Problem 3

We start approaching the problem by first sorting the first k elements and then making a max heap for the first k/2 elements and a min heap for next k/2 elements in the *optimalCost* function. As a result, for the lowest cost, all elements will converge to the lowest element of the min heap, which is our median. The cost is then estimated and the new element is added to the min heap or max heap based on the median. Similarly, the first element of the array must be removed in order for our window to maintain the same size as before. Now, if the size of the min heap and max heap are distorted, we restore them to their original sizes with the help of transferring an element.

Pseudo Code

```
// assuming operations available to us for min heap and max heap
/*size; // size of Heap
sum; // sum of all elements of heap
insert(x) // insert in heap
findMin()//minimum element of heap
extractMin()/extractMax()
//delete and return minimum element of heap
heapify(x) //delete x from heap */

balancing(minHeap h1, maxHeap h2, k) {
    if (h2.size > k / 2)
        h1.insert(h2.extractMax());
    else if (h2.size == k / 2)
        return;
    else
        h2.insert(h1.extractMin());

    return;
}

costing(minHeap h1, maxHeap h2, k) {
    median = findMin(h1);
    c1 = (median * k) / 2 + h2.sum;
    c2 = h1.sum - median * (k - (k / 2));
    return c1 + c2;
}



optimalCost(B, n, k) {
    for (i = 0 : k - 1)
        temp[i] = B[i];

    mergeSort(A);
    minHeap h1; maxHeap h2;


    for (i = (k / 2) - 1 : k - 1)
        h1.insert(A[i]);

    for (i = 0 : (k / 2) - 1)
        h2.insert(A[i]);

    res = costing(h1, h2, k);

    for (i = k : n - 1) {
        median = findMin(h1);

        if (B[i] >= median)
            h1.insert(B[i]);

        else
            h2.insert(B[i]);

        if (B[i - k] >= median)
            h1.heapify(B[i - k]);
        else
            h2.heapify(B[i - k]);

        balancing(h1, h2, k);

        res = min(costing(h1, h2, k), res);
    }

```

```
    return res;

}
```

Time Complexity

Merge Sort function is the only function in the algorithm which takes $\mathcal{O}(n \log n)$ time. All the other functions in the algorithm atmost take $\mathcal{O}(\log n)$, which basically are the operations like *insert* for the minHeap and maxHeap. So, our overall time complexity is $\mathcal{O}(n \log n)$

Proof of Correctness

- Let x, y be 2 numbers such that we have $x < y$. Consider a 3rd number z, such that z lies between the numbers x and y. It is obvious that $x < z$ and $z < y$. The cost incurred to converge or move to z is $(y - z) + (z - x)$ which is equal to $y - x$. Hence, it does not depend on the number z at all. So, from this we can clearly see that we get the optimal convergence when z is between x and y.
- So, start by pairing the kth element from start and end. If k is odd, we can directly say that all of the formed pairs will converge on the middle elements. If the value of k is even all pairs will converge one of the most middle pair value. We may assume that the cost measured for k elements is minimum because the min element of the min heap is either the middle element or one of the elements of the middle pair.
- At the end we took min of of all the k array costs. Hence, we can definitely say that our algorithm works correctly.

Problem 4

The approach to problem involves use of the *Greedy Paradigm*. We first calculate the maxLength (denoted by *maxLen* in algo) substring of S' starting at position *i* and which is present as a subsequence in S for each element of S' (*for(i=0:m-1)*). We then take the sum for all the *m* values, that is the length of the max of sub strings present in S' which occur as a sub sequence in S. Therefore, *maxLen* stores max length substring and we finally return *m-maxLen*.

Pseudo Code

```
count(S, Sdash) {

    maxLen < - 0;

    for (i = 0 : m-1) {
        currentPos < - i;
        for (j = 0 : n-1) {
            if (Sdash[currentPos] == S[j])
                currentPos++;
        }

        maxLen < - max(currentPos - i, maxLen);

    }

    return m - maxLen;
}
```

Time Complexity

For each start point S'[i] (*Sdash[i]*), the max length sub-sequence can be found in $\mathcal{O}(n + m)$. Here, *i* can take *m* values. So overall complexity since both the loops are nested turns out to be $\mathcal{O}(n * (n + m))$

Proof of Correctness

- We are allowed to add numbers only at beginning and end of S, so he numbers are added so that S' becomes a substring of S. We can clearly observe that some contiguous part of S' must be a subsequence of S if the numbers added to S are disregarded. So, to minimize the numbers to be added to S, we need to be maximize the contiguous part of S' matching S and when this maximization occurs the numbers not present in this contiguous part are added. *Thus, we can clearly see that finding the max length subsequence of S' will minimize the number of elements to be added.*

- Let us consider that S'[i] is the starting point for our max length subsequence which we aim to find. We start the process by finding first instance of starting point(S'[i]) in S, call it S[k]. We then look for S'[i+1] in the portion of S between k to n-1. Now we have done this for one single value of i. The proposed algorithm simply runs for all possible values of i. Using the greedy approach we find the immediate next instance of S'[i] in S, and thus we will be able to maximize the length of *max length subsequence*.