CSCI-GA.1170-001/002 Fundamental Algorithms November 24, 2015

Solutions to Problem 4 of Homework 9 (20 Points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, November 24*

Consider the problem of merging k sorted lists $L_1 \dots L_k$ of sizes $n_1, \dots, n_k$, where $n_1 + \dots + n_k = n$. 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 ℓ_1 and ℓ_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) (4 pts) The first (naive) hope is that the order of the merges does not matter "too much". For any $k < n$, given an example of k inputs $n_1 \dots n_k$ 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!

Solution:

Let $k = 8$ and $n_1 = 1, n_2 = 2, n_3 = 3, n_4 = 4, n_5 = 5, n_6 = 6, n_7 = 7, n_8 = 8 \implies n = 36, k > \log n$. Merge lists in the descending order i.e start merging n_8 with n_7 then with n_6 and so on. Therefore the total cost of merging these k lists will be $(15+21+26+30+33+35+36) = 196 = \Theta(nk)$ $\square$

(b) (4 pts) 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 $n_1 \dots n_k$, 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 $n_1 \dots n_k$ and the depths $d_1 \dots d_k$ of the initial k lists (leaves) in this tree.

Solution:

Without loss of generality assume that the merge order is $n_1, n_2, n_3, \dots, n_k$. In the first step when n_1, n_2 are merged, the cost is $n_1 + n_2$. Now we have a new list of size $n_1 + n_2$ that will be merged with n_3. So the total cost until now is $(n_1 + n_2) + (n_1 + n_2 + n_3)$. After $k-1$

merges the the cost will be $(n_1 + n_2) + (n_1 + n_2 + n_3) + \ldots + (n_1 + n_2 + \ldots + n_k)$.Notice that n_i would have occurred exactly d_i times in this summation. Therefore the total cost of merging these k lists will be $\sum_{i=1}^{k} n_i d_i$ $\square$

(c) (4 pts) 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.

Solution:

In the optimal tree for the Huffman code, first we merge the two characters with the least frequencies from all the given set of characters i.e if c_k, c_p are the first two characters to be merged then $f_k \leq f_p < \ldots = n_k \leq n_p < \ldots$ and then form a new alphabet c_{kp} whose frequency is $f_k + f_p$ and add this new alphabet to the existing set of alphabets.

In the optimal tree for the merging problem, first we merge the two lists with the least number of elements from all the given set of lists i.e if L_k, L_p are the first two lists to be merged then $n_k \leq n_p < \ldots$ and then form a new list L_{kp} whose size is $n_k + n_p$ and add this new list to the existing set of lists.

So there is a one to one correspondence between these two problems, therefore the optimal tree for the list merging problem is identical to the optimal tree for the Huffman code problem $\square$

(d) (4 pts) 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!

Solution:

Maintain a pointer for every leaf node to it's corresponding array and when any two leaf nodes are merged into a single node, this merged node contains a pointer to the merged array.

Proceeding this way up until the root, we finally get the resulting list formed by merging all the given k lists. This is identical to Huffman code problem and additionally at every step we are also merging two lists. Therefore the running time will be running time of Huffman code problem (with k leaf nodes $= k \log k$) and time taken to merge all the k lists

From part(b), we know that the cost of all the merges is $\sum_{i=1}^{k} n_i d_i = n \sum_{i=1}^{k} f_i d_i = nV$ where $V = \sum_{i=1}^{k} f_i d_i$. Hence the total running time is $O(k \log k + nV)$ $\square$

(e) (4 pts) 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?

Solution:

If the tree is a perfectly balanced binary tree then $d_i = \log k \;\; \forall i = 1$ to k
$\implies V = \frac{1}{n}\sum_{i=1}^{k} n_i d_i = \frac{\log k}{n}\sum_{i=1}^{k} n_i = \log k$
If the tree is not perfectly balanced i.e like the tree in Problem-3, part(b), i.e $d_i = i$, then $V = \frac{1}{n}\sum_{i=1}^{k} n_i d_i = \frac{1}{n}\sum_{i=1}^{k} n_i i \leq \log k$.
Therefore $V \leq \log k$. Substitute this value of V in part (d), we get the running time
$T(n) = O(k \log k + n \log k) = O(n \log k)$ i.e the greedy algorithm is not asymptotically significant than the original solution $\square$

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