell is not equal to 0 and increments the count for the corresponding section. Since there are $(\frac{rows*2)+1}{2}) * (\frac{(cols*2)+1)}{2})$ such cells, the time complexity of this function is $O((\frac{rows*2)+1}{2}) * (\frac{(cols*2)+1)}{2}))$. Lastly I sort my 4 element array that stores the section area count for all 4 sections, which takes $O(4) = O(1)$ and display results.

Therefore, the overall time complexity is $O(((rows * 2) + 1) * ((cols * 2) + 1))$.

Programming III (D. Disneyland)

Submission#: 197702418

The main function first reads in all point on map, then I call *convexHull* that follows **Graham Scan** to solve Convex Hull problem(GeeksforGeeks). The algorithm first sorts the given points based on their polar angles with respect to the point with the lowest y-coordinate (ties are broken by the x-coordinate). Therefore, the time complexity of sorting the points is $O(n\ log\ n)$.

Once the points are sorted, the algorithm uses a stack to keep track of the points that form the convex hull. The stack is initialized with the first three points from the sorted list. Then, for each subsequent point in the sorted list, the algorithm checks whether the point makes a left turn or a right turn with respect to the last two points on the stack. If the point makes a left turn, it is added to the stack. If the point makes a right turn, the algorithm pops the last point from the stack until the point makes a left turn with respect to the last two points on the stack.

This process continues until all the points have been processed. The time complexity of this process is $O(n)$. Finally, the algorithm computes the area of the convex hull using the Shoelace formula. This takes $O(n)$ time because it involves iterating over all the vertices of the convex hull. This program was originally intended to have large input points, put in our case we just have 4 points. Therefore, the overall time complexity is $O(4) = O(1)$.