**Image Quantization**

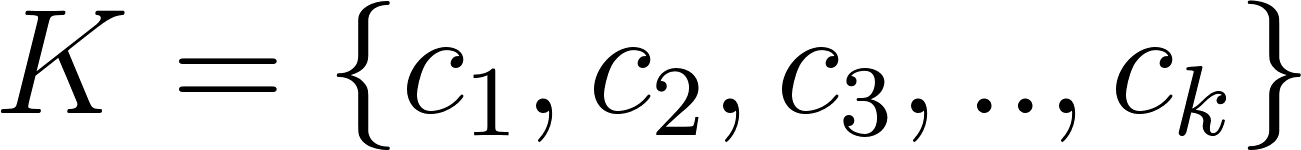
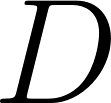
# What is Image Quantization?

**Quantization** is a lossy (some of the information is lost) compression technique achieved by compressing a range of values to a single quantum value. Image or color quantization is a technique where the number of colors is reduced; this is important for displaying images on devices that support a limited number of colors and for efficiently compressing certain kinds of images. Most editors and many operating systems have built-in support for color quantization. The reduced representative colors are called **color palette**. The quantization process should be optimized as possible so that not much information is lost and the new quantized image should differ as little as possible from the original image.

# Importance

1. **Image compression:** by reducing the number of bits per pixels without affecting the image view. It’s used as a step in the compression pipeline of most common formats like JPEG and MPEG.
2. **Image Filtering:** for noise smoothing and for deconvolving images that have been blurred.
3. **Target different devices:** color quantization is critical for displaying images with many colors on devices that can only display a limited number of colors

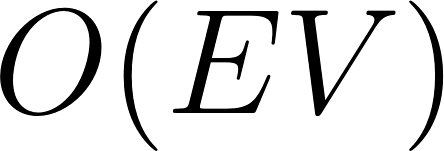
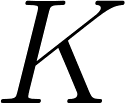
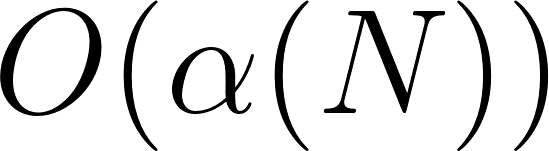
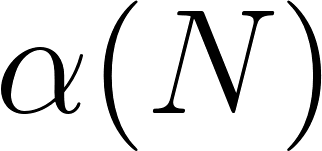
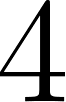
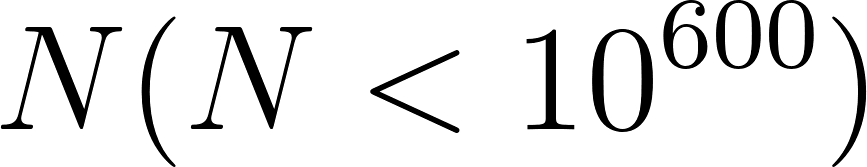
# Main Steps

1. **Color palette generation:** generating the smallest possible number of representatives [](https://www.codecogs.com/eqnedit.php?latex=K%20%3D%20%5C%7B%20c_1%20%2C%20c_2%20%2C%20c_3%20%2C%E2%80%A6..%E2%80%A6%2C%20c_k%5C%7D%0) from [](https://www.codecogs.com/eqnedit.php?latex=D%0) distinct color.
2. **Image Quantization:** by replacing every pixel [](https://www.codecogs.com/eqnedit.php?latex=RGB%0) with its representative.

**Original Image** **Quantized Image**



# Algorithms steps

1. Save only the distinct color in the image to avoid memory loss as a dense graph, where different color is node in the graph and it is connected to all other nodes.
2. The binary heap data structure was implemented to decrease the running time from [](https://www.codecogs.com/eqnedit.php?latex=O(EV)%0)to [](https://www.codecogs.com/eqnedit.php?latex=O(E%5Clog(V))%0) when constructing a minimum spanning tree.
3. Creation of a minimum spanning tree using **Prim’s** algorithm instead of **Kruskal’s** algorithm to avoid calculating all the edges weights then sorting them so the edge is only calculated when needed and all edges weights don’t necessarily need to be stored at first.
4. Creation of [](https://www.codecogs.com/eqnedit.php?latex=K%0) clusters using disjoint set union which runs at a complexity [](https://www.codecogs.com/eqnedit.php?latex=O(%5Calpha(N))%0), where [](https://www.codecogs.com/eqnedit.php?latex=%5Calpha(N)%0) is the inverse Ackermann function, which grows very slowly. In fact it grows so slowly that it doesn't exceed [](https://www.codecogs.com/eqnedit.php?latex=4%0) for all reasonable [](https://www.codecogs.com/eqnedit.php?latex=N(N%20%3C%2010%5E%7B600%7D)%0) approximately.
5. Calculating the representative of each cluster.
6. Replacing each color on the image with its representative.

# Algorithms Analysis

1.Construct Graph Function

|  |
| --- |
| public static RGBPixel[] constructGraph(RGBPixel[,] imageMatrix)  {  int height = GetHeight(imageMatrix), width = GetWidth(imageMatrix); *//Θ(1)*  Dictionary<string, int> distinctColors = new Dictionary<string, int>(); *//Θ(1) initializing*  int counter = 0; *//Θ(1) initializing*  string s = ""; *//Θ(1) initializing*  for (int i = 0; i < height; i++)  *// two for loops on the image dimensions complexity Θ(N^2)*  {  for (int j = 0; j < width; j++)  {  s = imageMatrix[i, j].blue.ToString() + '-';  *//using tostring and contains key which are Θ(1) so all body Θ(1)*  s += imageMatrix[i, j].green.ToString() + '+';  s += imageMatrix[i, j].red.ToString();  if (!(distinctColors.ContainsKey(s)))  {  distinctColors[s] = counter;  counter++;  }  }  }  Console.WriteLine("The Output is:"); *//showing output Θ(1)*  Console.Write("Number of Distinct colors: "); *//showing output Θ(1)*  Console.WriteLine(distinctColors.Count); *//showing output Θ(1)*  int ss = distinctColors.Count; *//Θ(1) initializing*  RGBPixel[] Graph = new RGBPixel[ss]; *//Θ(1) initializing*  foreach (var it in distinctColors)   *//Θ(N) iterates on distinct colors*  {  s = it.Key; *//Θ(1)*  string red = "", green = "", blue = ""; *//Θ(1)*  int length = s.Length; *//Θ(1)*  for (int i = 0; i < length - 2; i++)  *// Θ(1) loop it iterates always 9 times only*  *// (3 colors of 3 numbers + 2 signes - 2 from length) = 3\*3+2-2 = 9 times*  {  if (s[0] == '-' && s[1] == '+')  {  red += s[2];  s = s.Remove(2, 1); *//removing from constant string Θ(1)*  }  else if (s[0] == '-')  {  green += s[1];  s = s.Remove(1, 1); *//removing from constant string Θ(1)*  }  else  {  blue += s[0];  s = s.Remove(0, 1); *//removing from constant string Θ(1)*  }  }  Graph[it.Value].blue = Convert.ToByte(blue);  Graph[it.Value].green = Convert.ToByte(green);  Graph[it.Value].red = Convert.ToByte(red);  *//converting int to byte Θ(1)*    }  return Graph;  } |

2.Get Edge Weight Function

|  |
| --- |
| public static double GetEgdeWeight(RGBPixel Color1, RGBPixel Color2)  {  return Math.Sqrt((Math.Pow(Color1.blue - Color2.blue, 2)) + (Math.Pow(Color1.red - Color2.red, 2)) + (Math.Pow(Color1.green - Color2.green, 2)));  *//all built in math functions Θ(1)*  } |

3.HeapNode + ResultSet Class

|  |
| --- |
| public class HeapNode  {  *//initializing class Θ(1)*  public int vertex;  public double key;  public HeapNode()  {  }  }  public class ResultSet  {  *//initializing class Θ(1)*  public int parent;  public double weight;  public int current;  public ResultSet()  {  }  } |

4.MinHeap Class with its functions

|  |
| --- |
| public class MinHeap  {  *//initializing class Θ(1)*  public int capacity;  public int Size;  public HeapNode[] mH;  public int[] indexes;   public MinHeap(RGBPixel[] G)  {  this.capacity = G.Length; *//Θ(1)*  mH = new HeapNode[capacity + 1]; *//Θ(1)*  mH[0] = new HeapNode(); *//Θ(1)*  mH[0].key = double.MinValue; *//Θ(1)*  indexes = new int[G.Length]; *//Θ(1)*  Size = 0; *//Θ(1)*  }  public void insert(HeapNode x)  {  Size++; *//Θ(1)*  int idx = Size; *//Θ(1)*  mH[idx] = x; *//Θ(1)*  indexes[x.vertex] = idx; *//Θ(1)*  bubbleUp(idx); *//O(Log(V)) its analysis in the function*  }  public void bubbleUp(int pos)  {  int parentIdx = pos / 2; *//Θ(1)*  int currentIdx = pos; *//Θ(1)*    *//while loop average case complexity O(1) and worst case Complexity O(log(V))*   *//as its iterating on the levels (which is parentIdx/2) and the loop body O(1)*  *//adding values in variables and using Swap function which is Θ(1) so while loop complexity is O(log(V)).*  while (currentIdx > 0 && mH[parentIdx].key > mH[currentIdx].key)  {  HeapNode currentNode = mH[currentIdx];  HeapNode parentNode = mH[parentIdx];   indexes[currentNode.vertex] = parentIdx;  indexes[parentNode.vertex] = currentIdx;  swap(currentIdx, parentIdx);  currentIdx = parentIdx;  parentIdx = parentIdx / 2;  }  }  public void swap(int a, int b)  {  *//function swapping by using temp order Θ(1)*  HeapNode temp = mH[a];  mH[a] = mH[b];  mH[b] = temp;  }  public HeapNode extractMin()  {  HeapNode min = mH[1]; *//Θ(1)*  HeapNode lastNode = mH[Size]; *//Θ(1)*  indexes[lastNode.vertex] = 1; *//Θ(1)*  mH[1] = lastNode; *//Θ(1)*  mH[Size] = null; *//Θ(1)*  sinkDown(1);  *//calling sinkDown function which is upper bound O(log(V)).*  Size--; *//Θ(1)*  return min;  }   public void sinkDown(int k)  {   */\* Divide and Conquer function(similar to binary search)  Divide: check the child in left or right part(comparing to the size of the heap) Conquer: Recursively on 1 half Combine: Trivial\*/* *//Analysis of this recursive code is :* *//T(V) = T(V / 2) + Θ(1) solving using master method* *//Complexity will be Exact Θ(log(V))*  int smallest = k;  int leftChildIdx = 2 \* k;  int rightChildIdx = 2 \* k + 1;  if (leftChildIdx < heapSize() && mH[smallest].key > mH[leftChildIdx].key)  {  smallest = leftChildIdx;  }  if (rightChildIdx < heapSize() && mH[smallest].key > mH[rightChildIdx].key)  {  smallest = rightChildIdx;  }  if (smallest != k)  {  HeapNode smallestNode = mH[smallest];  HeapNode kNode = mH[k];  indexes[smallestNode.vertex] = k;  indexes[kNode.vertex] = smallest;  swap(k, smallest);  sinkDown(smallest);  }  }  public bool isEmpty()  {   return Size == 0; *//Θ(1)*  }   public int heapSize()  {  return Size; *//Θ(1)*  }  }  public static void decreaseKey(MinHeap minHeap, double newKey, int vertex)  {   int index = minHeap.indexes[vertex];   HeapNode node = minHeap.mH[index]; *//Θ(1)*  node.key = newKey; *//Θ(1)*  minHeap.bubbleUp(index); *//O(log(V))*  } |

5.Prim’s MST Function

|  |
| --- |
| public static void primMST(RGBPixel[,] imagematrix)   {  RGBPixel[] Graph = constructGraph(imagematrix);  *//calling function Θ(N^2)*  int size = Graph.Length; *//Θ(1)*  bool[] inHeap = new bool[size]; *//Θ(1)*  ResultSet[] resultSet = new ResultSet[size]; *//Θ(1)*  double[] key = new double[size]; *//Θ(1)*  HeapNode[] heapNodes = new HeapNode[size]; *//Θ(1)* *// for loop iterates V times(number of vertices) of body order //Θ(1)(initializations)* *// loop total complexity is order Θ(V)*  for (int i = 0; i < size; i++)   {  heapNodes[i] = new HeapNode();  heapNodes[i].vertex = i;  heapNodes[i].key = double.MaxValue;  resultSet[i] = new ResultSet();  inHeap[i] = true;  key[i] = double.MaxValue;  }  heapNodes[0].key = 0;  MinHeap minHeap = new MinHeap(Graph); *//Θ(1)*  *//calling function insert O(log(V)) in loop iterates V times times (number of vertices)*  *//loop total complexity is order O(V log(V))*  for (int i = 0; i < size; i++)  {  minHeap.insert(heapNodes[i]);  } */\* calling extractMin function with complexity O(log(V)) \*the While loop complexity Θ(V) ...so 1st part is O(V log(V)). Then the 2nd part is the body of the For loop calling GetEdgeweight function which is order Θ(1) and inside the if body calling decreaseKey function which is order of O(log(V)) \* the(While loop \* For loop) iterations which is the number of Edges Θ(E) which equals V ^ 2 as its Dense graph so its O(E log(V)). So Total complexity of while and for loops is O(V log(V)) + O(E log(V)) = O(E log(V)). So total complexity of primMST function is O(V log(V)) + Θ(V) + O(E log(V)) = O(E log(V)).\*/*  while (!minHeap.isEmpty())  {  HeapNode extractedNode = minHeap.extractMin();   int extractedVertex = extractedNode.vertex;  inHeap[extractedVertex] = false;   for (int i = 0; i < Graph.Length; i++)  {  if (inHeap[i])  {  double newKey = GetEgdeWeight(Graph[extractedVertex], Graph[i]);  if (key[i] > newKey)  {   decreaseKey(minHeap, newKey, i);  resultSet[i].parent = extractedVertex;  resultSet[i].weight = newKey;  resultSet[i].current = i;  key[i] = newKey;  }  }  }   }   double W = 0; *//Θ(1)*   Array.Sort(resultSet, (x, y) => y.weight.CompareTo(x.weight));  *//O(Dlog(D))*   for (int i = 0; i < Graph.Length; i++)  {  W += resultSet[i].weight;  } *//Θ(N) (number of distinct colors)*  Console.Write("Sum of MST is: ");*//Θ(1)*  Console.WriteLine(W);*//Θ(1)*  clustering(resultSet, Graph);  *//Clustering (analysis in the function)*  } |

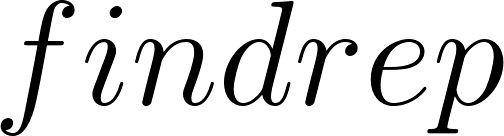
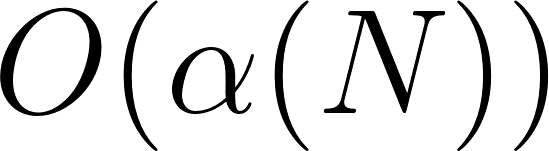
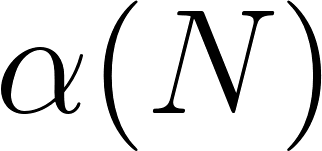
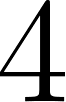
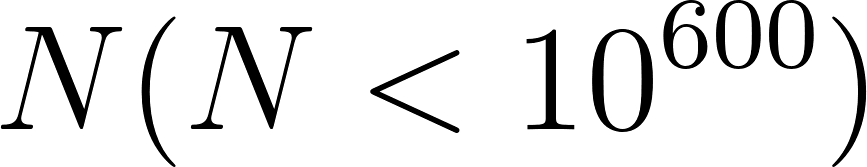
6.Clustering Function

|  |
| --- |
| public static Dictionary<string, RGBPixelD> clustering(ResultSet[] mST , RGBPixel[] Distinct )  {  *// this array contains the parent or representative that the index*   *// follows*   int[] repres = new int[Distinct.Length]; *// Θ(1)*  *// this array contains the number of elements that the index is*   *// parent of*   int[] size = new int[Distinct.Length]; *// Θ(1)*  RGBPixelD[] distClusters = new RGBPixelD[Distinct.Length]; *// Θ(1)*    *//this loop fills the array of RGBPixelD with values of the distict*   *//colours in the picture*   for ( int i = 0; i < Distinct.Length; i++) *// Θ(D)*  {  distClusters[i].blue = Distinct[i].blue; *// Θ(1)*  distClusters[i].green = Distinct[i].green; *// Θ(1)*  distClusters[i].red = Distinct[i].red; *// Θ(1)*  }  This loop initializes the two arrays repres and size   for ( int i = 0; i < Distinct.Length; i++) *// Θ(D)*  {  repres[i] = i; *// Θ(1)*  size[i] = 1; *// Θ(1)*  }  int clusters = Distinct.Length; *// Θ(1)*  *//this loop does the clustering in complexity O(DK)*   *//more on the complexity down below*  for (int i = mST.Length-1; i>=0; i--) O(D \* body Complexity)  {  clusters = uniteSet(mST[i].current, mST[i].parent, repres, size , clusters); *// (K) more on that below*  if (clusters == k) *// Θ(1)*  Break; *// Θ(1)*  }  bool[] visited = new bool[mST.Length ];*// Θ(1)*  RGBPixelD accumSum; *// Θ(1)*  accumSum.blue = 0; *// Θ(1)*  accumSum.green = 0; *// Θ(1)*  accumSum.red = 0; *// Θ(1)*    *// this loop gets the sum of all clusters by calling the recursing*   *// function getSum which accumulates the sum of a clusters colours in*  *// the representative cell*  for ( int i = mST.Length-1; i >=0; i--) *// Θ(D \* body complexity)*  {  repres[i] = getSum(i , visited , repres , distClu  sters , accumSum ,size); *// O(K)*  }   *// this loop calculates the mean by dividing the accumulated sum by*   *// The size of the cluster*   for (int i = mST.Length - 1; i >= 0; i--) *// Θ(D)*  {  if (size[i] == 0) *// Θ(1)*  Continue; *// Θ(1)*  distClusters[i].blue /= size[i]; *// Θ(1)*  distClusters[i].green /= size[i]; *// Θ(1)*  distClusters[i].red /= size[i]; *// Θ(1)*  }  *// map is a dictionary that stores each distinct colour as string to*   *// avoid collisions mapped to the colour of the representative*   *// of its cluster*  map = new Dictionary<string, RGBPixelD> ();   *// This is the loop that fills the map according to the previous*   *// description*  for (int i = mST.Length-1;i>=0;i-- ) *// Θ(D)*  {  string s = ""; *// Θ(1)*  s = Distinct[i].blue.ToString() + '-'; *// Θ(1)*  s += Distinct[i].green.ToString() + '+'; *// Θ(1)*  s += Distinct[i].red.ToString(); *// Θ(1)*   if (size[i] == 0) *// Θ(1) // Θ(1)*  {  map[s] = distClusters[repres[i]]; *// Θ(1)*  }  else  map[s] = distClusters[i]; *// Θ(1)*  }  return map; *// Θ(1)*  } |

7.DSU functions

|  |
| --- |
| public static int uniteSet(int a, int b, int[] repres, int[] size, int k )  {  a = findRep(a, repres);   b = findRep(b, repres);     if (a!=b) *//Θ(1)*  {  if (size[a] < size[b]) *//Θ(1)*  {  repres[a] = b; *//Θ(1)*  size[b] += size[a]; *//Θ(1)*  }  else  {  repres[b] = a; *//Θ(1)*  size[a] += size[b]; *//Θ(1)*  }  k--; *//Θ(1)*  }  return k; *//Θ(1)*  } |

|  |
| --- |
| public static int findRep(int a , int[] repres)  {  if (a == repres[a]) *//Θ(1)*  {  return a; *//Θ(1)*  }  else  return repres[a] = findRep(repres[a] ,repres);   } |

The disjoint set union using its two functions [](https://www.codecogs.com/eqnedit.php?latex=findrep%0) and [](https://www.codecogs.com/eqnedit.php?latex=uniteSet%0) maintains a complexity [](https://www.codecogs.com/eqnedit.php?latex=O(%5Calpha(N))%0), where[](https://www.codecogs.com/eqnedit.php?latex=%5Calpha(N)%0) is the inverse Ackermann function, which grows very slowly. In fact it grows so slowly that it doesn't exceed [](https://www.codecogs.com/eqnedit.php?latex=4%0) for all reasonable [](https://www.codecogs.com/eqnedit.php?latex=N(N%20%3C%2010%5E%7B600%7D)%0) approximately so it runs in almost constant time.

8.getSum Function

This function is upper bounded by the number of formed Clusters, as it recurse on the representatives of a single colour and a single colour can have no more representatives than the number of formed Clusters. So the function is [](https://www.codecogs.com/eqnedit.php?latex=O(K)%0).

|  |
| --- |
| public static int getSum(int a , bool [] visited , int[] parent , RGBPixelD[] distinct , RGBPixelD accumulativeSum , int[] size)  {  if (a == parent[a]) *//Θ(1)*  {  distinct[a].blue += accumulativeSum.blue; *//Θ(1)*  distinct[a].red += accumulativeSum.red; *//Θ(1)*  distinct[a].green += accumulativeSum.green; *//Θ(1)*  return a; *//Θ(1)*  }  else if(!visited[a]) *//Θ(1)*  {  visited[a] = true; *//Θ(1)*  accumulativeSum.blue += distinct[a].blue; *//Θ(1)*  accumulativeSum.red += distinct[a].red; *//Θ(1)*  accumulativeSum.green += distinct[a].green; *//Θ(1)*  }  size[a] = 0; *//Θ(1)*  return parent[a] = getSum(parent[a] , visited , parent , distinct , accumulativeSum ,size); *//O(K)*  } |