Foundations of Algorithms, Spring 2018: Homework 3

Due: Wednesday, March 7, 11:59pm

Problem 1 (8 points)

Consider the following algorithm for generating a binary tree representing a prefix code. Given a collection of symbols and their corresponding frequencies:

i) Sort the symbols by frequency in ascending order.

ii) Call the following recursive function, passing in the sorted list of symbols and frequencies. The function returns the root of the tree.

   (a) If the list contains only a single symbol, return that symbol as the root of the tree.

   (b) Else determine an index at which to cut the list into first and second sub-lists. This cut index should have the property that among all possible cut points, this one yields the minimum difference between the cumulative frequency of the two individual sub-lists.

   (c) Make a copy of the first sub-list.

   (d) Make a copy of the second sub-list.

   (e) Make recursive calls to generate the roots of the left and right sub-trees, in each case passing in the corresponding sub-list. Attach the root nodes of the left and right sub-trees to a parent node and return that parent node.

For example, given sorted frequencies: $\{10, 11, 15, 18, 22, 24\}$, the algorithm first splits the list into $\{10, 11, 15, 18\}$ and $\{22, 24\}$, having cumulative frequencies 54 and 46, respectively. The difference $|54 - 46| = 8$ is the minimum difference achievable by any partition. Continuing from there, the algorithm splits $\{10, 11, 15, 18\}$ into $\{10, 11\}$ and $\{15, 18\}$, as $|21 - 33| = 12$ is the minimum difference achievable when partitioning that list into two sub-lists. All lists of size two get split into two lists of size one.

Answer the following questions related to the algorithm. Assume there are n input symbols.

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

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.

3. Does this algorithm always generate an optimal prefix code? Either provide a proof that the algorithm always generates an optimal prefix code, or else provide a counter-example for which the algorithm does not generate an optimal prefix code, clearly arguing why the code is sub-optimal.

Problem 2 (12 points: 8 for implementation / 4 for writeup)

In a little town the roads are all laid out in a perfect grid. There is a North-South road every block. There is an East-West road every block. At every intersection, there is a traffic light. In order to make sure people obey the traffic lights, there are police cars stationed at some of the intersections. You have decided to open a corner donut store at one of the intersections, and need to identify the most profitable location for your store. The most profitable location will be the one that minimizes the sum of the distances that the traffic police have to travel to reach your store. The police must travel by using the existing roads (they can not travel diagonally).

You are given a collection of integer coordinates $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$ that represent the locations of traffic police. Give an $O(n)$ algorithm that determines the best corner location (given by (x_{best}, y_{best})) for your donut store. That is, (x_{best}, y_{best}) should be chosen such that $\sum_{i=1}^{n} |x_{best} - x_i| + |y_{best} - y_i|$ is as small as possible, and (x_{best}, y_{best}) should also be integer coordinates.

Problem 3 (12 points: 8 for implementation / 4 for writeup)

Consider the following variant of the interval scheduling problem. You are a contractor and want to complete as many jobs (intervals) as you can. You do work for two different employers. In order to keep both employers happy, you must alternate doing jobs for the employers. You can choose either employer for the first job, but must make sure never to do two consecutive jobs for the same employer.

Given are n intervals (jobs). The i^{th} interval is specified by (s_i, f_i, e_i), where the interval runs from starting time s_i to finishing time f_i, and the employer is e_i. Design an $O(n \log n)$ algorithm that determines the maximum number of compatible jobs that you can complete while keeping both employers satisfied (i.e. while alternating employers). Note that if job j has a finish time that is equal to the start time of job k, jobs j and k are considered compatible.

Problem 4 (15 points: 10 for implementation / 5 for writeup)

Consider a 2-backpack version of Knapsack. Given are two backpacks of capacities W_1 and W_2. Given are also n items $(w_1, c_1), (w_2, c_2), \ldots, (w_n, c_n)$, where w_i is the weight and c_i the cost of the i-th item. We want to find a set of items that can be split into two parts: one that fits in the first backpack and the other in the second backpack, and the sum of their costs is the largest possible. All the numbers are positive and W_1, W_2, and all the w_i's are integers. Give an $O(nW_1W_2)$ algorithm that finds the largest possible cost.

Problem 5 (15 points: 9 for implementation / 6 for writeup)

Given is a sequence of numbers $a_1, a_2, a_3, \ldots, a_n$. We say that $a_{j_1}, a_{j_2}, \ldots, a_{j_k}$ is a *subsequence* of this sequence if and only if $1 \leq j_1 < j_2 < \ldots < j_k \leq n$. The subsequence is *increasing* if and only if $a_{j_1} < a_{j_2} < \ldots < a_{j_k}$. In the *longest increasing subsequence* problem we search for an increasing subsequence of $a_1, a_2, a_3, \ldots, a_n$ that is the longest possible (i.e., k is as large as possible).

For example, for sequence 8,2,5,7,3,4,9,10, the longest increasing subsequences are 2,3,4,9,10 and 2,5,7,9,10 - either of them is a valid longest increasing subsequence.

This problem is about the longest increasing subsequence. You will implement a recursive approach and a dynamic-programming-based approach and compare their running times. In both cases we are interested only in finding the length of the sequence, not the sequence itself.

- Implement the following recursive approach. Implement the function `incrSubseqRecursive`(j, A), that computes the maximum length of an increasing subsequence of the sequence $a_1, a_2, \ldots, a_n$ (stored in the array A) that ends with the element a_j. This function tries to find a previous element a_i such that $i < j$ and $a_i < a_j$, and then it recursively searches for the maximum length of an increasing subsequence of $a_1, a_2, \ldots, a_i$. It tries up to $j - 1$ different i's and it chooses the one that gives rise to the maximum length. By concatenating a_j to the subsequence of $a_1, a_2, \ldots, a_i$ ending with a_i, we get a longest increasing subsequence of $a_1, a_2, \ldots, a_j$.
- Implement the dynamic programming approach to the longest increasing subsequence problem.
- Generate about 10 inputs for different values of n (how large can n be?) and report your observations on the running times of the two respective algorithms.

Note: Heart of the Algorithm

For any problem in which you develop a dynamic program (in this or any future homework!), to explain how your algorithm works (the correctness argument), describe the "heart of the algorithm" (you do not need to include any other explanation). Recall that the heart of the algorithm consists of three parts:

- Precise verbal description of the meaning of your dynamic programming array; see the slides for examples.
- A mathematical formula that describes how to compute the value of each cell of the array (using previously computed array values).
- Return value of the algorithm.

Note that the discussion of the complexity of your algorithm is separate and must be included as well.