private static int **Euclid\_Distance**(Point p, Point q)  **SHORTEST PATH**

{

return (int)Math.Sqrt((int)Math.Pow((p.X- q.X),2) + (int)Math.Pow((p.Y - q.Y), 2));

}

**CONNectedCELLS**

public int GetBiggestRegion(int[][] matrix)

{

maxHeight = matrix.Length-1;

List<int> sums = new List<int>();

HashSet<Point> visited = new HashSet<Point>();

for (int row = 0; row < matrix.Length; row++)

{

maxWidth = matrix[row].Length;

for (int column = 0; column < matrix[row].Length; column++)

{

if (matrix[row][column] == 1)

sums.Add(DFSScanNew(row, column, visited, matrix));

}

}

return sums.Max()

public int **DFSScan** (int row, int column, HashSet<Point> visited, int[][] matrix)

{

if (row < 0 || column < 0 || row > matrix.Length - 1 || column > matrix[row].Length - 1

|| visited.Contains(new Point(row, column))) {

return 0;

}

visited.Add(new Point(row, column));

if (matrix[row][column] == 0) return 0;

int sum = 1;

for (int r = row - 1; r <= row + 1; r++)

{

for (int c = column - 1; c <= column + 1; c++) {

if (r != row || c != column)

sum += DFSScanNew(r, c, visited, matrix);

} }

return sum;

}

bool **HasPathBFS**(Node source, Node destination)

{

HashSet<int> visited = new HashSet<int>();

Queue<Node> q = new Queue<Node>();

q.Enqueue(source);

while (q.Count > 0)

{

var current = q.Dequeue();

if (!visited.Contains(current.Value))

{

visited.Add(current.Value);

if (current.Value == destination.Value)

return true;

foreach (var item in current.Adjacents)

q.Enqueue(item);

} }

return false;

}

public class **Graph2** {

private Dictionary<int, Node> items = new Dictionary<int, Node>();

public class Node {

public int Value { get; }

public LinkedList<Node> Adjacents { get; set; }

public Node(int value)..

}

public void AddNode(int value)

public void AddEdge(int source, int destination)

public Node GetNode(int value)

StringPermutations(“”, ‘abc’)

private static void **StringPermutations**(string prefix, string input)

{

int n = input.Length;

if (n == 0) return;

else {

for (int i = 0; i < n; i++)

{

string rem = input.Substring(0, i) + input.Substring(i + 1);

StringPermutations(prefix+ input[i], rem);

} }

}

public int **LengthOfLongestSubstring**(string s)

{

int ans = 0, i = 0, j = 0;

int n = s.Length;

HashSet<char> table = new HashSet<char>();

while (i < n && j < n)

{

if (!table.Contains(s[j]))

{

table.Add(s[j++]);

ans = Math.Max(ans, j - i);

}

else

table.Remove(s[i++]);

}

return ans;

**Set bit :** A |= 1 << bit

**Clear bit :** A &= ~(1 << bit)

**Test bit :** (A & 1 << bit) != 0

static int **NumberOfSteps**(int n, Dictionary<int, int> map)

{

if (n == 0 || n == 1) return n;

if (!map.ContainsKey(n))

map.Add(n, NumberOfSteps(n - 1, map) + NumberOfSteps(n - 2, map));

return map[n];

}

return **ReverseStringRecursively**(value.Substring(1)) + value[0];

public static void **QuickSort**(int[] array, int start, int end)

{

int index = partition(array, start, end);

if (start < index - 1)

QuickSort(array, start, index - 1);

if (index < end)

QuickSort(array, index, end);

}

private static int partition(int[] array, int start, int end)

{

int pivot = array[(end+start)/2];

while (start <= end)

{

while (array[start] < pivot)

start++;

while (array[end] > pivot)

end--;

if (start <= end)

{

int temp = array[start];

array[start] = array[end];

array[end] = temp;

start++;

end--;

} }

return start;

}

public static void **MergeSort**(int[] array, int[] tempArray, int first, int last)

{

if (first >= last) return;

int mid = (last + first) / 2;

MergeSort(array, tempArray, first, mid);

MergeSort(array, tempArray, mid + 1, last);

MergeArrays(array, tempArray, first, last);

}

private static void MergeArrayGOOD(int[] array, int[] tempArray, int leftStart, int rightEnd)

{

int leftEnd = (rightEnd + leftStart) / 2;

int rightStart = leftEnd + 1;

int size = rightEnd - leftStart + 1;

int left = leftStart;

int right = rightStart;

int index = left;

while (left <= leftEnd && right <= rightEnd)

{

if (array[left] <= array[right])

tempArray[index++] = array[left++];

else

tempArray[index++] = array[right++];

}

while (left <= leftEnd)

{

tempArray[index++] = array[left++];

}

while (right <= rightEnd)

{

tempArray[index++] = array[right++];

}

Array.Copy(tempArray, leftStart, array, leftStart, size);