Shaun McThomas
Ian Schweer
CS 165
Project 2: Majority
04/13/2016

Introduction:

This problem is to implement a subroutine to determine the majority element in an array of two elements. Our subroutine could not access the raw data, and instead had to do a series of four element queries to determine the majority and minority elements based off the result of the query. It would either return 4,2 or 0 indicating all same elements, one different element, or even divide of elements respectively. The challenge was have the minimum number of average queries as possible.

Methodology:

To solve this problem, we looked first at what we could do for the brute force solution, and then we looked for optimizations. The naïve / brute force solution would be to take any group of four and get it's status. Calling this group the master group, we can then replace some element in the master group with another element and look at the new status. This would tell us what bin the new element belonged too. This method did not lend well to data where we had a tie however, or too one where there were not a consecutive group of four, which happened often enough. This method also lead to a $O(n-4)$ number of compares on average and worst.

We started to notice that we didn't really want the majority and minority groups, instead we just wanted to track two groups and their counts. Then whichever one was higher, we just wanted to return that. We also noticed that there was a natural divide in the data based on their query status, so we used that to come up with out next method.

We kept the idea of a master group, but broke them into equivalence classes. A all four group, three to one group, and rest group. The all four group we could easily by just swapping out one element and querying the resulting new set. The three to one group had a little more complicated logic. Depending on if we had a item in the all four group, we could compare it to that and determine the group of three to one's division easily. If we didn't have a group of four, then we would just do them against each other, claiming the first group as the master. The even divide, we simply counted two towards our bins, and we were set.

Data used for Analysis:

n= 10, max= 17, avg= 6.52
n= 20, max= 23, avg= 10.77
n= 200, max= 169, avg= 102.11
n= 2000, max= 1117, avg= 1017.37
n=10000, max= 5367, avg= 5121.65
n= 17, max= 23, avg= 8.90
n= 18, max= 27, avg= 10.53
n= 19, max= 28, avg= 11.44

Shaun McThomas
Ian Schweer
CS 165
Project 2: Majority
04/13/2016

Analysis:

Empirically, we can we had an average case of roughly (n/2), with some fluctuation that has to do with the probability of a certain sequence occurring during the runtime of the program. This will be further explained later.

Algorithmically, we start with n/4 compares to divide our dataset into groups of four. From there, we have to consider complexities within the groups.

All four groups:

Given m is the number of groups with four consecutive items; we compare all groups after the first group to the first group. We then either add four to our count of elements like group one, or against. This means we have an additional m – 1 queries. That means the worst and average case of this branch would $^{n}/_{4} + (m - 1)$.

Three to one groups:

This group was the most complicated. There were two stages: 1. Determining the representative bin item index. 2: Determining the bin. I will refer to the current group as c.

1.) This part was done the same regardless of other groups. What we would do is take the next element in the list, and substitute it into all the other positions of c. This would take at most 4 additional compares, although average case would be 3 $((1 * ¼) + (2 * ¼) + (3 * ¼) + (4 * ¼))$ assuming each item has equal probability of .25 of being the item that caused the group to be a four to one. We will perform this search until either we reached the last element, or we changed equivalence classes.
2.) Determining the bin requires us to compare too some master group. This is either a grouping of four, allowing us to only have one compare. However, if we have another group of 3, then we have do an additional query place the items. On worst case this will 5 compares, with the average case being closer to 3 or 4, using the distribution as above.

The worst case and average case of this process can fluctuate depending on the data. However, assuming that everything goes really bad, and we have only three to one groups, and all of them have the different element at the end, then the worst case will be $^{n}/_{4} + (4m + 1)$. The average case will be closer to $n/4 + 2m$ where m is the number of groups with a 3 to 1 division.

Even divide groups:

This grouping will incur no additional queries.

Shaun McThomas
Ian Schweer
CS 165
Project 2: Majority
04/13/2016

Conclusion:

So in total, our worst case theoretical number of comparisons will be because of all three to one groups. We first need 3 compares to setup the master group, every other time, we have (n/4) – 1 compares. Then we have the substitution checks, which at worst can take (4 * (n/4 -1)) compares, when the differing elements are always at the end. We then have a final 12 compares at the end for elements that aren't multiples of four. That brings our final complexity too

$$3 + \left(\frac{n}{4} - 1\right) + 4 * \left(\frac{n}{4} - 1\right) + 12 = \frac{5}{4}n + 10$$

The average case requires us to look at the probability of a certain sequences appearing. Our application ran 10000 times for each N, which means we would need at least 10 elements to ensure that we have all the possible scenarios happen (assuming the random number generation is iid). As n grows larger, we has less and less chance of a sequence occurring where we have all three to one groups with the final element in at the end. For example, when N = 20, that gives us 2^{20} possibilities. In that twenty spots, there are only 5 distinct spots where we can place a differing element. If we assume the sets won't alternate, then that gives us a probability of .000001 that we will get that exact sequence. That is very unlikely. It is also unlikely that we will have all the same elements. That means that we can expect there to be a nice distribution of cases where 1's and 0's are spread out throughout the dataset. That means we will probably have about 1/3 of the groups being ties, leaving us with

$$\frac{n}{4} + \frac{n}{4} + \left(\frac{n}{4} + 1\right) = \frac{3}{4}n$$

As N grows larger, and larger, we will even begin to see more cases of groupings of four. This explains all the discrepancies in our actually analysis.

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

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