COMPSCI 2XB3 L04 Lab Report #5

Mahad Aziz - azizm17 - 400250379 - azizm17@mcmaster.ca
Talha Amjad - amjadt1 - 400203592 - amjadt1@mcmaster.ca
Logan Brown - brownl33 - 400263889 - brownl33@mcmaster.ca
McMaster University

COMPSCI/SFWRENG 2XB3: Binding Theory To Practice

Dr. Vincent Maccio

TA: Amir Afzali

February 26, 2021

Building Heaps:

Predictions:

Bottom-up:

We predicted that bottom up will have O(nlog(n)) complexity. Looking at the implementation, we can see that the loop will always run the length of the list integer divided by 2 times minus 1 times which makes sense since the sink() function does not need to be called on the leaves of the heap. We know that sink() has a worst case complexity of O(logn) and since we run sink() n/2 times we can discard the coefficient 1/2 and end up with an overall time complexity of O(nlog(n)). In comparison to the other variations of heapsort, we believe that this implementation will have the fastest runtime since the for loop runs n/2 times.

Brick-by-Brick:

We predicted that the brick-by-brick heapsort variation should also have an O(n log(n)) complexity. In the brick-by-brick variation of heapsort, we utilized a for-loop that iterates through every element in the unsorted list which gives us a O(n) runtime. Then we used an empty heap and inserted each element of the unsorted list into the heap. We knew previously that the insert() function of heapsort has a worst case O(log (n)) time complexity so we predicted the resulting time complexity for the brick-by-brick variation of heapsort will be O(n log(n)). In comparison to the other variations of heapsort, we believe that this implementation will have the second best runtime since the loop runs n times.

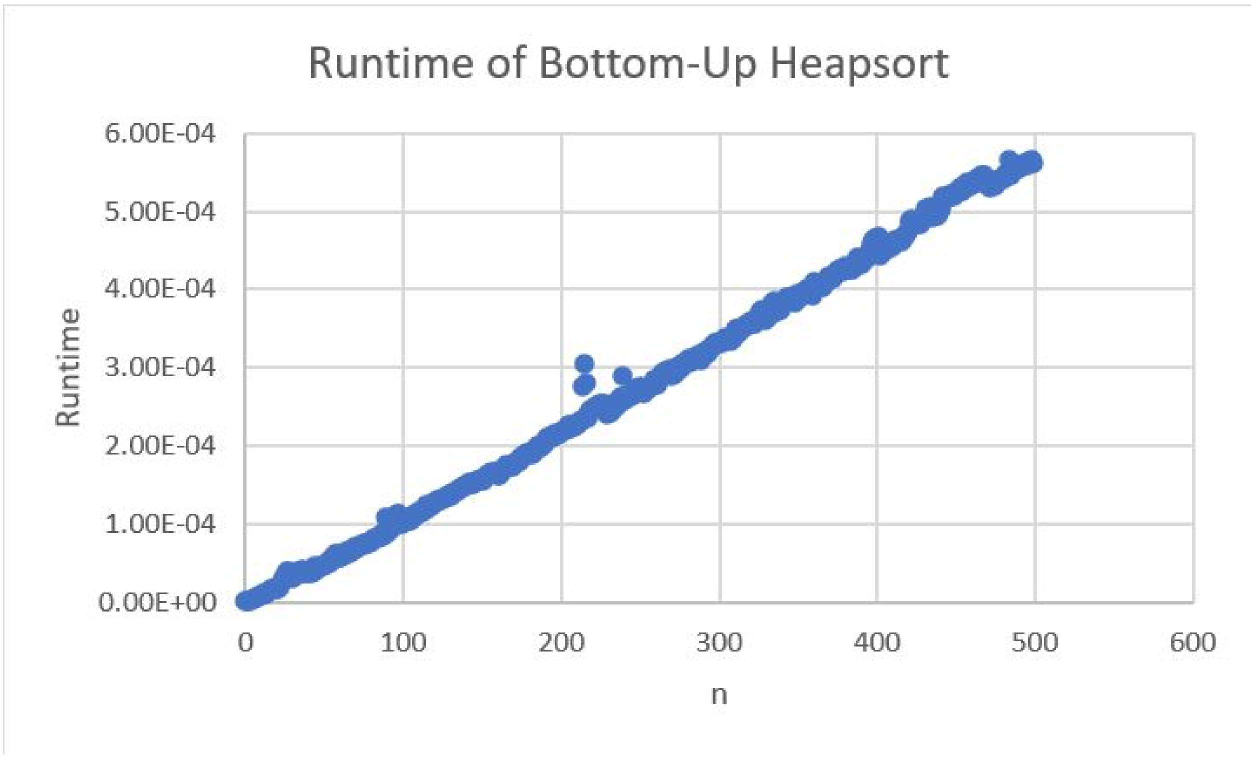
Sink top-down:

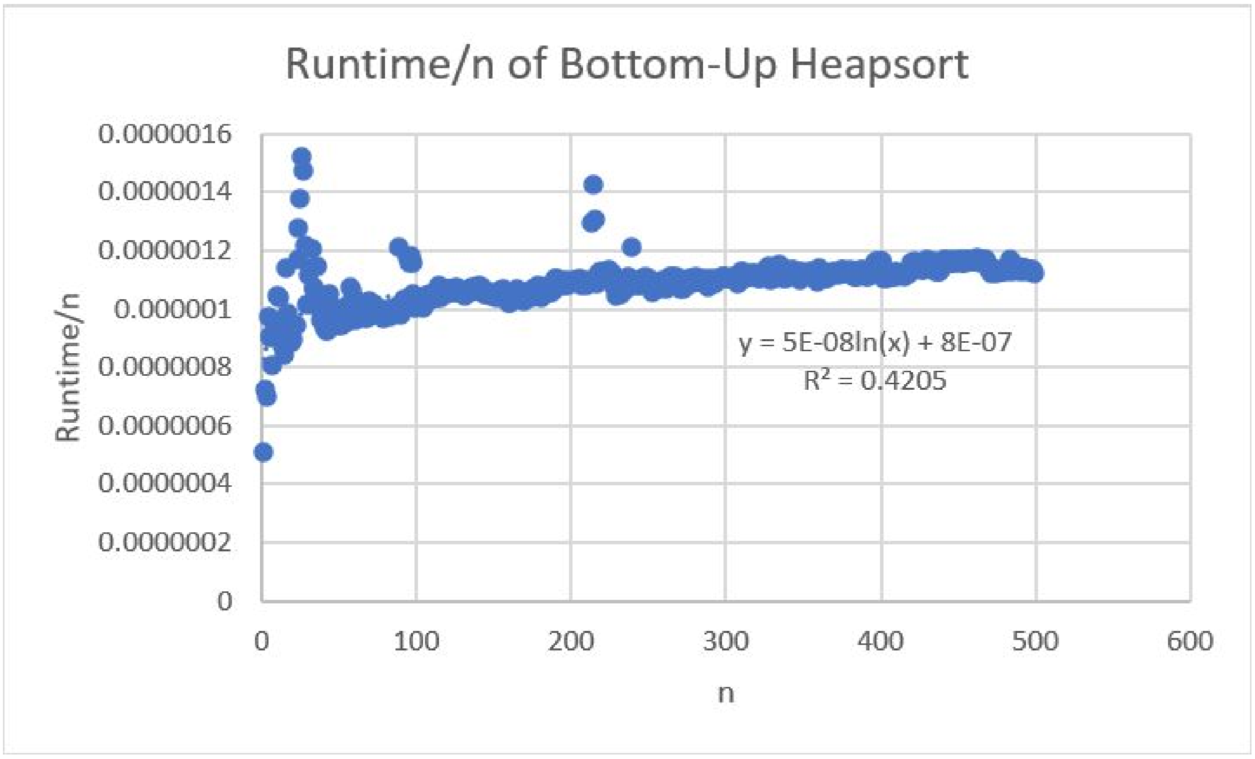
We predicted that the top-down implementation for heapsort will be O(nlog(n)) complexity as well. The top-down implementation uses the sink() method that has log(n) complexity. Furthermore, the top-down heapsort implementation utilizes a for loop that goes through each element in the list that is divided in half and then uses the sink() method to sort it into the heap. We then have a while loop that repeats this process until we get a heap. We also check if the data is in a heap structure which we implemented recursively with complexity O(nlog(n)). We determined that the while loop will only execute log(n) times (the height of the heap). This is due to the fact that every time we sink all of the nodes, the bottom layer is guaranteed to follow the heap property. Then the second last layer will follow the property and so on. This guarantees that the while loop will execute at maximum the number of layers of the heap which is log(n). Therefore with sink() having complexity O(log(n)), the for loop with complexity O(n), the while loop having complexity O(log(n)) and is_heap() having complexity O(nlog(n)) our final complexity is O(nlog(n)).

Results:

Bottom-up:

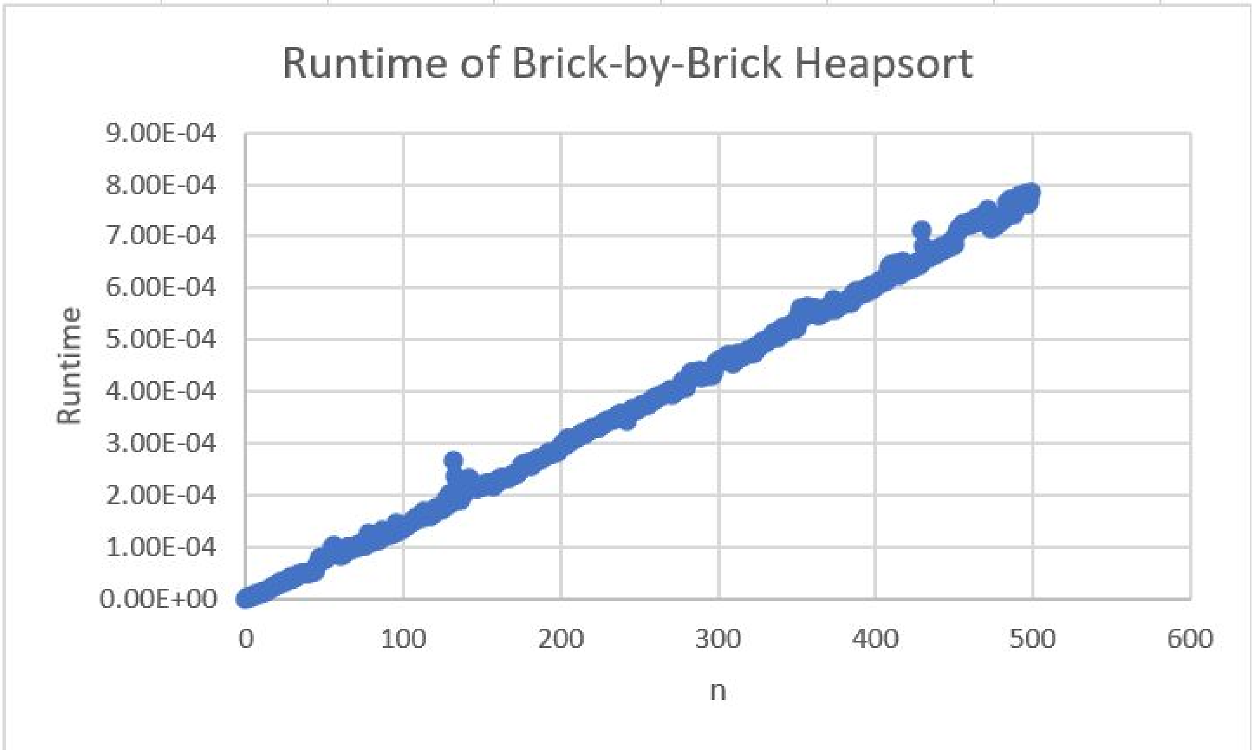
We tested the time complexity of the provided bottom-up variation of heapsort by using timeit. The results we got from timing the runtime of this heapsort variation proved our prediction that the runtime will be O(n log(n)). When initially plotting the runtime values on a spreadsheet, we noticed that the Runtime vs N graph created a linear trendline when looking at it. To confirm whether our prediction for this heapsort variation was wrong, we plotted another graph that compared the Runtime/N vs N values. For our prediction to be correct, this new graph should have a logarithmic trendline. After creating this new graph, we observed that the trendline was logarithmic with an R^2 value of 0.4205, which confirms that the time complexity of the bottom-up variation of heapsort is O(n log(n)). We have found through further research that building a heap has a complexity of O(n) so it is a possibility with further tests that we could see a linear relationship.

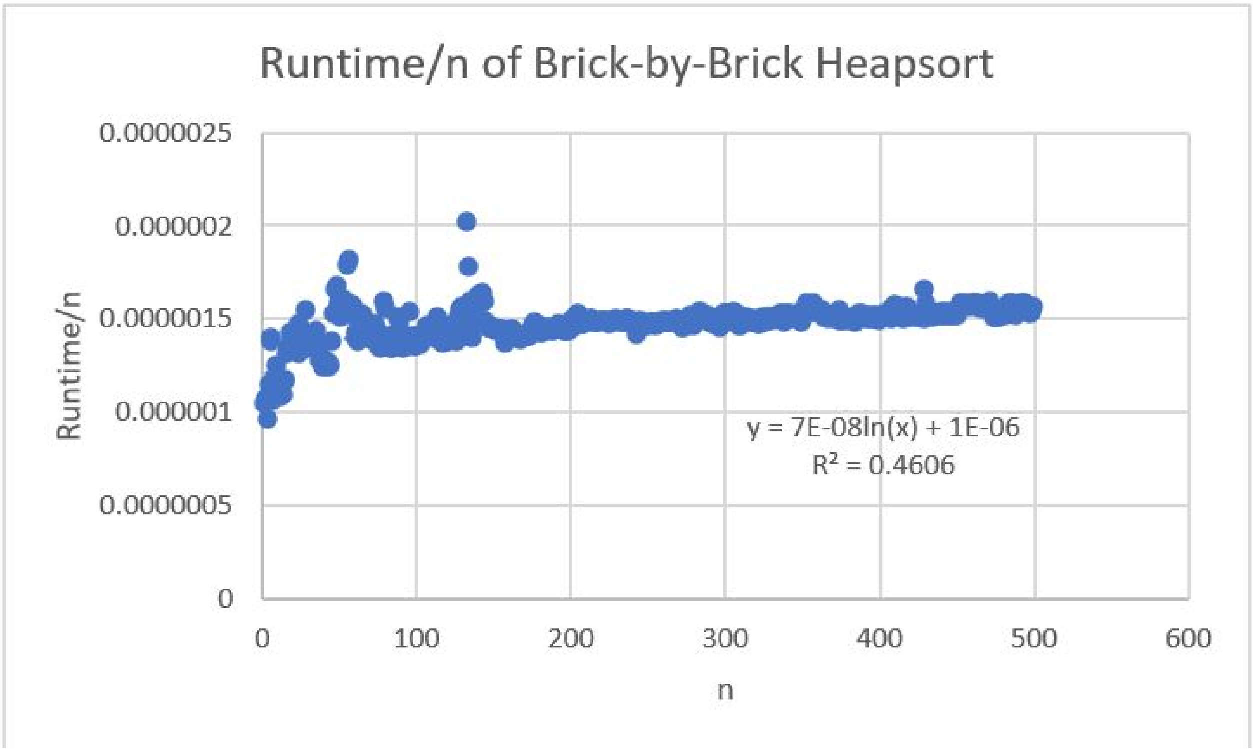




Brick-by-Brick:

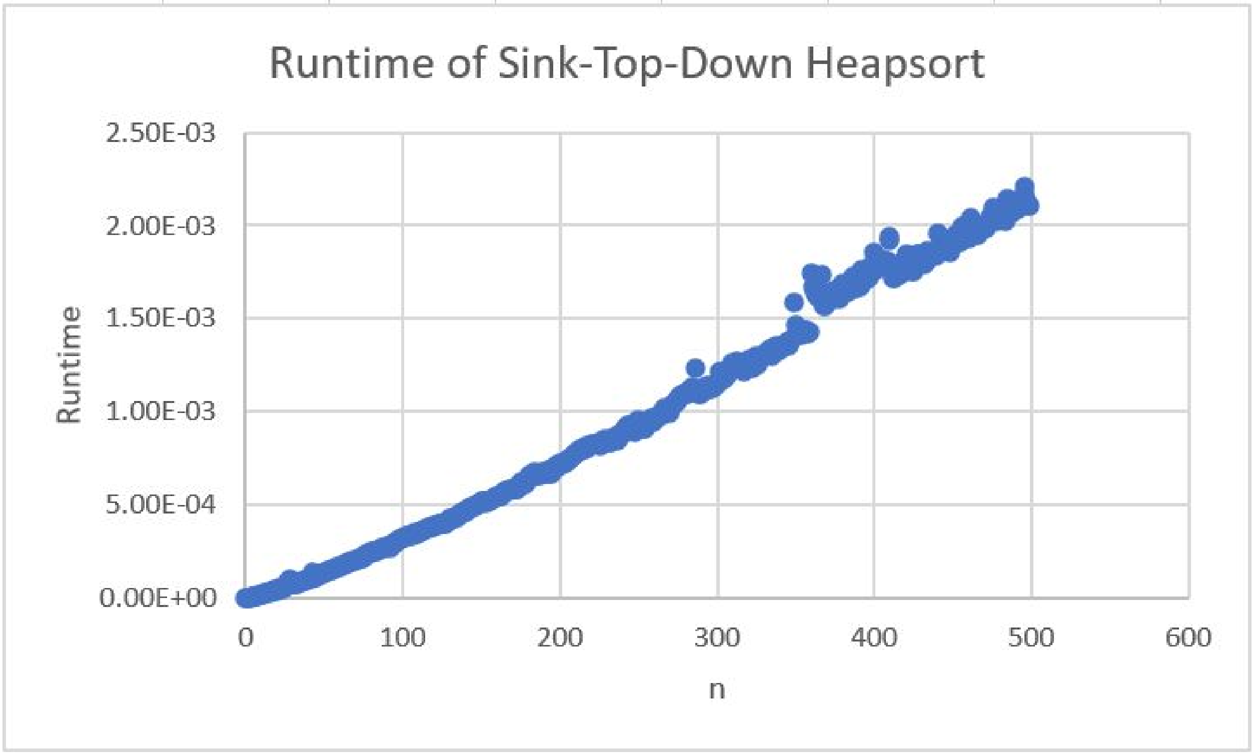
After implementing the brick-by-brick heapsort algorithm, we used timeit to see the performance of our brick-by-brick heapsort algorithm. We noticed that our prediction for this algorithm was indeed correct since the runtime vs n values that we got from our timing experiments produced a linearithmic graph. When plotting the runtime values on a spreadsheet, we noticed that the Runtime vs N graph looked like a linear function. To confirm however, whether or not our predictions for the runtime for the brick-by-brick algorithm was correct we decided to plot another graph, a Runtime/N vs N graph which would provide us with confirmation that brick-by-brick has a linearithmic complexity if the points formed a logarithmic trend. From the graph, we observed that the trendline for the points of Runtime/N vs N was indeed logarithmic and we got an R^2 value of 0.4606, confirming that the brick-by-brick algorithm of heapsort has linearithmic time complexity. We have found through further research that building a heap has a complexity of O(n) so it is a possibility with further tests that we could see a linear relationship.

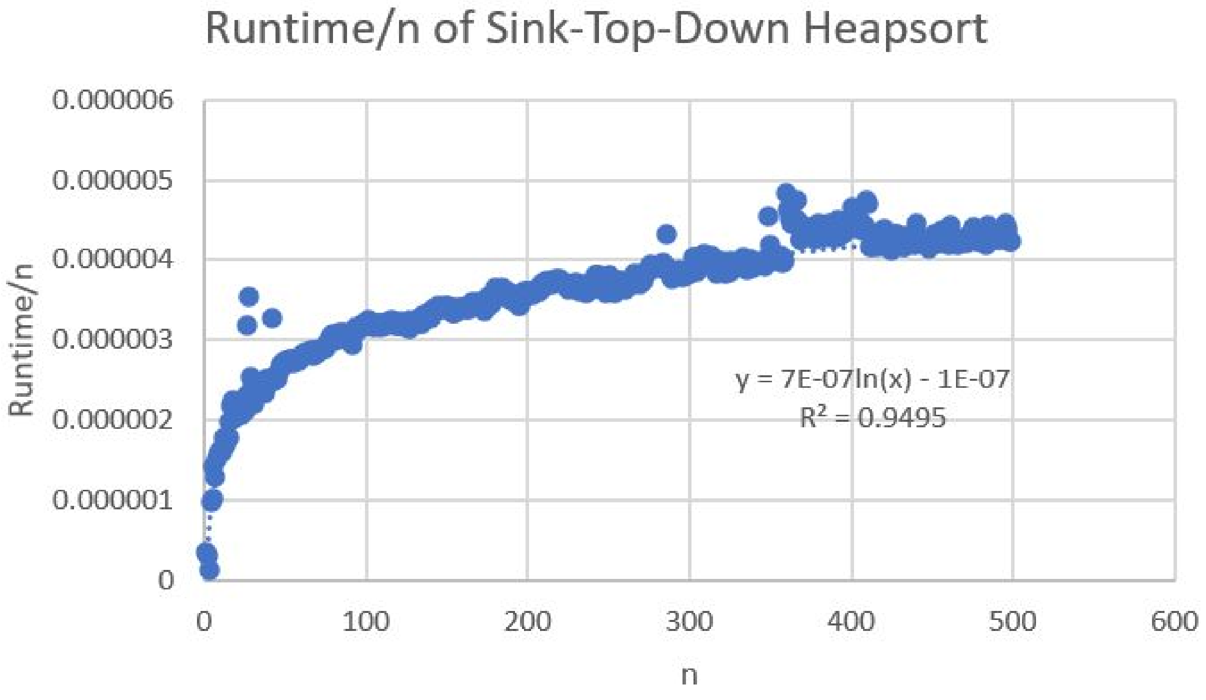

Runtime of Brick-by-Brick Heapsort
Runtime
n
9.00E-04
8.00E-04
7.00E-04
6.00E-04
5.00E-04
4.00E-04
3.00E-04
2.00E-04
1.00E-04
0.00E+00
0
100
200
300
400
500
600


Runtime/n of Brick-by-Brick Heapsort
Runtime/n
n
0.0000025
0.000002
0.0000015
0.000001
0.0000005
0
0
100
200
300
400
500
600
y = 7E-08ln(x) + 1E-06
R² = 0.4606

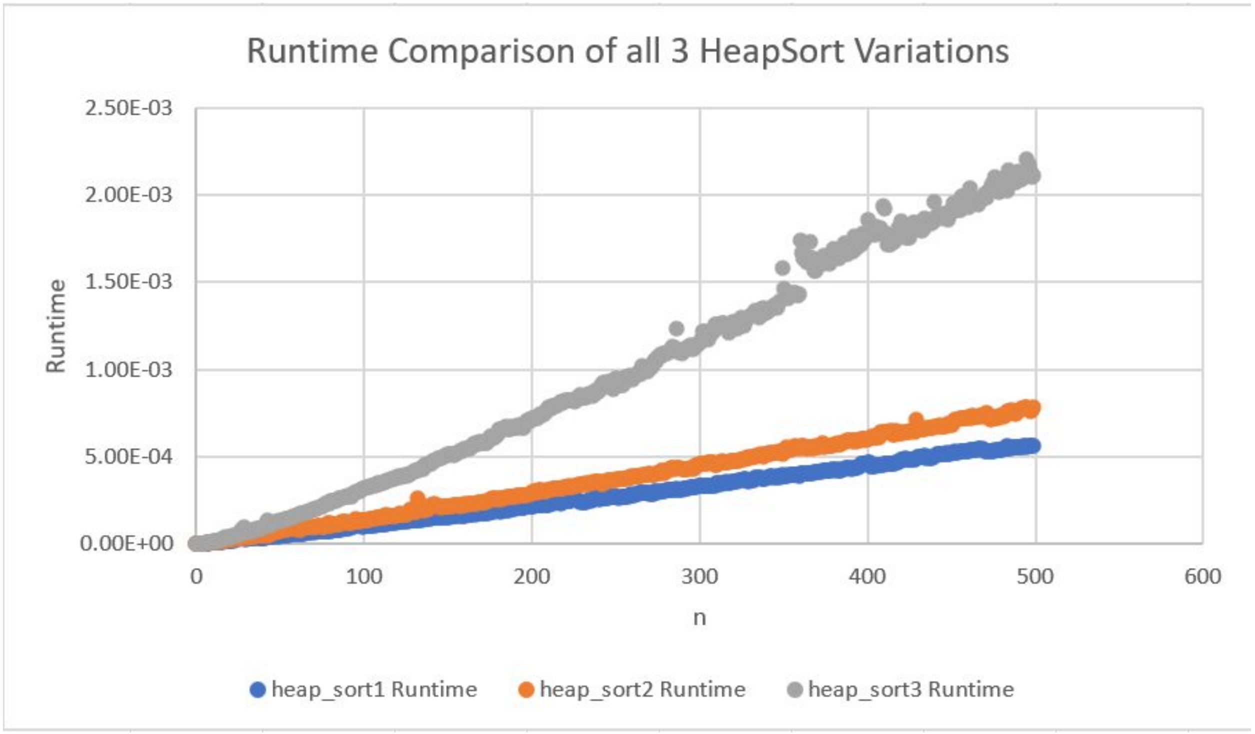
Sink-top-down:

After implementing the sink top-down variation for the heapsort algorithm, the timeit library was used to test the performance. After gathering and graphing the data, our predicate was correct regarding the linearithmic time complexity of this variation of heapsort. When initially plotting the values, the Runtime vs N graph looks linear. After further analyzing a new Runtime/N vs. N graph, we confirmed that the sink top-down heapsort did indeed have a linearithmic complexity. We observed that the trendline of the Runtime/N vs N was logarithmic and it gave a R^2 value of 0.9495.


Runtime of Sink-Top-Down Heapsort
2.50E-03
2.00E-03
1.50E-03
1.00E-03
5.00E-04
0.00E+00
Runtime
0
100
200
300
400
500
600
n


Runtime/n of Sink-Top-Down Heapsort
0.000006
0.000005
0.000004
0.000003
0.000002
0.000001
0
Runtime/n
y = 7E-07ln(x) - 1E-07
R² = 0.9495
0
100
200
300
400
500
600
n

As you can see in the graph below, the 3 Heapsort variations are shown. The Bottom-up variation of heapsort seems to be the best performing implementation compared to the brick-by-brick and sink-top-down implementation. These results line up nicely with our predictions.



For build_heap_3, can you identify an element of the implementation that causes poor performance? Could you think of a way to improve it?

From our results we saw that the build_heap_3 implementation of heapsort had the slowest runtime. We then discovered an aspect of the implementation that could improve the runtime so that it does not need to check if the list is in a heap every time the sink function is called on all the nodes. The element we found to cause poor performance in this implementation is the need to call the is_heap() function every time we loop through sinking each node. Having the number of times that we need to heapify the node ahead of time eliminates the purpose of the is_heap() function and will greatly improve the runtime. We can just sink every node a certain number of times and that will eliminate the amount of redundant checks that the code currently runs to determine if it is a heap or not. The maximum number of times you need to heapify every node before the entire list is a heap is $\log_2 n$ times, where n is the length of the list.

***K*-Heap:**

Advantages:

There are advantages when it comes to using a K-Heap over a binary heap where each node can have up to *k* children instead of being limited to 2 children. A significant advantage of k-heaps is that they have better memory cache behaviour which gives them a faster runtime in practice as compared to binary heaps. A source to this is included after advantages. Another advantage is that k-heaps allow the parent nodes to have up to *k* children which can be useful when dealing with very large amounts of data values since the height of the *k*-heap will be significantly shorter compared to the height of a traditional heap.

Source for K-Heap having better memory cache behaviour:

https://queue.acm.org/detail.cfm?id=1814327

Disadvantages:

One of the disadvantages for K-Heap is that it is often more complicated to think about in terms of implementation. Even though some of the advantages of K-Heap sort is that it can have a lot of children and have a longer width of a tree, the higher the k value, the runtime for K-Heap will become larger in worst case scenarios. Another disadvantage of the K-Heap is that some of the heap operations now have a worse runtime since the reorganization of the heap takes longer because of a greater number of children.

Complexity of the Sink function of a *K*-Heap:

The complexity of the sink function of a traditional heap is $O(\log_2 n)$. When you are actually going through the traditional binary heap and you apply the sink function to the parent nodes twice. This gives the sink function a runtime of $2 \log_2 n$ when it is actually going through the heap. In the case of a K-heap the runtime stays log(n) however the base changes from 2 to *k* since each parent node can have upto *k* children. When applying the heap sort to a list the sink function needs to be called *K* times since each parent has that many children nodes. Therefore, the runtime becomes K $\log_K n$ and when it comes to the overall general time complexity of the sink function it becomes O(K $\log_K n$).