Homework

Data structures, algorithms and complexity.

Task 1. What is the expected running time of the following C# code? Explain why.
Assume the array's size is n.

```
long Compute(int[] arr)
{
   long count = 0;
   for (int i=0; i<arr.Length; i++)
   {
      int start = 0, end = arr.Length-1;
      while (start < end)
         if (arr[start] < arr[end])
            { start++; count++; }
         else
            end--;
   }

   return count;
}
```

*Ok, we have two cycles – one for cycle and in it, one nested while loop. The for loop will do N operations (arr.length) or 0 operations if the array is empty. If we look closer the while loop will do exactly N operations too. But the while loop is activated for every iteration of the for cycle. That means we have N*N = N^2 operations.*
Earlier we mentioned that if the array is empty, no operations will be executed.
That means that we are in the case of the big O notation. Here are the asymptotic notation definitions:

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0, \exists n_0 : \forall n \geq n_0, 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)\}$$

$$O(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 : \forall n \geq n_0, 0 \leq f(n) \leq c.g(n)\}$$

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 : \forall n \geq n_0, 0 \leq c.g(n) \leq f(n)\}$$

$$o(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 : \forall n \geq n_0, 0 \leq f(n) < c.g(n)\}$$

$$\omega(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 : \forall n \geq n_0, 0 \leq c.g(n) < f(n)\}$$

The definition means that our function f(n) is part of set of functions that are trapped between another function c.g(n). In our case the performance is between 0 (constant function) and N^2 (quadratic function). So from here we conclude that our code has expected running time of ***$O(N^2)$.***

Task 2.
What is the expected running time of the following C# code? Explain why.

```
long CalcCount(int[,] matrix)
{
   long count = 0;
   for (int row=0; row<matrix.GetLength(0); row++)
      if (matrix[row, 0] % 2 == 0)
         for (int col=0; col<matrix.GetLength(1); col++)
            if (matrix[row,col] > 0)
               count++;
   return count;
}
```

The situation is similar to Task 1. Again we have one for loop and one nested loop inside it. The first loop will do exactly ***n*** *operations(number of the rows). The nested loop on the other hand depends on the matrix. The nested loop is only activated if the condition (****matrix[row, 0] % 2 == 0)*** *is true. So in the worst case all the first elements of the rows of the matrix will be even and the loop will execute* ***m*** *times(number of the columns). That makes it clear that our worst case scenario is n*m operatrions. There is a possibility that neither of the loops will be executed, so the expected execution time of this code is* ***O(n*m).***

Task 3. What is the expected running time of the following C# code? Explain why. Assume the input matrix has size of n * m.

```
long CalcSum(int[,] matrix, int row)
{
   long sum = 0;
   for (int col = 0; col < matrix.GetLength(0); col++)
      sum += matrix[row, col];
   if (row + 1 < matrix.GetLength(1))
      sum += CalcSum(matrix, row + 1);
   return sum;
}
Console.WriteLine(CalcSum(matrix, 0));
```

Things are little more complex here, because we have recursion. Also the way the code is written the program won't work well if the matrix is not symmetrical. Assuming it is symmetrical, the first loop will execute N times and after that the recursion will be activated. When the first recursion is called new (N-1) steps will be executed, when the next recursion is called we will have plus (N-2) steps. That means that we have N + (N-1) + (N-2) + (N-3) + … + (N-K). This is an arithmetic progression with sum $N(N-K)/2*K = (N^2.K – N.K^2) / 2 =$ ***$O(N^2)$.***