3

a.) There are a few scenarios that could occur

1. Both halves have the same majority - yes can infer

if t1 and t2 are true and e1 is equal to e2 then we can infer the solution for the whole vector since it is the majority for both halves

2. One half has a majority - potentially... but can't infer... must verify

if t1 is true but t2 isn't (or the other way around) the majority element from the half where t1 is true (e1) is a potential majority, but we have to go through the entire vector to be sure

3.) Different Majority Elements in each half - can't infer must verify

if t1 & t2 true but return different Stuff objects we must go through the entire vector and count occurences of both e1 & e2 to verify if there is actually a majority

4) No majority in either half - can infer no solution

no single element in either half so not going to be majority in full

4

b)

```
}
// Handle the case where the size of consecutivePairs is odd
// The last element can be considered an excess element
Stuff3 finalElement = extraOne;
if (consecutivePairs.size() % 2 != 0) {
    finalElement = consecutivePairs.at( n: consecutivePairs.size() - 1);
}
// Recurse with the new vector of pairs and the last unpaired element if
```

odd elements in vector means I can't be a pair so push that last element back to the consecutivePair vector as if it was a consecutive pair so that it can be considered

next page :) →

extra bonus

Version 1 O(n log n) proof

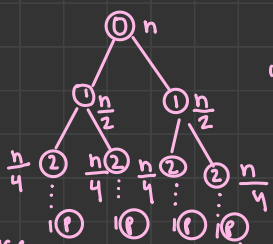
```cpp
#include <vector>
#include "Stuff3.h"

// you should modify the code for this one
// you can also change the parameters.
// If you do that. make sure you change that for the call from Decision1()
pair<bool, Stuff3> DecisionRecur1(vector<Stuff3>& vec, int firstElement, int lastElement)
{
    //Base Case: if length 1 return whatever element contained as potential majority
// you can change this line if you decide to change the parameters for DecisionRecur1
    if (firstElement == lastElement){
        return { x: true,  &: vec[firstElement]};
    }
// Find the middle
    int middleElement = firstElement + (lastElement - firstElement)/2;

//Recursive search for majority element in either half
    pair<bool, Stuff3>  leftSide = DecisionRecur1( &: vec,  firstElement,  lastElement: middleElement);
    pair<bool, Stuff3>  rightSide = DecisionRecur1( &: vec,  firstElement: middleElement + 1 ,  lastElement);
//If both halves have same majority element, return the element
    if (leftSide.first && rightSide.first && leftSide.second == rightSide.second){
        return rightSide;
    }
//If one side has a majority, check its validity over the entire segment
    //handle right side first
    int rightCount = 0;
    for (int i = firstElement; i <= lastElement; i++){
        if (rightSide.first && vec[i] == rightSide.second){
            rightCount++;
        }
    }
    if (rightCount > (lastElement - firstElement + 1)/ 2) {
        rightSide.first = true;
        return { &: rightSide.first,  &: rightSide.second};
    }
    //handle left
    int leftCount = 0;
    for (int i = firstElement; i <= lastElement; i++){
        if (leftSide.first && vec[i] == leftSide.second){
            leftCount++;
        }
    }
    if (leftCount > (lastElement - firstElement + 1)/ 2) {
        leftSide.first = true;
        return { &: leftSide.first,  &: leftSide.second};
    }
    //If not majority is found or its invalid return false
    return { x: false,  y: Stuff3()};
}
```

} → Recursion part takes $\log n$ operations to split

Splits recursively into left and right parts at depth 0 till vector



notice each step "level" takes $\frac{kn}{2}$ operations

assume p levels then total num operations

$= \frac{n}{2^p} = 1$

- at BASE CASE size of each subarray = 1

$p = 1$ since we split it up

$\frac{n}{2^p} = 1$

$n = 2^p$

$\log_2 n = p$ ∴ $O(\log n)$

Check over the entire vector segment to confirm majority

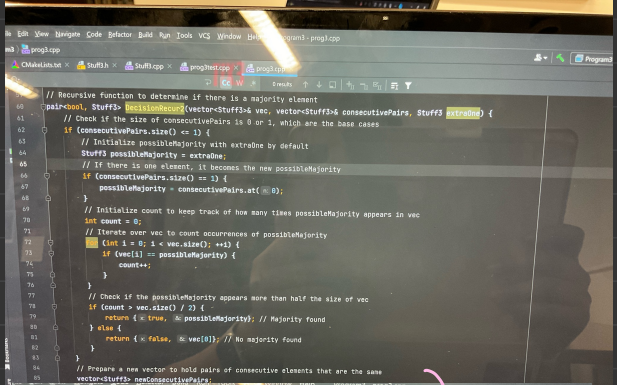
Outside of Recursive call performs a linear scan of the current segment. At each level the counting operation goes through elements in the vector once ∴ time complexity of $O(n)$ for non recursive parts

$O(\log n)$ at each recursive call and scans through at $O(n)$ as explained earlier resulting in an overall time complexity of $O(n) \cdot (\log n)$

∴ Overall time complexity $O(n \log n)$

O(n) time complexity proof

Initial operations to pass



```
// Recursive function to determine if there is a majority element
pair<bool, Stuff3> DecisionRecur2(vector<Stuff3>& vec, vector<Stuff3>& consecutivePairs, Stuff3 extraOne) {
    // Check if the size of consecutivePairs is 0 or 1, which are the base cases
    if (consecutivePairs.size() <= 1) {
        // Initialize possibleMajority with extraOne by default
        Stuff3 possibleMajority = extraOne;
        // If there is one element, it becomes the new possibleMajority
        if (consecutivePairs.size() == 1) {
            possibleMajority = consecutivePairs.at( n: 0);
        }
        // Initialize count to keep track of how many times possibleMajority appears in vec
        int count = 0;
        // Iterate over vec to count occurrences of possibleMajority
        for (int i = 0; i < vec.size(); ++i) {
            if (vec[i] == possibleMajority) {
                count++;
            }
        }
        // Check if the possibleMajority appears more than half the size of vec
        if (count > vec.size() / 2) {
            return { x true, & possibleMajority}; // Majority found
        } else {
            return { x false, & vec[0]}; // No majority found
        }
    }
    // Prepare a new vector to hold pairs of consecutive elements that are the same
    vector<Stuff3> newConsecutivePairs;
    // Iterate through consecutivePairs in steps of 2
    for (int i = 0; i < consecutivePairs.size() - 1; i += 2) {
        if (consecutivePairs.at( n: i) == consecutivePairs.at( n: i + 1)) {
            newConsecutivePairs.push_back(consecutivePairs.at( n: i));
        }
    }
    // Handle the case where the size of consecutivePairs is odd
    // The last element can be considered an excess element
    Stuff3 finalElement = extraOne;
    if (consecutivePairs.size() % 2 != 0) {
        finalElement = consecutivePairs.at( n: consecutivePairs.size() - 1);
    }
    // Recurse with the new vector of pairs and the last unpaired element if any
    return DecisionRecur2( & vec, & newConsecutivePairs, extraOne: finalElement);
}
// Function to kickstart the recursive decision-making process
pair<bool, Stuff3> Decision2(vector<Stuff3>& vec) {
    if (vec.empty()) {
        return { x false, & vec[0]}; // Handle empty vector case
    }
    // Initial extra element, possibly a default-constructed Stuff3 object
    Stuff3 extraOne;
    // Create a copy of vec to be used as consecutivePairs for recursion
    vector<Stuff3> consecutivePairs = vec;
    // Start recursion with the full vector
    return DecisionRecur2( & vec, & consecutivePairs, extraOne);
}
```

initial pairing goes through vector one time so time complexity of O(n)

IF there are multiple pairs with same element (or last odd element pushed back) then recursive call.

The recursive call reduces the vector by half each time

n

function checks only Half of current vector each time

$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} \cdots$ until base case of < 2

first call is $\frac{n}{2}$

$\therefore$ Overall time complexity is $O(n)$ for initial pairing and $O(n)$ for recursive vector calls

$\therefore$ overall time complexity is

$O(n+n) = O(2n) = \boxed{O(n)}$