**Team ID – T013**
**TA – Mohamed Magdy**

# **Project:** Image Quantization

Team Members

|  |  |  |
|---|---|---|
| 20191700732 | <---- | هانى محمد سيد احمد |
| 20191700740 | <---- | هيثم محمود السيد محمود |
| 20191700742 | <---- | وسيم عبده فتحى عبده |

## Project Requirements:

### 1. Construct the graph by and calculating distances between them:

### Code:

**finding distinct colors:**

```
public static List<int> DistinctColors(RGBPixel[,] colors) // θ(N*N)
  {
     byte[,,] visited = new byte[256, 256, 256]; //θ(1)
     List<int> DistinctColours; DistinctColours = new List<int>();  //θ(1)

    for (int i = 0; i < colors.GetLength(0); i++)   //θ(N)  N -> length
     {
        for (int j = 0; j < colors.GetLength(1); j++) //θ(N)   N -> wigth
         {
           if (visited[colors[i, j].red, colors[i, j].green, colors[i, j].blue] == 0)
            {
              visited[colors[i, j].red, colors[i, j].green, colors[i, j].blue] = 1;
              num = (colors[i, j].red << 16) + (colors[i, j].green << 8) + colors[i,
j].blue; //θ(1)
                     DistinctColours.Add(num);  //θ(1)
             }
         }
     }
          return DistinctColours;
  }
```

### calculating distances between them:

```
public static double Distance_between_two_colors(int x, int y)//θ(1)
        {
            return Math.Sqrt(Math.Pow((byte)(x >> 16) -(byte)(y >> 16), 2) +
Math.Pow((byte)(x) - (byte)(y), 2) + Math.Pow((byte)(x >> 8) - (byte)(y >> 8),
2));//θ(1)
        }
```

### Explanation:

A. Find number of distinct colors:
We loop through each color in the image matrix and check if we didn't visit the color yet then we will add that color to the list then return the distinct colors.

B. Distance between two colors:
This function takes two colors and returns Euclidean distance between them.

### Analysis Of Code:

**A. Find number of distinct colors: O(N*N)**
**B. Distance between two colors: Θ(1)**

## 2. Find the minimum spanning tree from the graph;

## Code:

```
public List<Edge> graphOfMST(List<int>distinct_colors)
   {
       List<Edge> adjaceny_list = new List<Edge>();

       primMst(distinct_colors);                         // O(V log(V))
       for (int i = 0; i < num_of_vertices; i++)         //θ(V)
       {
           if (root[i] >= 0)                             //θ(1)
           {
               Edge temp = new Edge(distinct_colors[root[i]], distinct_colors[i],
min_weights[i]);                                         //θ(1)
               adjaceny_list.Add(temp);                  //θ(1)
           }
     }

        List<Edge> sorted = adjaceny_list.OrderBy(x => x.weight).ToList();
 //O(E log (E))
            return sorted;
        }

        public void primMst(List<int> distinct)             // O(V log(V))
        {
            // Initializing needed Variables
            double distinctColor_B = System.Environment.TickCount;


            for (int i = 0; i < num_of_vertices; i++)   //θ(V)
            {
                // Constructing the Minimum Heap
                Node temp = new Node();                 //θ(1)
                temp.vertex = i;                        //θ(1)
                if (i!=0)                               //θ(1)
                    temp.weight = double.MaxValue;      //θ(1)
                else
                {
                    temp.weight = 0;                     //θ(1)
                }
                minheap.insert(temp);                    //θ(1)
                root[i] = int.MaxValue;                  //θ(1)
                min_weights[i] = double.MaxValue;        //θ(1)
                is_in_heap[i] = true;                    //θ(1)
                key[i] = int.MaxValue;                   //θ(1)
            }

            double distinctColor_a = System.Environment.TickCount;      //θ(1)
            double distinctColor_r = distinctColor_a - distinctColor_B; //θ(1)
            distinctColor_r /= 1000; //θ(1)
            var s = distinctColor_r;  //θ(1)
            root[0] = -1;                                //θ(1)
            min_weights[0] = 0;                          //θ(1)
            while (!minheap.isEmpty())                 //O(Log(V)) --> θ(V*Log(V))
            {
                Node minimum = minheap.extractMin();              //θ(1)
                int vertex_of_minimum_node = minimum.vertex;      //θ(1)
                is_in_heap[vertex_of_minimum_node] = false;       //θ(1)

                for (int i = 0; i < num_of_vertices; i++)         //O(V)
                {
                    if(is_in_heap[i])                             //θ(1)
                    {
                        double temp_weight = 
ImageOperations.Distance_between_two_colors(distinct[vertex_of_minimum_node], 
distinct[i]); //θ(1)

                        if (temp_weight < key[i])                    //θ(1)
                        {
                            // we are updating the value of the
                            int at_index = minheap.indices[i];          //θ(1)
                            Node temp_node = minheap.node[at_index];    //θ(1)
                            temp_node.weight = temp_weight;             //θ(1)
                            minheap.bubbleUp(at_index);                 //θ(1)

                            root[i] = vertex_of_minimum_node;           //θ(1)
                            min_weights[i] = temp_weight;               //θ(1)
                            key[i] = temp_weight;                       //θ(1)
                        }
                    }
                }
            }
        }

public double min_sum_MST()                           //θ(V)
    {
        double total_sum = 0;                         //θ(1)

        for (int i = 0; i < num_of_vertices; i++)  //θ(V)
            total_sum += min_weights[i];           //θ(1)

        return Math.Round(total_sum, 2);           //θ(1)
    }
```

**<u>Explanation:</u>**

First function(graphOfMST):
Here we declare an adjacency list to get each node its adjacent node then we call primMST function with an argument of dictinct colors its internal functionality will be discussed later and then we loop over each vertex and assign each node and its minimum weight to adjaceny_list and after finishing the loop we sort the adjaceny_list (Edges) based on weight and put it in sorted list then return the sorted list.

Second function(primMST):
In the first for loop, we construct the minimum heap.
After the first loop, the first one in the root will always be -1 because it’s the parent and it doesn’t have any weights.
The last nested loop, as long as the minimum heap is not empty, we extract the minimum node and put its vertex into a variable named vertex_of_minimum_node and then that minimum one will be removed by setting it as false then in the inner for loop, we will loop over each node except the minimum one and then calculate the Euclidean distance between two colors which is the weight.

Third function(minimumSumOfMST):
Here after we get the minimum weights from the whole we loop over all minimum weights and add them to a variable named total_sum and after loop it returns the total minimum sum.

**Analysis Of Code:**

**O(V*Log(V))**

## 3. Extract the K clusters from the minimum spanning tree with maximal spacing between them:

**Code:**

```
public static List<HashSet<int>> getClusters(List<Edge> MST, int num_cluster,
List<int> distinctColor) // O(D)
    {
        Dictionary<int, HashSet<int>> neighbours = new
        Dictionary<int,HashSet<int>>();                              //θ(1)
         HashSet<int> reached = new HashSet<int>();                  //θ(1)
         List<HashSet<int>>clusters = new List<HashSet<int>>();      //θ(1)

         for (int i = 0; i < num_cluster - 1; i++)                   //θ(K)
         {
             MST.RemoveAt(MST.Count - 1);                            //O(1)
         }

         for (int j = 0; j < distinctColor.Count; j++)               //θ(D)
         {
             neighbours.Add(distinctColor[j], new HashSet<int>());   //θ(1)

         }

         for (int j = 0; j < MST.Count; j++)                         //θ(E)
         {
             neighbours[MST[j].from].Add(MST[j].to);                 //θ(1)
             neighbours[MST[j].to].Add(MST[j].from);                 //θ(1)
         }

         foreach (var var in neighbours)                             //θ(D)
         {
             if (!reached.Contains(var.Key))                         //θ(1)
             {
                 HashSet<int> cluster = new HashSet<int>();          //θ(1)
                 dfs(var.Key, ref reached, ref neighbours, ref cluster);
                 clusters.Add(cluster);                              //θ(1)
             }
         }
         return clusters;                                            //θ(1)
    }
private static void dfs(int cur, ref HashSet<int> reached, ref Dictionary<int,
HashSet<int>> neighbours, ref HashSet<int> cluster)
    {
         cluster.Add(cur);                                                 //θ(1)
         reached.Add(cur);                                                 //θ(1)

         foreach (var neighbour in neighbours[cur])                        //θ(E)
         {
             if (!reached.Contains(neighbour))                             //θ(1)
                 dfs(neighbour, ref reached, ref neighbours, ref cluster); //θ(1)
         }
    }
```

**Explanation:**
After getting sorted list of edges from (graphOfMST) function.
So, in the first loop: we remove the last element in edges list (maximum weight) until the number of cluster (K) = 0.

Second loop: we add (Distinct colors) as keys in Dictionary(neighbors).
Third loop: we add list of neighbors for every Distinct_color as value in dictionary(neighbors).
After that, we apply Depth First Search algorithm(dfs) to adjacency list value in dictionary(neighbors) to get list of Hashset clusters then return it.

**Analysis Of Code:**
**O(D)**

## 4. Find the representative color of each cluster:

**Code:**

```
public static RGBPixel[,,] Palette(List<HashSet<int>> clusters)    //θ(D)
    {
        int sum_red, sum_green, sum_blue;                             //θ(1)
        RGBPixel[,,] palette = new RGBPixel[256, 256, 256];           //θ(1)
        List < RGBPixel> color_of_cluster = new List<RGBPixel>();     //θ(1)

        for (int x = 0; x < clusters.Count; x++)                      //θ(D)
        {
            RGBPixel average_color;
            sum_red = 0; sum_green = 0; sum_blue = 0;

            foreach (var cur in clusters[x])
            {
                sum_red += (byte)(cur >> 16);                        //θ(1)
                sum_green += (byte)(cur >> 8);                       //θ(1)
                sum_blue += (byte)(cur);                             //θ(1)
            }
            average_color.red = (byte)(sum_red / clusters[x].Count);     //θ(1)
            average_color.green = (byte)(sum_green / clusters[x].Count); //θ(1)
            average_color.blue = (byte)(sum_blue / clusters[x].Count);   //θ(1)

            color_of_cluster.Add(average_color);
        }

        for (int i = 0; i < clusters.Count; i++)                     //θ(D)
        {
            foreach (var cluster in clusters[i])
            {
                palette[(byte)(cluster >> 16), (byte)(cluster), (byte)(cluster
>> 8)] = color_of_cluster[i];                                        //θ(1)
            }
        }

        return palette;                                              //θ(1)
    }
```

**Explanation:**
Here we have clusters which is in hash set which means all distinct items
So, in the first loop, we calculate average color of each cluster.

Second loop, we update color of all distinct colors in each cluster to average color of its.

**Analysis Of Code:**

**O(D)**

**5. Quantize the image by replacing the colors of each cluster by its representative color:**

**Code:**

```
public static RGBPixel[,] quantizeImage(RGBPixel[,] imageMatrix,RGBPixel[,,]
new_colors)                                                        //θ(N*N)
    {
        int height = ImageOperations.GetHeight(imageMatrix);          //θ(1)
        int width = ImageOperations.GetWidth(imageMatrix);            //θ(1)
        RGBPixel[,] quantized_image = new RGBPixel[height, width];    //θ(1)

        for (int i = 0; i < height; i++)   //θ(N*N)     //θ(N)          N->height
        {
            for (int j = 0; j < width; j++)             //θ(N)          N->width
            {
                quantized_image[i, j] = new_colors[imageMatrix[i, j].red,
imageMatrix[i, j].blue, imageMatrix[i, j].green]; //θ(1)
            }
        }
        return quantized_image; //θ(1)
    }
```

**Explanation:**

It loops over each pixel and replaces colors in the original matrix with the new colors (Palette from ExtractColors function) after clustering.

**Analysis Of Code:**

**O(N*N)**