Image quantization project
Algorithms course

Classes:

class edge:

```
public class edge: IComparable<edge>
```

A class which holds each edge weight, start point and end point.

```
public double distance = 0;
public int[] points = new int[2];
```

and implements function CompareTo to enable sorting according to the distances

```
public int CompareTo(edge other) => other.distance.CompareTo(this.distance) * -1;
```

class ImageOperations:

A class to hold all functions to perform distinct colors extracting, MST constructing, clustering, pallet generating, DisplayImage and Quantizing the image

Functions description and complexity:

OpenImage:

```
public static RGBPixel[,] OpenImage(string ImagePath, ref int[,,] mapper, ref List<RGBPixel> colorSet)
```

parameters:

ImagePath => which takes the path of the image to open.

Mapper => As reference 3D Boolean array to map every color to a position according to red, green, blue values to create a simple set which adds the color if mapper[red, green ,blue] = false which means the color is not present in the list "colorSet" and changes the value to true to avoid adding the color again to the list.

colorSet => As reference List which initially holds no colors and after the execution of the function it holds all distinct colors from the image to pass it to the other functions later.

Complexity: while reading the image it checks for every color "O (N^2)" in this snippet

```
for (y = 0; y < Height; y++) //O(N)
    for (x = 0; x < Width; x++) //O(N)
```

if it is present in the mapper or not "O (1)" in this snippet

```
if (colorsBoolArray[Buffer[y, x].red, Buffer[y, x].green, Buffer[y, x].blue] == false)// O(1)
{
colorsBoolArray[Buffer[y, x].red, Buffer[y, x].green, Buffer[y, x].blue] = true; //O(1)
colorSet.Add(Buffer[y, x]); //O(1)
}
```

Which means over all "O (N^2)" complexity for extracting the distinct colors.

CalculateMST:

```
public static List<edge> CalculateMST(int numberOfDistinctColors, List<RGBPixel> colorSet)
```

parameters:

numberOfDistinctColors: an integer value which holds the number of distinct colors.

colorSet: the list which holds the distinct colors itself.

Attributes:

```
bool[] partnersOfTheTree = new bool[numberOfDistinctColors];
int least = 0, localLeast = 0;
double min, dist;
edge tmp;
List<edge> MST = new List<edge>();

```

partnersOfTheTree: a Boolean array which maps every color from colorSet depending on the index of each color to chow if the color is already in the tree or not

least , localLeast: an integer value which holds the index of the starting point of distance calculations which is considered the point which optimal edge found (initially set the first color in index 0)

min: a double value to hold the minimum edge weight.

dist: a double value to hold the current edge weight to compare it to min and add it to the MST(if selected).

tmp: a pointer to hold an edge from the MST to avoid re calculating in every access.

MST: a list to hold the selected edges for creating the minimum spanning tree and to be returned as a result for the function.

Complexity:

The looping the separated to 2 phases
 phase 1:

Looping from 0 to all other vertices and calculating the distances for every edges

```
dist = Math.Sqrt(Math.Pow(colorSet[localLeast].red - colorSet[j].red, 2) +
Math.Pow(colorSet[localLeast].green - colorSet[j].green, 2) +
Math.Pow(colorSet[localLeast].blue - colorSet[j].blue, 2));
```

Then creates the initial MST which holds all edges from color 0 to all other colors ... which makes the first feasible solution and not the optimal solution (No cycle but not the best cost) and has a size of (V-1) then chooses the least distance to point to the next edge to loop from (like Dijkstra)

```
int k = 1;
          // loops from 0 to all other vertcies
          while (k < numberOfDistinctColors)//O(N)
          {
             // calculating the distance
             byte red1 = colorSet[localLeast].red;//O(1)
             byte green1 = colorSet[localLeast].green;//O(1)
             byte blue1 = colorSet[localLeast].blue;//O(1)
             byte red2 = colorSet[k].red;//O(1)
             byte green2 = colorSet[k].green;//O(1)
```

```
            byte blue2 = colorSet[k].blue;//O(1)
            double red = (red1 - red2) * (red1 - red2);//O(1)
            double green = (green1 - green2) * (green1 - green2);//O(1)
            double blue = (blue1 - blue2) * (blue1 - blue2);//O(1)
            dist = Math.Sqrt(red + green + blue);//O(1)
            // adding the edge to the initial MST
            MST.Add(new edge(dist, 0, k));//O(1)
            // checks if it is the minimmum edge to start from its end
            if (dist < min)//O(1)
            {
                // change the minimum
                min = dist;//O(1)
                // save the index of the next start over
                least = k;//O(1)
            }
            k++;
        }
```

Which is done in O(N) as it loops only for all other vertices.

Phase 2:

Looping from all vertices except of the first one to every other vertex to compare with the initial MST to check if better way(distance) to reach a specific vertex and replace it.

It is done in O(1) as comparing with the pervious best wat is done with indexing the MST as it is kind of sorted and the order is not changing in the entire code.

```
k = 1;
        // looping from every vertix exept of 0 to the others
        while (k < numberOfDistinctColors)//O(N)
        {
            //reset the min for every vertix
            min = 99999;
            //set the startPoint as optiml way found
            partnersOfTheTree[least] = true;
            // reserve the start point index to prevent fomr modifications
            localLeast = least;
            // check the distance to all others from the current start point
            for (int j = 1; j < numberOfDistinctColors; j++)//O(D)
            {
                // exclude the optimal points if found
                if (partnersOfTheTree[j] == false)//O(D)
                {
                    // calculating the distance
                    byte red1 = colorSet[localLeast].red;//O(1)
                    byte green1 = colorSet[localLeast].green;//O(1)
                    byte blue1 = colorSet[localLeast].blue;//O(1)
                    byte red2 = colorSet[j].red;//O(1)
                    byte green2 = colorSet[j].green;//O(1)
                    byte blue2 = colorSet[j].blue;//O(1)
                    double red = (red1 - red2) * (red1 - red2);//O(1)
                    double green = (green1 - green2) * (green1 - green2);//O(1)
                    double blue = (blue1 - blue2) * (blue1 - blue2);//O(1)
                    dist = Math.Sqrt(red + green + blue);//O(1)
                    // hold the edge to reduce access time
                    tmp = MST[j];//O(1)
                    //check if it is a better way to this vertix
                    if (dist < tmp.distance)//O(1)
                    {
                        //replace the distance
                        MST[j].distance = dist;//O(1)
                        //replace the starting index
                        MST[j].points[0] = localLeast;//O(1)
                    }
                    //check if it is the best edge in the tree to set the next start point
                    if (tmp.distance < min)//O(1)
                    {
                        //change the min
                        min = tmp.distance;//O(1)
                        // reserve the index
                        least = tmp.points[1];//O(1)
                    }
                }
            }
            k++;
        }
```

Which means over all “O (N^2)” complexity for calculating the distance from every vertex to all other vertices and extracting the MST edges.

clustering:

```
public static List<List<int>> clustering (int numberOfDistinctColors, List<edge> MST, int k)
```

parameters:

numberOfDistinctColors => an integer value which holds the number of distinct colors.

MST => the list that holds the edges which creates the minimum spanning tree.

K => the number of desired clusters for this image

```
List<List<int>> clusters = new List<List<int>>();
int[] indexer = new int[numberOfDistinctColors];
int cluster = 1, unClusterd = numberOfDistinctColors, numberOfClusters = 0 , p1 , p2 , I1, I2;
edge tmp;
```

clusters => list which holds a list of clusters that is extracted from the function and to be returned.

Indexer => an integer array to map every color from colorSet to it is cluster Number

Cluster => defines the number of the current cluster to be added and initially set to 1

unClusterd => the number of not yet clustered vertices initially set to numberOfDistinctColors and decrements

numberOfClusters => the number of clustered vertices initially set to 0 and increments.

p1 , p2 , I1, I2 => pointers to point for every wanted index from indexer or color to avoid re calculating

tmp: a pointer to hold an edge from the MST to avoid re calculating in every access.

Complexity:

first step is to sort the MST List to access the least distance ascendingly

```
MST.Sort(); // O(NlogN)
```

Clustering is done by accessing the MST from the least distance ascendingly and combining every to points for an edge according to some conditions in 2 phases

First phase: has 4 conditions

First condition is if the 2 vertices are clustered to different clusters, then we will combine the 2 clusters

```
 if (I1 != 0 && I2 != 0 && I1 != I2) // O(1)
{
    int keep = I2;
    clusters[keep - 1].ForEach(l => indexer[l] = I1);//O(D)
    clusters[I1 - 1].AddRange(clusters[keep - 1]);//O(D)
    clusters[keep - 1].Clear();//O(1)
    numberOfClusters--;//O(1)
}
```

The second and third conditions are the same to check if only one from those points is clusterd then we add the not clustered one to the other's cluster

```
 else if (I1 != 0)
{
    indexer[p2] = I1;
    clusters[I1 - 1].Add(p2);
    unClusterd--;

}
else if (I2 != 0)
{
    indexer[p1] = I2;
    clusters[I2 - 1].Add(p1);
    unClusterd--;
}
```

The fourth condition is if the 2 points are not clustered yet then we add the 2 points to a new cluster

```
 else
{
   List<int> tempCluster = new List<int>();
   tempCluster.Add(tmp.points[0]);
   tempCluster.Add(tmp.points[1]);
   clusters.Add(tempCluster);
   indexer[p1] = cluster;
   indexer[p2] = cluster;
   cluster++;
   numberOfClusters++;
   unClusterd -= 2;
}
```

The second phase has 3 conditions and is started by the condition:

```
if (numberOfClusters + unClusterd == k)
```

and the 2 main conditions is checking for the remaining edges if any 1 of 2 points is not clustered yet

```
if (I1 == 0)//O(1)
{
    List<int> tempCluster = new List<int>();//O(1)
    tempCluster.Add(p1); //O(1)
    clusters.Add(tempCluster); //O(1)
    indexer[p1] = cluster; //O(1)
    cluster++;//O(1)
    numberOfClusters++;//O(1)
    unClusterd--;//O(1)
}
if (I2 == 0)
{
    List<int> tempCluster = new List<int>();//O(1)
    tempCluster.Add(p2); //O(1)
    clusters.Add(tempCluster); //O(1)
    indexer[p2] = cluster; //O(1)
    cluster++;//O(1)
    numberOfClusters++;//O(1)
    unClusterd--;//O(1)
}
continue;

```

and the third condition is if all vertices are clustered then break the loop

```
if (unClusterd == 0) break;
```

Which means over all “O (D^3)” complexity for clustering the colors

generatePallete:

```
public static List<RGBPixel> generatePallete(List<List<int>> clusters, List<RGBPixel> colorSet, ref int[,,] mapper)
```

parameters:

clusters: the list of clusters that holds all vertices with all clusters

colorSet: the list of distinct colors

mapper: an integer array thar maps between every color and it’s cluster representative color

attributes:

```
List<RGBPixel> palletaa = new List<RGBPixel>();
int clusterCounter = 0, count = clusters.Count;
double red = 0, green = 0, blue = 0;
int numberOfClusterElements
RGBPixel current;
```

palletaa: is the list that holds every cluster representative color

clusterCounter: an integer to hold clusters ID to assign to the mapper

current : a pointer to hold the color to add to pallet later.

Complexity:

we loop on clusters to get the list of specific cluster

```
for (int i = 0; i < count; i++) //O(K)
```

then we loop on those colors in this cluster to calculate avg for every red , green , blue

```
foreach (var j in clusters[i]) //O(D)
{
    red = red + colorSet[j].red; // O(1)
    green = green + colorSet[j].green; // O(1)
    blue = blue + colorSet[j].blue; // O(1)
    mapper[colorSet[j].red, colorSet[j].green, colorSet[j].blue] = clusterCounter; // O(1)
}
```

Then we assign this color to pallet and assign the same id to every color in this cluster to help with mapping every color to its cluster representative color

```
current.red = (byte)Math.Ceiling(red / numberOfClusterElements);
current.green = (byte)Math.Ceiling(green / numberOfClusterElements);
current.blue = (byte)Math.Ceiling(blue / numberOfClusterElements);
palletaa.Add(current);
clusterCounter++;
```

as the number of loops will be bounded by the number of distinct colors then the over all “O (D)” complexity for generating pallet

Quantize:

```
public static RGBPixel[,] Quantize(RGBPixel[,] ImageMatrix, List<RGBPixel> p, int[,,] mapper)
```

ImageMatrix: the image which we want to perform quantization on

P: the color pallet

Mapper: the array that maps every color to its representative color

Complexity:

We loop on the image matrix to get every pixel and we check it on the mapper to get it’s cluster id and check it in the pallet list to change the color to its representative color

```
for (int i = 0; i < Height; i++)// O(N)
 for (int j = 0; j < Width; j++)// O(N)
  {
   RGBPixel quantized = new RGBPixel(); // O(1)
  quantized = p[mapper[ImageMatrix[i, j].red, ImageMatrix[i, j].green, ImageMatrix[i, j].blue]]; //O(1)
   Filtered[i, j].red = quantized.red; // O(1)
   Filtered[i, j].green = quantized.green; // O(1)
   Filtered[i, j].blue = quantized.blue; // O(1)
  }

```

Then we return the quantized image

Which means over all “O (N^2)” complexity for mapping each color to its representative color

The Code of image operations:

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
using System.Drawing.Imaging;
using System.Linq;
using System.Diagnostics;
///Algorithms Project
///Intelligent Scissors
///
namespace ImageQuantization
{
    public class edge : IComparable<edge>
    {
        public double distance = 0;
        public int[] points = new int[2];
        public edge(double distance, int point1, int point2)
        {
            this.distance = distance;
            this.points[0] = point1;
            this.points[1] = point2;
        }
        public int CompareTo(edge other) => other.distance.CompareTo(this.distance) * -1;
    }
    public struct color
    {
        public byte red, green, blue;
    }
    /// <summary>
    /// Holds the pixel color in 3 byte values: red, green and blue
    /// </summary>
    public struct RGBPixel
    {
        public byte red, green, blue;
    }
    /// <summary>
    /// Library of static functions that deal with images
    /// </summary>
    public class ImageOperations
    {
        public static Stopwatch overAllTime = new Stopwatch();
        public static double phase1Time;
        /// <summary>
        /// Open an image and load it into 2D array of colors (size: Height x Width)
        /// </summary>
        /// <param name="ImagePath">Image file path</param>
        /// <returns>2D array of colors</returns>
        public static RGBPixel[,] OpenImage(string ImagePath, ref int[,,] mapper, ref List<RGBPixel> colorSet)
        {
            //List<edge> edges = new List<edge>();
            colorSet = new List<RGBPixel>();
            bool[,,] colorsBoolArray = new bool[256, 265, 265];
            Bitmap original_bm = new Bitmap(ImagePath);
            int Height = original_bm.Height;
            int Width = original_bm.Width;
            RGBPixel[,] Buffer = new RGBPixel[Height, Width];
            Stopwatch DistinctTime = new Stopwatch();
            DistinctTime.Start();
            overAllTime.Start();
            unsafe
            {

                BitmapData bmd = original_bm.LockBits(new Rectangle(0, 0, Width, Height), ImageLockMode.ReadWrite, original_bm.PixelFormat);
                int x, y;
                int nWidth = 0;
                bool Format32 = false;
                bool Format24 = false;
                bool Format8 = false;
                if (original_bm.PixelFormat == PixelFormat.Format24bppRgb)
                {
                    Format24 = true;
                    nWidth = Width * 3;
                }
                else if (original_bm.PixelFormat == PixelFormat.Format32bppArgb || original_bm.PixelFormat == PixelFormat.Format32bppRgb || original_bm.PixelFormat == PixelFormat.Format32bppPArgb)
                {
                    Format32 = true;
                    nWidth = Width * 4;
                }
                else if (original_bm.PixelFormat == PixelFormat.Format8bppIndexed)
                {
                    Format8 = true;
                    nWidth = Width;
                }
                int nOffset = bmd.Stride - nWidth;
                byte* p = (byte*)bmd.Scan0;
                for (y = 0; y < Height; y++)//O(N)
```

```
              {
                  for (x = 0; x < Width; x++)//O(N)
                  {
                      if (Format8)
                      {
                          Buffer[y, x].red = Buffer[y, x].green = Buffer[y, x].blue = p[0];
                          p++;
                      }
                      else
                      {
                          Buffer[y, x].red = p[2];
                          Buffer[y, x].green = p[1];
                          Buffer[y, x].blue = p[0];
                            if (Format24) p += 3;
                            else if (Format32) p += 4;
                        }
                        // check if the color is present in the list
                        if (colorsBoolArray[Buffer[y, x].red, Buffer[y, x].green, Buffer[y, x].blue] == false)//O(1)
                        {
                            // sets the color is present in the list
                            colorsBoolArray[Buffer[y, x].red, Buffer[y, x].green, Buffer[y, x].blue] = true;//O(1)
                            // adds the color to list
                            colorSet.Add(Buffer[y, x]);//O(1)
                        }
                    }
                    p += nOffset;
                }
                original_bm.UnlockBits(bmd);
            }


            //Console.WriteLine(colorSet.Count + " Distinct colors ");
            DistinctTime.Stop();
            overAllTime.Stop();
            MessageBox.Show(colorSet.Count.ToString() + " Number of distinct colors in time " + DistinctTime.ElapsedMilliseconds + " ms");

            return Buffer;
        }
        /// <summary>
        /// Get the height of the image
        /// </summary>
        /// <param name="ImageMatrix">2D array that contains the image</param>
        /// <returns>Image Height</returns>
        public static int GetHeight(RGBPixel[,] ImageMatrix)
        {
            return ImageMatrix.GetLength(0);
        }
        /// <summary>
        /// Get the width of the image
        /// </summary>
        /// <param name="ImageMatrix">2D array that contains the image</param>
        /// <returns>Image Width</returns>
        public static int GetWidth(RGBPixel[,] ImageMatrix)
        {
            return ImageMatrix.GetLength(1);
        }
        /// <summary>
        /// Display the given image on the given PictureBox object
        /// </summary>
        /// <param name="ImageMatrix">2D array that contains the image</param>
        /// <param name="PicBox">PictureBox object to display the image on it</param>
        public static void DisplayImage(RGBPixel[,] ImageMatrix, PictureBox PicBox)
        {
            // Create Image:
            //==============
            int Height = ImageMatrix.GetLength(0);
            int Width = ImageMatrix.GetLength(1);
            Bitmap ImageBMP = new Bitmap(Width, Height, PixelFormat.Format24bppRgb);
            unsafe
            {
                BitmapData bmd = ImageBMP.LockBits(new Rectangle(0, 0, Width, Height), 
ImageLockMode.ReadWrite, ImageBMP.PixelFormat);
                int nWidth = 0;
                nWidth = Width * 3;
                int nOffset = bmd.Stride - nWidth;
                byte* p = (byte*)bmd.Scan0;
                for (int i = 0; i < Height; i++)
                {
                    for (int j = 0; j < Width; j++)
                    {
                        p[2] = ImageMatrix[i, j].red;
                        p[1] = ImageMatrix[i, j].green;
                        p[0] = ImageMatrix[i, j].blue;
                        p += 3;
                    }

                    p += nOffset;
                }
                ImageBMP.UnlockBits(bmd);
            }
            SaveFileDialog dialog = new SaveFileDialog();
            ImageBMP.Save("myfile.png", ImageFormat.Png);
            PicBox.Image = ImageBMP;
        }
        /// <summary>
        /// Apply Gaussian smoothing filter to enhance the edge detection
        /// </summary>
        /// <param name="ImageMatrix">Colored image matrix</param>
        /// <param name="filterSize">Gaussian mask size</param>
        /// <param name="sigma">Gaussian sigma</param>
        /// <returns>smoothed color image</returns>
        public static List<edge> CalculateMST(int numberOfDistinctColors, List<RGBPixel> colorSet)
        {
            Stopwatch MSTTime = new Stopwatch();
            MSTTime.Start();
            overAllTime.Start();
            // bool array to check optimized vertcies
            bool[] partnersOfTheTree = new bool[numberOfDistinctColors];
            int least = 0, localLeast = 0;
            double min = 99999, dist;
            // holds the edge to reduce access time
            edge tmp;
            List<edge> MST = new List<edge>();
            // adds a fake edge to start the MST indexing from 1
            MST.Add(new edge(0, 0, 0));//O(1)
            int k = 1;
            // loops from 0 to all other vertcies
            while (k < numberOfDistinctColors)//O(N)
            {
                // calculating the distance
                byte red1 = colorSet[localLeast].red;//O(1)
                byte green1 = colorSet[localLeast].green;//O(1)
                byte blue1 = colorSet[localLeast].blue;//O(1)
                byte red2 = colorSet[k].red;//O(1)
                byte green2 = colorSet[k].green;//O(1)
                byte blue2 = colorSet[k].blue;//O(1)
                double red = (red1 - red2) * (red1 - red2);//O(1)
                double green = (green1 - green2) * (green1 - green2);//O(1)
                double blue = (blue1 - blue2) * (blue1 - blue2);//O(1)
                dist = Math.Sqrt(red + green + blue);//O(1)
                // adding the edge to the initial MST
                MST.Add(new edge(dist, 0, k));//O(1)
                // checks if it is the minimmum edge to start from its end
                if (dist < min)//O(1)
                {
                    // change the minimum
                    min = dist;//O(1)
                    // save the index of the next start over
                    least = k;//O(1)
                }
                k++;
            }
            k = 1;
            // looping from every vertix exept of 0 to the others
            while (k < numberOfDistinctColors)//O(N)
            {
                //reset the min for every vertix
                min = 99999;
                //set the startPoint as optiml way found
                partnersOfTheTree[least] = true;
                // reserve the start point index to prevent fomr modifications
                localLeast = least;
                // check the distance to all others from the current start point
                for (int j = 1; j < numberOfDistinctColors; j++)//O(D)
                {
                    // exclude the optimal points if found
                    if (partnersOfTheTree[j] == false)//O(D)
                    {
                        // calculating the distance
                        byte red1 = colorSet[localLeast].red;//O(1)
                        byte green1 = colorSet[localLeast].green;//O(1)
                        byte blue1 = colorSet[localLeast].blue;//O(1)
                        byte red2 = colorSet[j].red;//O(1)
                        byte green2 = colorSet[j].green;//O(1)
                        byte blue2 = colorSet[j].blue;//O(1)
                        double red = (red1 - red2) * (red1 - red2);//O(1)
                        double green = (green1 - green2) * (green1 - green2);//O(1)
                        double blue = (blue1 - blue2) * (blue1 - blue2);//O(1)
                        dist = Math.Sqrt(red + green + blue);//O(1)
                        // hold the edge to reduce access time
                        tmp = MST[j];//O(1)
                        //check if it is a better way to this vertix
                        if (dist < tmp.distance)//O(1)
                        {
                            //replace the distance
                            MST[j].distance = dist;//O(1)
                            //replace the starting index
                            MST[j].points[0] = localLeast;//O(1)
                        }
                        //check if it is the best edge in the tree to set the next start point
                        if (tmp.distance < min)//O(1)
                        {
                            //change the min
                            min = tmp.distance;//O(1)
                            // reserve the index
                            least = tmp.points[1];//O(1)
                        }
                    }
                }
                k++;
            }
            double MST_SUM = 0;
            //calculating the MST SUM
            for (int i = 0; i < MST.Count; i++) MST_SUM = MST_SUM + MST[i].distance;//O(N)
            MSTTime.Stop();
            overAllTime.Stop();
            phase1Time = overAllTime.ElapsedMilliseconds;
            //Console.WriteLine(MST_SUM + " MST SUM");
            //int MSTCOUNT = MST.Count()-1;
            MessageBox.Show(MST_SUM.ToString() + " MST SUM IN TIME " + MSTTime.ElapsedMilliseconds + " ms");
            //Console.WriteLine(MSTCOUNT - 1 + " MST COUNT");
            //double MEAN = MST_SUM / (MST.Count() - 1);
            //Console.WriteLine(MEAN + " MST MEAN");
            ////List<double> std = new List<double>();
            //double stdSUM = 0;
            //int current = 0;
            //double currentMean = MEAN;
            //int current_MSTCOUNT = MSTCOUNT;
            //for (int i = 1; i < MSTCOUNT; i++) stdSUM += Math.Pow((MST[i].distance - MEAN), 2);
            //stdSUM /= (MSTCOUNT - 1);
            //stdSUM = Math.Sqrt(stdSUM);
            //Console.WriteLine(stdSUM + " MST STD");
            //Console.WriteLine("------------------------------------------------------------");
            return MST;
        }
        public static List<List<int>> clustering(int numberOfDistinctColors, List<edge> MST, int k)
        {

            Stopwatch clusteringTime = new Stopwatch();
            clusteringTime.Start();
            overAllTime.Restart();
            List<List<int>> clusters = new List<List<int>>();
            int[] indexer = new int[numberOfDistinctColors];
            int cluster = 1, unClusterd = numberOfDistinctColors, numberOfClusters = 0, p1, p2, I1, I2;//O(1)
            edge tmp;
            MST.Sort(); // O(Nlog(N))
            // loop on every edge to get it's 2 points
            for (int i = 1; i < numberOfDistinctColors; i++)//O(E)
            {
                // holds the edge to reduce access time
                tmp = MST[i];
                // holds the edge points and their indexer value to reduce access time
                p1 = tmp.points[0]; p2 = tmp.points[1]; I1 = indexer[p1]; I2 = indexer[p2];
                // if the clusters are found
                if (numberOfClusters + unClusterd == k)//O(1)
                {
                    //if all points are clusterd then break ;
                    if (unClusterd == 0) break;//O(1)
                    // if the point refrences 0 as not clustered
                    if (I1 == 0)
                    {
                        // create a new cluster to hold this point
                        List<int> tempCluster = new List<int>();//O(1)
                        tempCluster.Add(p1);//O(1)
                        clusters.Add(tempCluster);//O(1)
                        indexer[p1] = cluster;//O(1)
                        cluster++;//O(1)
                        numberOfClusters++;//O(1)
                        unClusterd--; //O(1)
                    }
                    // if the point refrences 0 as not clustered
                    if (I2 == 0)//O(1)
                    {
                        // create a new cluster to hold this point
                        List<int> tempCluster = new List<int>();//O(1)
                        tempCluster.Add(p2);//O(1)
                        clusters.Add(tempCluster);//O(1)
                        indexer[p2] = cluster;//O(1)
                        cluster++;//O(1)
                        numberOfClusters++;//O(1)
                        unClusterd--;//O(1)
                    }
                    continue;
                }
                // if both are clusterd and not the same cluster
                if (I1 != 0 && I2 != 0 && I1 != I2)//O(1)
                {
                    int keep = I2;//O(1)
                    //change every value of the indexer of the 2nd point to referance the cluseter of the 1st point
                    clusters[keep - 1].ForEach(l => indexer[l] = I1);//O(N)
                    //union the 2 clusters
                    clusters[I1 - 1].Union(clusters[keep - 1]);//O(Log(N))
                    // clears the derprecated cluster
                    clusters[keep - 1].Clear();//O(1)
                    // reduce the number of clusters
                    numberOfClusters--;//O(1)
                }
                //if only one point is clusterd
                else if (I1 != 0)
                {
                    //set the referance to the other point to referance he smae cluster as the first
                    indexer[p2] = I1;//O(1)
                    //add the other to its cluster
                    clusters[I1 - 1].Add(p2);//O(1)
                    // reduce the number of unClusterd
                    unClusterd--;//O(1)

                }
                //if only one point is clusterd
                else if (I2 != 0)//O(1)
                {
                    //set the referance to the other point to referance he smae cluster as the first
                    indexer[p1] = I2;//O(1)
                    //add the other to its cluster
                    clusters[I2 - 1].Add(p1);//O(1)
                    // reduce the number of unClusterd
                    unClusterd--;//O(1)
                }
                // if both are not clusterd
                else
                {
                    // create new cluster
                    List<int> tempCluster = new List<int>();
                    // add both points
                    tempCluster.Add(tmp.points[0]);//O(1)
                    tempCluster.Add(tmp.points[1]);//O(1)
                    clusters.Add(tempCluster);//O(1)
                    // set the referance to both points as the cluster number
                    indexer[p1] = cluster;//O(1)
                    indexer[p2] = cluster;//O(1)
                    cluster++;//O(1)
                    numberOfClusters++;//O(1)
                    unClusterd -= 2;//O(1)
                }
            }
            clusteringTime.Stop();
            overAllTime.Stop();
            MessageBox.Show("the number of clusters " + k + " in time " + clusteringTime.ElapsedMilliseconds + " ms");
            return clusters;
        }
        public static List<RGBPixel> generatePallete(List<List<int>> clusters, List<RGBPixel> colorSet, ref int[,,] mapper)
        {
            overAllTime.Start();
            Stopwatch generatePalleteTime = new Stopwatch();
            generatePalleteTime.Start();
            //re intiailize the mapper to over write any old data
            mapper = new int[256, 256, 256];//O(1)
            List<RGBPixel> palletaa = new List<RGBPixel>();
            int clusterCounter = 0, count = clusters.Count;
            for (int i = 0; i < count; i++)//O(N)
            {
                // holds the color
                RGBPixel current;//O(1)
                double red = 0, green = 0, blue = 0;//O(1)
                int numberOfClusterElements = clusters[i].Count();//O(1)
                // clculates the avarage of red , green and blue
                foreach (var j in clusters[i])
                {
                    red = red + colorSet[j].red;
                    green = green + colorSet[j].green;
                    blue = blue + colorSet[j].blue;
                    //sets the referance to the color to it's cluster number
```

```
                        mapper[colorSet[j].red, colorSet[j].green, colorSet[j].blue] = clusterCounter;//O(1)
                    }
                    // overcoming the error from clustring step by skipping every empty cluster
                    if (clusters[i].Count == 0) continue;
                    // round to handle double numbers error
                    current.red = (byte)Math.Round(red / numberOfClusterElements);//O(1)
                    current.green = (byte)Math.Round(green / numberOfClusterElements);//O(1)
                    current.blue = (byte)Math.Round(blue / numberOfClusterElements);//O(1)
                    // add the color to the pallet list in the same refernce to the mapper
                    palletaa.Add(current);//O(1)
                    // change ther referance
                    clusterCounter++;//O(1)
                }
                generatePalleteTime.Stop();
                overAllTime.Stop();
                MessageBox.Show("generating Pallet done in  time " + generatePalleteTime.ElapsedMilliseconds + " ms");
                return palletaa;
            }
            public static RGBPixel[,] Quantize(RGBPixel[,] ImageMatrix, List<RGBPixel> p, int[,,] mapper)
            {
                Stopwatch QuantizeTime = new Stopwatch();
                QuantizeTime.Start();
                overAllTime.Start();
                int Height = GetHeight(ImageMatrix);
                int Width = GetWidth(ImageMatrix);
                RGBPixel[,] Filtered = new RGBPixel[Height, Width];
                //loop to get every pixel in the picture to replace
                for (int i = 0; i < Height; i++)//O(N)
                    for (int j = 0; j < Width; j++)//O(N)
                    {
                        RGBPixel quantized = new RGBPixel();
                        //gets the pallet referance from the mapper then saves the color
                        quantized = p[mapper[ImageMatrix[i, j].red, ImageMatrix[i, j].green, ImageMatrix[i, j].blue]];//O(1)
                        //replace the old color by it's representitive
                        Filtered[i, j].red = quantized.red;//O(1)
                        Filtered[i, j].green = quantized.green;//O(1)
                        Filtered[i, j].blue = quantized.blue;//O(1)
                    }
                QuantizeTime.Stop();
                overAllTime.Stop();
                MessageBox.Show("image mapping colors done in time " + QuantizeTime.ElapsedMilliseconds + " ms");
                MessageBox.Show("over all time is  " + (overAllTime.ElapsedMilliseconds + phase1Time) + " ms");
                MessageBox.Show("over all time is  " + ((overAllTime.ElapsedMilliseconds + phase1Time) / 1000) + " S");
                MessageBox.Show("over all time is  " + ((overAllTime.ElapsedMilliseconds + phase1Time) / 1000 / 60) + " M");
                return Filtered;
            }
        }
    }
```

The code of MainForm.cs:

```
 using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace ImageQuantization
```

```
{
    public partial class MainForm : Form
    {
        public MainForm()
        {
            InitializeComponent();
        }

        RGBPixel[,] ImageMatrixOriginal, ImageMatrixQuantized;
        List<RGBPixel> p = new List<RGBPixel>() , colorSet = new List<RGBPixel>();
        List<List<int>> clustersList = new List<List<int>>();
        List<edge> MST;
        int[,,] mapper = new int[256, 256, 256];
        private void btnOpen_Click(object sender, EventArgs e)
        {
            OpenFileDialog openFileDialog1 = new OpenFileDialog();
            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                //Open the browsed image and display it
                string OpenedFilePath = openFileDialog1.FileName;
                ImageMatrixOriginal = ImageOperations.OpenImage(OpenedFilePath, ref mapper, ref colorSet);
                ImageOperations.DisplayImage(ImageMatrixOriginal, pictureBox1);
                MST = ImageOperations.CalculateMST(colorSet.Count, colorSet);
            }
            txtWidth.Text = ImageOperations.GetWidth(ImageMatrixOriginal).ToString();
            txtHeight.Text = ImageOperations.GetHeight(ImageMatrixOriginal).ToString();
        }

        private void btnGaussSmooth_Click(object sender, EventArgs e)
        {
            int clusters = (int)nudMaskSize.Value;
            clustersList = ImageOperations.clustering(colorSet.Count, MST, clusters);
            p = ImageOperations.generatePallete(clustersList, colorSet, ref mapper);
            ImageMatrixQuantized = ImageOperations.Quantize(ImageMatrixOriginal, p, mapper);
            ImageOperations.DisplayImage(ImageMatrixQuantized, pictureBox2);
        }

        private void pictureBox1_Click(object sender, EventArgs e)
        {

        }

        private void pictureBox2_Click(object sender, EventArgs e)
        {

        }

        private void label1_Click(object sender, EventArgs e)
        {

        }

        private void label2_Click(object sender, EventArgs e)
        {

        }

        private void label3_Click(object sender, EventArgs e)
        {

        }

        private void label4_Click(object sender, EventArgs e)
        {

        }

        private void txtHeight_TextChanged(object sender, EventArgs e)
        {

        }

        private void nudMaskSize_ValueChanged(object sender, EventArgs e)
        {

        }

        private void txtWidth_TextChanged(object sender, EventArgs e)
        {

        }

        private void label5_Click(object sender, EventArgs e)
        {

        }

        private void label6_Click(object sender, EventArgs e)
        {

        }

        private void txtGaussSigma_TextChanged(object sender, EventArgs e)
        {

        }

        private void panel1_Paint(object sender, PaintEventArgs e)
        {

        }

        private void panel2_Paint(object sender, PaintEventArgs e)
        {

        }
    }
}
```