Problem 9-1 (Uninterrupted Television) 14 Points

Little Johnny is extremely fond of watching television. His parents are off for work for the period \([S, F]\), and he wants to make full use of this time by watching as much television as possible; in fact, he wants to watch TV non-stop the entire period \([S, F]\). He has a list of his favorite \(n\) TV shows (on different channels), where the \(i\)-th show runs for the time period \([s_i, f_i]\), so that the union of \([s_i, f_i]\) fully covers the entire time period \([S, F]\) when his parents are away.

(a) (10 points) Little Johnny doesn’t mind switching to a show that has already started running, but is very lazy to switch the TV channels, and so he wants to find the smallest set of TV shows that he can watch, and still stay occupied for the entire period \([S, F]\). Design an efficient \(O(n \log n)\) greedy algorithm to help Little Johny. Do not forget to carefully argue the correctness of your algorithm, using either the “Greedy Always Stays Ahead” or the “Local Swap” argument.

(b) (4 points) Assume now that Little Johnny will only watch shows from the beginning till end (except the shows starting before \(S\) or ending after \(F\)), but now he fetches another TV from the adjacent room, so that he can potentially watch up to two shows at a time. Can you find a strategy that will give the smallest set of TV shows that he can watch on the two TVs, so that at any time throughout the interval \([S, F]\) he watches at least one (and at most two) shows. (Hint: Try to examine your algorithm in part (a).)

Problem 9-2 (Arithmetic vs. Geometric) 8 points

Design optimal Huffman codes for the following frequencies \(f_0, \ldots, f_7\). In each case, draw the Huffman tree incrementally, until you arrive at your final solution. After you finish, which Huffman code is more “balanced”: “arithmetic” or “geometric”?

(a) (4 points) Arithmetic: \(f_i = 10 + i\), for \(i = 0 \ldots 7\).

(b) (4 points) Geometric: \(f_i = 10 \cdot 2^i\), for \(i = 0 \ldots 7\).

Problem 9-3 (List Merging) 20 Points

Consider the problem of merging \(k\) sorted lists \(L_1 \ldots L_k\) of sizes \(n_1, \ldots, n_k\), where \(n_1 + \ldots + n_k = n\). We know that using a priority queue of size \(k\), we can implement this merge it time \(O(n \log k)\) (by repeatedly extracting smallest element from priority queue and replacing it by the next element of the list it came from).
Here we will design an alternative algorithm which repeatedly finds two sorted lists and merges them, so that after \( k - 1 \) merges of two lists we are left with a single sorted list. Assume merging two lists of size \( \ell_1 \) and \( \ell_2 \) takes time \( \ell_1 + \ell_2 \) (irrespective of the actual elements inside the lists). We would like to find the order of the \( k - 1 \) merges which minimizes the total cost.

E.g., when \( k = 3 \), we have three choices depending on which two lists we merge first. If we start with \( L_1 \) and \( L_2 \) (costing \( n_1 + n_2 = n - n_3 \)), and then merge the result with \( L_3 \) (cost \( (n_1 + n_2) + n_3 = n \)), we pay \( (n - n_3) + n = 2n - n_3 \). Similarly, if we start with \( L_2 \) and \( L_3 \) (costing \( n_2 + n_3 = n - n_1 \)), and then merge the result with \( L_1 \) (cost \( (n_2 + n_3) + n_1 = n \)), we pay \( (n - n_1) + n = 2n - n_1 \). Finally, if we start with \( L_1 \) and \( L_3 \) (costing \( n_1 + n_3 = n - n_2 \)), and then merge the result with \( L_2 \) (cost \( n_1 + n_3 + n_2 = n \)), we pay \( (n - n_2) + n = 2n - n_2 \). Thus, the best cost achievable is \( \min(2n - n_1, 2n - n_2, 2n - n_3) = 2n - \max(n_1, n_2, n_3) \), which means the we should exclude the largest list from the first merge (i.e., merge the two smallest lists first).

(a) (3 points) The first (naive) hope is that the order of the merges does not matter “too much”.

For any \( k < n \), give an example of \( k \) inputs \( n_1 \ldots n_k \) summing to \( n \), and a really poor choice of the merge order, so that the total cost of the merges is \( \Theta(nk) \). I.e., for \( k > \log n \) this is much worse than simply sorting the \( n \) total numbers from scratch!

(b) (4 points) Represent any valid order of \( (k - 1) \) merges as a binary tree, where the \( k \) leaves are labeled by the \( k \) initial lists with sizes \( n_1 \ldots n_k \), and every merge of two lists corresponds to creating a parent node of size equal to the sum of the two lists (children) just merged. Given a particular tree (i.e., order of merges), write the total cost of all the merges as a function of \( n_1 \ldots n_k \) and the depths \( d_1 \ldots d_k \) of the initial \( k \) lists (leaves) in this tree.

(c) (4 points) Consider the Huffman code problem with \( k \) characters \( c_1 \ldots c_k \), where frequency of \( c_i \) is \( f_i = n_i/n \). Using part (b), argue that the optimal tree (order or merges) for the list merging problem is identical to the optimal tree (i.e., prefix-free code) for the Huffman code problem.

(d) (4 points) Based on part (c), develop an optimal greedy algorithm for the list merging problem. Express the running time of the list merging solution (not just determining the order or merges, but also the merges themselves!) as the function of \( n, k \) and \( V \), where \( V \) is the optimal solution value for the Huffman code problem introduced in part (b). Do not forget to count the time use to actually solve the Huffman code problem!

(e) (4 points) Prove that \( V \leq \log k \) (think of one solution which is always an option),\(^1\) and substitute \( V = \log k \) into the formula you got in part (d). How does it compare with the original \( O(n \log k) \) solution?

\(^1\)It turns out that one can prove a much tighter bound on \( V \): \( V \in [H, H+1] \), where \( H = \sum_{i=1}^{k} \frac{n_i}{n} \log_2(\frac{n_i}{n}) \) is called the entropy of the probability distribution \( (\frac{n_1}{n}, \ldots, \frac{n_k}{n}) \). Moreover, for many “skewed” distribution, \( H, V \ll \log k \).