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) = i * 4 + 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).

   - We keep track of an index of the node of the minimum value among the four children and swap with the parent if the parent is larger than any of the four children. We used a while loop to go down the heap to look for potential swap that needs to make. In the while loop, we used a for loop to check the value of four children.
   - One challenge we encountered was setting the condition of the while loop. Since we want to keep going as long as the parent is larger than any of the children, the condition should be something like while (parent>children). We want to

compare the parent with the smallest child, but the for loop that compares the parent with the children is inside of while loop and it is updating a variable called min(initialized outside of while loop) to keep track of that child's index. Thus, the condition of while loop becomes something like while(parent>min). We set the min to be parent in the beginning and use a do while to make sure it executes the statement at least once(since the if we use a regular while loop, it is not going to enter it at all because parent=min in the beginning). If we found a min(smallest child index) in the for loop, we check the condition of while loop(parent>min), if it is true, we swap the parent and child and set the parent equals to min(so that later on in for loop if there is no child smaller than the parent, we will jump out of while loop because parent=min in next round) in the beginning of while loop and continue to for loop.

3. In this question, you will run a variety of experiments and report back on the resulting behavior. You can find the experiments in the `analysis.experiments` package. For each of the three experiments, answer the following:
    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).

- Experiment 1 is testing the time to run sort top K(500 in this case) elements for 10 times within a changing-sized list.
- Experiment 2 makes a list with fixed size and records the time of sorting top K(value of K is changing in this case)elements 10 times in the list.
- Experiment 3 carries three tests where each of those has different ways to generate hash code for each character array. Thus to test the efficiency(time) to insert the list of array into the dictionary.

    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.

- Since the Experiment 1 has an unchanged K value throughout the for loop, it implements the topKSort method, but the list sizes increases throughout the experiment, so we should get the increasing nlog(K) runtime which in this case K is equal to 500 for 10 times.
- Experiment 2 should have the larger outcome when K is larger since the searcher runtime is nlog(k). The reason why the runtime is nlog(k) is because we loop through the whole input which is n and (1)put the first k item(runtime for insertion is log(k) because worst case k item in heap) from the input in the heap and then (2)compare the rest(index from k+1 to last item in input), insert if it is larger than min in current heap(insertion runtime log(k) because k item in heap). In conclusion, it is n times log(k) no matter we are on (1) or (2). When k get larger, nlog(k) should be larger.
- Experiment 3 should have three distinct runtime among three tests since each of them has different hashcode method. Test 1's hashcode method simply returns the sum of the first four elements in the array. With increasing k, more and more collision would occur because the number of sum of the first four elements is finite. However, Test 2 loops through the entire array to generate a hashcode for each array we want to put in the dictionary, the collision rate should be less than the first method. Test 3 actually does the same thing as the second method with a lower collision rate because it still iterates the entire array, but the final code times 31 lowers the collision rate because it adds more variation and separate the elements apart to avoid situations in the second method like the list“[a],[b],[c]” collides with the “[c],[b],[a]”.

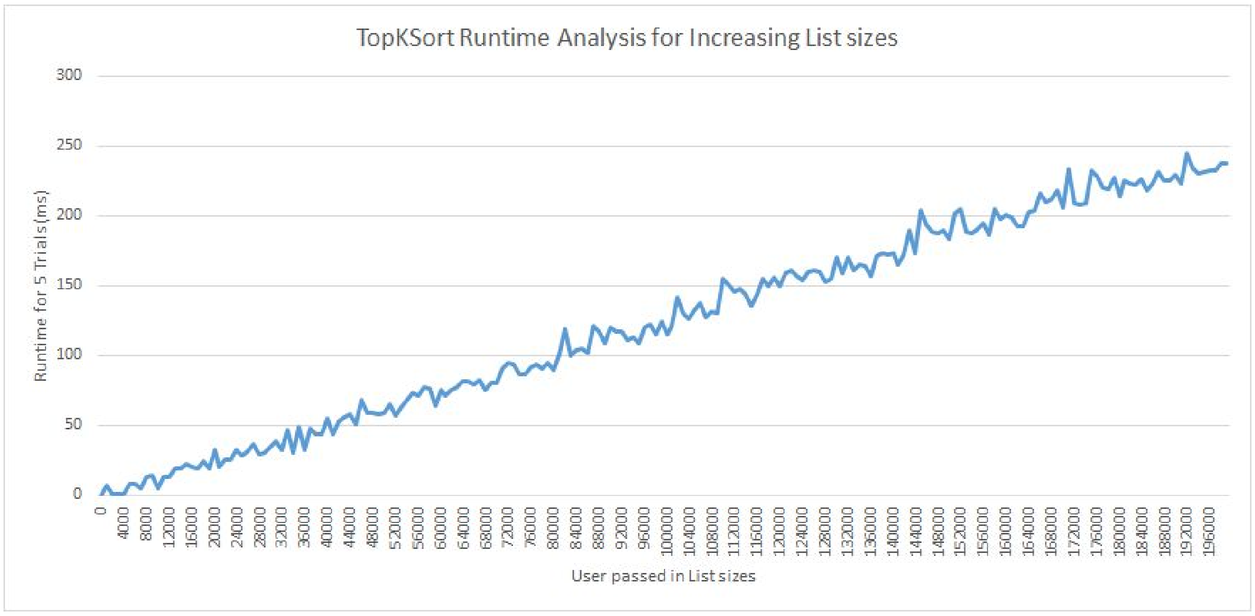
3. Run the experiment code. Each `Experiment*.java` file should generate a CSV file in the `experimentdata` folder. Import the CSV file into Microsoft Excel, Google

Sheets, or any other analysis of your choice and generate a plot of the data. Include this plot as a image inside your writeup.

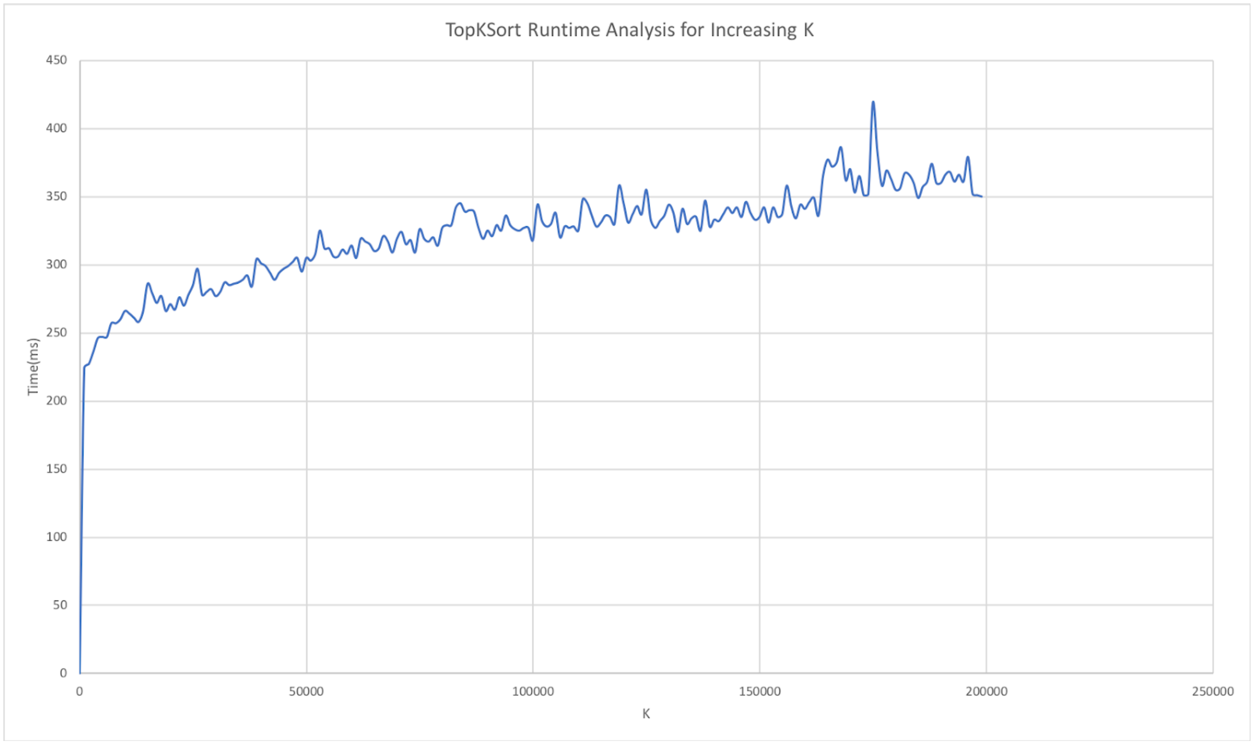
If the CSV file contains multiple result columns, plot them all in the same chart.

You will be graded based on whether you produce a reasonable plot. In particular, be sure to give your plot a title, label your axes (with units), include a legend if appropriate, and relabel the columns to something more descriptive than "TestResult".

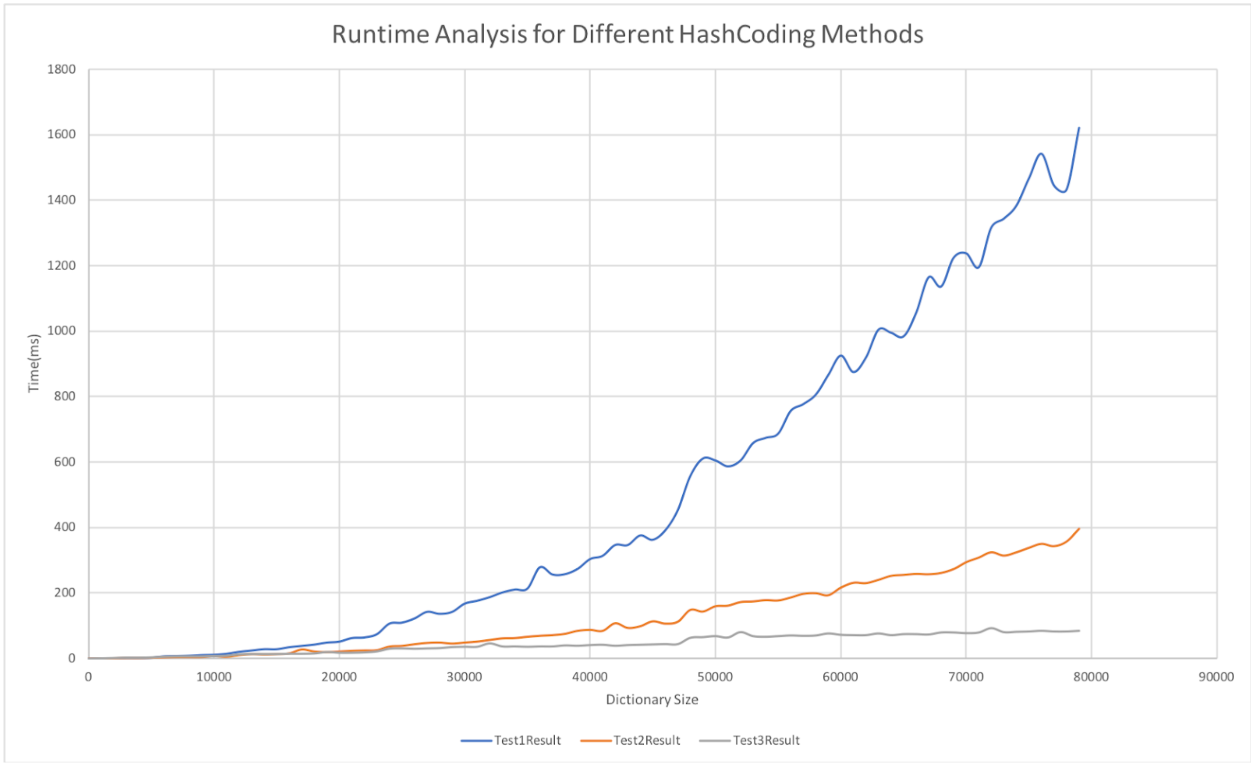
- Experiment 1:



- Experiment 2:



- Experiment 3:



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

- Experiment 1: The experiment 1 result behaves the same as our hypothesis, the total runtime increases as the input list sizes increases. According to the top KSort method, the runtime should change with the O(nlog(k)) function. Since the k stays constant in this experiment, and the list sizes increases as an increment of 1000 from 0 to 200000, the nlog(k) function should increases and behaves as a linear function, same trend predicted in our hypothesis. Although there are some waves along the trend towards the end, but the overall tendency is linear.
- Experiment 2: Our hypothesis aligns with the results. But we did not expect the run time spikes when the k value jumps from 0 to approximately 1000. From 1000 to 200000, the trend follows O(nlog(k)) function. According to our TopKSearcher method, when k equals 0, the method immediately returns an empty list, thus the runtime is close to 0. Then when k is greater than 0 the algorithm starts to go through the input list and add things into the heap, which is represented by the O(nlog(k)) runtime. Therefore, there is a big runtime spike from k=0 to k=1000. After k=1000, each of the later runs of the topKSort method follows the n(log(k)) increment trend. Since n is constant in this experiment, the trend can be represented as a log function graph shown above.

- Experiment 3: Our hypothesis matches with the results. The runtime of each hash coding methods differ from each other. Test 1 takes the longest time due to high collision rate, because using the sum of 4 characters has a high chance to result in same number. No matter how we resize the dictionary, the hashcode returned is going to be mapping to the same spots because the combination of the 4 elements is a finite number. High collision rate drags down the runtime. Test 2 takes significantly less time since it is using the sum of the array. Even though there is still a chance to have the same hashcode, the chance is much less. Test 3 is basically using the Java hashcode and is much faster due to low collision rate because of more variation, 31 is a large enough prime that the number of buckets is unlikely to be divisible by it. The experiment shows that Java hashcode method is efficient.