Steven Bui, Ethan Ng, Alex Young
CS 325

Group Assignment 1

Description of Algorithm

For our algorithm, we applied divide and conquer to break the original problem into smaller sub-problems, similar to merge sort. Given an n amount of delegates, the algorithm recursively splits into two halves until the delegates are left in pairs or singles. The first time the split function is called, it takes in 0 as the lowest index and (n - 1) as the highest index. In the split function, if the sub-problem has three or more candidates, it will recursively call on split twice, splitting the sub-problem into a lower and upper half. This process continues until all delegates have been separated into pairs or singles. At this point, the two delegates of each pair are compared with the same_party function.

If the pair of indexes that meet have delegates from the same party, then the function will recursively pass on one of the index values with the total number of delegates that met. For example, if the function was just called, then a count of 2 would be returned as each delegate only knows that they themselves are of the party. On the other hand, if one of the delegates met someone else that had met 3 other delegates from the same party, and they themselves had met 2 same party delegates, then the one delegate that goes onward would have a count of 5.

If a single index or delegate is being checked for, it will be recursively passed on without changing the count. Since we can rely on one party having a majority of delegates, we can pass the odd delegate up by default and check it against the next matching pair of delegates without having to worry.

If the pair of indexes that meet are not from the same party, then the function will first update the count numbers to compare each index. If will then check if one delegate has a greater count of same party members that they have met. This greater count delegate will continue onward as well as check the previous pairs that met up with the lower count delegate to see if any were from the same party. If so, the count will be updated to reflect these same party members on the other side of the recursion tree. In the case where the two delegates from different parties have the same count, neither will be passed on in the recursive tree and instead an index that points to (-1) with a count of 0 will be returned.

If a delegate meets up with no one (an index of -1), it will pass itself and update its count after checking to see if there were any delegates that met up with this -1 index that were in the same party.

The split function returns two values. One is the index of a majority delegate and the other value is a count that keeps track of how many delegates inside the high and low indexes are of the majority party. By returning both of these values, we can check the majority party of one section of the index against another while keeping track of how many of a given political party there are for a comparison if the two parts have different majority groups. If the delegates don't share a party, -1 is returned as an index and 0 as the count, which signals that there isn't a match in the segment being examined. At the end, the majority count is passed through the majority_party_size function and is returned to the main function.

The count function takes in a low index, a high index and a delegate that you want to check against. It runs a while loop to check the desired delegate against the delegates between the low and high index. If the same_party function returns True, it adds 1 to the count variable. Regardless of result, 1 is added to the checking index. The result is the number of times the majority party appears between the low and high indexes, and the number is typically added to the running count of the majority party, or a party being checked against another majority party.

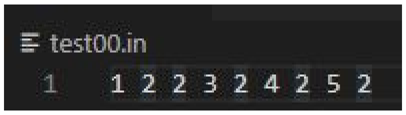
Runtime Analysis

$T(n) = 2T(\frac{n}{2}) + O(n) = O(nlogn)$

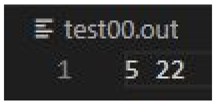
The split function is working with an n amount which is divided in half at every step so the number of handshakes that occurs is O(logn). The count function contains a while loop which has to iterate through delegates that shake hands so its runtime is O(n). Therefore, the total runtime (number of handshakes) for the algorithm is O(nlogn).

Correctness of Algorithm

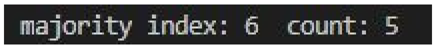
Example 1



```
test00.in
1 2 2 3 2 4 2 5 2
```



```
test00.out
5 22
```



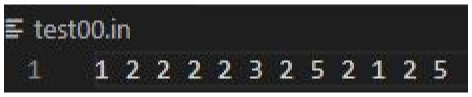
```
majority index: 6  count: 5
```

In this example we used the test input that was provided. Here it shows that the program was able to find that the 6th index had a majority party delegate in 22 same_party calls. It found that the count of the majority party was 5.

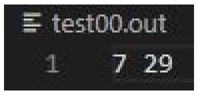
This is correct because index 6 has a delegate with party 2, and there are 5 total delegates with party 2.

This is in nlogn time because 22 < 28.

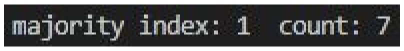
Example 2



```
test00.in
1   1 2 2 2 2 3 2 5 2 1 2 5
```



```
test00.out
1   7 29
```



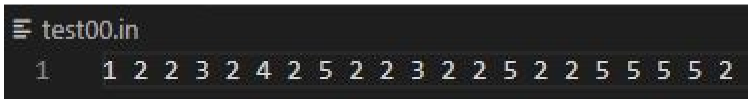
```
majority index: 1  count: 7
```

In this example we used a random test input with 12 total delegates and 7 delegates that have majority party 2. Here it shows that the program was able to find that the 1st index had a majority party delegate in 29 same_party calls. It found that the count of the majority party was 7.

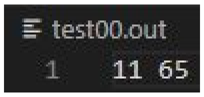
This is correct because index 1 has a delegate with party 2, and there are 7 total delegates with party 2.

This is in nlogn time because 29 < 43.

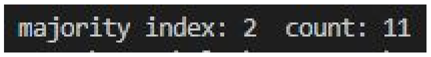
Example 3



```
test00.in
1   1 2 2 3 2 4 2 5 2 2 3 2 2 5 2 2 5 5 5 5 2
```



```
test00.out
1   11 65
```



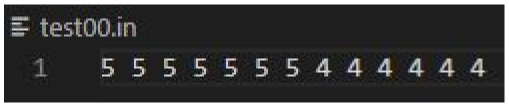
```
majority index: 2  count: 11
```

In this example we used a random test input with 21 total delegates and 11 delegates that have majority party 2. Here it shows that the program was able to find that the 2nd index had a majority party delegate in 65 same_party calls. It found that the count of the majority party was 11.

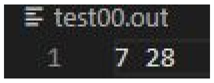
This is correct because index 2 has a delegate with party 2, and there are 11 total delegates with party 2.

This is in nlogn time because 65 < 92.

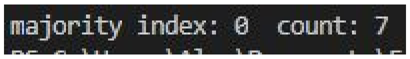
Example 4



```
test00.in
1   5 5 5 5 5 5 5 4 4 4 4 4 4
```



```
test00.out
7 28
```



```
majority index: 0  count: 7
```

In this example we used a random test input with 13 total delegates and 7 delegates that have majority party 5. Here it shows that the program was able to find that the 0 index had a majority party delegate in 28 same_party calls. It found that the count of the majority party was 7.

This is correct because index 0 has a delegate with party 5, and there are 7 total delegates with party 5.

This is in nlogn time because 28 < 48.