CS3330  Algorithms
Midterm Exam (100 points)
Closed books and notes (except two sheets of notes)
April 26, 2018

1. (25 points) Below is a weighted undirected graph. (a) Show the DFS (depth-first search) tree found by the 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, C, G, E, D, B, F, H.
(b) Prim’s MST in the middle figure, the order is: A, D, E, H, B, C, G, F
(c) Dijkstra’s shortest paths in the right figure, the order is: A, D, C, E, B, H, G, F

[Graphs and trees shown]

2. (15 points) Let S = [.5, .7, .3, .4, .5, .3, .6, .3, .4] 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, .3], [.4, .5], [.6, .3], [.4] };
BF uses 5 bins: { [.5, .4], [.7, .3], [.5, .3], [.6, .3], [.4] };
FFD uses 4 bins: { [.7, .3], [.6, .4], [.5, .5], [.4, .3, .3] }, which is optimal;
BFD uses 4 bins: { [.7, .3], [.6, .4], [.5, .5], [.4, .3, .3] }, which is optimal;
NF uses 5 bins: { [.5], [.7, .3], [.4, .5], [.3, .6], [.3, .4] };
Counter example to FFD and BFD: [0.5, 0.4, 0.4, 0.3, 0.2, 0.2], which has an optimal solution of 
{ [0.5, 0.3, 0.2], [0.4, 0.4, 0.2] }, but both FFD and BFD produce 
{ [0.5, 0.4], [0.4, 0.3, 0.2], [0.2] }.

3. (30 points) Given a list \( L = [a_1, a_2, \ldots, a_n] \) of integers, the longest non-decreasing subsequence problem is to find the length of longest non-decreasing subsequences of \( L \), denoted by \( \text{LNDS}(L) \). For example, if \( L = [2, 2, 1, 7, 3, 8] \), then \([2, 2, 7, 8]\) is a longest non-decreasing subsequence of \( L \) and \( \text{LNDS}(L) = 4 \). Please design an efficient algorithm to compute \( \text{LNDS}(L) \) 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 \( \text{LNDS}(L) \): Sort \( L \) into \( Y \) in non-decreasing order, then \( \text{LNDS}(L) = \text{LCS}(L, Y) \). This is because a common subsequence \( Z \) of \( L \) and \( Y \) is also a subsequence of \( L \) and \( Z \) is non-decreasing because \( Y \) is non-decreasing. 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 \( \text{LNDS}(L) \) directly: For \( 1 \leq j \leq n = |L| \), define \( \text{lnds}(j) \) to be the length of the longest non-decreasing subsequence of \( L \) including and ending in \( L[j] \). Obviously, \( \text{lnds}(j) \geq 1 = \text{lnds}(1) \). If \( i < j \) and \( L[i] \leq L[j] \), then \( \text{lnds}(j) \geq \text{lnds}(i) + 1 \). Actually,
\[
\text{lnds}(j) = \max \{ \text{lnds}(i) + 1 \mid i < j, L[i] \leq L[j] \}.
\]
So the pseudo code is:

**Proc LNDS(L)**
- For \( j \) from 1 to \( |L| \) do
  - \( \text{lns}[j] = 1; \)
  - For \( i \) from 1 to \( j-1 \) do
    - If \( (L[i] \leq L[j]) \) \( \text{lns}[j] = \max(\text{lns}[j], \text{lns}[i] + 1); \)
    - \( i = 1; \)
  - For \( j \) from 2 to \( |L| \) do \( i = \max(i, \text{lns}[j]); \)
- Return \( i \)

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

4. (30 points) Let \( S = \{ L_i \mid 1 \leq i \leq n, L_i \text{ is a sorted list of numbers} \} \), and for each list \( L_i \), let \( a_i = |L_i| \) be the number of elements in \( L_i \). We are given a function \( \text{merge}(L, L') \) which takes time \( O(|L| + |L'|) \) to merge \( L \) and \( L' \) into one sorted list, and returns the result. To merge all lists in \( S \) into one sorted list, we may use the following algorithm:

**mergeAll(S)**
- // assume \( S = \{ L_i \mid 1 \leq i \leq n \} \), \( |S| = n \)
- while \( (|S| > 1) \) {
  - \( S' = \text{if} \ (|S| \% 2 == 1) \ \text{then} \ \{ L_{|S|} \} \ \text{else} \ \{ \}; \)
  - for (\( i = 1; i < |S|; i = i+2 \) ) \( S' = S' \cup \{ \text{merge}(L_i, L_{i+1}) \} \)
  - \( S = S'; \)
}

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

This is a continuation of the $\text{mergeAll}$ problem from the first midterm, where $\text{mergeAll}$ 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 lists of sizes 1k, 1k, 1k, 100k, respectively, where $k$ is an integer. The algorithm $\text{mergeAll}$ will take 2 iterations and the total cost is $4 \times 103k = 206k$. If we merge the first two lists, the cost is $2k$, we then merge the result of the first merge with the third list of size 1k, the cost is $3k$. Finally, we merge it with the list of 100k, the cost is $103k$. The total cost is $2k + 3k + 103k = 106k$. So $\text{mergeAll}$ takes almost twice the cost comparing to the cumulative merge.

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

```
    greedyMergeAll(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, merge(L1, L2));
        }
    }
```

Let $d_i$ be the number of times that the original $L_i$ is involved in the merge operation. Since the queue operations removeMin and insert take $O(\log n)$ time, the total complexity of greedyMergeAll 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 merge. When all $a_i$ are equal, $d_i = \log n$, and the total cost is the same as $O(A \log n)$.