1. What is the running time of the below code?

Algorithm 1 Test function

```
Test(a, start, end) :
 1: n = end - start ;
 2: if n <= 1
 3:     return a[n] ;
 4: else
 5:     newEnd = start + n/6;
 6:     newEnd2 = newEnd + 2·n/6;
 7:     sol1 = Test(a, start, newEnd);
 8:     sol2 = Test(a, newEnd + 1, newEnd2);
 9:     sol3 = Test(a, newEnd2 + 1, end);
10:     combineSol = combine(a, start, newEnd, end) //T(n) = O(n)
11:     return min([sol1,sol2, sol3, combineSol]);
12: end
```

2. You are given a matrix called buildings that has locations of all the buildings at a university in a two-dimensional coordinate. We would to construct paved paths that connected the buildings to each other. Implement an algorithm to calculate the minimum budget required to finish the constructions.

1. How would you find the minimum amount to construct the paths?

   This problem is an instantiation of the Minimum Spanning Tree problem. Concretely, we want to find the set of edges that connect the buildings with the condition that sum of the edge weights of the formed tree is minimal compared to any other construction. To see why the problem is the MST problem we note that if we were to obtain a solution for a system that had N points and M edges where $M \geq N$ we could remove edges that were in a cycle until we had $N-1$ edges, which would both produce a tree and would be a smaller weighted solution then the previous one. The brute-force solution is to generate all possible spanning trees and then choose the one with minimum weight. Since the MST will be computed on a complete graph, by Cayley's formula there would be N^{N-2} possible spanning trees, making this approach intractable. It is the case then we apply one of the greedy algorithms for finding the minimum spanning tree, namely Prim's or Kruskal's. We note that an optimal implementation of Prim's or Kruskal's on an adjacency list representation of a graph is $O(E \cdot log(V))$. Since the graph that will be generated from the dataset is complete, this solution degenerates to $O(V^2 \cdot log(V))$. We know that we can do a simple implementation of Prim's on a adjacency matrix in $O(V^2)$ which will be the approach used.

2. Write the pseudocode for the best algorithm you came up with.

Algorithm 2 minimum budget cost

generate_building(buildings) :

```
V = buildings.rows()
G[V][V] = {0} // initialize matrix to have all zero's
for i = 0 to V :
    for j = i + 1 to V :
        x_i = buildings[i][0]
        y_i = buildings[i][1]
        x_j = buildings[j][0]
        y_j = buildings[j][1]
        cost = sqrt((x_i − x_j)^2 + (y_i − y_j)^2)
        G[i][j] = cost
        G[j][i] = cost
return G
```

Prim(G) :

```
cost = [inf for i in G]
cost[s] = 0
T = ∅
while |T| ≠ |V| − 1:
    curr = min(cost) and not in T
    T = T ∪ curr
    for(i = 0;i < |V(G)|;i++):
        if G[curr][v]≠ 0 and G[curr][v] < cost[v] and v ∉ T:
            cost[v] = G[curr][v]
return sum(cost)
```

total_minimum_cost(buildings) :

```
G = generate_graph(buildings)
total_cost = prims(G)
return total_cost
```

3. Implement your answer using any programming language you want to.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <deque>
#include <limits>

using namespace std;

vector<vector<double>> generate_graph(const vector<vector<int
    >>& buildings){
    int m = buildings.size();
    vector<vector<double>> graph(m, vector<double>(m, 0.0));
        // initialize mxm graph
```

```
12      for(int i = 0;i < graph.size();i++){
13          for(int j = i + 1;j < graph[0].size();j++){
14              int xi = buildings[i][0];
15              int yi = buildings[i][1];
16
17              int xj = buildings[j][0];
18              int yj = buildings[j][1];
19
20              double dist = sqrt(pow(xi - xj, 2) + pow(yi - yj,
                    2));
21              // matrix is symmetric since undirected
22              graph[i][j] = dist;
23              graph[j][i] = dist;
24          }
25      }
26      return graph;
27  }
28
29  int select_min(const deque<bool>& in_mst, const vector<double
        >& cost){
30      int chosen_vertex = -1;
31      double vertex_cost = std::numeric_limits<int>::max();
32      for(int i = 0;i < cost.size();i++){
33          if(!in_mst[i] && cost[i] < vertex_cost){
34              chosen_vertex = i;
35              vertex_cost = cost[i];
36          }
37      }
38      return chosen_vertex;
39  }
40
41  double prims(const vector<vector<int>>& buildings){
42      auto adjmat = generate_graph(buildings); // O(V**2)
43      int V = adjmat.size();
44      deque<bool> in_tree(V, false);
45      vector<double> cost(V, std::numeric_limits<double>::max())
            ;
46      cost[0] = 0.0;
47      int iterations = 0;
48      while(iterations < V - 1){
49          int curr = select_min(in_tree, cost); // select next
                valid candidate
50          in_tree[curr] = true; // O(V)
51          iterations++;
52          for(int i = 0;i < V;i++){ // O(V)
53              if(adjmat[curr][i] > 0 && !in_tree[i] && adjmat[
                    curr][i] < cost[i]){
54                  cost[i] = adjmat[curr][i];
55              }
56          }
57      }
58      // adding up the edge weights of the mst
59      // O(V)
60      double result = 0;
61      for(double d : cost){
62          result += d;
63      }
```

```
64      return result;
65  }
```

4. What is the time complexity of your answer?

   We analyze the code. Let $|V|$ denote the number of buildings in the dataset. The first line of the prim function explictly generates an adjacency matrix of size $|V|\times|V|$ by computing all pairwise euclidean distances of the buildings. This is $\binom{|V|}{2} = \frac{|V|\cdot(|V|-1)}{2} = O(|V|^2)$. The while loop which iterates $|V|-1$ times, as this will be number of edges on the tree, and does two computations. First it selects a smallest valid candidates from the vertices on the frontier of the current MST. This is $O(|V|)$ as we loop through all vertices. The other computation checks the neighbors of the selected candidate and checks if a better cost has been found, and if so updates it. Since we have to check all $|V|$ vertices, this is also $O(|V|)$. The last portion of the algorithm sums up the weights of the edges found in the mst, this will be $O(|V|)$. In total

$$\underbrace{O(|V|^2)}_{\text{generate graph}} + \underbrace{O(|V|^2)}_{\text{while loop and selecting min}} + \underbrace{O(|V|^2)}_{\text{while loop and checking neighbors}} + \underbrace{O(|V|)}_{\text{computing total cost}} = O(|V|^2)$$

3. You are given an adjaccency matrix that has 0s and 1s in it. Implement an algorithm to find the exact number of connected components on the map.

1. How can you find the total number of connected components?

   A connected component is a maximal subgraph such that there exist a walk between any two vertices in that subgraph. We know that if we choose an arbitrary vertex and generate a walk to all vertices that it could walk to via DFS or BFS then that constitutes a connected component. If a vertex has not been visited then, since the connected component is maximal, this vertex must not be in the connected component and we compute the walks again for that connected component. This gives us a way to compute the connected components. we start a counter initialized to 0. We iterate through the vertex set and if we have not seen a particular vertex that means we have a new connected component, compute DFS on that vertex to find and mark all nodes that are visited. Incrementing the counter by one and again iterate through the vertex set computing DFS and then incrementing the counter if we have an unseen vertex.

2. Write the pseudocode for the best algorithm you came up with.

Algorithm 3 Connected components

connected_components(G) :

```
1: m = G.rows()
2: n = G.cols()
3: cc = 0
4: visited[m][n] = {false} // initialize all vertices to not visited
5: for(i = 0;i < m;i++) :
6:     for(j = 0;j <n;j++) :
7:         if not visited[i][j] and G[i][j]==1 :
8:             DFS(G,i, j, m, n, visited)
9:             cc++
10: return cc
```

DFS(G,i, j, m, n, visited) :

```
1: visited[i][j] = true;
2: possible_neighbors = {[0, 1], [0, -1], [1, 0], [-1, 0], [1, 1], [1, -1], [-1, 1], [-1,
   -1]} // neighbors can be to the left,right and diagonals of a possible vertex
3: for neighbor in possible_neighbors :
4:     new_i = neighbor[0] + i
5:     new_j = neighbor[1] + j
6:         if not out_of_bounds(new_i, new_j, m, n) and not visi-
   ted[new_i][new_j] and G[new_i][new_j] == 1 :
7:         DFS(G, new_i, new_j, m, n, visited)
```

3. Implement your answer using any programming language you want to.

```
1  #include <iostream>
2  #include <vector>
3  #include <deque>
4
5  using namespace std;
6
7  bool in_bound(int x, int y, int m, int n){
8      return x >= 0 && x < m && y >= 0 && y < n;
9  }
10
11 void dfs(const vector<vector<int>>& adj, int x, int y, deque<
       deque<bool>>& marked){
12     int m = adj.size();
13     int n = adj[0].size();
14     marked[x][y] = true;
15     vector<vector<int>> neighbors = {{-1, 0}, {1, 0}, {0, -1},
           {0, 1}, {-1, -1}, {-1, 1},{1, -1}, {1, 1}};
16     for(const auto& neighbor : neighbors){
17         int new_x = x + neighbor[0];
18         int new_y = y + neighbor[1];
19         if(in_bound(new_x, new_y, m, n) && !marked[new_x][
               new_y] && adj[new_x][new_y] == 1){
20             dfs(adj, new_x, new_y, marked);
21         }
22     }
```

```
}

int connected_components(const vector<vector<int>>& adj){
    int cc = 0;
    int m = adj.size();
    int n = adj[0].size();
    deque<deque<bool>> marked(m, deque<bool>(n, false));
    for(int i = 0;i < m;i++){
        for(int j = 0;j < n;j++){
            if(!marked[i][j] && adj[i][j] == 1){
                dfs(adj, i, j, marked);
                cc++;
            }
        }
    }
    return cc;
}
```

4. What is the time complexity of your answer?

   We analyze the pseudocode, suppose that we are given a matrix with no 1's, that is a matrix with 0 connected components. The algorithm will iterate through all entries and never compute DFS. Since the matrix will contain $m \cdot n$ elements this will be $\Theta(m \cdot n)$. Suppose the matrix is all 1's. Then DFS will be called on the leftmost element and continously be called throughout the matrix since the graph is fully connected. Any call will have $\Theta(1)$ work but there will be $\Theta(m \cdot n)$ calls resulting in $\Theta(m \cdot n)$. Once the connected component has been computed, the algorithm will still iterate through the rest of the vertex set, since there exists $m \cdot n$ total elements, this is $\Theta(m \cdot n)$. The total running time for this case is $\Theta(m \cdot n) + \Theta(m \cdot n) = \Theta(m \cdot n)$. Any other case would be a mixture of these two cases and so the algorithm in general is $\Theta(m \cdot n)$.