Datastructures and Algorithms GD

Big-O-Highscores

HVA

16 september 2017

Sjors Gielen (500765899) & Marc Griepspoor (500710937)

Datastructures and Algorithms GD

Big-O-Highscores

Before any assignment was started Sjors made a simple test runner to keep track of tests in a consistent manner. The test runner looks like this:

```
namespace Datastruct_and_algo_excersizes
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WindowWidth = (int)(Console.LargestWindowWidth /1.5);
            //setup a timer
            Stopwatch timer = new Stopwatch();
            Stopwatch testTimer = new Stopwatch();
            //setup excersizes to run.
            Dictionary<Excersize, int> excersizes = new Dictionary<Excersize, int>();
            excersizes.Add(new Excersize_1(), 5000);//make excersize 1 and I want to run it 5000 times
            List<string> testResults = new List<string>();

            foreach(KeyValuePair<Excersize, int> entry in excersizes)
            {
                int[] results = new int[entry.Value];
                double[] times = new double[entry.Value];
                testTimer.Restart();
                for(int i = 0; i < entry.Value; i++)
                {
                    timer.Restart();//make sure the timer is ready to rock
                    var result = entry.Key.run();//run the excersize
                    timer.Stop();
                    times[i] = timer.Elapsed.TotalMilliseconds;
                    results[i] = result;
                    //comment and un-comment this line if you want to keep track of test progress(recommended to have the line on if debugging or running tests that take a large amount of time)
                    Console.WriteLine("{0,-60} has completed in {1,10:0.0000} milliseconds Result was {2,5} this was run {3,6}", entry.Key, times[i], result, (i + 1));
                }
                testTimer.Stop();
                string testResult = string.Format("{0,-60} All {4,8} tests completed " +
                    "\r\nAverage time = {1,30:0.000000000000000} milliseconds, full test time: {2,30:0.000000000000000} milliseconds(includes upper for loop for tests)" +
                    "\r\nAverage result = {3,30:0.000000}\r\n", entry.Key, CalcAvg(times), testTimer.Elapsed.TotalMilliseconds, CalcAvg(results), entry.Value);
                testResults.Add(testResult);
            }
            //echo out each tests results
            foreach(string s in testResults)
            {
                Console.WriteLine(s);
            }

            Console.WriteLine("press enter key to close");
            Console.ReadLine();
        }
```

The test runner uses a dictionary to easily add in the excerisze classes along with an integer of how many times the test suite is meant to run the code.
An example of the test suite running Excersize_1 5000 times 3 times would look like this in the console:

```
Datastruct_and_algo_excersizes.Excersize_1                   All     5000 tests completed
Average time =              0,021754520000000 milliseconds, full test time:           109,791200000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,593400

Datastruct_and_algo_excersizes.Excersize_1                   All     5000 tests completed
Average time =              0,021594599999999 milliseconds, full test time:           108,838200000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,435400

Datastruct_and_algo_excersizes.Excersize_1                   All     5000 tests completed
Average time =              0,023314819999999 milliseconds, full test time:           117,503400000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,253000

press enter key to close
```

Question 1:

Write a program that generates an array with 100 random numbers (high scores between 0 and 10000) and then checks if the numbers are unique - output is yes or no:

For question one we simply made an array of size 100 and used a for loop to iterate through each.

With the data setup we made a number of methods. First we did the straight forward solution of having a double for loop run through it. But we ended up creating a hashmap buffer to

which we would attempt to add new indexes. Once an index was impossible to be added we would have our result. This way the algorithm is O(f(n)), Ω(f(1)).

```
32  public static class HasDupe
33  {
34      public static int HasDuplicate(this IEnumerable<int> source, bool counting = false)
35      {
36          int count = 0;
37          if (source == null)
38          {
39              throw new ArgumentNullException(nameof(source));
40          }
41
42          var checkBuffer = new HashSet<int>();
43          foreach (int item in source)
44          {
45              if (!checkBuffer.Add(item))
46              {
47                  //Console.WriteLine("could not add" + item);
48                  if (counting)
49                  {
50                      count++;
51                  }
52                  else
53                  {
54                      return 1;
55                  }
56              }
57          }
58
59          return count;
60      }
61  }
```

Question 2:

Execute the program above 5000 times, and report how many times the array contains at least one duplicate. What is the probability that a high score list contains at least one duplicate value?

Mathematically we can define the odds of a number existing twice in the array by doing the following math:

$$chance = 1 - \frac{PossbileNumbers!}{(possibleNumbers)^{indexes} * (possibleNumbers - indexes)!}$$

Where ! is the factorial operation.

Which in this case would be

$$x = 1 - \frac{10000!}{(10000)^{100} * (10000 - 100)!}$$

Resulting in x to approximately be equal to 0.3914.

This value is accurately represented within our test suite. we even decided to run it for 5 million times to get more significance.

```
Datastruct_and_algo_excersizes.Excersize_1                  All     5000 tests completed
Average time =              0,010312120000000 milliseconds, full test time:            52,555000000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,401200

Datastruct_and_algo_excersizes.Excersize_1                  All     5000 tests completed
Average time =              0,008971500000000 milliseconds, full test time:            45,917900000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,400000

Datastruct_and_algo_excersizes.Excersize_1                  All 50000000 tests completed
Average time =              0,008070610313354 milliseconds, full test time:        412870,447700000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,390553

Datastruct_and_algo_excersizes.Excersize_1                  All     5000 tests completed
Average time =              0,008030960000000 milliseconds, full test time:            41,067000000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,390400

Datastruct_and_algo_excersizes.Excersize_1                  All     5000 tests completed
Average time =              0,007845940000000 milliseconds, full test time:            40,105600000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,386600

press enter key to close
```

Question 3

Write a program that initializes an array of length 10000 with all values at 0. Then, add +1 to a random element, and repeat 100 times. Now check if the array contains a value greater than 1 (output is a boolean).

For this we created a new file that we would run through our test suite. The new file is as follows:

```
10  class Excersize_1_Q3 : Excersize
11  {
12      private static int entities = 100;
13      public override int run()
14      {
15          int[] data = new int[10000];//must be declared in the run method otherwise we get cross contamination
16          Random random = new Random();
17          for(int i = 0; i<entities; i++)
18          {
19              data[random.Next(data.Length)]++;
20          }
21          return HasMore.HasTwoOrMore(data);
22      }
23  }
24
25  public static class HasMore
26  {
27      public static int HasTwoOrMore(this IEnumerable<int> source, bool counting = false)
28      {
29          int count = 0;
30          if (source == null)
31          {
32              throw new ArgumentNullException(nameof(source));
33          }
34          var orderedBuffer = source.OrderByDescending(i => i);
35
36          foreach (int item in orderedBuffer)
37          {
38              if (item > 1)
39              {
40                  if (counting)
41                  {
42                      count++;
43                  }
44                  else
45                  {
46                      return 1;
47                  }
48              }
49              else if (!counting)//when we are not counting the first index tells is weither or not there are duplicates
50              {
51                  return 0;
52              }
53          }
54          return count;
55      }
56  }
```

As the orderby from LINQ uses stable quicksort the big-O we are dealing with here is O(f(n log(n))).

Question 4

Execute the program above 5000 times, and report how often at least one duplicate value is found. Again, what is the probability that a high score list contains at least one duplicate value?

```
Datastruct_and_algo_excersizes.Excersize_1_Q3               All     5000 tests completed
Average time =              0,985666779999996 milliseconds, full test time:          4931,143100000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,435200

Datastruct_and_algo_excersizes.Excersize_1_Q3               All     5000 tests completed
Average time =              0,953149839999999 milliseconds, full test time:          4767,802700000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,411200

Datastruct_and_algo_excersizes.Excersize_1_Q3               All     5000 tests completed
Average time =              0,953563120000001 milliseconds, full test time:          4769,846400000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,398600

Datastruct_and_algo_excersizes.Excersize_1_Q3               All     5000 tests completed
Average time =              0,952668999999994 milliseconds, full test time:          4765,407500000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,429800

Datastruct_and_algo_excersizes.Excersize_1_Q3               All     5000 tests completed
Average time =              0,958087619999997 milliseconds, full test time:          4792,464600000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,365600

press enter key to close
```

The probabilities are found on the average result section of the reports.

Question 5

Are the probabilities found under 2) and 4) equal? Explain why (not).

In our testing suite we do find the same probability, or at least a close enough estimate over 5 tests.

```
C:\Users\sjors\Desktop\HvA\Jaar 2\Year2 repo\Datastruct and algo excersizes\Datastruct and algo excersizes\bin\Debug\Datastruct and algo excersizes.exe
Datastruct_and_algo_excersizes.Excersize_1_Q3               All     5000 tests completed
Average time =              0,984631839999999 milliseconds, full test time:          4925,770700000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,352000

Datastruct_and_algo_excersizes.Excersize_1_Q3               All     5000 tests completed
Average time =              0,955269800000003 milliseconds, full test time:          4778,381200000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,438400

Datastruct_and_algo_excersizes.Excersize_1_Q3               All     5000 tests completed
Average time =              0,952812300000001 milliseconds, full test time:          4766,029400000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,373800

Datastruct_and_algo_excersizes.Excersize_1_Q3               All     5000 tests completed
Average time =              0,955386399999999 milliseconds, full test time:          4779,048100000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,379200

Datastruct_and_algo_excersizes.Excersize_1_Q3               All     5000 tests completed
Average time =              0,965166320000003 milliseconds, full test time:          4827,974700000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,408800

Datastruct_and_algo_excersizes.Excersize_1                  All     5000 tests completed
Average time =              0,007905780000000 milliseconds, full test time:            40,405500000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,390600

Datastruct_and_algo_excersizes.Excersize_1                  All     5000 tests completed
Average time =              0,007794560000000 milliseconds, full test time:            39,854900000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,384600

Datastruct_and_algo_excersizes.Excersize_1                  All     5000 tests completed
Average time =              0,007778400000000 milliseconds, full test time:            39,768300000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,388000

Datastruct_and_algo_excersizes.Excersize_1                  All     5000 tests completed
Average time =              0,009031000000000 milliseconds, full test time:            46,217400000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,397400

Datastruct_and_algo_excersizes.Excersize_1                  All     5000 tests completed
Average time =              0,008425140000000 milliseconds, full test time:            43,099100000000000 milliseconds(includes upper for loop for tests)
Average result =                      0,385600

press enter key to close
```

Another interesting note is the huge speed difference here. The original formula is about 100 times faster. This may be related to how the data is setup. Perhaps the test suite should have data be defined outside of the scope of the stopwatch?

More likely thought is our difference in big o notation. The sorting process simply takes up too much time currently.

Question 6 & 7

Consider the program you created to answer question 1 and 3. What is the time complexity (in terms of Big-O) of this program? Motivate your answer.

Time complexities were already handled on questions 1 and 3. For summary they were.
For the first solution O(f(n)).
This is the result of simply adding items directly to a hasmap which will be impossible if the hasmap already contains the value, via this we detect a duplicate value. This wat we would only have to iterate once through the entire dataset.
For the second solution O(f(n log(n))).
Due to the sorting process we require the slower O(f(n log(n))). Quicksort may be fast but it doesn't shine when it only gets a small amount of indexes. It reverts back to Selection sort far too fast(instead of binary sort). Keep in mind that n is assumed to be the arraysize here and not the playercount. Increasing the playercount should have no effect on the speed on this algorithm

Question 8 & 9

Taking n as the number of players, fill in the following table for your solution of 1 & 3. Make sure you take the average time over at least 1000 execution runs. Do these results confirm your analysis reported in question 6)?

To make testing this faster and more flexible we modified the scripts to make the variables in question be present in the constructors of the class. This would let us test several configurations in a swift manner. The testing suite received some small modifications to support this as well.

```
Datastruct_and_algo_excersizes.Excersize_1_Q3               All     1000 tests completed
Average time =              1,048942900000000 milliseconds, full test time:          1049,841700000000000 milliseconds(includes upper for loop for tests)
Average result =                    0,514000 variable n was:        100

Datastruct_and_algo_excersizes.Excersize_1_Q3               All     1000 tests completed
Average time =              0,987760000000000 milliseconds, full test time:           988,209700000000000 milliseconds(includes upper for loop for tests)
Average result =                    0,867000 variable n was:        200

Datastruct_and_algo_excersizes.Excersize_1_Q3               All     1000 tests completed
Average time =              1,042101800000000 milliseconds, full test time:          1042,571300000000000 milliseconds(includes upper for loop for tests)
Average result =                    1,000000 variable n was:        400

Datastruct_and_algo_excersizes.Excersize_1_Q3               All     1000 tests completed
Average time =              1,059864100000000 milliseconds, full test time:          1060,307600000000000 milliseconds(includes upper for loop for tests)
Average result =                    1,000000 variable n was:        800

Datastruct_and_algo_excersizes.Excersize_1                  All     1000 tests completed
Average time =              0,008382900000000 milliseconds, full test time:             8,575200000000000 milliseconds(includes upper for loop for tests)
Average result =                    0,389000 variable n was:        100

Datastruct_and_algo_excersizes.Excersize_1                  All     1000 tests completed
Average time =              0,010676100000000 milliseconds, full test time:            10,879400000000000 milliseconds(includes upper for loop for tests)
Average result =                    0,855000 variable n was:        200

Datastruct_and_algo_excersizes.Excersize_1                  All     1000 tests completed
Average time =              0,013450600000000 milliseconds, full test time:            13,639500000000000 milliseconds(includes upper for loop for tests)
Average result =                    1,000000 variable n was:        400

Datastruct_and_algo_excersizes.Excersize_1                  All     1000 tests completed
Average time =              0,018650900000000 milliseconds, full test time:            18,851700000000000 milliseconds(includes upper for loop for tests)
Average result =                    1,000000 variable n was:        800

press enter key to close
```

As expected the Q3 solution does not change in terms of time as the respective n value for it was not altered. More players just means that the chance it returns true increases.

For the Q1 solution though we can see that the time slowly increases. Again this is along with what we would expect. A larger list means the chance the first duplicate is found increases slower.

Amendment

After writing the main points through this document we realized that it would currently be far more efficient to remove the sorting from the Q3 solution. In doing so we reduce the big O down to n! – (n-size)! instead of nlog(n). As the probability to find a 2 on the first index goes up with more players.

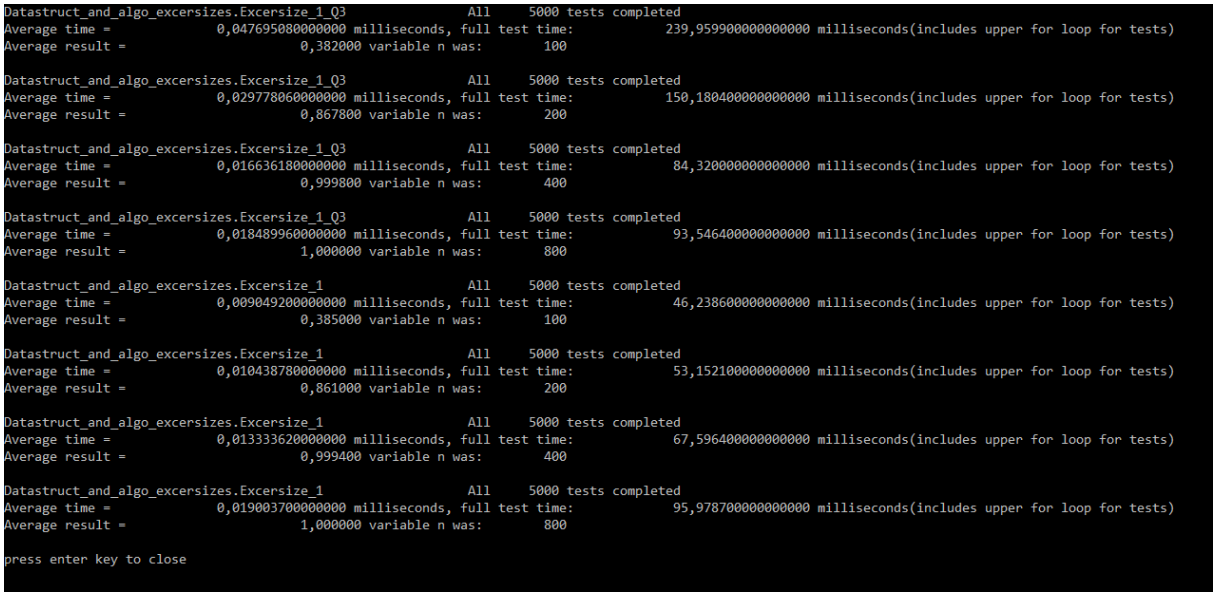
This means the code has moved to this:

```
32      public static int HasTwoOrMore(this IEnumerable<int> source, bool counting = false)
33      {
34          int count = 0;
35          if (source == null)
36          {
37              throw new ArgumentNullException(nameof(source));
38          }
39          //var orderedBuffer = source.OrderByDescending(i => i);
40          //lets not order as we then don't have to go through O=nlog(n)
41          //but instead O=n
42
43          foreach (int item in source)
44          {
45              if (item > 1)
46              {
47                  if (counting)
48                  {
49                      count++;
50                  }
51                  else
52                  {
53                      //Console.WriteLine("found a 2 or higher : " + item);
54                      return 1;
55                  }
56              }
57          }
58          return count;
59      }
```

And the eight piece test now looks like:

Also please note that on some of the earlier tests we had the wrong random seed setting in that random parameter of Q3. This results in Q3 having too high probabilities sometimes as the seeds isn't properly being modulated.

This is achieved by doing: `Random random = new Random(Guid.NewGuid().GetHashCode());` instead of `Random random = new Random();`



The differences in time is now more in-line with what we expect from our big O notation. The main reason we see Q3 400 be faster than Q3 800 is that the data-setup takes much longer compared on Q3 compared to Q4. A thought is to make the dataset production be handled outside of the stopwatch timers to remove this part of the process from being timed.

Keep in mind that the data creation on both ends is big O = n, therefor we did not take them into account when comparing the actually methods which execute the mutation on the dataset.