1. **In lecture, we gave you formulas for finding the parent, left child, and right child within the array when implementing a binary heap.**

   For this assignment, you needed to implement a 4-heap – a heap where each child had four equations. This meant you needed to create two equations: a formula parent(i) to find the parent of some node i, and a formula child(i,j) to find the j-th child of some node i.

   What were those formula?

   (To get full credit on this question, you just need to correctly define parent(i) and child(i,j). You do not need to justify your definitions.)

   parent(i) = (i - 1) / 4

   child(i,j) = 4 * i + j

2. **When implementing the percolateDown algorithm in ArrayHeap, you probably noticed that manually checking each of the four children was tedious and redundant.**

   How did you refactor this redundancy? Were there any challenges you ran into along the way? If so, how did you handle those challenges?

   Justify the design decisions you made. (Your answer should be at most 1 to 2 paragraphs long).

   Since we know that we are implementing a 4-heap, we decided to use a for loop that executes 4 times to do the same work that needs to be done on each child. We also used recursion in percolateDown because we don't necessarily know how far we need to go on this method.

   One of the challenges we ran into along the way is determining whether or not this node has any children. We handled this challenge by first checking whether the index of the first child of the current node is within the current size or not. This will not cause a NullPointerException as well as preventing an ArrayIndexOutOfBoundsException in the for loop of the work that needs to be done on the children.

Experiment 1

1. **Briefly, in one or two sentences, summarize what this experiment is testing. (As before, treat this as an exercise in reverse-engineering unknown code).**

   This experiment is testing the time it takes to execute topKSort ten times while manipulating the sizes of the list.

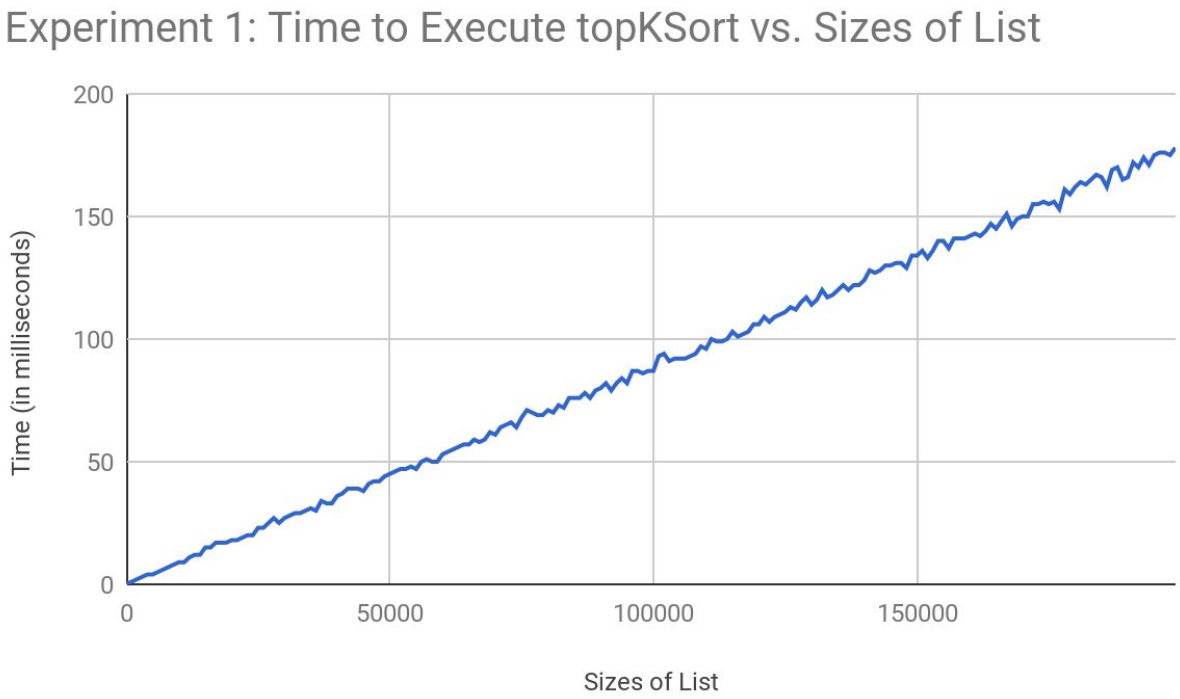
2. **Predict what you think the outcome of running the experiment will be.**

   Your answer should be at most one or two paragraphs. There is no right or wrong

answer here, as long as you thoughtfully explain the reasoning behind your hypothesis.

We think that the outcome of running the experiment will be that the time it takes to execute topKSort one time is O(nlog(k)), and the time it takes to execute topKSort ten times is O(10nlog(k)), which is still within O(nlog(k)) because we can ignore the constant factor here. This is because O(nlog(k)) is the efficiency we aim to implement for the topKSort method, and we do that by making sure the size of the ArrayHeap used for topKSort never exceeds the given K value.

3.



4. **How do your results compare with your hypothesis? Why do you think the results are what they were?**

The results are different with our hypothesis in the sense that our hypothesis didn't incorporate the sizes of list, but we had the correct idea. The results are what they were because as the size of the list, the n in O(nlog(k)), goes up, more values in the list need to be examine. But the time, in milliseconds, it takes to execute topKSort only goes up linearly because the method runs in O(nlog(k)) time, where k is independent from n.

Experiment 2

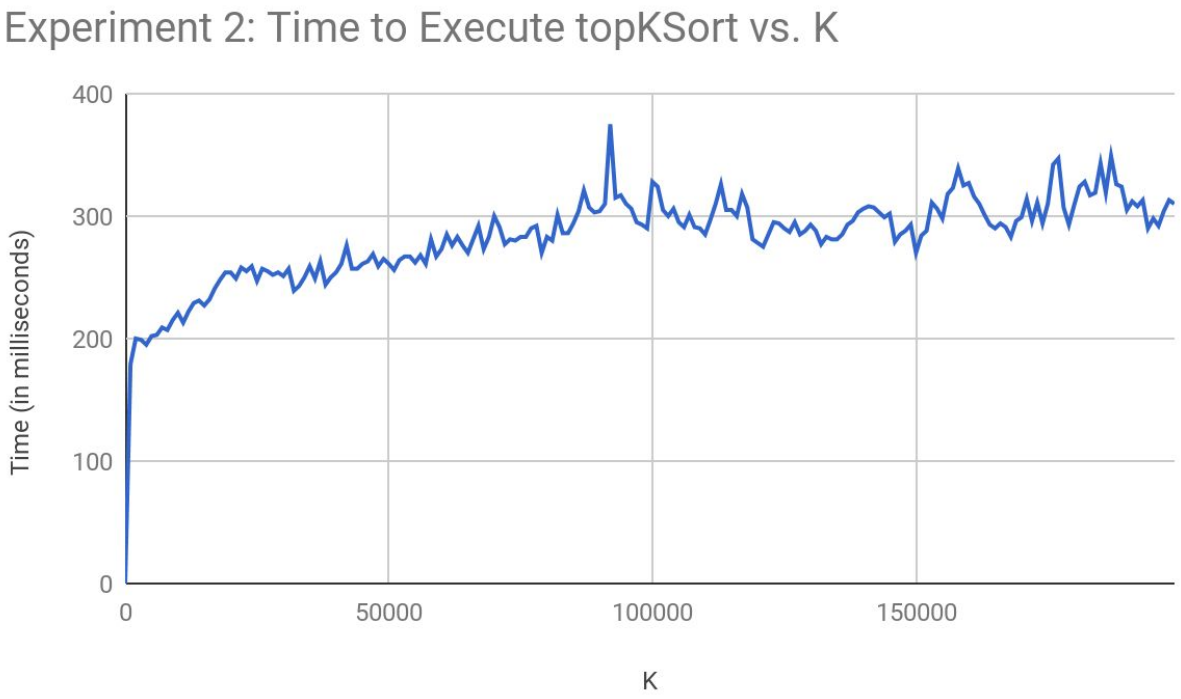
1. **Briefly, in one or two sentences, summarize what this experiment is testing. (As before, treat this as an exercise in reverse-engineering unknown code).**

This experiment is testing the time it takes to execute topKSort ten times, but it's different from Experiment 1 in the sense that the given K value changes for different tests in Experiment 2.

2. **Predict what you think the outcome of running the experiment will be.**

   Your answer should be at most one or two paragraphs. There is no right or wrong answer here, as long as you thoughtfully explain the reasoning behind your hypothesis.

   We think the outcome of running the experiment will be that as the given K value increases, the time, in milliseconds, it takes to execute topKSort will increase by log(K). This is because O(nlog(k)) is the efficiency for the topKSort method. In this case, n, the size of the list, remains the same, so the only factor that is causing the change is the different K values.

3. 

4. **How do your results compare with your hypothesis? Why do you think the results are what they were?**

   The results confirms with our hypothesis, which is as the given K value increases, the time, in milliseconds, it takes to execute topKSort will also increase. The time to execute the method increases logarithmically with K. The results are what they were because as the K value increases, the size of the ArrayHeap used for topKSort increases with it, which causes the time calling the method to go up since more sorting needs to be done, potentially.

Experiment 3

1. **Briefly, in one or two sentences, summarize what this experiment is testing. (As before, treat this as an exercise in reverse-engineering unknown code).**

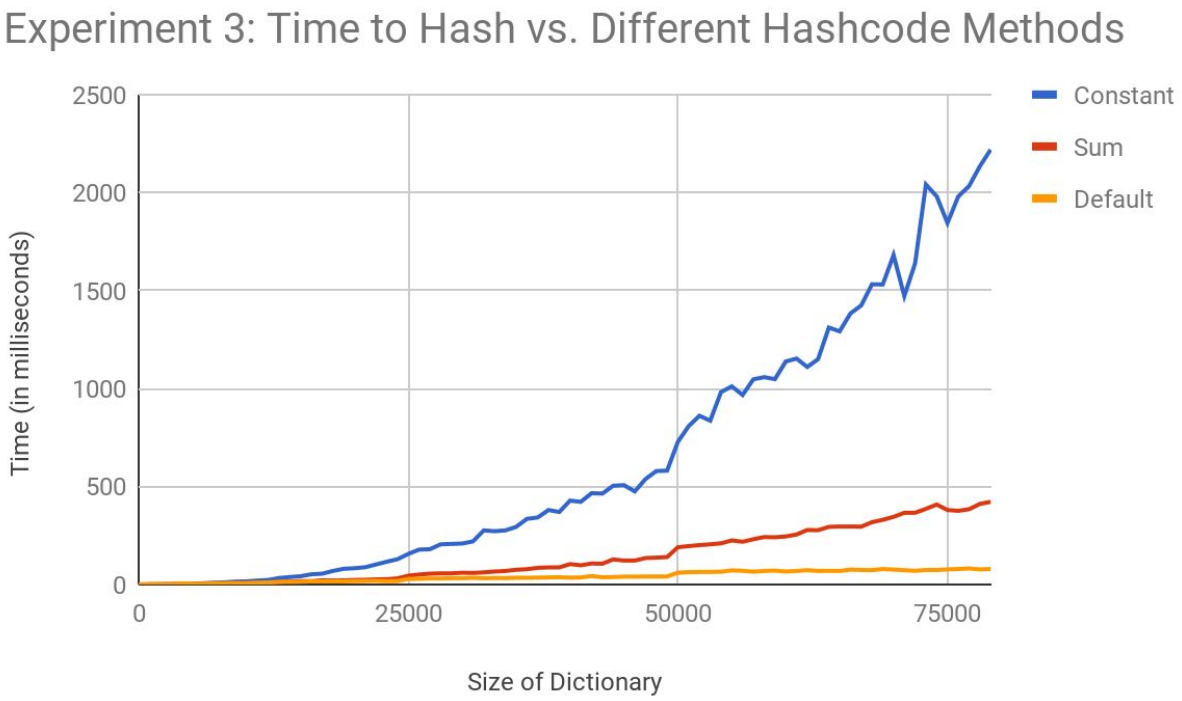
   This experiment is testing the time it takes to hash a ChainedHashDictionary using three different hashcode methods: constant, the sum of the characters, and a method that's similar to Java's default hashcode.

2. **Predict what you think the outcome of running the experiment will be.**

   Your answer should be at most one or two paragraphs. There is no right or wrong answer here, as long as you thoughtfully explain the reasoning behind your hypothesis.

   We think the outcome of running the experiment will be that the time it takes to hash a ChainedHashDictionary will be the slowest for the constant hashcode method which is the sum of the first four characters in the array, followed by the method of using the sum of the characters, and then the method that's similar to Java's default hashcode will be the fastest as the size of the dictionary goes up. This is because for the hashcode method that uses a constant integer, all the elements in that ChainedHashDictionary will be hashed to the same spot, which can cause collisions between the current data and the new one. This in turn defeats the purpose of a ChainedHashDictionary. The hashcode method using the sum of the characters will perform better than the constant one, but it still has a high chance of mapping to the same spot. A good example is the same characters in different sequences.

3.



4. **How do your results compare with your hypothesis? Why do you think the results are what they were?**

   The results confirms with our hypothesis, which is that it takes the least amount of time to hash a ChainedHashDictionary using a method that's similar to Java's default hashcode, followed by the method of using the sum of the characters, and then the constant hashcode method, as the size of the dictionary increases. The results are what they were because all the values will be hashed to the same spot in the ChainedHashDictionary using the constant hashcode method, which makes the dictionary meaningless since then the dictionary is essentially just an one-dimensional array. Like we said in our hypothesis, although there is still a high chance for the method using the sum of the characters to hash to the same spot in the ChainedHashDictionary, it's significantly better than the previous method. This is also proved by the graph where the constant hashcode method is in $O(n^2)$ time, yet using the sum of the characters is much faster. The method that's similar to Java's default hashcode performs the best in this case because it adds more variety in its method, which prevents each value to be hashed to the same spot that often. In the end, this experiment proves that Java's current hashcode method is efficient.