Problem 1:

1) What is a recurrence that describes step ii) of the algorithm?

$$T(n) = \begin{cases} d & \text{if } n=1 \\ 2.T(n/2) + dn & \text{if } n>1 \end{cases}$$

2) What is the asymptotic complexity of the algorithm? Provide an answer for both the best case (sub-lists equal size) and worst case (one sub-list has all but one symbol). No proof is required ?

Complexity: Best Case O(nlogn) Worst Case: O(nlogn)

3) No the algorithm does not always generate an optimal prefix code. The example is {1,2,3,4,5} for which the algorithm requires 1.2 bits more for encoding compared to the Huffman algorithm which is proved to be optimal. Hence it is proven that this algorithm is sub-optimal otherwise it would require the same number of bits as Huffman (as Huffman is optimal).

Problem 2:

Algorithm:

1) To find the point which minimizes the sum of distance to the store is a intersection point which is the center point. This is the definition of a point which is the median point.
2) To solve this problem in O(n) time we have to find the median in O(n) time.
3) We have used the Median of Medians algorithm to find the median point.
4) We stored X and Y coordinates in 2 different arrays in O(n) time.
5) Then applied the median of median algorithm on the 2 subarrays separately. This is again O(n) time for each array of X and Y.
6) To find the distance we looped through the data each time adding the distance from previous iteration to the distance of the current iteration in O(n) time.

Complexity:
Input: O(n)
Median of Median for X: O(n)
Median of Median for Y: O(n)
Output: O(n)
Total: O(n)

Problem 3:

Algorithm:

1) Divide the jobs into two separate arrays based on employer 1 and employer 2

2) Sort the individual arrays based on their finish time
3) Assign the jobs :
    a) Base case - The first job added to the solution will be either of first two jobs of individual array with the one having least finish time for first job.
    b) Check if the next job from employer array 1 has start time > finish time of last added job and both have different employers. If yes , then add it to solution Else check the same for employer array 0.
    c) Increase the pointer in both the arrays.
    d) Repeat till all the jobs have been seen in both the arrays.

Complexity Analysis:

Step -1 takes - O(n) time
Step -2 takes - O(n log n)
Step -3 takes - O(n) time

Total Time complexity = $O(n \log n)$

Proof of correctness :

The algorithm described divided the input into arrays , each consisting of jobs from respective employers. Both the arrays are then sorted based on their finishing time.
We initialize two pointers for both the arrays and start with adding the job from employer which has least finishing time. After that, we check each job from both the arrays to be optimal with previous job added to solution and check if two consecutive jobs are not from same employer. In this way , we cover all the jobs and each new job is just compared with the previously added job.

Problem 4:

Algorithm: For this problem, we can just extend the idea of knapsack problem with one knapsack. For two knapsacks we can create a 3D array[x][y][z]
X- number of items
Y - weight of knapsack 1
Z - weight of knapsack 2

Heart of the algorithm --

1) S[i][v1][v2] = max value in knapsack given access to items 1...i with v1 units of weight Available in knapsack 1 and v2 units of weight available in knapsack 2.

2) Solution -- S[n][w1][w2]

3) Recurrence -- S[i][v1][v2] = max(S[i-1][v1][v2], s[i-1][v1- w_i][v2] + $Cost_i$,
s[i-1][v1][v2- w_i] + $Cost_i$)

Complexity Analysis:
Three loops are run to compute each cell of the dynamic programing array. Hence, complexity

Total Time complexity = $O(n\ w1\ w2)$

Problem 5:

Algorithm: For this problem, we directly apply the longest increasing subsequence algorithm. This is done by using Dynamic Programming approach and a recursive approach.

Heart of the Algorithm:

1) S[k] = Length of longest increasing subsequence given access to elements 1….k and requiring a_k is part of the longest increasing subsequence.
2) Solution: $Max_k S[k]$
3) Recurrence: S[k] = 1 + S[j] where j is the index that maximizes S[j] such that j<k and $a_j < a_k$.

Complexity:

1) The complexity of the Dynamic programming approach is $O(n^2)$.
2) The complexity of the Recursive approach is $O(2^n)$.

Running time comparisons in nanoseconds(ns).

N	Dynamic Programming	Recursive Algorithm
Input-1 10	5105	56159
Input-2 10	18963	47043
Input-3 20	8387	4925996
Input-4 30	41573	1735788259
35	17504	66305009833

40	36468	> 3 minutes
50	99556	> 3 minutes
100	129459	> 3 minutes
150	743567	> 3 minutes
1000	4408160	> 3 minutes
Input-5 10,000	212253941	> 3 minutes

For a value of about n=40, the recursive problem has a time which exceeds 3 minutes. The Dynamic programming approach works for input of all sizes.