Sandhya Chandramohan
Naveena Kannan

CS 260P Project #1

To find k-largest elements in an array

Build Instructions

Compile using: `gcc MAIN.c -o MAIN`
Run using: `./MAIN`

Algorithm

The Algorithm used is as follows:

1. Build a Min Heap consisting of the indexes of the first k elements in the array
2. From the (k+1)th to nth element compare each element against the root of the Min Heap
3. If the element is greater than the root of the Min Heap then replace the root of the Min Heap with its index and re-balance the Min Heap
4. At the end, the Min Heap will contain the indexes of the K largest elements
5. The K largest elements are then sorted using Binary Insertion sort to obtain the indexes of the K largest elements in descending order

Analysis

- The Min Heap of the indices of the first k elements is formed at worst in O(k) time
- The re-balancing of Min Heap on addition of (n-k) elements at worst case happens in O(((n-k)*logk)
- Finally the Binary Insertion sort of k elements in the Min Heap is performed in O(k^2 + klogk) in which O(k^2) dominates

Theoretical Worst Case

Now let's assume that the array has n random elements and the k largest elements are equally likely to be in any of the n positions.

The worst possible case is when the array is in ascending order and the k-largest elements are the last k elements of the array. The number of comparisons taken in that case will be:

1. To Build the Min Heap of first k elements ->
   For some integer d = $\lceil \log k \rceil$
   In general, level i of a full and complete binary tree will contain 2i nodes, and that those nodes are d – i – 1 levels above the leaves.

   Comparisons = $2\sum_{0}^{d-1} 2i(d - i - 1)$ = $\lceil 2k - \lceil \log k \rceil \rceil$

   But since the elements are in ascending order, building the Min Heap of the first k elements will take k comparisons.

2. In each iteration of the elements from k+1 to the nth element, the root of the Min Heap will be replaced and Min Heap rebalanced so comparisons in each heapify operations ->
   The number of levels of the Min Heap is at most 1 + log(k)
   Each iteration of the loop moves the target value a distance of 1 level, the loop will perform no more than log(k) iterations.
   In worst case, we perform two comparisons.
   Comparisons = 2log(k)
   There are (n-k) elements so the total comparisons = (n-k) * 2logk

3. Finally, when sorting the k largest elements in descending order using Binary Insertion Sort ->
   To search an element (using binary search) it takes log(k) comparisons in worst case, and to move the elements it takes k time.
   To do this for k elements,
   Total comparisons = (k(logk+k)
   For n=100 and k=10,
   Total comparisons at worst case = k + (n-k)*2logk + k(logk+k) = 300

Theoretical Expected Worst Case

Let's say the algorithm is run on 100 random arrays. For the worst case scenario to occur, the array will have to be in ascending order.

The probability of that happening is = 1 – P(Array is not in ascending order) = $1 - (99/100)^{100}$

= 0.633967 => nearly 63.4% chance => So the expected value of this scenario is 300∗0. 633967 = 190.19

Let's think of the next worst case, in which all but one element are in ascending order.

The probability of that happening is = [1 – P(99 elements are not in increasing order, 1 element is)] * P(100 elements are not in increasing order)
= [1 – (99/100)^{100}] (99/100)^{100} = 0.2320 => 23.30% chance

Now assuming the total comparisons at worst case improves by a uniform factor of constant of (300/100) = 3 for each element not in ascending order.

=> expected value = (300 - 3) * 0.2320 = 68.904

Which brings us to a total expectation of = 259.094

So if we extend this logic, then the expected value of the scenario where the worst case is 300–3i is

$E=(300-3i)(1-(\frac{99}{100})^{100})\ ((\frac{99}{100})^{100})^{i} = 298.267$

We add all those expected values up and we end up with approximately 298.267

So if we ran this algorithm 100 times on an array of size 100, we'd expect the worst case we observe to be about 298 or 299 comparisons.

Theoretical Average Case

For the average case, we take each possible case, and multiply the number of comparisons done in that case by its probability then sum them together. Since we are considering an array in which it is equally likely for the k-largest elements to be in any of the n positions, in this case all the probabilities are 1/300 :

1/300 + 2/300 + …. + 299/300 + 300/300 = 150.5 comparisons

Observed Worst Case and Average Case

The Observed Worst case is: 312

The Observed Average case is: 244.91

The theoretical worst case = 300 and the expected theoretical worst case = 298.267 are close to the Observed Worst case = 312 and to each other. The deviation observed could be due to the approximation made at calculating the theoretical worst case and also due to the fact that the variation of total comparisons with each element not in ascending order is approximated. The values of the inputs n and k and the number of samples tested (100 in this case) also determine the divergence.

The theoretical average case = 150.5 and the observed average case = 244.91. The deviation could be due to the reason that the probabilities of different scenarios occurring might not be uniformly distributed, some input cases might be more likely to occur in reality in a 1000 runs of the algorithm. As a result, the theoretical average

case which does not account for it is lower than the observed average case. The values of the inputs n and k and the number of samples tested (100 in this case) also determine the divergence.