Assignment 1: Analysis of KthLargest Algorithm

Bastien Gliech

January 30, 2018

A. Mathematical Analysis of Algorithm

In this program, there are several key steps that are undergone to ensure that the algorithm functions properly. To analyze the entire algorithm, each individual component must be examined first.

(i) The algorithm finds a value larger than any value in the array. As this has to make one comparison per index in the array, the worst-case complexity is:

$$O(n)$$

where n is the size of the array

(ii) The algorithm finds the largest value that is under a "maximum". This requires one pass through the entire array, once again leading to a worst-case complexity of:

$$O(n)$$

(iii) It should be stated that (ii) occurs more than once. In the first pass of (ii), the "maximum" is the value found in (i). The amount of passes is not static, but rather is determined from whether the kth largest element has been discovered. For every pass, the algorithm tallies how many elements equal the value found in (ii). This is repeated for decreasing values from (ii) until the sum of all tallies is greater than k. In the worst case, k is tallied only once per traversal of the array. Thus, the worst case complexity is:

$$O(k * n)$$

Thus, if $k = 1$, then the complexity of the algorithm is $O(1 * n) = O(n)$. This will be true for any case when $k \in \mathbb{Z}$. When $k = n/2$, the worst case, determined from (iii), will be:

$$O(n^2)$$

B. Experimental Results

As n remained constant and k increased linearly, the time of completion for the algorithm increased linearly. Thus, the complexity was $O(n)$.

As n increased linearly and k remained constant, the time of completion for the algorithm increased linearly. Thus, the complexity was $O(n)$.

So far, this confirms the previous analysis of the algorithm. However, when both n and k increased linearly, the analysis would suggest that the complexity should be $O(n^2)$. However, this is not the case. As both n and k increase linearly, the complexity is $O(n)$. This is due to the increasing number of duplicate values, as an inverse relationship exists between number of duplicate values and number of array traversals.

The algorithm functions very well with a small range of values, but as soon as the range of values increases, the time complexity of the algorithm approaches $O(n^2)$. In order to solve this, the approach to solving the problem would have to change entirely. If an array of size k was used to store data, the process of referencing the last used maximum would be much easier. However, this could become costly for the memory for higher values of n. Knowing this, the algorithm functions very well for a small range of values without being extremely costly.