ALGORITHM-ASSIGNMENT-2

NAME-ANUBHAV ANAND
ENROLLMENT NUMBER-2020CSB102
DEPARTMENT-COMPUTER SCIENCE AND TECHNOLOGY(CST)
SUBJECT-ASSIGNMENT-2 OF ALGORITHM LAB

Q1>. 1. You need to implement the polygon triangulation problem.

(a) Prepare a dataset for the convex polygon with increasing number of arbitrary vertices.

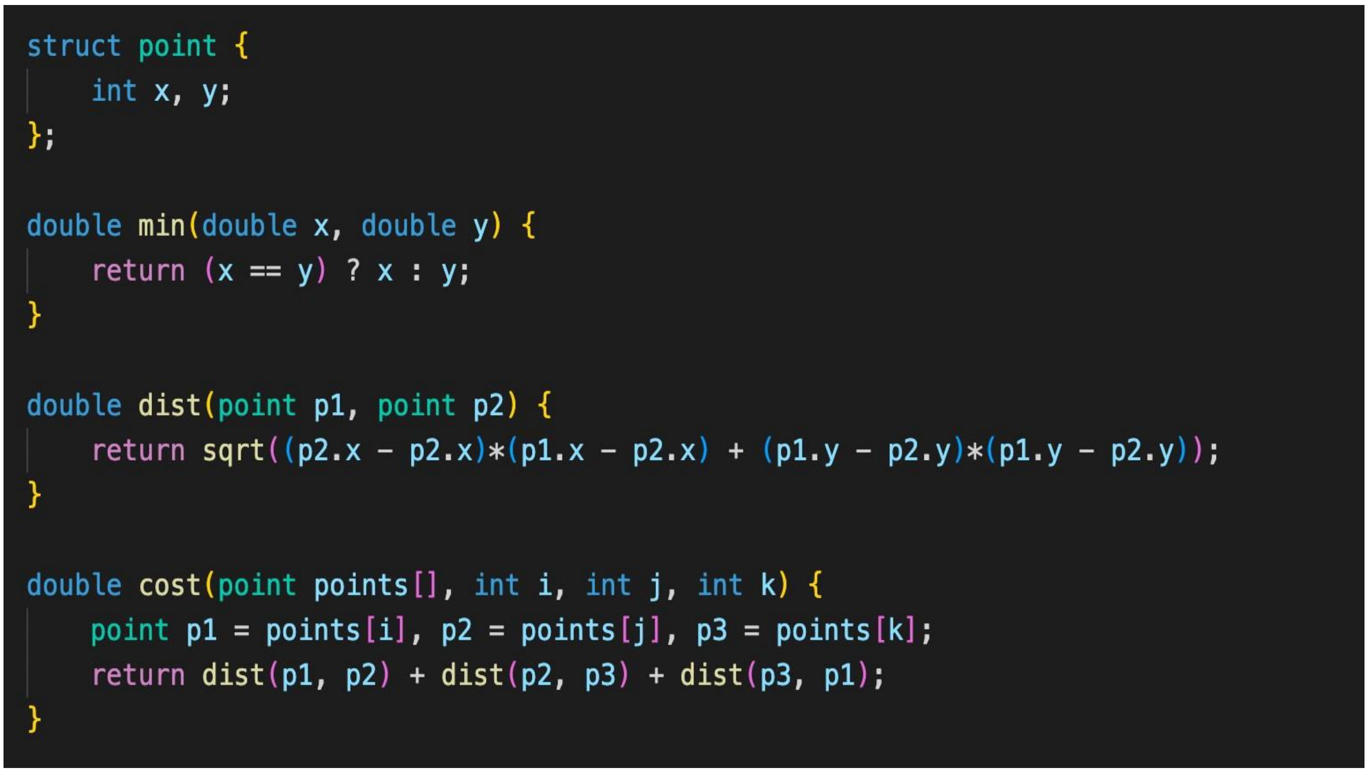
(b) Consider a brute force method where you do an exhaustive search for finding the optimal result.

(c) Next apply dynamic programming, in line with the matrix chain multiplication problem.

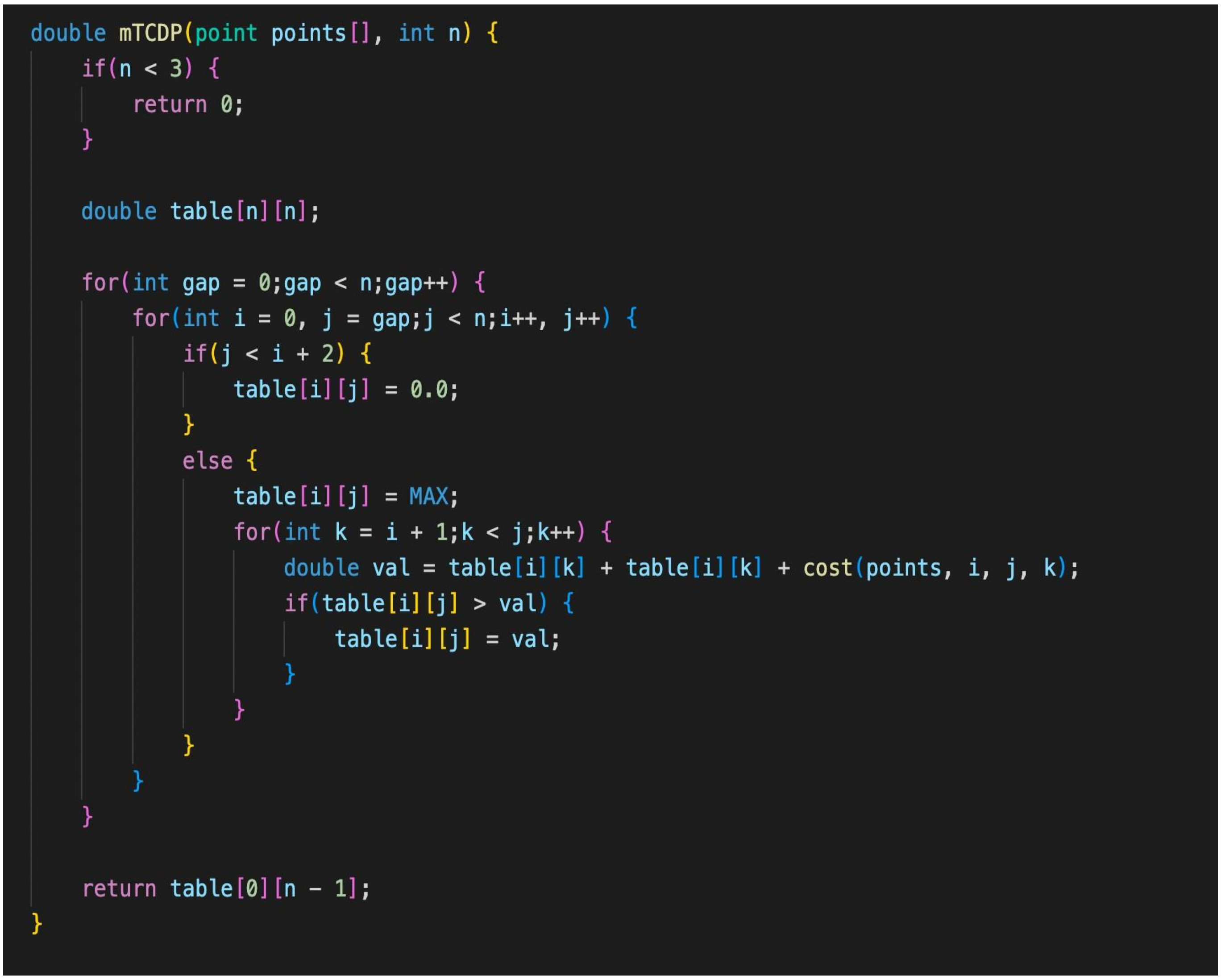
(d) Then implement a greedy strategy to solve the same problem by choosing non-intersecting diagonals in sorted order.

(e) Finally compare the results and suggest whether the DP and Greedy approach results show mismatch.

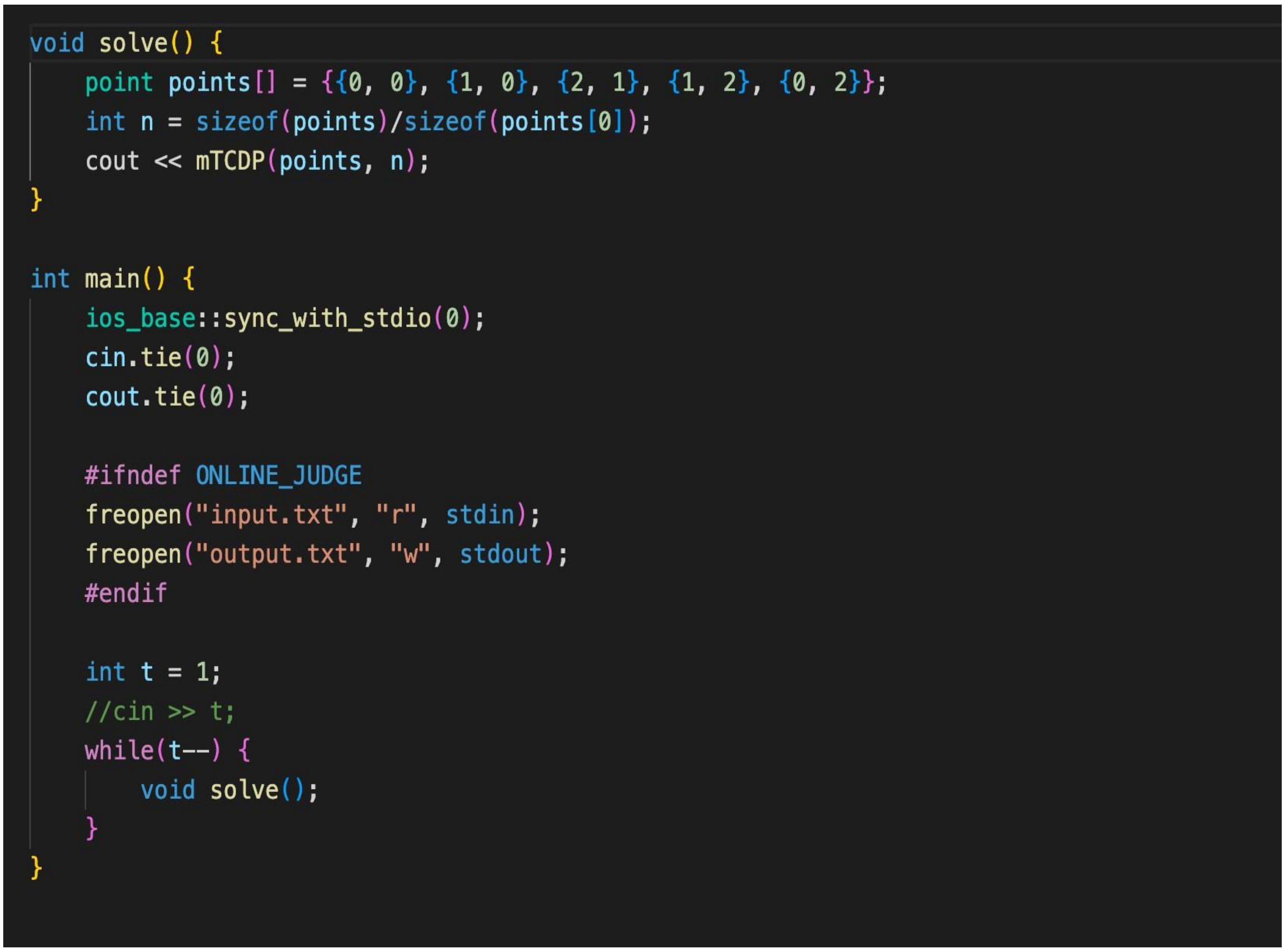
Ans-In c++



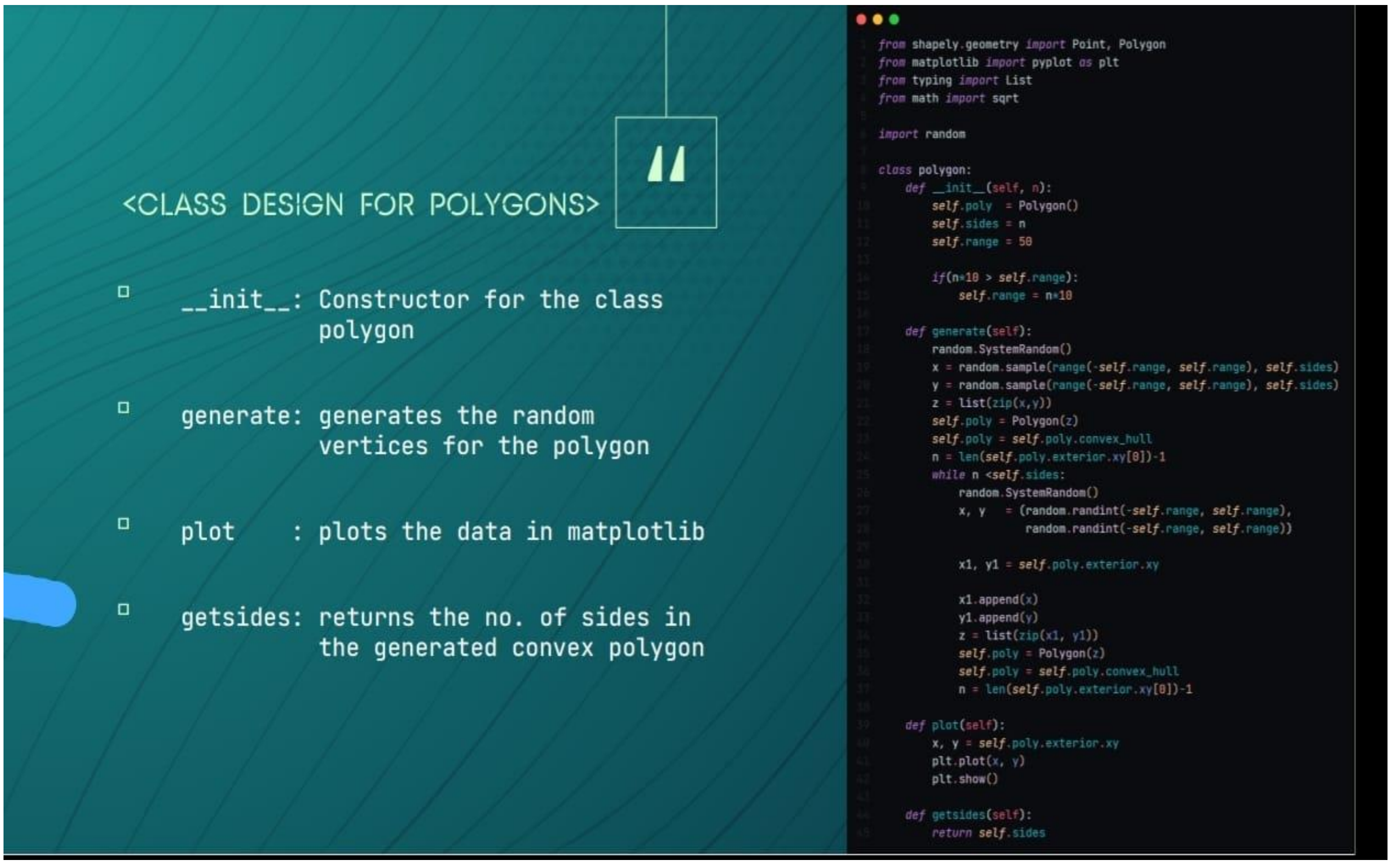
```cpp
struct point {
    int x, y;
};

double min(double x, double y) {
    return (x == y) ? x : y;
}

double dist(point p1, point p2) {
    return sqrt((p2.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y));
}

double cost(point points[], int i, int j, int k) {
    point p1 = points[i], p2 = points[j], p3 = points[k];
    return dist(p1, p2) + dist(p2, p3) + dist(p3, p1);
}
```



```cpp
double mTCDP(point points[], int n) {
    if(n < 3) {
        return 0;
    }

    double table[n][n];

    for(int gap = 0;gap < n;gap++) {
        for(int i = 0, j = gap;j < n;i++, j++) {
            if(j < i + 2) {
                table[i][j] = 0.0;
            }
            else {
                table[i][j] = MAX;
                for(int k = i + 1;k < j;k++) {
                    double val = table[i][k] + table[i][k] + cost(points, i, j, k);
                    if(table[i][j] > val) {
                        table[i][j] = val;
                    }
                }
            }
        }
    }

    return table[0][n - 1];
}
```


```cpp
void solve() {
    point points[] = {{0, 0}, {1, 0}, {2, 1}, {1, 2}, {0, 2}};
    int n = sizeof(points)/sizeof(points[0]);
    cout << mTCDP(points, n);
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);

    #ifndef ONLINE_JUDGE
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
    #endif

    int t = 1;
    //cin >> t;
    while(t--) {
        void solve();
    }
}
```

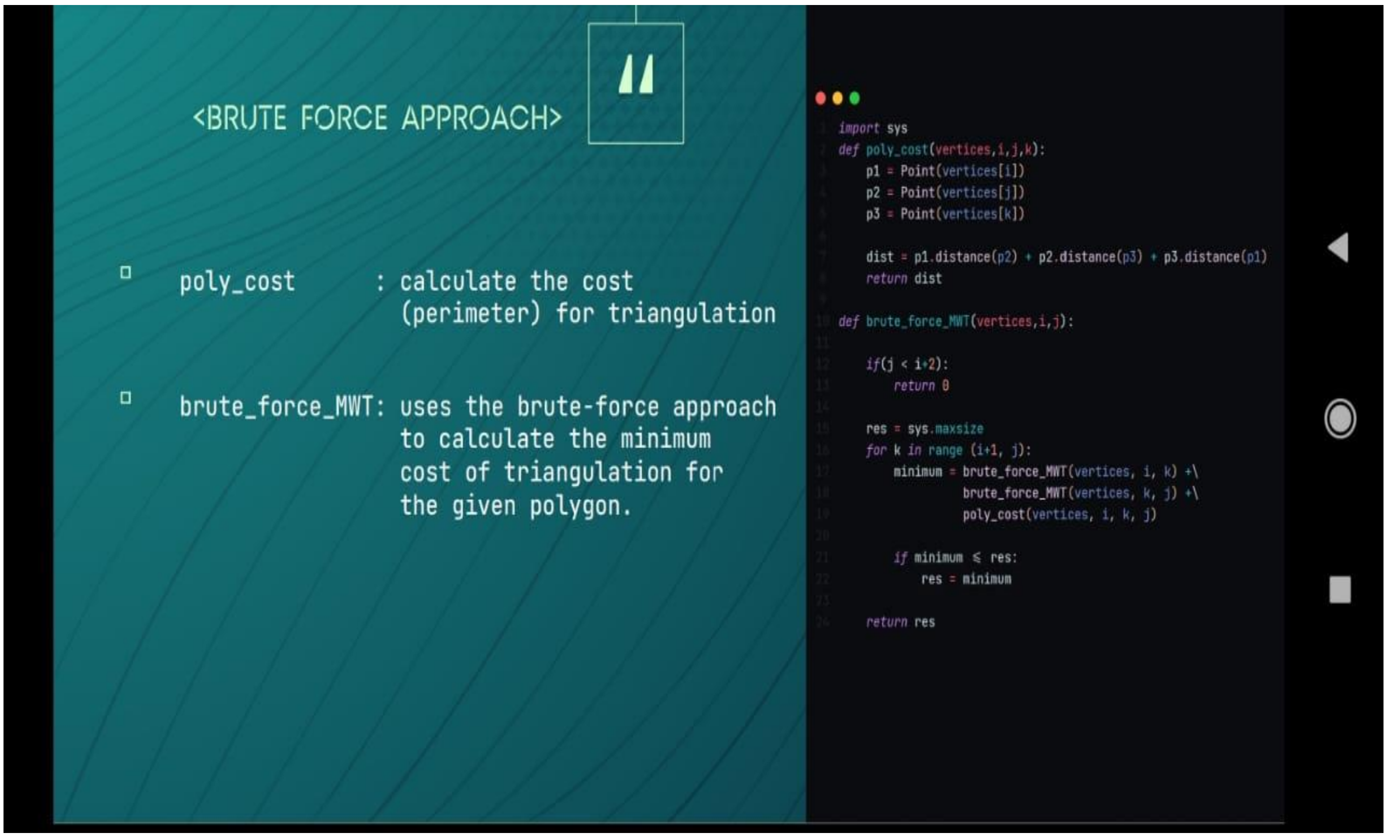
In python-



<CLASS DESIGN FOR POLYGONS>

- `__init__`: Constructor for the class polygon
- `generate`: generates the random vertices for the polygon
- `plot` : plots the data in matplotlib
- `getsides`: returns the no. of sides in the generated convex polygon

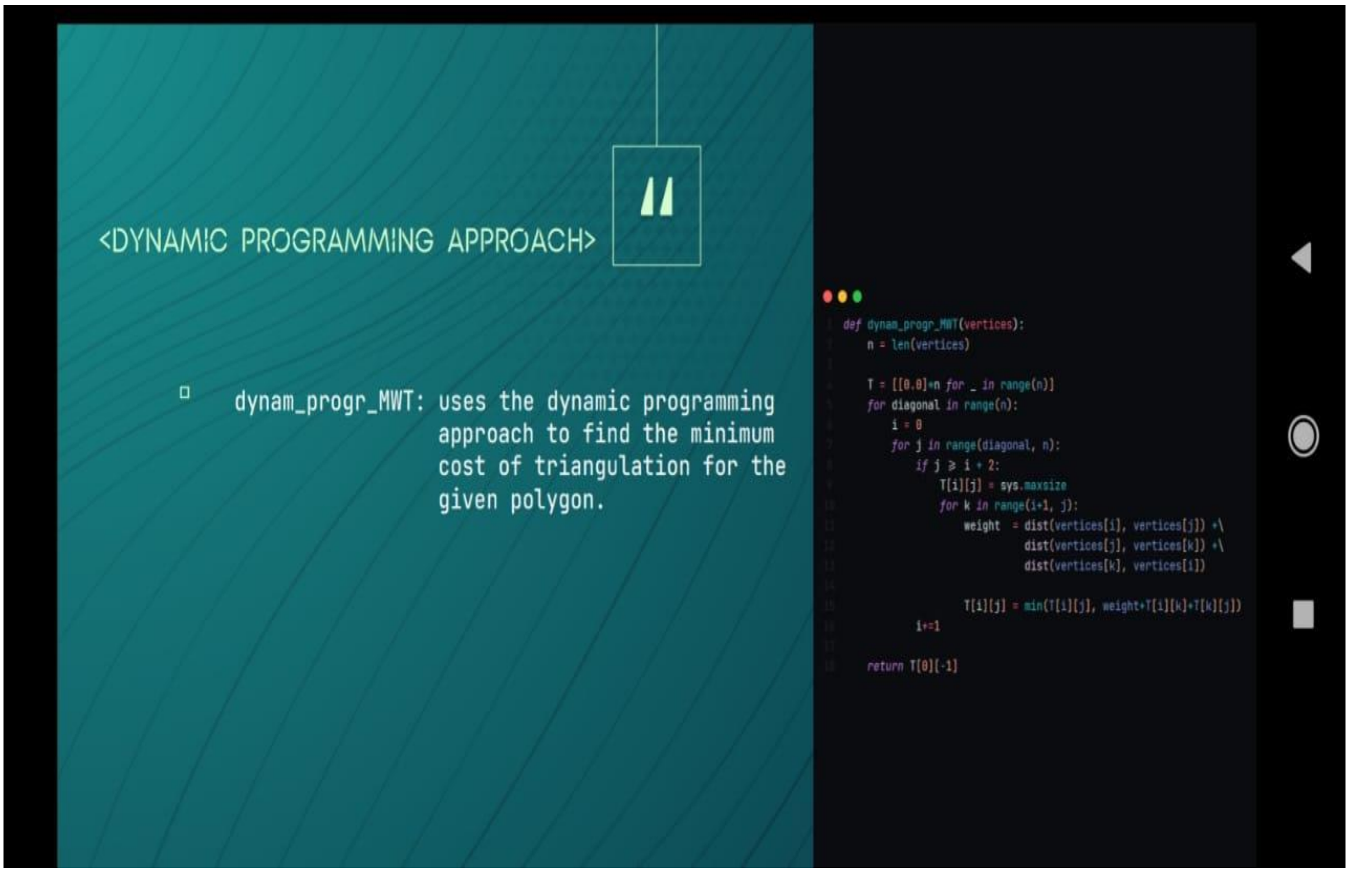
```python
from shapely.geometry import Point, Polygon
from matplotlib import pyplot as plt
from typing import List
from math import sqrt

import random

class polygon:
    def __init__(self, n):
        self.poly  = Polygon()
        self.sides = n
        self.range = 50

        if(n*10 > self.range):
            self.range = n*10

    def generate(self):
        random.SystemRandom()
        x = random.sample(range(-self.range, self.range), self.sides)
        y = random.sample(range(-self.range, self.range), self.sides)
        z = list(zip(x,y))
        self.poly = Polygon(z)
        self.poly = self.poly.convex_hull
        n = len(self.poly.exterior.xy[0])-1
        while n <self.sides:
            random.SystemRandom()
            x, y   = (random.randint(-self.range, self.range),
                          random.randint(-self.range, self.range))

            x1, y1 = self.poly.exterior.xy

            x1.append(x)
            y1.append(y)
            z = list(zip(x1, y1))
            self.poly = Polygon(z)
            self.poly = self.poly.convex_hull
            n = len(self.poly.exterior.xy[0])-1

    def plot(self):
        x, y = self.poly.exterior.xy
        plt.plot(x, y)
        plt.show()

    def getsides(self):
        return self.sides
```



<BRUTE FORCE APPROACH>

- `poly_cost` : calculate the cost (perimeter) for triangulation
- `brute_force_MWT`: uses the brute-force approach to calculate the minimum cost of triangulation for the given polygon.

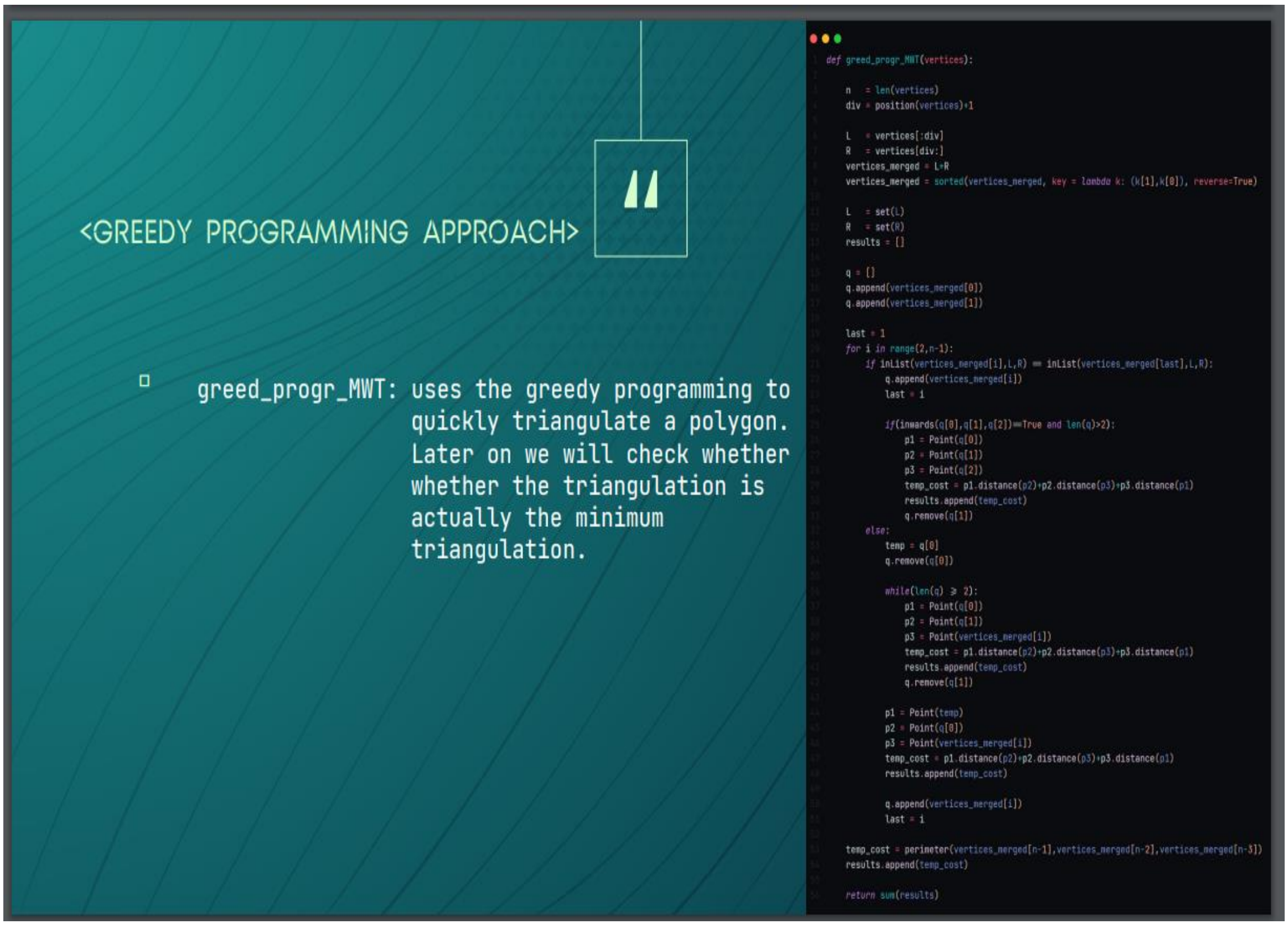
```python
import sys
def poly_cost(vertices,i,j,k):
    p1 = Point(vertices[i])
    p2 = Point(vertices[j])
    p3 = Point(vertices[k])

    dist = p1.distance(p2) + p2.distance(p3) + p3.distance(p1)
    return dist

def brute_force_MWT(vertices,i,j):

    if(j < i+2):
        return 0

    res = sys.maxsize
    for k in range (i+1, j):
        minimum = brute_force_MWT(vertices, i, k) +\
                  brute_force_MWT(vertices, k, j) +\
                  poly_cost(vertices, i, k, j)

        if minimum ≤ res:
            res = minimum

    return res
```



<DYNAMIC PROGRAMMING APPROACH>

- dynam_progr_MWT: uses the dynamic programming approach to find the minimum cost of triangulation for the given polygon.

```python
def dynam_progr_MWT(vertices):
    n = len(vertices)

    T = [[0.0]*n for _ in range(n)]
    for diagonal in range(n):
        i = 0
        for j in range(diagonal, n):
            if j ≥ i + 2:
                T[i][j] = sys.maxsize
                for k in range(i+1, j):
                    weight  = dist(vertices[i], vertices[j]) +\
                              dist(vertices[j], vertices[k]) +\
                              dist(vertices[k], vertices[i])

                    T[i][j] = min(T[i][j], weight+T[i][k]+T[k][j])
            i+=1

    return T[0][-1]
```

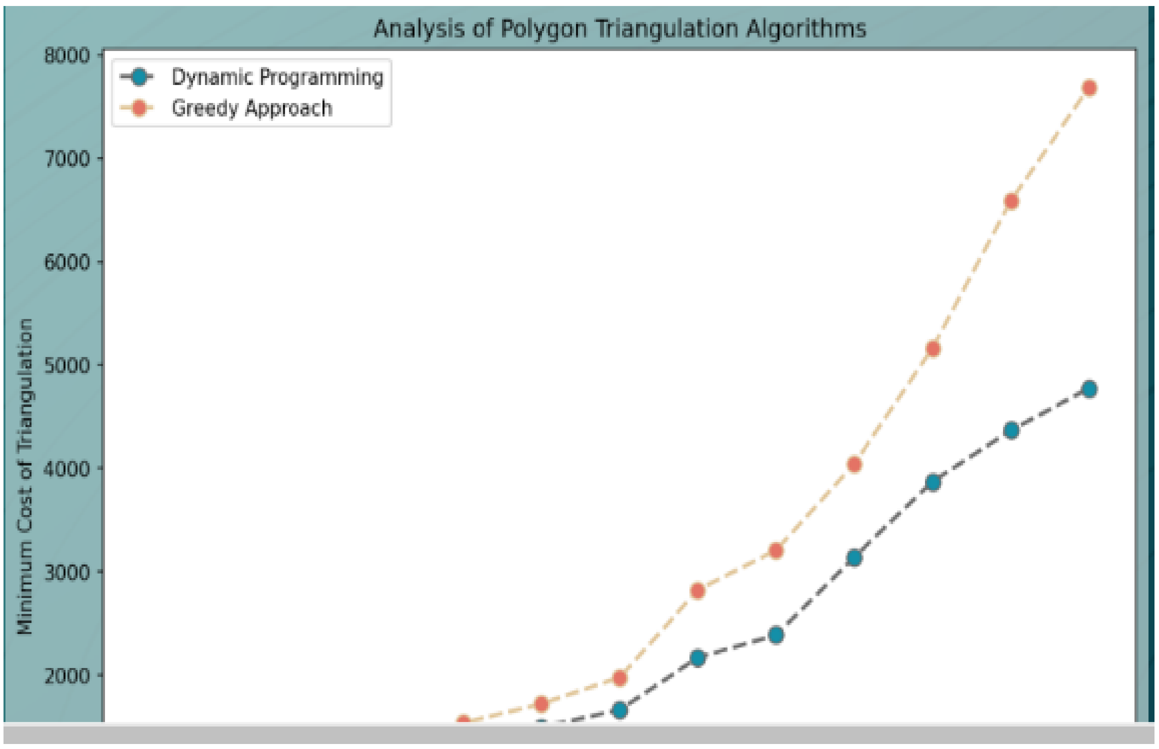


<GREEDY PROGRAMMING APPROACH>

- greed_progr_MWT: uses the greedy programming to quickly triangulate a polygon. Later on we will check whether whether the triangulation is actually the minimum triangulation.

```python
def greed_progr_MWT(vertices):

    n   = len(vertices)
    div = position(vertices)+1

    L   = vertices[:div]
    R   = vertices[div:]
    vertices_merged = L+R
    vertices_merged = sorted(vertices_merged, key = lambda k: (k[1],k[0]), reverse=True)

    L   = set(L)
    R   = set(R)
    results = []

    q = []
    q.append(vertices_merged[0])
    q.append(vertices_merged[1])

    last = 1
    for i in range(2,n-1):
        if inList(vertices_merged[i],L,R) == inList(vertices_merged[last],L,R):
            q.append(vertices_merged[i])
            last = i

            if(inwards(q[0],q[1],q[2])==True and len(q)>2):
                p1 = Point(q[0])
                p2 = Point(q[1])
                p3 = Point(q[2])
                temp_cost = p1.distance(p2)+p2.distance(p3)+p3.distance(p1)
                results.append(temp_cost)
                q.remove(q[1])
        else:
            temp = q[0]
            q.remove(q[0])

            while(len(q) ≥ 2):
                p1 = Point(q[0])
                p2 = Point(q[1])
                p3 = Point(vertices_merged[i])
                temp_cost = p1.distance(p2)+p2.distance(p3)+p3.distance(p1)
                results.append(temp_cost)
                q.remove(q[1])

            p1 = Point(temp)
            p2 = Point(q[0])
            p3 = Point(vertices_merged[i])
            temp_cost = p1.distance(p2)+p2.distance(p3)+p3.distance(p1)
            results.append(temp_cost)

            q.append(vertices_merged[i])
            last = i

    temp_cost = perimeter(vertices_merged[n-1],vertices_merged[n-2],vertices_merged[n-3])
    results.append(temp_cost)

    return sum(results)
```

Plot-



Analysis –

The dynamic programming approach takes O(n3) while the implementations of Seidel's Algorithm in the Greedy Approach takes the complexity of O(n*logn) which in this case was O(n2logn).

However it is noticeable that the Greedy approach does not always give the minimum triangulation of the polygon, especially in those with larger number of sides. But it is considerably faster so it can be used as an alternative for the dynamic programming approach if accuracy is not a very important factor

Q2. We need to scale up the things - from toy problems to real life problems. For this, you need to study at least following two datasets, provided by two reputed Universities. I am talking about:
(a) SNAP - Stanford Network Analysis Project - datasets collected by Stanford University.
(b) KONECT - Koblenz Network Collection - datasets compiled by Koblenz University, Deutschland (Germany).

First study and feel the vastness of these networks from the dynamic and disjoint set operations point of view. Then try to implement graph algorithms like Connected Components and Minimum Spanning Tree using these datasets.

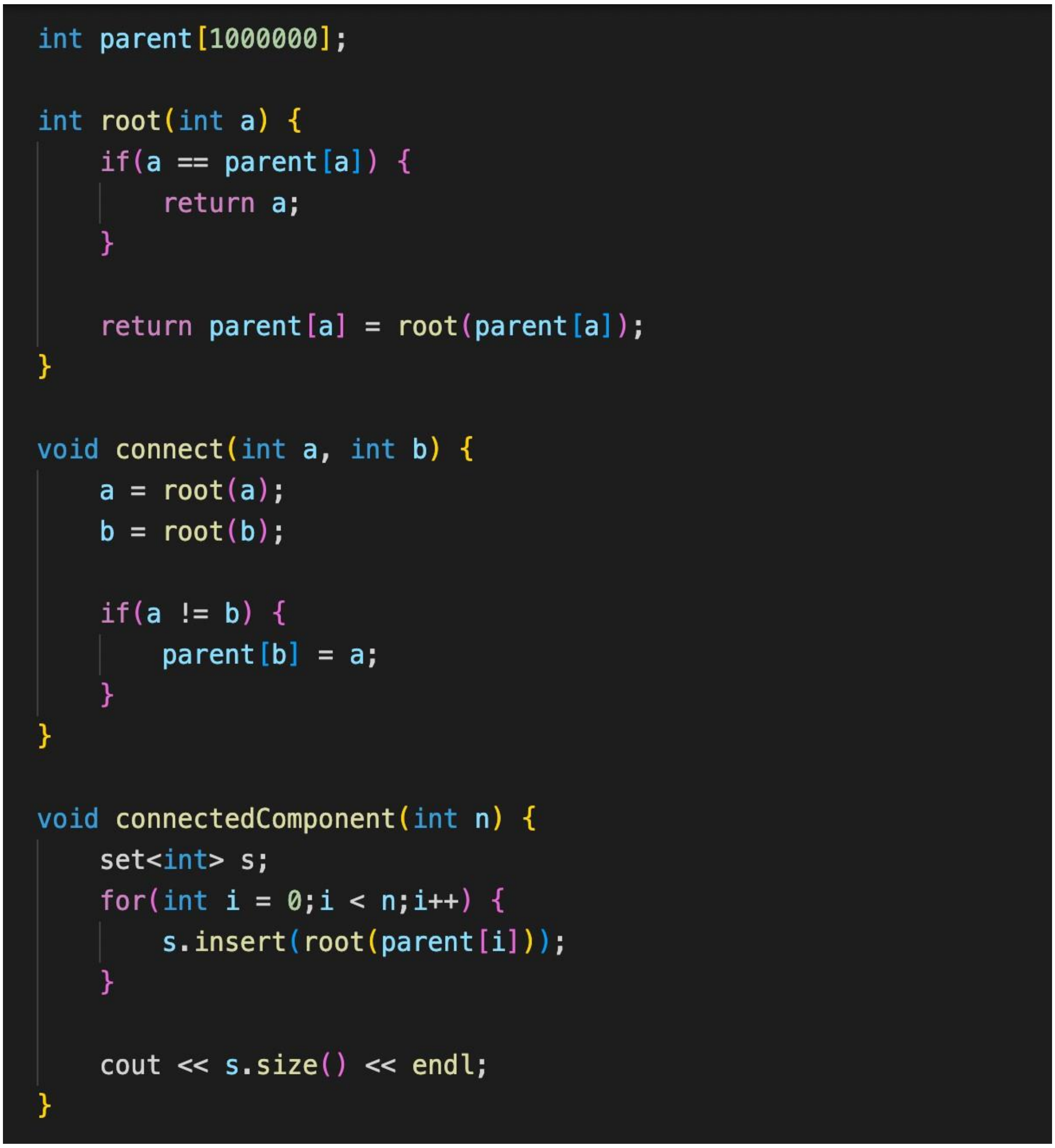
Ans-EXPLORING DATABASES Exploring the SNAP and KONECT databases and try running different algorithms like MST, Disjoint Sets etc

▫ Stanford Network Analysis Platform (SNAP)-is a general purpose network analysis and graph mining library. It is written in C++ and easily scales to massive networks with hundreds of millions of nodes, and billions of edges. It efficiently manipulates large graphs, calculates structural properties, generates regular and random graphs, and supports attributes on nodes and edges. ▫ It is also supported on Python under Snap.py

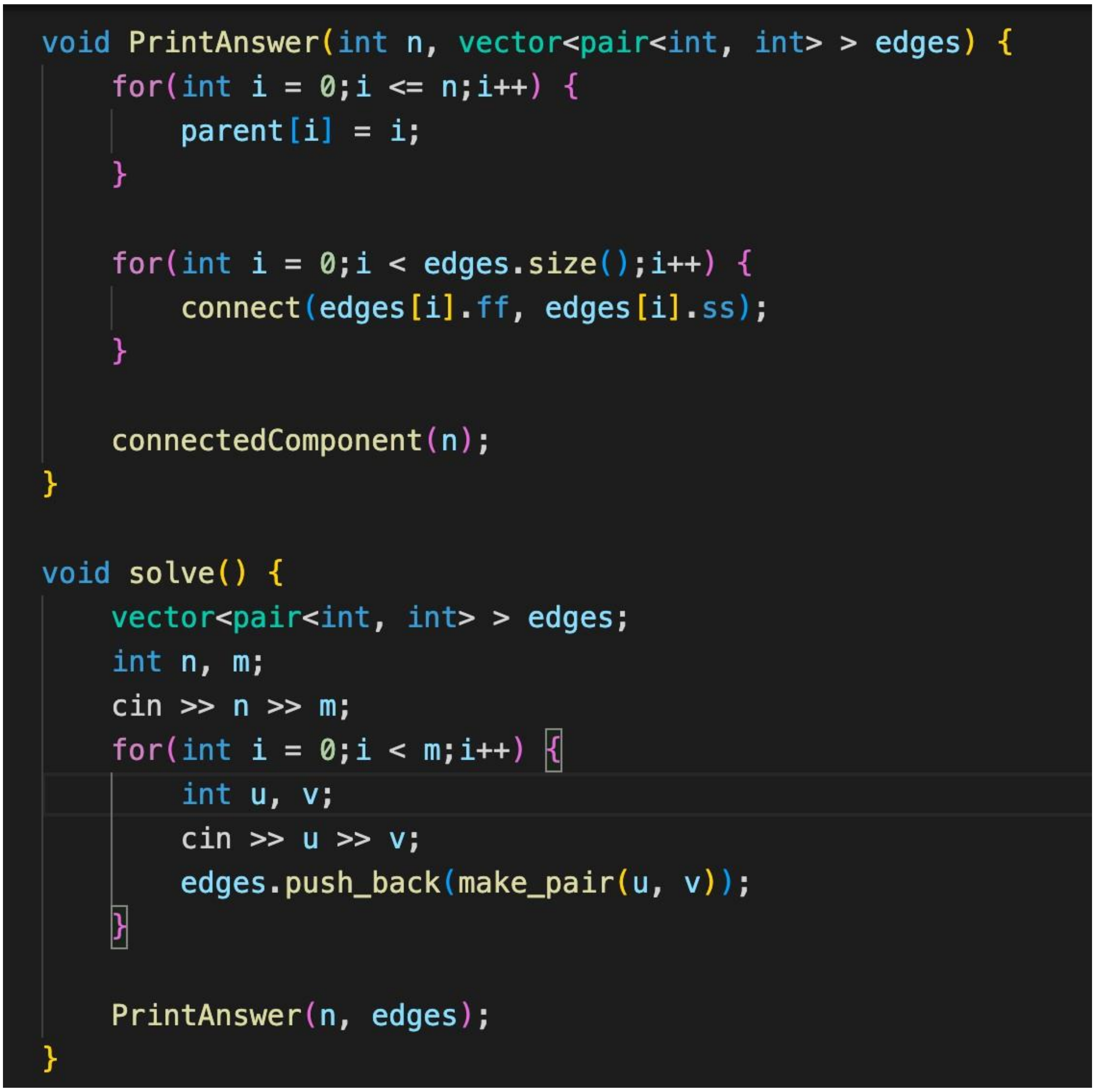
▫Koblenz Network Collection, as a project has roots at the University of Koblenz-Landau in Germany. All source code is made available as Free Software, and includes a network analysis toolbox for GNU Octave, a network extraction library, as well as code to generate webpages, including statistics and plots. KONECT is run by research group around Jérôme Kunegis at the University of Namur, in the Namur Center for Complex Networks. ▫ The KONECT project has 1,326 network datasets in 24 categories

▫ Given n vertices and m predefined edges in a graph , one has to find a subset of m edges so that the n vertices are divided into disjoint sets. ▫ All vertices will be first initialized as disjoint the edges will be used to connect the sets as long as it does not make a cycle. After all the edges are connected keeping the previous condition in mind, we get the final answer. ▫ This algorithm can be implemented using two types of data structures ▫ Linked List ▫ Tree Based

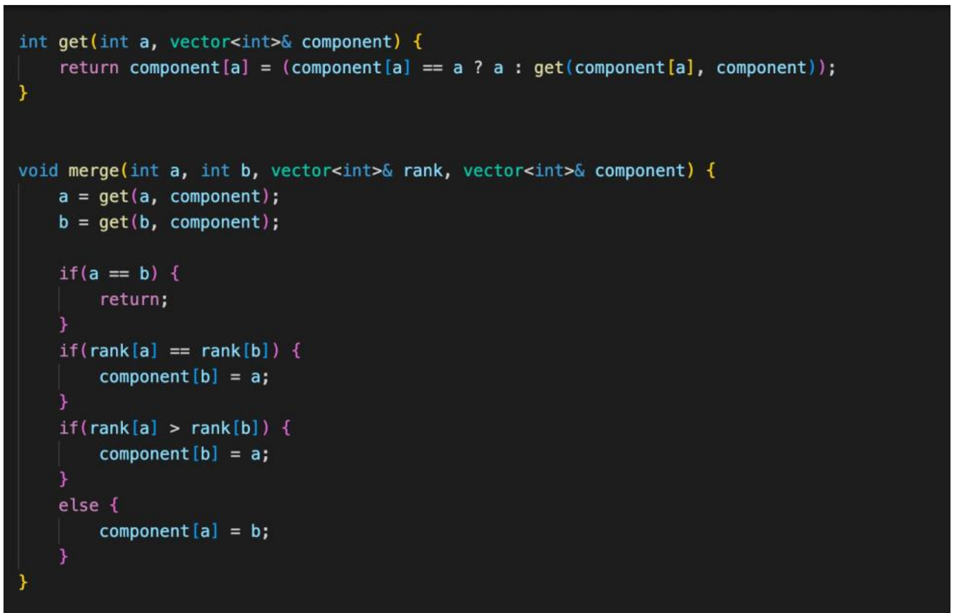
▫ Two linked lists are needed for this purpose, one for dynamically managing number of connected components of the graph (REPRESENTATIVE ELEMENT), and the other to maintain the list of vertices inside that connected component. ▫ The connected component list is doubly linked while the vertices list is singly linked.
▫ We only use vertex as nodes, which contain the rank of the node and a reference to the parent node(mostly the main root node) . ▫ The rank of the node basically signifies how big the subtree with the node as root is.



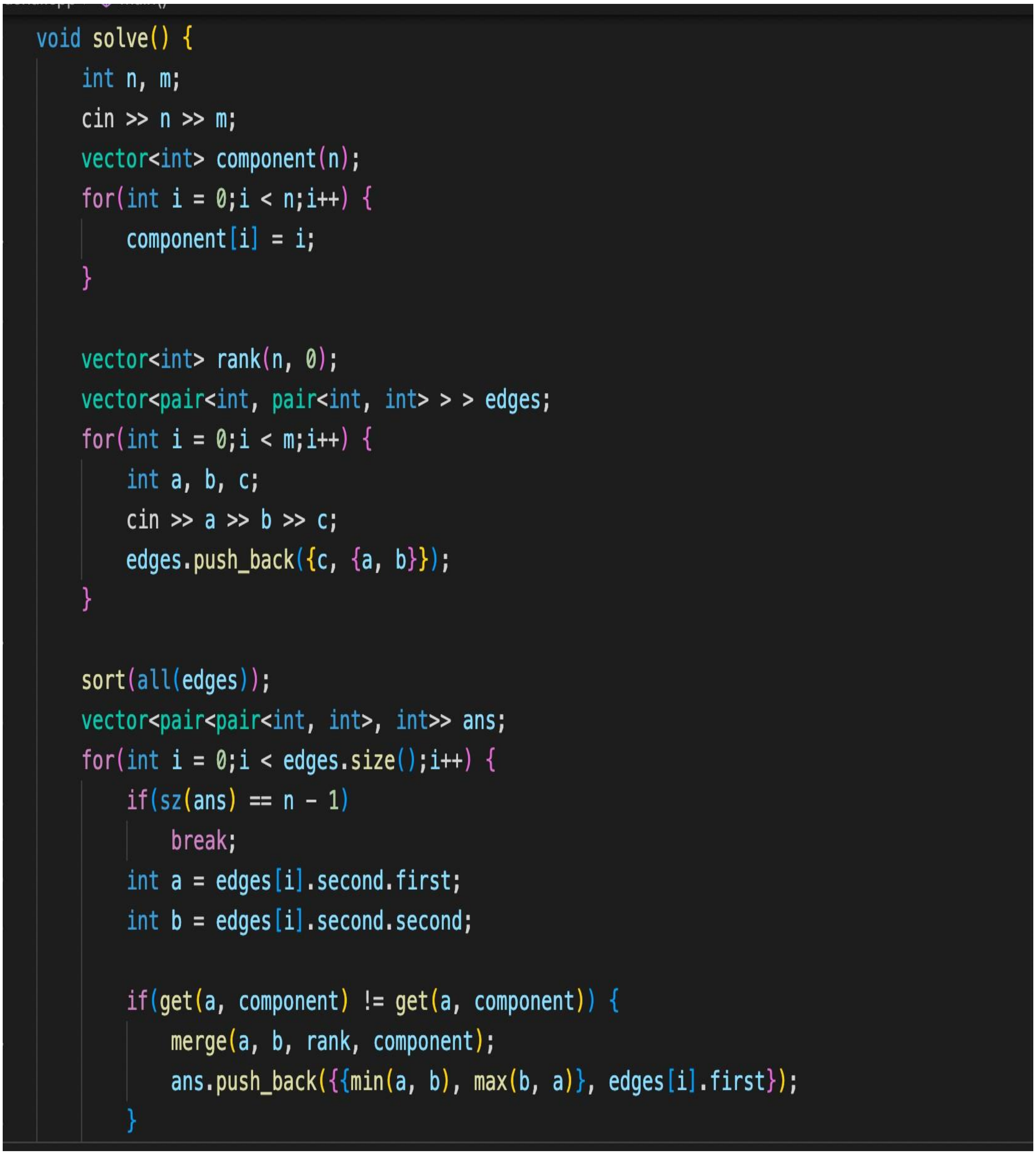
```cpp
int parent[1000000];

int root(int a) {
    if(a == parent[a]) {
        return a;
    }

    return parent[a] = root(parent[a]);
}

void connect(int a, int b) {
    a = root(a);
    b = root(b);

    if(a != b) {
        parent[b] = a;
    }
}

void connectedComponent(int n) {
    set<int> s;
    for(int i = 0;i < n;i++) {
        s.insert(root(parent[i]));
    }

    cout << s.size() << endl;
}
```



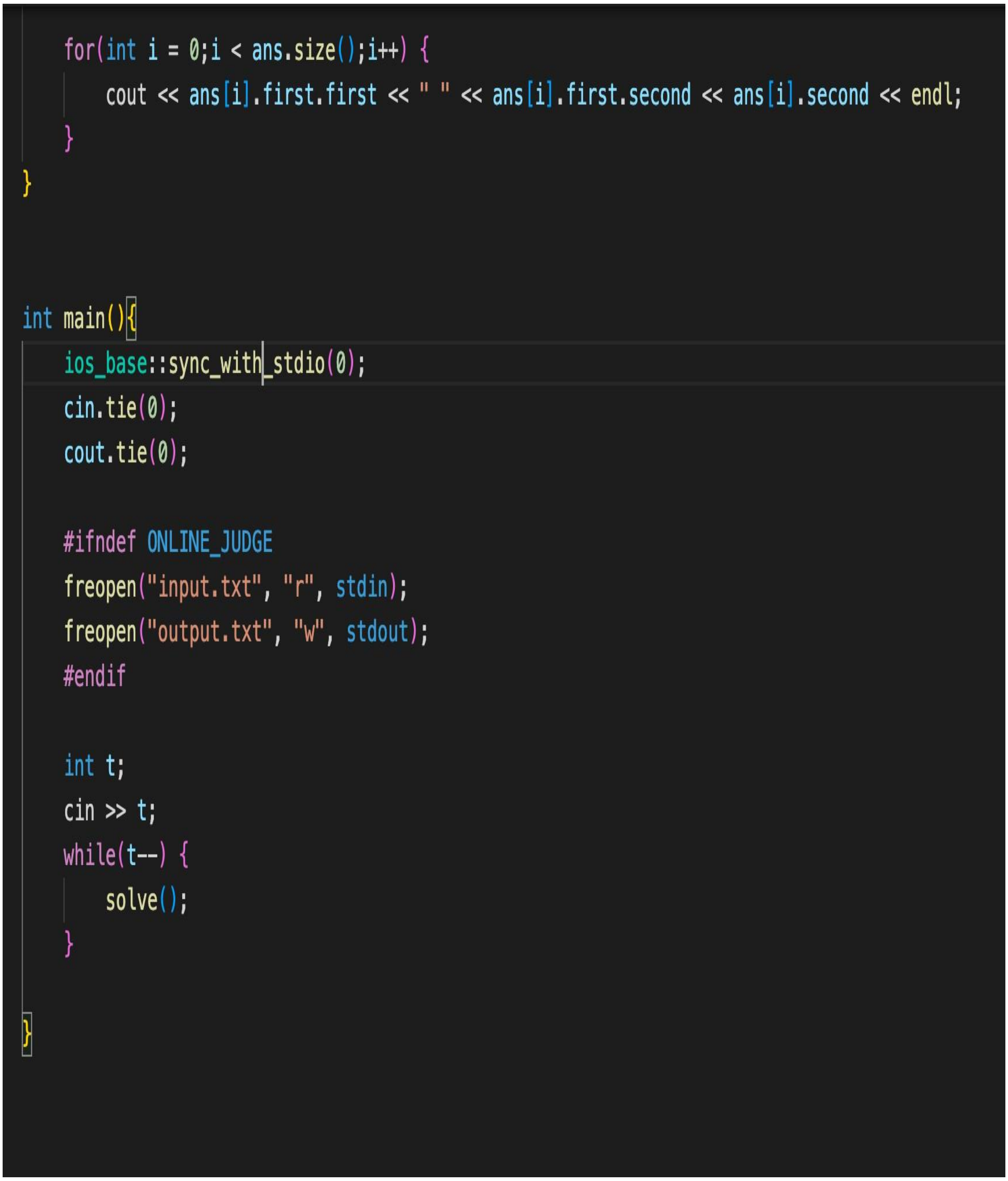
```cpp
void PrintAnswer(int n, vector<pair<int, int> > edges) {
    for(int i = 0;i <= n;i++) {
        parent[i] = i;
    }

    for(int i = 0;i < edges.size();i++) {
        connect(edges[i].ff, edges[i].ss);
    }

    connectedComponent(n);
}

void solve() {
    vector<pair<int, int> > edges;
    int n, m;
    cin >> n >> m;
    for(int i = 0;i < m;i++) {
        int u, v;
        cin >> u >> v;
        edges.push_back(make_pair(u, v));
    }

    PrintAnswer(n, edges);
}
```



```cpp
int get(int a, vector<int>& component) {
    return component[a] = (component[a] == a ? a : get(component[a], component));
}


void merge(int a, int b, vector<int>& rank, vector<int>& component) {
    a = get(a, component);
    b = get(b, component);

    if(a == b) {
        return;
    }
    if(rank[a] == rank[b]) {
        component[b] = a;
    }
    if(rank[a] > rank[b]) {
        component[b] = a;
    }
    else {
        component[a] = b;
    }
}
```



```cpp
void solve() {
    int n, m;
    cin >> n >> m;
    vector<int> component(n);
    for(int i = 0;i < n;i++) {
        component[i] = i;
    }

    vector<int> rank(n, 0);
    vector<pair<int, pair<int, int> > > edges;
    for(int i = 0;i < m;i++) {
        int a, b, c;
        cin >> a >> b >> c;
        edges.push_back({c, {a, b}});
    }

    sort(all(edges));
    vector<pair<pair<int, int>, int>> ans;
    for(int i = 0;i < edges.size();i++) {
        if(sz(ans) == n - 1)
            break;
        int a = edges[i].second.first;
        int b = edges[i].second.second;

        if(get(a, component) != get(a, component)) {
            merge(a, b, rank, component);
            ans.push_back({{min(a, b), max(b, a)}, edges[i].first});
        }
```



```cpp
    for(int i = 0;i < ans.size();i++) {
        cout << ans[i].first.first << " " << ans[i].first.second << ans[i].second << endl;
    }
}


int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);

    #ifndef ONLINE_JUDGE
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
    #endif

    int t;
    cin >> t;
    while(t--) {
        solve();
    }

}
```

Thoughts-

These databases provide not only the facility for generating random directed and non-directed graphs, they also provide the means for order statistics on various operations that can be performed on said graphs. ▫ According to the recorded observations, tree based approach is much faster than linked list based approach. Also the time taken mainly depends on the number of edges, while the space taken depends on the number of vertices. ▫ However the full analysis of the datasets cannot be possibly done in a Laptop environment