James Sweetman (jts2939)
Jung Yoon (jey283)
Unique #51835

HW #4

1.

a. **Algorithm:** The algorithm will be very similar to mergesort except instead of sorting we will be comparing the molecules. The array will be split in the same fashion as mergesort. We will assume that majority means strictly greater than half of the array.
Two cases will occur:
    1) Array has size 1. Return the molecule because it is the majority
    2) Array has size 2, we do an ImgComp to compare the two molecules. If they are both the same, we return the molecule as the majority molecule. Otherwise we return that there is no majority molecule.

   From here on, we will have four checks on the two results of the two split arrays.
    1) Both return no majority molecule.
    2) Right side returns a majority molecule and left side doesn't.
    3) Left side returns a majority molecule and right side doesn't.
    4) Both return a majority element.

   If there is a majority element being returned, then it is a potential candidate to be the majority molecule and so we will loop through the combined array with counter(s) for the majority molecule(s). If the counter comes out to be a majority, then return the molecule as the majority molecule.
   This will continue until you hit the top level and either have a majority element or not.

b. **Correctness:**
   Base case: Let the number of merges be 0 and there be either 1 or 2 molecules in the subarray. Let the size be 1, then the majority of the element is the only element that is present. If the size is 2, then CompImg will return Yes if the two molecules are the same. If they are, then that is the majority molecule and it will be returned. Else "no majority" element will be returned.
   Inductive Hypothesis:
   Given an array with k molecules after n merges, the function MergeComp(k,n) will return the majority if it exists or return no majority if there is not one.
   Inductive Case:
   Assume M(n,c), we will prove M(2k, n+1). The reason for 2k molecules is because the number of entries in the combined array will be double the small array. The algorithm divides it in halves, so it will combine the halves.
   Let x be the sub array returned by M(n,c) and y be the other sub array returned by M(n,c). Then we have 4 situations:

1) No majority is returned by x and y. Since both are returning no majority, the combined will have no majority. This is because at most, an element will only be able to make up half of the combined array which does not fit the definition of being a majority provided by the algorithm.
2) Left is majority and right returns no majority. The combined array will either return a majority, being the majority from the left array, or no majority. The reason for overriding the majority with a no majority is if after counting through the array, it does not make up more than half of the other array.
3) Right is majority and left returns no majority. Same exact thing as scenario 2 but swap the subarrays.
4) Both return a majority. The correct majority (or no majority) will be returned because a count over the larger array will happen and keep track of which element has the most. If the most is greater than half then it will return that. If neither are greater than half then it will return no majority.

c. **<u>Runtime:</u>** At each level, two calls are made on the halves of the array. The non-recursive cost is looping through the array and comparing each element. This will occur at most twice, if there are two majority elements that were returned from a lower level in the recursion tree. Therefore the recurrence for this algorithm is $T(n) = 2T(n/2) + 2n$. Using the Master Thm. we find that the runtime is $O(n\log n)$.

d. **Extra credit:** In order to do it in one passing, we can use a variation of Moore's voting algorithm.

<u>2.</u>

a. Algorithm: In the else statement, determine what quadrant minVert is located in, call it q. Then return the min(minVert, FindLocalMinimum(q)).

b. Correctness:

c. Runtime: $T(n) = T(n/4) + 14n$. By master's thm., $T(n)$ is $O(n)$.

<u>3.</u>

a. **Proof by counterexample:**
Let's say that there is a sequence of nodes with weights n1= 9, n2=10, n3=9. Then, by the greedy algorithm given, we would pick the middle node, n2. On the other hand, the maximum weight independent set would be comprised of the n1 and n3 nodes with weights of 9. Thus, the algorithm does not always find an independent set of maximum total weight.

b. **Proof by counterexample:**
Let's say that there is a sequence of nodes with weights n1=7, n2=5, n3=6, n3=7. Then, by the algorithm given the n1 and n3 will be selected. However, the maximum independent set would be comprised of n1 and n4. Thus, the algorithm does not always find an independent set of maximum total weight.

c. Let’s say that we have an independent set s_i for each n_i from n_1. Let’s also assume that n_0 = 0 and n_1 = 1. That is, node_0’s weight is 0 and node_1’s weight is 1.

Then, for a node n_i whose weight greater than 1, either n_i belongs in the independent set s_i or it doesn’t. That is to say that there are two cases.

Knowing this, we can compute values of weights in ascending order from weight = 1 to weight = n. And since the weight that we ultimately want is n_n, we can just compute s_n, taking advantage of the two cases we mentioned before, by tracing back the computations and finding the max, depending on whether or not the node/weight in question belongs to the corresponding independent set.

Since this is just a variation of the greedy algorithm given in part A, we know that this algorithm is valid.

As for runtime, our algorithm computes in constant time per iteration and it’s over n iterations so we know that the runtime for this is O(n).