Role of Algorithms

CMSC 330 Ungraded Homework Documentation

3/27/2016

University of Maryland, University College

Megan Tran

1. **Project Description and Algorithm Analysis.**

**Description and Approach**

1. This project has been coded in Java. My goal for this project was to create methods for two program fragments, run the methods 50 times for different values of N and give average running time and O(n) for each run.
2. As per assignment requirements, I created a Java class named Algorithms. In the Algorithms class, I have coded the two program fragments into methods called algorithmOne and algorithmTwo
3. To test the average running time and O(n) for the two algorithms, I created two additional methods called runAlgorithmOne and runAlgorithmTwo to run each algorithm a number of times according to the parameter passed in. An additional method named printAverages is called to print out the average elapsed time and average O(n) for each run
4. Test cases and run time tests are defined in the main method of the program. There are a total of three test cases defined for the test plan, as well as run time tests for both algorithms. They are clearly labeled with comments in the source code.

**O(N) Analysis**

1. For this first fragment, we can see that there are two for-loops. In the first for-loop, there will be a total of N iterations from the part where it states I<N. This brings the big-Oh for this algorithm to O(N).
2. For the second loop, the loop will iterate I number of times. This means that the total iterations for this for loop can go up to N number of times. This brings the big-Oh for this fragment to O(N2).
3. The second program fragment is more complex than the first fragment. The first for-loop with iterate a total of N amount of times, giving it O(N) iterations. In the second loop, there will be a total of N\*N iterations because the loop will iterate as long as J < I\*I. This brings the total O(N) to O(N2).
4. There is an If statement puts some restriction on when the third for-loop will run. The third loop will only run if J%I == 0. However, I believe the If statement will matter little when analyzing big-Oh for this algorithm because it is not a loop.
5. The third for-loop will be able to iterate N2 times because K < J, and as seen in the last for-loop, J can be as much as (I\*I), while I can only be as much as N. This brings the total O(N) for the algorithm to O(N2 \* N2) = O(N4).
6. **Test Plan**

In the main method of the class Algorithms, I have defined three tests cases. They are clearly labeled in the comments. When you run the driver method of the program, you should be able to see the results printed out. The test cases and results are highlighted below.

**Test Case 1:**

1. I declared an instance of Algorithms with a default no-arg constructor. I have to declare an instance to use the methods in the class because the methods I created are not static, and therefore need to be called by an instance of the class.
2. I am testing plugging in negative values for both algorithms 1 & 2. The output value for every call to these two algorithms is 0. For algorithmOne, I tested negative values -10 and -100, and for algorithmTwo I tested negative values -20 and -200.
3. The output of 0 is an expected result. A negative value for N would not make sense for these algorithms because the for-loops do not use any negative numbers. The for-loops starts are the int value of 0 and will not iterate for negative values. Furthermore, negative results are not desirable for the two algorithms.

**Test Case 2:**

1. I will be using the same instance of Algorithms used in test case 1. This test case will focus on the resulting output for the algorithmOne method. All values used to test algorithmOne are positive.
2. The output should display the sums that are created with algorithmOne with N = 10, 50, 100, 250, and 500. One sum is displayed on each line of the test case. The sums should be 45. 1225. 4950, 31125, and 124750, respectively.
3. The results for these sums make sense to me. This algorithm’s worst case big-Oh is O(n2). The sums do not have a linear relationship to the values that were input into the method (like if the big-Oh of the method were to be O(n)), and display more of an exponential relationship.

**Test Case 3:**

1. I will be using the same instance of Algorithms used in the previous test cases. This test case will focus on the resulting output for the algorithmTwo method, and is similar to the last test case. All values used to test algorithmTwo are positive.
2. The output should display the sums that are created with algorithmTwo with N = 10, 50, 100, 250, and 500. One sum is displayed on each line of the test case. The sums should be 870. 730100. 12087075, 481794250, and 7760510375, respectively.
3. As expected, the resulting sums are a lot larger than the sums found in the second test case when running algorithmOne. Since the big-Oh for this algorithm is O(n4), I knew the sums would be big. The sums for the plugged values 500 and 1000 are so big that an int variable cannot hold it – I had to use a return value of long.
4. **Run Time Tests**
5. Run time tests for both algorithms can be found after the test cases in the main method of the Algorithms class. For each algorithm, I averaged the elapsed time and O(n) for 50 runs. To do this, I used an instance of the Algorithms class and called runAlgorithmOne and AlgorithmTwo methods for different N values.
6. I have a total of 6 run time tests for Algorithm 1. I used the N values 10, 50, 100, 250, 500, and 1000. I ran the algorithm 50 times to get the average results displayed in the output. There is one average on each line. I do the same tests for Algorithm 2.
7. For the run time tests make sense with some anomalies. I saw that running algorithmOne 50 times with N = 250 is usually faster than when N = 100 by about half the time. However, the average O(n) for N = 250 is considerably bigger than for N = 100.
8. Run time tests for algorithmTwo take an obviously longer time to finish executing than for algorithmOne, as expected. The tests for algorithmOne are finished nearly in an instant compared to algorithmTwo. This makes sense because the second algorithm is more complex and as a big-Oh of O(n4).
9. The last run time test value (1000) for algorithmTwo takes especially long to finish. On my test run, it takes an average of 4824520250 milliseconds to complete one run. The average O(n) is a whopping O(124583708250).

Below I have included sample output for the run time tests. Average O(n) should be the same for each run, but the elapsed times varies.

Run Time Tests for Algorithm 1

----------------------

Average elapsed time for 50 runs with N = 10 is 250 ms

Average O(n) is O(45).

Average elapsed time for 50 runs with N = 50 is 11547 ms

Average O(n) is O(1225).

Average elapsed time for 50 runs with N = 100 is 5518 ms

Average O(n) is O(4950).

Average elapsed time for 50 runs with N = 250 is 2578 ms

Average O(n) is O(31125).

Average elapsed time for 50 runs with N = 500 is 11232 ms

Average O(n) is O(124750).

Average elapsed time for 50 runs with N = 1000 is 25943 ms

Average O(n) is O(499500).

----------------------

Run Time Tests for Algorithm 2

----------------------

Average elapsed time for 50 runs with N = 10 is 1252 ms

Average O(n) is O(870).

Average elapsed time for 50 runs with N = 50 is 220902 ms

Average O(n) is O(730100).

Average elapsed time for 50 runs with N = 100 is 1803234 ms

Average O(n) is O(12087075).

Average elapsed time for 50 runs with N = 250 is 34493184 ms

Average O(n) is O(481794250).

Average elapsed time for 50 runs with N = 500 is 377282001 ms

Average O(n) is O(7760510375).

Average elapsed time for 50 runs with N = 1000 is 4824520250 ms

Average O(n) is O(124583708250).

----------------------

1. **Lessons learned and Possible Improvements**

**Lessons Learned**

1. One of the main problems I was encountering was that during run time tests (especially for algorithmTwo), the average O(n) would result in a negative number. The negative numbers baffled me and I had no idea why it happened. I searched the internet and asked around for some answers. A classmate illuminated me in saying that it may be a problem of my numbers being too big for an int variable, and that I may want to switch to long.
2. This solution proved to be my answer. In my running algorithm methods, I changed the variables that were holding the average O(n) value and total O(n) for all the runs from int to long. My numbers were getting too big for int! However, the problem arose again when I tested O(n) for bigger and bigger values of N, the problem came up again!
3. Originally, the return value for algorithmOne and algorithmTwo was int. I changed the return value to long. It turns out that this solved all of the unwanted negative/strange outputs! The sums I was getting were just too big for ints. This was a great lesson learned for me. Long is 64 bits while ints are 32 bits. If I am expecting very large numbers, I should switch over to using long.
4. Analyzing O(N) for each algorithm made me practice analyzing algorithm complexity and efficiency. The program also gave me good experience in testing and averaging run time. Although I’ve conducted run time tests before, I have never used the technique for writing a method that will run the algorithm X number of times for you. This is a very helpful method to have when testing!

**Possible Improvements**

1. For this project, I have only tested for values of N that were 1000 or less. When analyzing O(N), I hear that it is best to for large values of N. As a possible future improvement, I would do tests for values such as 5,000 and 10,000.
2. I understand now that a possible problem that I may run in to is the sum values generated from the methods can be large, and some were too large for int. If I am to test with larger numbers, I may have to modify the program to use the BigInteger class to handle large numbers that a long type variable cannot hold.
3. The methods runAlgorithmOne and runAlgorithmTwo are similar methods to each other save for one line of code (which method to run). As a possible improvement, I would like to merge these two methods into one so that I can eliminate more duplicate code.

**References**

I used the following reference to help me with completing the assignment.

Elizes, R. (2016). Running time and determining O(N) for an algorithm. Retrieved from <https://learn.umuc.edu/d2l/lms/news/main.d2l?ou=126936>