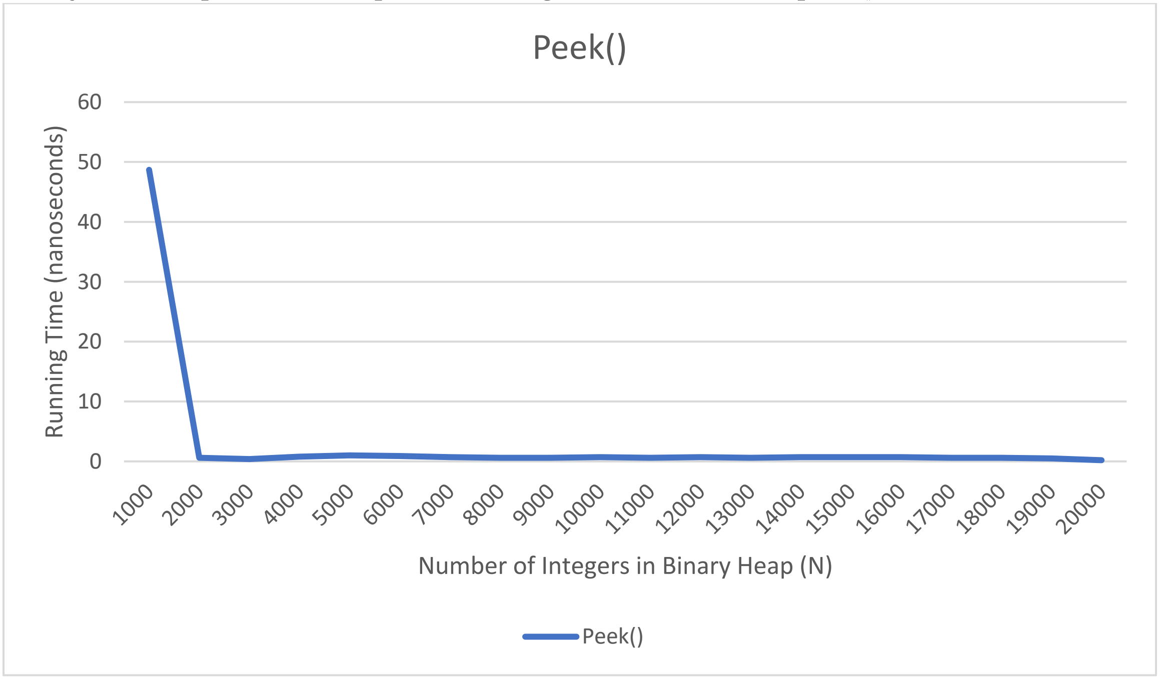
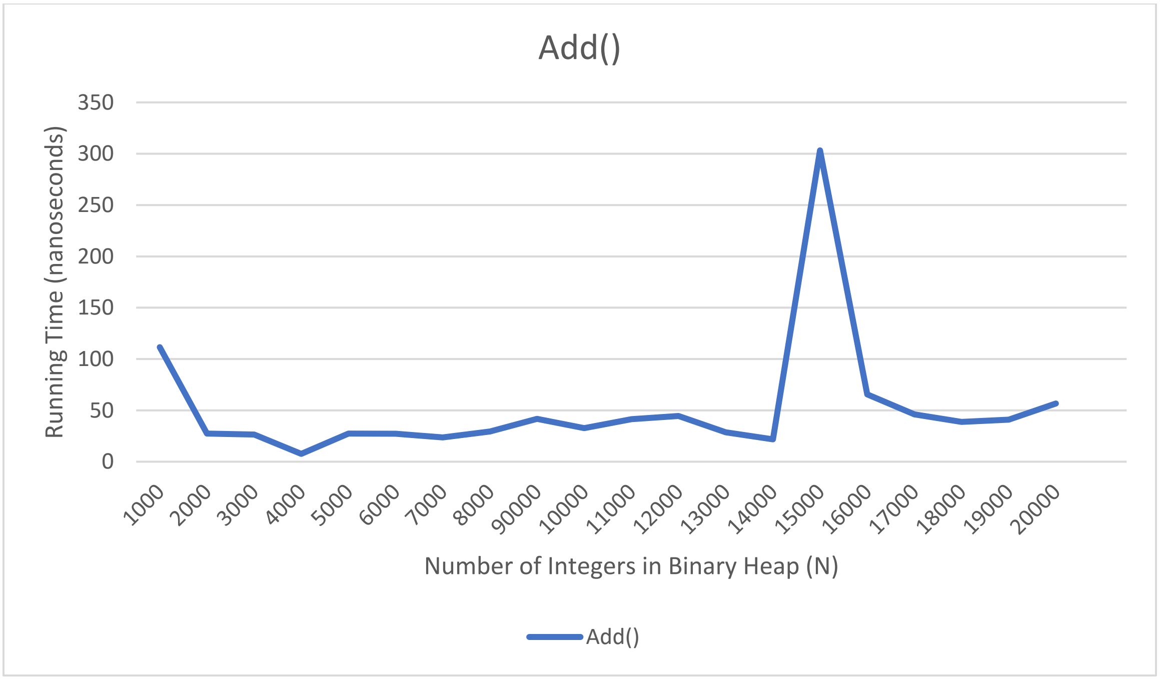
Assign 10 Analysis**:**

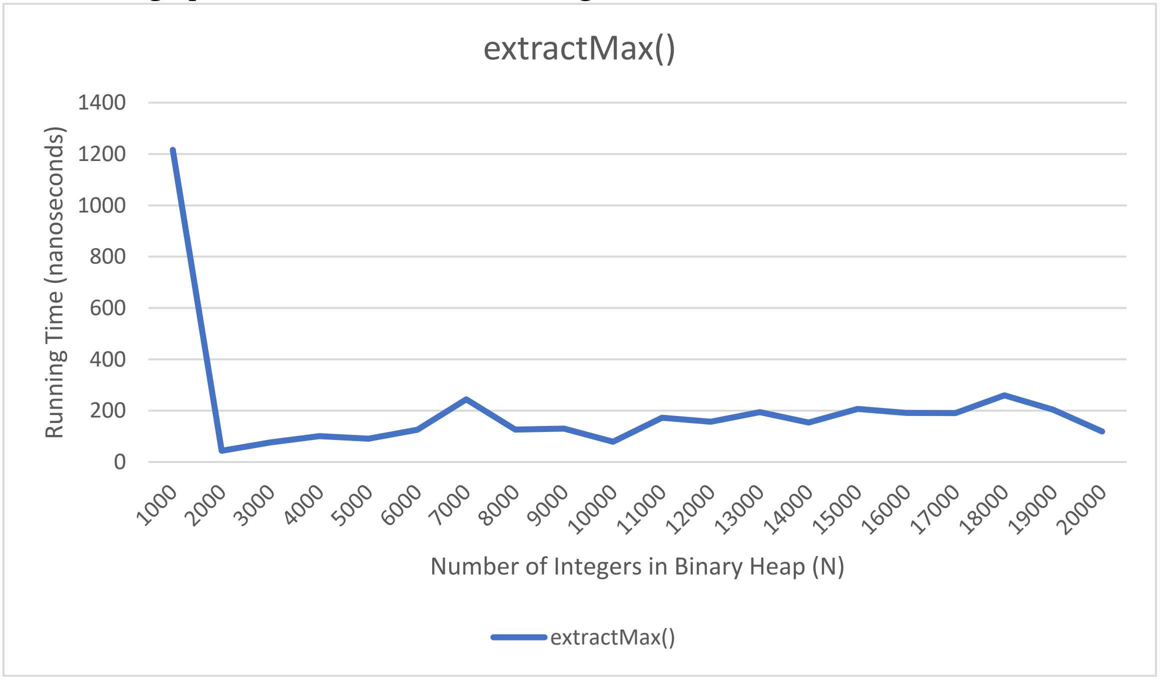
1. Peek() Timing: For this experiment, I began by creating a binary max heap and adding N random integers to it. After I created the heap, I started a timer and called the peek() function to time how long it took to return the maximum value in the heap. As is clear below, the peek function is constant and runs in O(1) time. The only discrepancy in the plot below is that the beginning time is a lot higher than the rest of the peek() times, but I believe this is simply due to the background processes needed to start the timer impacting the original time. However, in general, the graph shows that the number of items in the binary max heap does not impact how long it takes to call the peek() function.



Add: For this experiment, I set up a binary max heap with N integers contained in it. After I created the heap, I started a timer and added a random value to the heap, then called extractMax() to keep the problem size of the heap the same. I repeated this 1000 times to get the average, and subtracted the time it took to extract the maximum value after each call to add. The plot below shows that calling add() is typically constant. I believe the discrepancies at N=1000 and N = 15000 are due to background processes of the computer (as the wifi in my apartment has been struggling today) and are not an accurate reflection of the actual complexity of the add() function. Apart from those data points, the rest of the graph is pretty consistently constant. This shows that the number of items in the binary max heap does not impact how long it takes to add a new value to the heap. It also could be a reflection of the worst case of add(), which would run in O(logN) time if we had added a new maximum value and thus had to percolate all the way back up the heap after adding the value to the end.

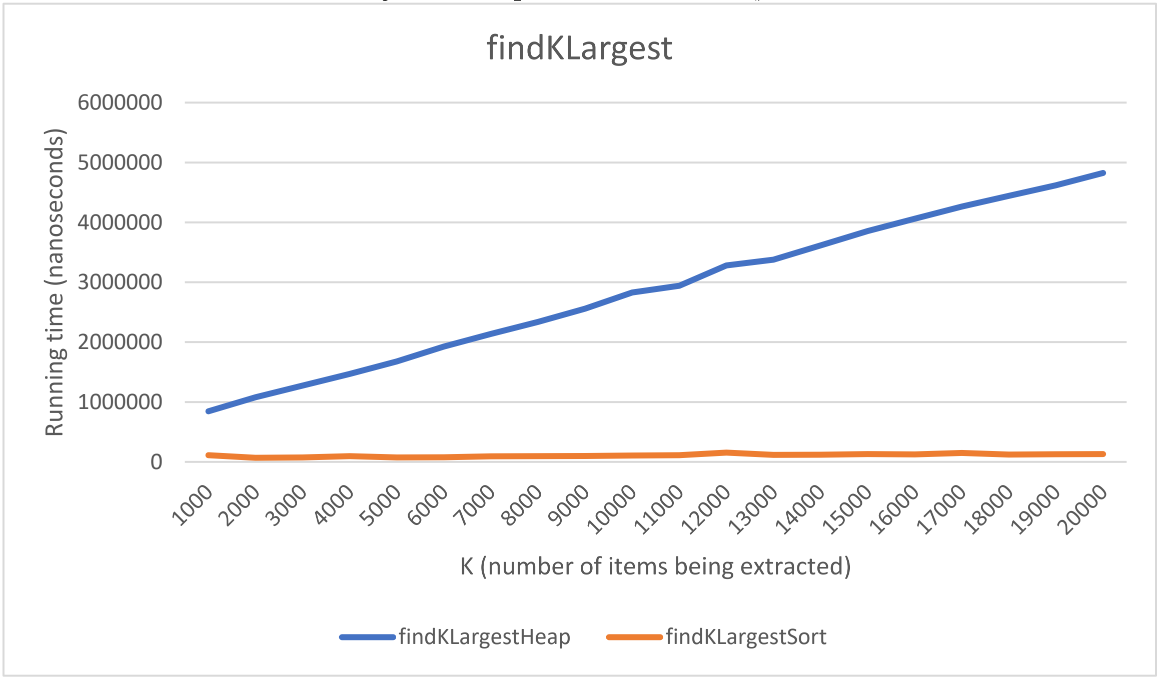


Extract Max: For this experiment, I once again created a binary max heap with N integers contained in it. After I created the heap, I then started a timer and called extractMax(), then added a random integer to ensure the problem size of the binary heap stayed the same. To calculate the actual time it took to run extractMax(), I subtracted the amount of time it took to add a random integer and took the average of the new value. The plot below is a little difficult to interpret, because there is a bit of variability in the slope of the graph. Additionally, the run times are so small that it is difficult to get a good reading of the overall graph, as the times are so close together.



2. The running time growth rates of peek(), add(), and extractMax() do reflect the expected big-O behavior of the methods. Peek() is supposed to run in constant time, as the method only needs to return the value contained at index one in the backing array. The graph shows a relatively flat line with 0 slope, which is typical of a constant running time. Additionally, the actual calculations between the expected and the actual run time have a roughly consistent ratio when using constant time to calculate it, so that also shows that it is constant. Add(), in the average case, is supposed to run in constant time as well. I feel that the graph does reflect this, as it is a mostly constant line with zero slope. However, there is obviously a few points where the graph spikes up because it took longer to add a value to the graph. I believe this is partially due to computer process discrepancies as well as the possibility that this is a reflection of add's worst case behavior, which is O(logN) which occurs when we add a new maximum value to the binary heap and we must percolate all the way up through the heap. The actual/expected calculations do have roughly the same proportion as well when I calculate it as constant time. Even though the proportion does have a relatively large range from 25-75, I believe this is only because it is very difficult to get an accurate timing of such fast processes, and this proportion change is just due to that issue. The spike in the graph did have a different calculation, however. It had a ratio within the expected range when I calculated it with O(logN) time, which shows that it actually was the worst-case version of add(), and I must have added a new maximum value to the heap. The extractMax() graph actually does not reflect the expected complexity, O(logN). It seems to be an O(NlogN) graph, which is difficult to see from the graph, but when I calculated the actual/expected, the ratio was constant when the expected was NlogN, not logN. I believe this is due to an inefficiency in our percolateDown() method, as my partner and I did have some difficulty getting it to work correctly while keeping it efficient. Our extratMax() method is very simple apart from the percolateDown() section, which is the only section that actually has an impact on the running time so the issue must lie in that method.
3. K is the number of items we want to extract from the heap, while N is the actual number of total items in the heap itself. The overall behavior when k is much smaller than N is for smaller values of N, the overall complexity is closer to O(N) as opposed to (N+ logN). However, for larger values of k where it is close to N, the complexity will end up being closer to O(NlogN), and is much slower than when the values of k are smaller than N.
4. For this experiment, I created an ArrayList with 25000 integers listed in it. I passed that arrayList into the method, and passed in k from values from 1000 to 20000 incrementing by 1000 each time. This way, the list stayed the same size, but I tested for different values of k as they got closer to N, the length of the list. I started a timer, and then called the findKLargestHeap() function, and then repeated the experiment over for the findKLargestSort() function. From the graph below, it is clear that Java's Sort routine is much more efficient than using our binary heap for this problem. The run time for Java's sort method seems to be constant, because instead of creating an entire binary tree it just sorts the arrayList in place then takes the first k values in that list, while our method must

take the time to create a binary max heap then extractMax() k times.



5. Neither of the growth rates seem to match the expected Big-O behaviors. The graph above makes it seem like findKLargestHeap is linear, while the findKLargestSort is constant. However, when I calculated the actual/expected, none of the values stayed consistent for the above complexities. They also did not stay consistent when I used the expected Big-O behavior. I tried a few different complexities to see if I could figure out the actual complexity, but none of the basic complexities created a roughly consistent proportion. I believe that the findKLargestHeap has a different complexity because the complexity of our extractMax() function is not correct, as analyzed above. Our findKLargestHeap method relies on creating a binary max heap from the given list, then calling extractMax() k times to get the k largest values. Thus, because the complexity for extractMax() is not as expected, it also impacted findKLargestHeap's complexity to be not as expected. I believe that findKLargestSort complexity difference is because we chose to sort the parameter list, and then use a for loop to loop through the list k times starting from the end to get k max items, instead of just sorting items and creating a sublist from the end of items to the value at list.length() – k. This would have been a more efficient way to execute this problem.