Practical 2: Analysis of Algorithms

**Question 1: Big-O notation**

1. In proportion, so
2. Constant time, regardless of input
3. Has , so complexity is of class

**Question 2:**

c)

**if**

**for** **to** **do**

**else**

**for** **to** **do**

**for** **to** **do**

It’s easy to see that the worst-case complexity is of class because of the nested for loop in the **else** clause.

The exact efficiency function depends on how often happens.

Let be the probability of happening.

Then, the probability of not happening is , where :

* , we say never happens
* , we say always happens

The parameter characterising the input is the natural number .

The basic operation is , assuming a computation time of .

We need to setup a summation:

* For the first for loop, happens, , where is added to itself for as how many times the for loop runs. This depends on a going from to . The compact way to say this in mathematical notation is:
* To factor in the fact that this only happens if an event happens, we need to use the probability of happening (which is ), by multipliying with the above complexity:
* Do the same for the **else** clause, knowing that the probability of **not** happening is .

Therefore, the efficiency function is:

Since is a constant, and is the complexity of the basic operation, the worst-case time complexity is of class , because is the highest power here.

In a game, chance of winning $100, and chance of losing $50. If we play for a long time, we are earning an average of:

**Question 3**: Big-O analysis of algorithms

a) Algorithm to check if an array is sorted:

**ALGORITHM** *IsSorted*()

**if**

**return** true

**else**

**for** **to** **do**

**if**

**return** false

**return** true

The parameter characterising the input is the number of elements in the given array , which is called .

The basic operation is .

Argue that the worst case only happens when the array **is sorted**, and that there are more than 1 element (because if there is only one element, the efficiency function would be )

For , and the array is sorted, the full for-loop will be computed, and the complexity is:

Therefore, the time complexity is , since the highest power multiplying is .

b) Algorithm to find the minimal separation between any two elements in an array

**ALGORITHM** *MinDistance*()

**for** **to** **do**

**for** **to** **do**

**return**

Parameter characterising the size of the input is the number of elements in the given array , which is called .

Basic operation:

Since there is no branching, there is only one scenario, and it is the worst case:

Firstly, find the number of times the inner for loop occurs:

( added to itself for times)

Substitute back to the :

Now we need to work out the summation with :

Note: If we know an arithmetic series, we will find

However, we can also work out the summation ourself:

Add the above together:

Since is added to itself times:

Divide both sides by 2:

Substitute back into :

From here, we can see that the highest power multiplying is , therefore, the time complexity is .

c) Algorithm to reverse an array.

**ALGORITHM** ReverseArray()

**for** **to** **do**:

The parameter characterising the input is the number of items in the given array , which we call .

The basic operations are:

Or, we could also say, the basic operation is the swap, with a composition of the above basic operation.

We know that

This means:

Therefore, the worst case is when .

Since the highest power multiplying , and is , the time complexity is of class .

**Question 5:**

namespace Empirical

{

internal class Program

{

static void Main(string[] args)

{

Console.WriteLine("Size\t|\tCount");

for (int n = 1000; n <= 20000; n += 1000)

{

int count = RunAlgorithm(n);

Console.WriteLine($"{n}\t|\t{count}");

}

}

/// <summary>

/// Run the algorithm with a random array of size

/// n.

/// </summary>

/// <param name="n">The number of elements in the array</param>

/// <returns>The number of key comparisons</returns>

static int RunAlgorithm(int n)

{

int[] arr = new int[n];

Random rnd = new Random();

for (int i = 0; i < n - 1; i++)

{

arr[i] = rnd.Next(int.MinValue, int.MaxValue);

}

return SortAnalysis(arr);

}

static int SortAnalysis(int[] arr)

{

int n = arr.Length;

int count = 0;

for (int i = 1; i <= n - 1; i++)

{

int v = arr[i];

int j = i - 1;

// Insert if statement to increase counter by one

// even if j < 0

if (j < 0) count = count + 1;

while (j >= 0 && arr[j] > v)

{

count = count + 1;

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = v;

}

return count;

}

}

}

Size | Count

1000 | 248721

2000 | 996045

3000 | 2237172

4000 | 4044519

5000 | 6162658

6000 | 8971443

7000 | 12227552

8000 | 16068650

9000 | 20190535

10000 | 25033650

11000 | 30334758

12000 | 36230987

13000 | 42083148

14000 | 48866291

15000 | 56298616

16000 | 64095426

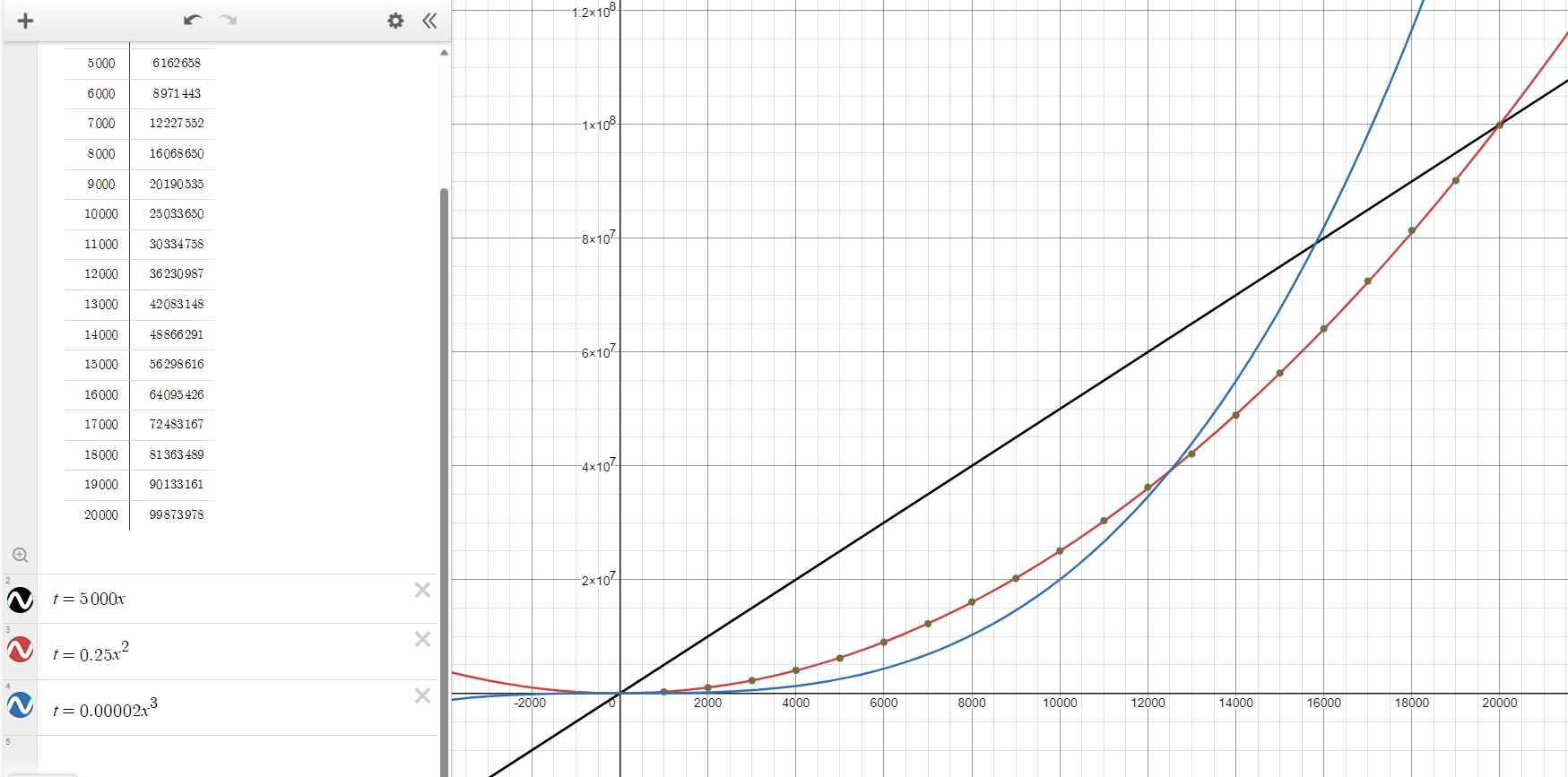
17000 | 72483167

18000 | 81363489

19000 | 90133161

20000 | 99873978

Plot the size of the array on the axis, and the count of key comparisons on the axis. Here, we observe that the line fits the best, so the time complexity is of class .



If we already assume that , we can find by doing:

If we don’t have a hypothesis to start with, i.e., unknown , can observe how grows as grows by:

Find and , and divide . Since as doubles, the number of operations quadruples, which could imply a quadratic time efficiency function.

Try a few more to be sure, like .