1. (25 points) Below is a weighted undirected graph. (a) Show the DFS (depth-first search) tree found by the non-recursive DFS algorithm and list the vertices in the order of adding to the DFS tree. (b) Show the MST (minimum spanning tree) found by Prim’s algorithm and list the vertices in the order of adding to the MST. (c) Compute the shortest paths from vertex A to all other vertices using Dijkstra’s algorithm and list the vertices in the order of adding to the cloud. For all the above questions, we start with vertex A and ties are broken by alphabet order of vertices.

**Answer:** (a) DFS in the left figure, the order is: A, B, E, F, G, C, H, D.
(b) Prim’s MST in the right top figure, the order is: A, E, B, C, D, G, F, H
(c) Dijkstra’s shortest paths in the right bottom figure, the order: A, E, B, C, D, F, G, H

2. (15 points) Let S = [.5, .3, .7, .4, .6, .5, .4, .6] be an instance of the Bin Packing Problem. Please display the solutions of First Fit (FF), Best-Fit (BF), First Fit Decreasing (FFD), Best Fit Decreasing (BFD), and Next Fit methods. If one method produces the optimal solution for S, then construct a counter example to show that method cannot always produce an optimal solution.

**Answer:** FF uses 5 bins: { [.5, .3], [.7], [.4, .6], [.5, .4], [.6] };
BF uses 5 bins: \{ [.5, .4], [.7, .3], [.6, .4], [.5], [.6] \};
FFD uses 4 bins: \{ [.7, .3], [.6, .4], [.6, .4], [.5, .5] \}, which is optimal;
BFD uses 4 bins: \{ [.7, .3], [.6, .4], [.6, .4], [.5, .5] \}, which is optimal;
NF uses 5 bins: \{ [.5, .3], [.7], [.4, .6], [.5, .4], [.6] \};

Counter example to FFD and BFD: \{ .5, .4, .4, .3, .2, .2 \}, which has an optimal solution of
\{ .5, .3, .2 \}, \{ .4, .4, .2 \}, but both FFD and BFD produce \{ .5, .4 \}, \{ .4, .3, .2 \}, \{ .2 \}.

3. (30 points) Given a list \( L = [a_1, a_2, ..., a_n] \) of integers, the longest non-increasing subsequence problem is to find the length of longest non-increasing subsequences of \( L \), denoted by \( \text{LNIS}(L) \). For example, if \( L = [7, 7, 8, 2, 6, 1] \), then \( [7, 7, 2, 1] \) is a longest non-increasing subsequence of \( L \) and \( \text{LNIS}(L) = 4 \). Please design an efficient algorithm to compute \( \text{LNIS}(L) \) and analyze its complexity.

**Answer:** From the class, we have an algorithm \( \text{LCS}(X, Y) \), which returns the length of the longest common subsequence of \( X \) and \( Y \). We may use \( \text{LCS}(X, Y) \) to compute \( \text{LNIS}(L) \):

Sort \( L \) into \( Y \) in non-increasing order, then \( \text{LNIS}(L) = \text{LCS}(L, Y) \). This is because a common subsequence \( Z \) of \( L \) and \( Y \) is also a subsequence of \( L \) and \( Z \) is non-increasing because \( Y \) is non-increasing. The complexity of this algorithm is the complexity of sorting \( L \) plus the complexity of \( \text{LCS}(L, Y) \), which is \( O(n \log(n) + n^2) = O(n^2) \), where \( n = |L| \).

An alternative solution is to compute \( \text{LNIS}(L) \) directly: For \( 1 \leq j \leq n = |L| \), define \( \text{lnis}(j) \) to be the length of the longest non-increasing subsequence of \( L \) including and ending in \( L[j] \). Obviously, \( \text{lnis}(j) \geq 1 = \text{lnis}(1) \). If \( i < j \) and \( L[i] \geq L[j] \), then \( \text{lnis}(j) \geq \text{lnis}(i) + 1 \). Actually, \( \text{lnis}(j) = \max\{\text{lnis}(i) + 1 | i < j, L[i] \geq L[j] \} \).

So the pseudo code is:

```plaintext
Proc LNIS(L)
  For j from 1 to |L| do
    lnis[j] = 1;
    For i from 1 to j-1 do
      If (L[i] \leq L[j]) lnis[j] = max(lnis[j], lnis[i] + 1);
    i = 1;
    For j from 2 to |L| do i = max(i, lnis[j]);
  Return i
```

The complexity of \( \text{LNIS}(L) \) is \( O(n^2) \).

4. (30 points) We wish to append strings in \( S = \{ s_i | 1 \leq i \leq n, |s_i| = a_i \} \) into a single string. Suppose \( w = \text{append}(u, v) \) takes time \( O(|u|+|v|) \) to append strings \( u \) and \( v \) into \( w \). To append all strings in \( S \) into one string, we may use the following algorithm:

```plaintext
appendAll(S) { // assume S = { L_i | 1 \leq i \leq n }, |S| = n
  while (|S| > 1) {
    S' = if (|S| \% 2 == 1) then \{ L_{|S|} \} else \{ \};
    S = append(S', S);  // append S' to S
  }
  Return S
}
```
for (i = 1; i < |S|; i = i+2)  \( S' = S' \cup \{ \text{append}(L_i, L_{i+1}) \} \)
\( S = S' \); }}

(a) Analyze the complexity of appendAll(S) in terms of n and \( a_i = |s_i| \) for \( 1 \leq i \leq n \).
(b) Construct a counter example S showing appendAll(S) is not optimal.
(c) Design an efficient algorithm to solve this problem for the general input.

If we replace “append” by “merge” and a string is replaced by a list of numbers, then this problem is a continuation of the appendAll problem from the first midterm, where mergeAll is shown to be optimal when all \( L_i \) have the same length.

(a) Let \( A = \sum_{1 \leq i \leq n} a_i \), where \( a_i \) is the size of original \( L_i \). \( n = |S| \) is halved after each loop. So the body of the while loop will execute \( \log(n) \) times. The total cost of each for loop is \( O(A) \). So the total complexity is \( O(A \log n) \).

(b) Suppose S contains 4 strings of lengths 1k, 1k, 1k, 100k, respectively, where k is an integer. The algorithm appendAll will take 2 iterations and the total cost is \( 2 \times (103k) = 206k \). If we append the first two strings, and then append with the third string of size 1k, the cost will be \( 2k + 3k \). Finally, we append the result from the first three strings with the last string of 100k, the cost is \( 103k \). The total cost is \( 2k + 3k + 103k = 106k \). So appendAll takes almost twice the cost comparing to the cumulative append.

(c) When \( L_i \) have various lengths, the optimal method is to borrow the idea from the Huffman code: append two shortest strings into one, until the total number of strings become one.

```plaintext
greedyAppendAll(S) {
    Q = createQueue(S); // create the min-queue from S, using |L_i| as the key.
    while (Q.size > 1) {
        L1 = removeMin(Q);
        L2 = removeMin(Q);
        insert(Q, append(L1, L2));
    }
}
```

Let \( d_i \) be the number of times that the original \( L_i \) is involved in the append operation. Since the queue operations removeMin and insert take \( O(\log n) \) time, the total complexity of greedyAppendAll is \( O(n \log n + \sum_{1 \leq i \leq n} d_i a_i) \). Using the proof for Huffman’s coding, this is optimal for all the algorithms using append. When all \( a_i \) are equal, \( d_i = \log n \), and the total cost is the same as \( O(A \log n) \).