CS 325 Group Assignment 1

Jacob Cheney, Brandon Lam, Philip Warton

October 13, 2020

1 Algorithm Description and Proof of Correctness

Our algorithm can be described as a divide and conquer algorithm, because we divide our problem size in half every time we recur. Let $D = [d_0, d_1, \cdots, d_{n-1}]$ be the array of delegates, where d_i is the party that the delegate at index i belongs to. If our array is longer than 1, we choose the midpoint of our array, $m = \lfloor \frac{n-1}{2} \rfloor$, and then think of our problem as two sub-arrays,

$$A = [d_0, \cdots, d_m] \qquad\qquad B = [d_{m+1}, \cdots, d_{n-1}]$$

We know that the majority party must have $m + 1$ delegates (assuming we do truncated integer division when computing m). By the pigeonhole principle, we must have more that $\frac{m+1}{2}$ delegates of our majority party in either A or B, meaning that the majority party of D will also be the majority party of A or B, or of both A and B.

Case 1: Our majority party dominates A or B, not both Let maj(D) be the majority party element of D. Then either maj$(D) =$ maj(A) or maj$(D) =$ maj(B). By recursion we can assume that the majority size and element of A and B are both known. We use a count function to count the number of occurrences of maj(B) in A, and the number of occurrences of maj(A) in B. This will call the `same_party()` function $m + m = n$ times total. Then we compare the count of maj(A) in D, to the count of maj(B) in D. Whichever is larger, we take to be the size of our majority party in D, and we know also which party is our majority.

Case 2: Our majority party dominates A and B In this case we check that maj(A) =maj(b), calling `same_party()` once and simply combine the size of the majority in each.

When the size of our array is 1, we know that maj(D) is going to be d_0, and that the size of the majority party is 1. This will be our base case for recursion.

2 Running Time Analysis

We can check the condition for the base case in constant time with a simple if statement, which starts us at $O(1)$ complexity. To get the information for our sub-arrays A and B, we make 2 recursive calls to our function with an input size of $\frac{n}{2}$. We will work to compute the cost of this further on. Assuming that we have the majority party, and its size for A and B, we make one call to `same_party()` to check if we have Case 1 or Case 2. This check is constant time, meaning excluding the recursive step, we still have $O(1)$ time complexity. Since Case 2 is constant time, our worst case scenario is one in which we have Case 1 each step of the way. Assuming that we have Case 1, we must call `same_party()` n times, as in our algorithm description. This means that our function will operate in linear time, $O(n)$, excluding the recursion.

To account for recursion, we must add $T(\frac{n}{2})$ to our running time twice, since we recur on both halves. This gives us $T(n) = O(n) + T(\frac{n}{2}) + T(\frac{n}{2}) = O(n) + 2T(\frac{n}{2})$. This will continue to branch out until $\frac{n}{2^i} = 1$. This will finally result in the following sum:

$$\begin{aligned}
T(n) &= O(n) + 2O\left(\frac{n}{2}\right) + 2^2 O\left(\frac{n}{2^2}\right) + 2^3 O\left(\frac{n}{2^3}\right) + \cdots + 2^{\log_2(n)} O\left(\frac{n}{2^{\log_2(n)}}\right) \\
&= O(n) + O(n) + O(n) + \cdots + O(n) \\
&= \log(n)\, O(n) \\
&= O(n \log(n))
\end{aligned}$$