Author: Brandon Fowler
EWUID: 00639348

Class: CSCD320

Quarter: Spring 2014

Project Report

Abstract: My goal is to find the top ten thousand richest people on a much larger given list of many people's income; along with the line index that these top ten thousand values are found in the file. Due to memory limits, I can only store ten thousand records at one time. In order to solve this task, my idea is to read the first ten thousand records and store their values and line indexes in an array of node structures; each node holding a value and a line index. Then I will convert this array into a min heap, which will store the smallest value on top. I will then continue to move through the file, and each time I find a value in the file that is larger than the value at the top of the heap, I will swap the values and line indexes, then reform the heap. This is easy to manage, because it guarantees that we are trading the smallest stored value each time for a bigger one; and the heap is easy to reform, because by this point it is already guaranteed that everything stored below the swapped values is already a min heap. This idea should accomplish my goal within the specified parameters.

Report:

In detail, the problem given is to find the ten thousand largest numbers in a text document containing a huge amount of numbers, separated line by line. The problem specified that my code cannot store more than ten thousand records at any time, and that I must also keep track of what line index in the file that the values are found. Then, I must output the values I have found.

In order to solve this problem, I first decided that the structure of my records should be designed as a nodes, which store file values along with the line index they were found on. These records are held in an array capable of holding ten thousand records. The next decision I made, was to immediately read in the first ten thousand values from the file into an array. Next, I wrote algorithms to turn this array into a min heap. I wrote the standard min heapify method, then in another method I walked through the array and recursively used heapify in order to convert the array into a min heap. Next, I wrote a simple method that would swap the value stored at the top of the heap(the smallest) for another number passed in, then called heapify to insure min heap consistency. Afterwards, I simply continued to read file values and anytime I

found a value bigger than the top of the heap, I would just pass it to my swap method. Finally, after reading all of the file values, I could simply print out every value and line index in my heap array to an output file, to show that I found the largest ten thousand records from the input file.

The main time cost of this algorithm comes from reading/comparing "n" values from the input file, and calling the heapify method every time a larger value is found. In the worst case if the list given is sorted in ascending order(every new value is larger than the last), then heapify will have to be called for every new value. We know that the heapify method takes O(logn) time, and that we will be reading in "n" values and at worst be calling heapify for every value; so the total time cost of my algorithm should be O(nlogn) . Reading in the first ten thousand records and printing out the highest ten thousand, are just large constants and are therefore ignored in my time complexity calculations.

I experimentally tested this solution with a list of one million randomly generated numbers, and to the best of my testing abilities determined that it functions as intended, and swiftly for the problem size. For several runs of a list of 1 million records, it averaged about 4 seconds each time to output results on my own PC. I did several smaller tests to verify the correctness of the algorithms results, and the results were correct every time as best as I could determine. Testing with a huge data set, was much harder to physically verify correct results; however nothing except the problem size should change between the two, therefore it should still be correct. The ten thousand largest records are output to a .txt file, in the order that they are stored in the heap array.

To conclude, I believe that I successfully solved this problem satisfactorily to its specifications, and the algorithm itself should be reasonably quick. I would be interested again to use a similar strategy in another class or when I move on to solve real world problems, and will save the code to build upon.

Note: I tried to generate a set of test data with 1 billion records, and my computer could not even handle it without freezing. I would say that a data set so large, is simply too much regardless of the algorithm, unless the machine involved is equipped to handle it.