Group Assignment #1: Report

James Luo, James Barry, Tarren Engberg
CS 325 - Analysis of Algorithms
October 10th, 2017

Algorithm Description

This report covers and algorithm meant to find the kth smallest element from the union of m sorted arrays of length n. Each array on its own is sorted, but they have no relation to each other. This algorithm must leverage the fact that each is sorted in order to produce a lower run time. The arrays will be taken as input from a set of data files, and will not be read into memory due to runtime and data restrictions; the data files will potentially hold millions of numbers. Simply joining and sorting the arrays is also not possible due to the runtime restraints.

In order to author an effective algorithm to find and return the *k*th smallest element in *m* collective arrays, we first must handle the base cases. Per specification, there will be 3 data parameters which will describe the data files' attributes: *m* describing the amount of arrays, *n* describing the amount of elements in each array and *k* describing the desired in-order element from the union of all *m* of the arrays.

Our base case will handle the lowest allowable values of *m* & *n*. The specification defines the lowest input for each *m* & *n* as 1 with special instructions for each scenario. In the situation where *m* = 1, we know that we have a single ordered array and we can simply use *k* as a seek selector to return the *k*th element. In the scenario that *n* = 1, we will treat each array as unordered and be forced to examine each array element at a complexity of O(N).

Our code first reads in the *input.txt* file and stores the previously mentioned m, n & k as variables for use in our algorithm. Due to the fact that our input data is significantly large, we must design our algorithm to work with data files without storing all of the elements. For the purposes of our algorithm we have chosen to use file I/O that is compatible with binary file content and utilize the 'seek' function to pick values as needed from the file.

The algorithm we have designed takes the halfway point of the first array m_1 [n/2] and records that value v. After finding value v, the algorithm seeks through the other data files using binary search. If it determines that the value v does not exist in a file, there are three possibilities. Either all the values in the file are less than v, all the values in the file are greater than v, or v is not present in the file and the value v falls between two other values that are in the file. The algorithm will start out by checking the first and last index of each file to decide early on if the file even needs to be searched. If all values in the file are less than v, the search will return the final index (which would be equal to n-1) as a base case. In the circumstance that all values are

greater than v, the search will return -1 as a base case. If v falls between two values, binary search will perform as usual and ultimately return the index of the lower value. If v does exist, binary search will also perform as usual but simply return the index of the value v.

After the binary search section of the algorithm has been performed on all of the input files, the indexes will be added together. Then, m will be added to that value to correct the difference between the 0-based index of our seek method and the 1-based index of the input. This total value is equal to the hypothetical k value of v, meaning that there are k-1 values less than v. Note that this k is not the k received in input; this new k value will be referred to as k2. k2 is then compared to the k received in input. This is to determine if the value of k falls below or above or is equal to k2. If they are equal, return the value v. If k2 is less than k, the k value we need to find is greater than v. This means k does not exist before v, and the algorithm will move on to the midpoint of the upper half of the file. This allows us to effectively cut off half of the array without needing to search it. If k2 is greater than k, it means k does not exist after v, and the algorithm will move on to the midpoint of the lower half. After this new midpoint is determined, binary search will once again be performed on all of the other data files as before. Once all of this data file is effectively checked by the algorithm, it will move on to the midpoint of the next file. The process will then repeat until k is found.

Once the kth element has been returned our program writes it to a file titled *output.txt*

Running Time Analysis

The core of this algorithm is binary search. Binary search is used to look for value v in a data file. The runtime of this is log(n) because, in the worst case, we repeatedly cut the searched data file in half until none of the data file is left. This is exactly what taking the log of n is - dividing it by 2 until it reaches 1 [1].

Binary search must always be performed on m data files in order to find the hypothetical k value of value v.

Because we repeatedly find the hypothetical k values of midpoints in a data file, we search for the k value of log(n) values for each file, and each search takes log(n) time as described above.. Because of this, searching one file takes m $\log^2(n)$ time. This then must be done m times in worst case, because every data file must be searched.

Therefore, the total runtime of the algorithm in worst case is $O(m^2 \log^2(n))$, which matches the specified runtime restraints for the algorithm. .

References

1. Running time of binary search - Khan Academy - https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/running-time-of-binary-search
2. Discussion after class with Amir Nayyeri