Okay, let's tackle these problems step-by-step.

**A. [6.0 Points] Solving the Recurrence Relation using Iterative Substitution**

We have the recurrence:  
T(n) = 7T(n/2) + cn², for n > 1, and T(1) = 1

Let's expand the recursion:  
T(n) = 7 \* T(n/2) + cn²  
T(n/2) = 7 \* T(n/4) + c(n/2)²  
T(n/4) = 7 \* T(n/8) + c(n/4)²

Substitute the second equation into the first:  
T(n) = 7 \* [7 \* T(n/4) + c(n/2)²] + cn²  
T(n) = 7² \* T(n/4) + 7c(n²/4) + cn²

Substitute the third equation:  
T(n) = 7² \* [7 \* T(n/8) + c(n/4)²] + 7c(n²/4) + cn²  
T(n) = 7³ \* T(n/8) + 7²c(n²/16) + 7c(n²/4) + cn²

After *k* iterations, we get:  
T(n) = 7ᵏ \* T(n/2ᵏ) + c \* n² \* [7⁰ + 7¹/4¹ + 7²/4² + ... + 7ᵏ⁻¹/4ᵏ⁻¹]

We stop the recursion when n/2ᵏ = 1, which implies that k = log₂(n)

Thus, T(n) = 7^log₂(n) \* T(1) + cn² \* Σ[ (7/4)ⁱ ] from i = 0 to log₂(n) - 1

Since T(1) = 1:  
T(n) = n^log₂(7) + cn² \* Σ[ (7/4)ⁱ ]

The summation is a geometric series. Because the ratio (7/4) is greater than 1, the series is dominated by its last term.

Σ[ (7/4)ⁱ ] ≈ O((7/4)^log₂(n))  
= O(n^log₂(7/4))

n^log₂(7/4) = O(n^0.8), which is less than O(n²) so the geometric series is dominated by n² term.

So, we have two terms n^log₂(7) and cn². n^log₂(7) is greater than n² because log₂(7) ≈ 2.8, so we can conclude that the dominant part is n^log₂(7).

Therefore, **T(n) = Θ(n^log₂(7))**

**B. [11 Points] Merging m Sorted Lists in O(n log m) Time**

A min-heap (priority queue) can efficiently solve this problem.

**Algorithm:**

1. **Initialization:**
   * Create a min-heap.
   * Insert the first element from each of the m sorted lists into the min-heap along with the index of the list.
2. **Iteration:**
   * While the min-heap is not empty:
     + Extract the smallest element and its corresponding list index from the heap.
     + Add the extracted element to the final result list.
     + If there are more elements in that list, insert the next element from that list into the min-heap.

**Time Complexity Analysis:**

* Insertion of initial m elements: O(m log m)
* Extracting n elements and potentially inserting another element from the lists: O(n log m)
* Total: O(m log m) + O(n log m) = O((n+m) log m). If n > m then we can write it as **O(n log m)**

**C. [3.0 Marks] Huffman Algorithm Time Complexity**

The Huffman algorithm builds an optimal prefix code. Here's the breakdown:

1. **Frequency Counting:** O(n), where *n* is the total number of characters.
2. **Min-Heap Construction:** O(k log k), where *k* is the number of unique characters.
3. **Tree Construction:** O(k log k), because we have k-1 merging steps, and each merging step takes O(log k).
4. **Code Generation:** O(k), which is the number of nodes in the Huffman Tree.

Total Complexity: O(n) + O(k log k) + O(k log k) + O(k) = **O(n + k log k)**.  
In most cases, k is less than or equal to n, thus the algorithm has O(n log n) as the worst time complexity scenario.

**D. [8.0 Marks] MST Weight in a Simple Graph with Unit Edge Weights**

In a simple graph with *n* vertices where all edges have a weight of 1, any MST will have exactly *n*-1 edges.

Since each edge contributes 1 to the total weight, the weight of the MST is simply the number of edges in the MST: (n-1) \* 1= **n-1**.

**E. [11.0 Marks] Finding Minimum Absolute Sum Pair**

We are given an array *S* with *k* integers. The goal is to find two numbers *x*, *y* such that |*x* + *y*| is minimized.

**Algorithm:**

1. **Sort:** Sort the array *S* in ascending order. O(k log k).
2. **Two Pointers:** Initialize two pointers: left at the beginning (index 0) and right at the end (index k-1).
3. **Iterate and Update:**
   * Calculate the absolute sum: abs\_sum = abs(S[left] + S[right]).
   * Initialize min\_abs\_sum = abs\_sum, x = S[left], and y = S[right].
   * Move the pointers according to the sum:
     + If the sum (S[left] + S[right]) is positive, decrement right.
     + If the sum is negative, increment left.
   * Update min\_abs\_sum, x, and y if a smaller abs\_sum is found.
   * Continue until left crosses right.
4. Return x,y

**Time Complexity Analysis:**

* Sorting: O(k log k)
* Iteration using two pointers: O(k)
* Total Time Complexity: O(k log k) + O(k) = **O(k log k)**

**F. [11 Points] Computing In-Degree and Out-Degree from Adjacency Lists**

With an adjacency list representation of a graph:

1. **Initialization:**
   * Create two arrays: in\_degree and out\_degree, initialized to 0 for all vertices.
2. **Iteration:**
   * Iterate through each vertex u in the adjacency list.
     + For each neighboring vertex v in u's list:
       - Increment out\_degree[u] (because *u* has an outgoing edge to *v*).
       - Increment in\_degree[v] (because *v* has an incoming edge from *u*).

**Time Complexity Analysis:**

* Initialization: O(V)
* Iterating through lists: We go through all the edges in the graph which takes O(E) time, as adjacency list represent the edges of a graph
* Total: **O(|V| + |E|)** where V is the set of vertices and E is the set of edges

If you'd like any part explained further, just ask.