Question 1:

1.1 A

To find the smallest $|S_i - S_j|$ where $i \neq j$, use a red-black tree to store data. Then start at the left-most bottom node and do the following:

1. Compare left-child to parent $|S_{lc} - S_p|$ and store difference.
2. Compare right-child to parent $|S_{rc} - S_p|$ and compare to the stored difference, keep the lower of the two values.
3. Continue pattern for all parent children combos in red-black tree
4. Return the stored value.

1.2 B

To return this query in constant time we simply need to augment the red-black tree to hold an extra value. I have defined this value to be: smallest difference in a node's subtree. This way to find the smallest difference in our data-set we just extract the value from the root node. The algorithm would be the same but each difference would be compared to the internal difference value stored in a node. So:

1. Compare left-child to parent $|S_{lc} - S_p|$ and store difference in parent node.
2. Compare right-child to parent $|S_{rc} - S_p|$ and compare to the stored difference, replace value stored in parent if new comparison is smaller.
3. Continue pattern for all parent children combos in red-black tree, but if the stored smallest difference in a subtree is smaller than its parent's, then update the parent's to be equal to that of the child.
4. Return the stored value at the root.

To update the tree after its insertion would take $O(\log n)$ time since after the insertion we only need to percolate back up and update the parents keeping the smaller difference each time until we get back to the root.

1.3 C

To handle deletions, the process would be the same as in a red-black tree deletion but we would need to update the stored differences for the values affected by the deletion. In many cases this would just required one comparison, since once a child replaces its deleted parent, and assuming red-black properties hold, then you would only need to compare to the new child. For more complicated deletions involving rotations, the number of comparisons increases but for a left rotation you would only need to recheck the left-subtree and vice-versa for a right rotation.

Question 2:

To count the number of 'flips' in an array in $O(n \log n)$ time, first sort the array in descending order using an efficient sorting algorithm that completes the process in $O(n \log n)$ time, e.g. heapsort. After you have the sorted array, since it is in descending order, the number of flips is simply $\Sigma_{i=1}^{n-1} i$ where n is the size of the array. This sum is done in constant time so we have $O(n \log n)$ time.

Question 3:

To solve this problem, I will use the method described in the online lecture videos for LCS but modified to accommodate for the ability to insert/remove/replace. The algorithm would function in much the same way. Starting off we need an array of size m x n to store results of subproblems, this size represents the size of the two strings. Now we compare the last characters of each string to each-other, if they are the same then no editing is required and we move on, recursing on the two string with the last characters removed from each one and storing that result in our memo-array. If on the other hand the last two characters do not match, then we would recurse on each of the possible resolutions to make them match, i.e. insert or replace or remove. However, since we only care about the least number of changes in order to turn one string into another we wrap the recursive calls in a min function so we would have min(remove, replace, insert). In our array we would store $1 + min$(remove, replace, insert). The one represents whichever change is made. At each recursion call we check the cell associated with that call to see if we have already solved this problem to avoid repeating ourselves. Continuing in this way, the cell at m x n in our memo-array would contain our final answer.

This dynamic solution would solve this problem in $O(m \cdot n)$ time since we are making sure that we are never repeating subproblems by using memoization.