Shaun McThomas
Ian Schweer
CS 165
Project 1: Selection
04/13/2016

Introduction:

This problem is to implement an algorithm when given a list of n number, find the k largest using the fewest calls provided COMPARE in worst case scenarios. For this project we assume that COMPARE is a "black box" function call, which is expensive but required.

Methodology:

In developing this algorithm, we devised several methods. The first solution was the naïve solution. This solution is to first find the largest value and stores it in Best, then repeat k times. Because this is an O (k*n) solution, it was never developed.

The next solution was to use the partition function of quickselect to first partition off the k largest elements. After this partition, we then would sort those k elements. This algorithm seemed to have a good complexity ($O\ (n\ +\ k\ log\ k)$) on paper but did not in practice. This was quickly scrapped for different solutions.

Another idea was to use a max heap. In this implementation, we first heapify the list of n elements. Then we pop off the k top elements. This implementation yielded much better than any other previous solutions. This is the case, even though the complexity of this is $O(n\ +\ k\ log\ n)$ that is worse than the quickselect method.

The next implementation was the solution that leads us closer to our final solution. This is to use a min heap to hold the largest k elements seen so far. We first heapify the first k element to create a min heap. Then we walk through the remaining n-k elements comparing it to the min on the heap. If the value is larger than that element we add it to heap. After we have seen all the elements, we pop all the element off the heap in sorted order. This is a $O(n\ +\ n\ log\ k)$ solution.

We also implemented an idea to use an AVL tree of size k to track the largest k elements. This implementation gave us the most consist and easy to analyze solution but not the best performing.

After returning to the idea of the min heap of elements, we decide to try caching comparisons to avoid duplicate comparisons. The cache only saved us $O(klogk)$ compares. This became our final, and submitted, solution.

Data used for Analysis:

Data from various implementations:

quick select

n= 10, k=1: maximum= 32, avg= 14.33
n= 100, k=10: maximum= 625, avg= 267.98

Shaun McThomas
Ian Schweer
CS 165
Project 1: Selection
04/13/2016

n=10000, k=40: maximum= 48512, avg=20950.01
n=10000, k=100: maximum= 61074, avg=22176.52

Max Heap
n= 10, k=1: maximum= 15, avg= 13.02
n= 100, k=10: maximum= 294, avg= 278.25
n=10000, k=40: maximum= 19951, avg=19751.58
n=10000, k=100: maximum= 21399, avg=21228.38

Min heap of Maxes
n= 10, k=1: maximum= 9, avg= 9.00
n= 100, k=10: maximum= 296, avg= 233.31
n=10000, k=40: maximum= 12467, avg=12049.25
n=10000, k=100: maximum= 16523, avg=15804.27

Min heaps of maxes with Caching
n= 10, k=1: maximum= 9, avg= 9.00
n= 100, k=10: maximum= 275, avg= 215.36
n=10000, k=40: maximum= 12267, avg=11885.22
n=10000, k=100: maximum= 16107, avg=15441.85

AVL
n= 10, k=1: maximum= 9, avg= 9.00
n= 100, k=10: maximum= 340, avg= 302.11
n=10000, k=40: maximum= 51595, avg=48036.81
n=10000, k=100: maximum= 60789, avg=58355.60

Data from Final Implementation:

n= 10, k=1: maximum= 9, avg= 9.00
n= 100, k=10: maximum= 275, avg= 215.37
n=10000, k=40: maximum= 12267, avg=11885.22
n=10000, k=100: maximum= 16107, avg=15441.85

Constant K:
n= 10, k=10: maximum= 38, avg= 30.32
n= 16, k=10: maximum= 70, avg= 55.66
n= 32, k=10: maximum= 139, avg= 101.70
n= 64, k=10: maximum= 213, avg= 164.67
n= 128, k=10: maximum= 322, avg= 259.25
n= 256, k=10: maximum= 483, avg= 416.70
n= 512, k=10: maximum= 803, avg= 706.12
n= 1024, k=10: maximum= 1336, avg= 1248.07
n= 2048, k=10: maximum= 2396, avg= 2303.71
n= 4096, k=10: maximum= 4522, avg= 4379.69
n= 8192, k=10: maximum= 8633, avg= 8508.44

Shaun McThomas
Ian Schweer
CS 165
Project 1: Selection
04/13/2016

n=10000, k=100: maximum= 16107, avg=15441.85

Constant n

n=10000, k=1: maximum= 9999, avg= 9999.00
n=10000, k=2: maximum= 10031, avg=10015.42
n=10000, k=4: maximum= 10111, avg=10070.78
n=10000, k=8: maximum= 10318, avg=10217.52
n=10000, k=16: maximum= 10744, avg=10574.88
n=10000, k=32: maximum= 11636, avg=11408.34
n=10000, k=64: maximum= 13709, avg=13231.47
n=10000, k=65: maximum= 13873, avg=13301.36
n=10000, k=66: maximum= 13856, avg=13357.97
n=10000, k=70: maximum= 14100, avg=13611.88
n=10000, k=99: maximum= 16025, avg=15392.15
n=10000, k=100: maximum= 16002, avg=15427.41

Analysis:

The strategy our team took was using a min heap that contains the maximum k elements of the dataset, the largest element being a leaf. We also used caching with linear probing to reduce the number of comparisons. Building the heap of k elements take k -1 compares in the best case and 2 k in worst case. Compare the remaining elements to the smallest element always requires n-k compares. If the value is less than the smallest of the max elements, it must replace the min element in the heap and the heap must be re-heapified. This can have zero time or as much as n-k time, each time requiring 2 to 2 $\log_2$ k compares each time. In building this heap all necessary compare to pop the elements have been seen and cached. Therefore, all the k log k compares have been swapped by cache lookups and are not counted. The compare complexity for the best case ended up being:

$$(k-1)+(n-k)$$

and on worst we have:

$$2k+(n-k)+(n-k)(2\log_2(k))$$

These are marginally off because the ideal circumstances to match these complexities rarely happen.

Running the Code:

Follow the Evaluation Process as describe in the project specifications.:

- Place your doalg.c file in his directory that contains the class-supplied files: MAIN.c and COMPARE.c
- Compile the program on Linux using the command: gcc MAIN.c -o MAIN
- Run the program using the command: `MAIN > output`

Shaun McThomas
Ian Schweer
CS 165
Project 1: Selection
04/13/2016

Other Notes:

- The caching implementation used a self-designed linear probing hash map. The hashing function was taken from http://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-that-accepts-an-integer-hash-key which is Knuth's multiplicative method.
- Other Implementations we coded are turned in under folder "Other Implementations"

Conclusion:

This problem has various solutions ranging from O(kn) to O(n log k). The best solution we found is to use a minimum heap to store the largest values then remove them from the heap to sort them. This has the best case performance of (k-1) +(n-k) and worst performance of $2k + (n-k) + (n-k)(2 \log_2 k)$. These extreme cases are very rare. The average case is believed to lie around $n + .5k\, n \log_2 k + k \log_2 k$. but imperial evidence show that it is actually much lower.