Shubham Gupta **Theoretical Assignment: 2**
180749 **Part: 1**

- ## Problem 1

  Only articulation points when removed can destroy the correctness of a graph G.
  To solve our problem we will use DFS, find articulation point and a method to reverse/transpose the graphs.
  After transposing G_1(for non empty set S) and G_2(for non empty set T), we will take an intersection of these 2 and get a new graph G_new, we find the articulation points in G_new. If no articulation point is found the graph is connected or else it is not.

  - **Pseudo-code**

```
\\ Transposing the graph
\\ function to add edge from source to destination
AE (adj[], s, d)    {}

\\ main function using adjacency list of graph and adjacency list of transpose
graph
transpose(adj[], trn[], cnt){
        For i = 0 to cnt
                For j = 0 to adj.size
                        AE(trn, adj[i][j], i);
}

\\ Finding articulation point
DFS(v)
visited(v) ← true; DFN[v] ← dfn++; highpt ← ∞;
for each neighbout w of v {
        if (Visited(w) = false) {
                Parent(w) ← v; DFS(w);
                highpt(v) ← min (highpt(v), highpt(w));
                if (highpt(w) >= DFN[v])    {AP[v] ← true;}
        }
        Else if (Parent(v) !=w)    {highpt(v) = min (highpt(v), DFN[w]);}
}
DFS_traversal(G){
        dfn ← 0;
        For each vertex v in V     {visited(v) ← false; AP[v] ← false;}
        For each vertex c in V {
                if (vertex(v) ← false)    {DFS(v);}
        }
}
```

- **Analysis of Time Complexity**
  We used transpose of the graph that runs in O(m+n) and finds articulation point that takes O(m+n) time. Therefore, the total time complexity comes out to be O(m+n).

- **Analysis of Space Complexity**
  Since it takes n elements in the graph, the space complexity is O(n).

Problem 2

We will use the merge sort algorithm to tackle the problem. This problem is similar to the counting inversion problem; in addition to that, we need to handle A[i] > 2*A[j].

- **Pseudo-code**

```
Merge (A[], B[], k, mid, i){
        a ← i; b ← mid + 1; c ← 0;
        While (a <= mid - 1 and b <= k){
                If(A[a] > 2*A[b])    {c3 ← c3 + mid - a;   b++;}
                Else    {a++;}
        }
        a ← i; b ← mid + 1;
        While (i <= mid - 1 and b <= k){
                If (A[i] >= A[b])    {B[c++] ← A[b++];}
                Else    {B[c++] ← A[a++];}
        }
        While (a <= mid - 1)    {B[c++] ← A[i++];}
        While (b <= k)    {B[c++] ← A[b++];}
        return c3;
}

Merge_sort (A[], i, k){
        If (i = k) return 0;
        Else{
                mid ← (i + k) / 2;
                c1 ← Merge_sort (A[], i, mid);
                c2 ← Merge_sort (A[], mid + 1, k);
                B[k] ← 0; \\ array of size k
                c3 ← Merge (A[], B[], k, mid, i);
                A ← B;
                return c1+c2+c3;
        }
}
```

- **Analysis of Time Complexity**

  c1 runs in T(n/2)
  c2 runs in T(n/2)
  c3 (Merge) runs in O(n)

  If n = 1, T(n) = c for some constant c
  If n > 1,

  ```
      T(n) = c*n + 2*T(n/2)
           = O(n*log(n))          *Using Master Theorem
  ```

Problem 3

As all the x coordinates are the same, the minimum distance could be found by just checking the y coordinates. Euclidean distance could calculate this, and we will use $(x - x_s)^2 + (y_i - y_s)^2 + (x - x_t)^2 + (y_i - y_t)^2$. As all the given points are not sorted, and sorting is prohibited, we will use Balanced BST to solve it.
For all this, we need two data structures, one Node to make nodes in the balancing BST and another Cell to store the x and y coordinates of each point. We also need some helper functions to calculate the distance between two points, balance the BST, insert in a BST to keep it balanced, and finally, the primary recursive function to find the minimum cost.

- **Pseudo-code**

```
struct Node (val, l, r, size) {}
struct Cell (x, y) {}

Distance ($x_1$, $x_2$, $y_1$, $y_2$) {Euclidean distance between ($x_1$, $y_1$) and ($x_2$, $y_2$)}

Balance (root node) {
    A[size of root] ← 0;
    preorder traversal to fill A;
    Create the tree using divide and conquer;
    return root;
}

InsertNode (root, value) {insert nodes in a way to keep it balanced}

Solve(node, s, t, cost) {
    d ← min(Distance(x, node.value, s.x, s.y),
```

```
                    Distance(x, node.left.value, s.x, s.y,),
                    Distance(x, node.right.value, s.x, s.y));
        If (cost < d)    {return;}
        cost ← d;
        If (d = Distance(x, node.left.value, s.x, s.y))     {Solve(node.left, s, t, cost);}
        If (d = Distance(x, node.right.value, s.x, s.y)) {Solve(node.right, s, t, cost);}
}
```

- **Analysis of Time Complexity**
  Insertion in the tree will take place in log(n) time; since there are n queries, the whole tree will form in O(n*log(n)) time.
  It is given, and we have made the function for computing distance between a pair of points takes O(1) time, and computing the cost of a 2-path takes time O(1) time.
  For each query operation, we are searching in a balanced BST for each query operation, which can be done in O(log(n)) time.

- **Analysis of Space Complexity**
  We are maintaining a BST of n elements; therefore, a space of O(n) is required.