Analysis document: BinaryMaxHeap

● Graded

Student

HARRY KIM

Total Points

34 / 35 pts

Question 1

Question 1

1 / 1 pt

✓ **– 0 pts** Correct

Question 2

Question 2

4 / 4 pts

✓ **– 0 pts** Correct

Question 3

Question 3

8 / 8 pts

✓ **– 0 pts** Correct

Question 4

Question 4

3 / 4 pts

✓ **– 1 pt** `add()` (worst case) complexity not correctly observed or expected

Question 5

Question 5

5 / 5 pts

✓ **– 0 pts** Correct

Question 6

Question 6

8 / 8 pts

✓ **– 0 pts** Correct

Question 7

Question 7

5 / 5 pts

✓ **– 0 pts** Correct

Question 8

Late Penalty

0 / 0 pts

✓ **– 0 pts** submitted before deadline or penalty applied to code submisison

Questions assigned to the following page: 1, 2, and 3

Harry Kim
U1226472
11/19/2020
CS 2420

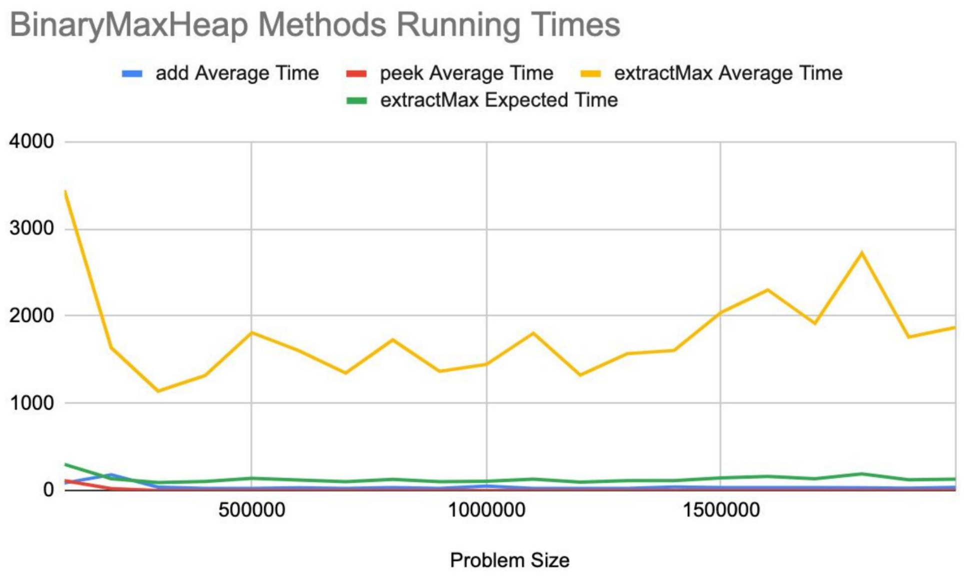
Analysis Document — Assignment 10

1. I did complete the programming portion of this assignment with a partner. My programming partner is Braden Morfin, and I am the one who submitted the assignment to Gradescope.

2. Our binary heap implementation stored the root item at index 1 in order to avoid the special case where finding the children of the root node by using the equations LeftChildIndex = ParentIndex * 2 and LeftChildIndex = ParentIndex * 2 would be problematic if the index of the root was at 0 and return the wrong index. By implementing the root stored at index 1, we can avoid this special case and keep the code simpler and faster.

3. In order to assess the efficiency of the basic priority queue methods (e.g., *add*, *peek*, *extractMax*), we first ran the test through a for loop which incremented the problem size to be used in each loop by sizes of 100,000 and went up to no greater than 2,000,000. We then created two heaps of type integer and populated each heap with a random number ranging from 0 to the problem size - 1. We then ran timing tests to create plots with 4 lines representing the average time of the *add*, *peek*, and *extractMax* methods. We only performed a check analysis on extractMax, it's expected run time being logarithmic, since it would be pointless to do the others (*add* and *peek*) as their expected runtimes are constant and so we would just be printing the same plots over each other. While timing the methods, we also made sure to make method calls loop 10000 times in order to reduce the noise of the plots.

Questions assigned to the following page: 3, 4, 5, and 6



According to this graph, the *add* and *peek* methods display very fast constant time while the extractMax is not clearly discernible as to what type of growth rate it displays. Because we used a check analysis on extractMax, by dividing the average time by the expected growth (logarithmic) and the expected time converges to a positive number, we can conclude that the *extractMax* method displays a logarithmic running time.

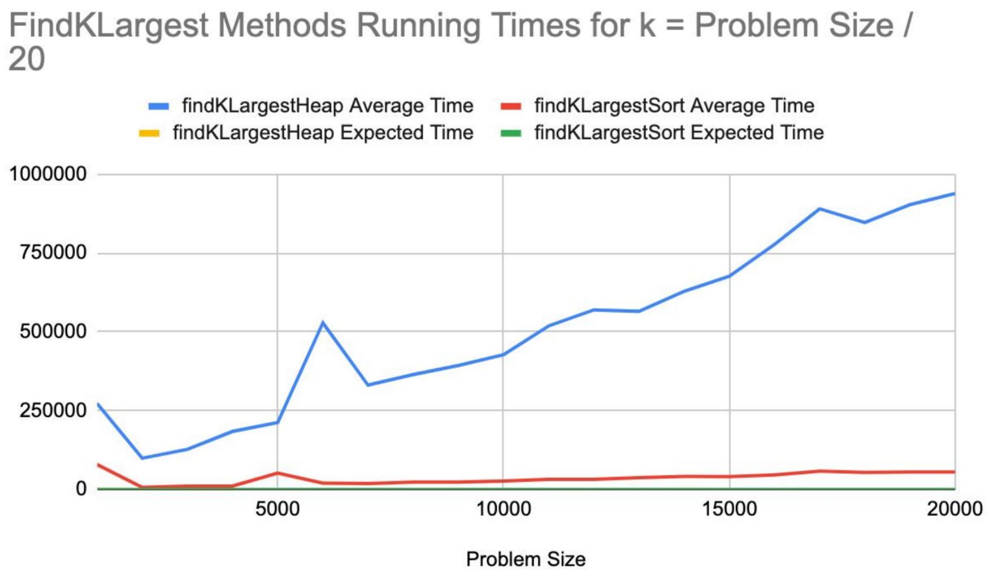
4. The running-time growth rates observed in our experiment matches the expected Big-O behaviors because the growth rate of add and peak are expected to be constant and the growth rate of extractMax is logarithmic, and the graph shows us that the growth rates display the expected growth rate.

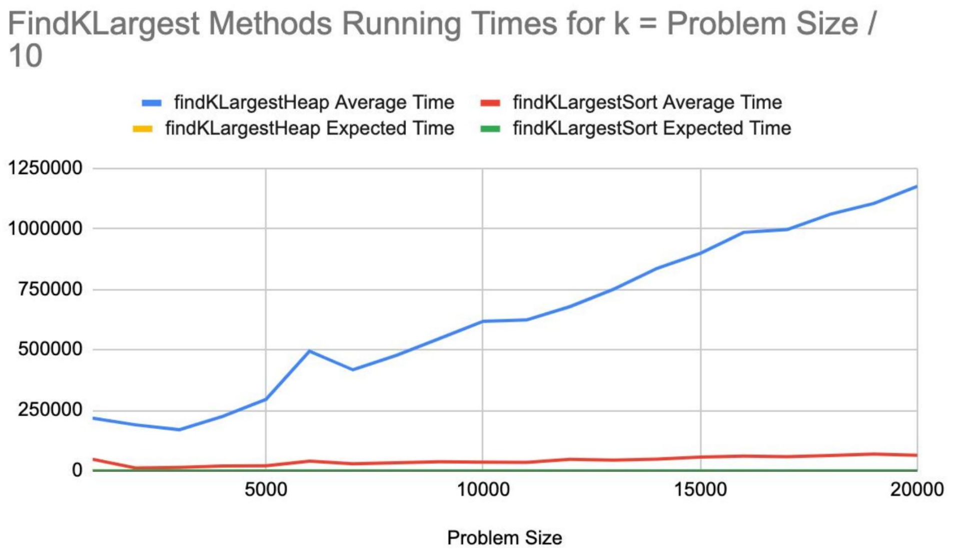
5. In our *findKLargestHeap* methods the expected behavior is O(N + k * logN). The variable k means how many of the largest items we would want to return (5, 4, and 3 from the list of numbers 5, 4, 3, 2, 1 if k = 3 for example). The variable N means the total number of items within the heap. When k is much smaller than N, it means that the numbers returned are all closer to 0 than the problem size and thus displays linear behavior. When k is close to N, it means that the numbers returned are close to all the numbers in the list and thus displays linearithmic behavior.

6. In order to assess and compare the efficiency of solving the find-k-largest problem using a binary heap to using Java's sort routine (*findKLargestHeap* vs. *findKLargestSort*), we first ran the test through a for loop which incremented the problem size to be used in each loop by sizes of 1,000 and went up to no greater than 20,000. We then created a list and populated it with integers from 0 to the problem size -1. We then put the list in as a parameter for the *findKLargestHeap*, which uses our *BinaryMaxHeap* class, and

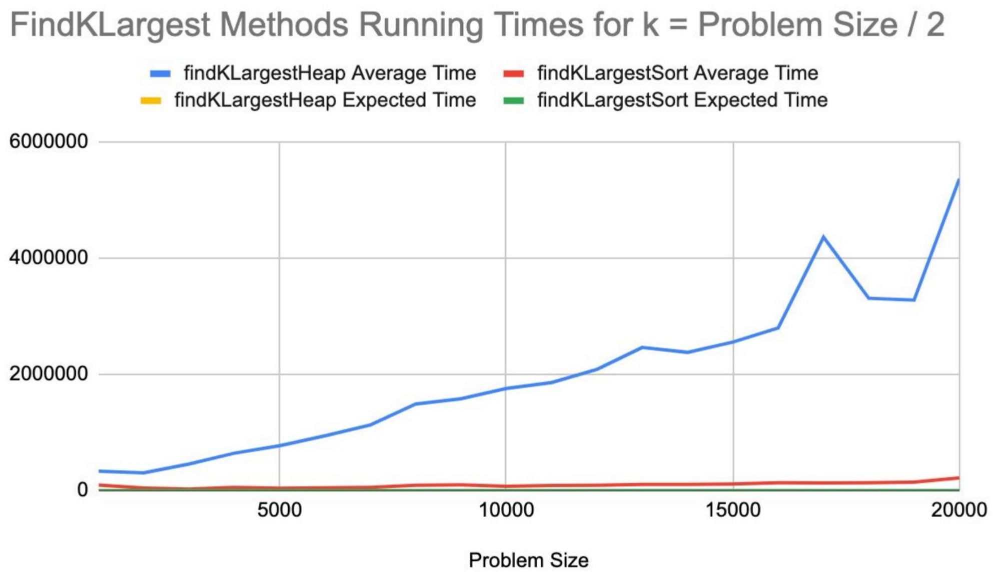
Question assigned to the following page: 6

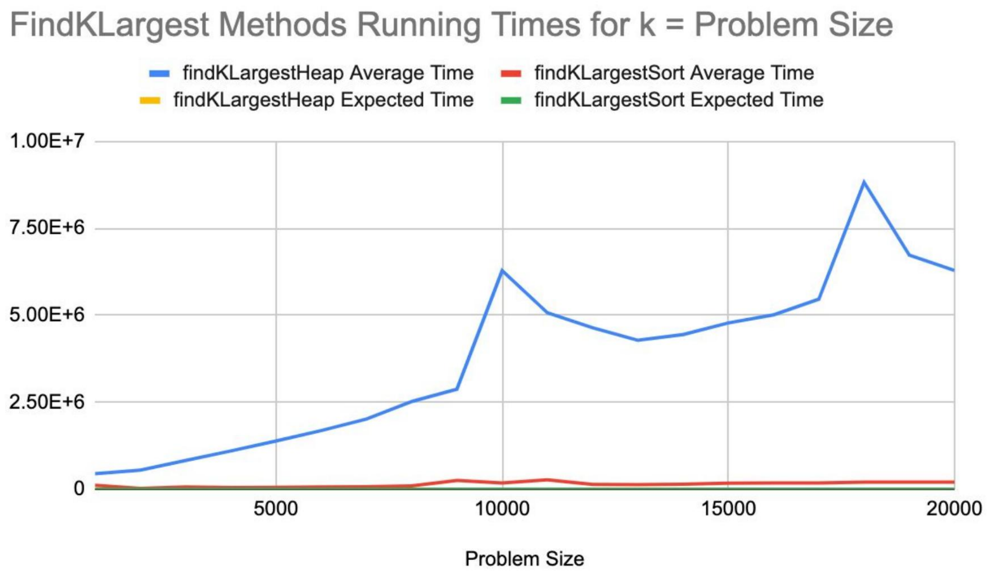
findKLargestSort, which uses Java's sort routine. We then ran timing tests to create plots with 4 lines representing the average time of both *findKLargestHeap* and *findKLargestSort* methods along a check analysis for both of them. While timing the methods, we also made sure to make method calls loop 500 times in order to reduce the noise of the plots. In order to test different k values, we created an int k variable which would equal the problem size divided by some number. We ran the test multiple times to get the running times for different values of k for both methods.





Question assigned to the following page: 6





From our run times of both methods, it would seem that *findKLargestSort*, which uses Java's sort routine, vastly outperformed our own method even in cases where k was smaller. Java's sorting must be faster by using some quicksort algorithm and simply returning the k largest values whereas our BuildHeap method itself within our BinaryMaxHeap itself has an expected runtime of O(NlogN) (the same as java's sorting) along with extracting the max item k times which adds on a complexity of O(logN) for each k.

7. The running-time growth rates observed in our experiment matches the expected Big-O behaviors because the expected runtime of our findLargestHeap is O(N + k * logN) while the expected runtime of findKLargestSort, which uses Java's sort routine, is O(NlogN). Through our runtimes along with our check analysis, the graph shows us that the growth rates display the expected growth rate.