Solutions to Practice Problems 4

Problem 1. This problem is about demonstrating the limitations of the greedy algorithm for interval scheduling. In this problem, suppose you are optimizing over total time of scheduled intervals, rather than number of intervals.

1. Devise a counterexample where the "earliest finish first" greedy algorithm fails to find the optimal value.
2. Suppose instead that at every step you schedule the job that takes the longest time. Show how this greedy algorithm fails to find an optimal value with a specific counterexample.
3. Show that scheduling the job that takes the longest time at every step can be no more than 3 times worse than the ideal solution.

Answers:

1. Consider two jobs, Job 1 with interval $[0, 2)$ and Job 2 with interval $[0, 1)$. The "earliest finish first" algorithm schedules Job 2, since it finishes first, and then cannot schedule Job 1, which conflicts with it. The resulting schedule has total time 1, while the optimal schedule consists of Job 1 alone and has total time 2.
2. Consider three jobs: Job 1 with interval $[0, 2)$, Job 2 with interval $[3, 5)$, and Job 3 with interval $[1, 4)$. The "longest job first" strategy schedules Job 3, of length 3, first. Job 3 conflicts with both other jobs, so the final schedule has total time 3. The optimal schedule consists of Jobs 1 and 2, with total time 4.
3. Let $j_1, \ldots, j_r$ be the jobs chosen by the greedy algorithm, with intervals $[a_i, b_i)$ and lengths $\ell_i = b_i - a_i$. For each i define the enlarged interval $I_i = [a_i - \ell_i,\ b_i + \ell_i)$, which has length $3\ell_i$.

   We claim that every job j of an optimal schedule lies entirely within some I_i. If j were not scheduled by greedy, it must conflict with some greedy job (otherwise greedy would have added it); let j_k be the longest greedy job that conflicts with j. If j is not contained in I_k, then j extends more than ℓ_k beyond j_k on one side, so j is longer than j_k. But then, when greedy chose j_k, the job j was still available (every greedy job chosen earlier is at least as long as j_k, and among those, the ones conflicting with j would contradict the choice of j_k), and greedy would have chosen j instead of j_k. Hence $j \subseteq I_k$. If j was scheduled by greedy, it is some j_i and lies in I_i.

   The jobs of the optimal schedule are pairwise disjoint, so the total length of those contained in I_i is at most $|I_i| = 3\ell_i$. Summing over i, the total time of the optimal schedule is at most $\sum_i 3\ell_i$, which is three times the total time of the greedy schedule.

Problem 2. This problem concerns a greedy approximation algorithm for the load balancing problem. In this problem, you are given m machines and k jobs, and each job takes t_i time. You may schedule any job on any machine, and your goal is to minimize the total makespan, or the longest time taken by any machine for its jobs. You will be analyzing a greedy load-balancing algorithm which runs over the jobs in any order, and always adds the next job to the machine with the shortest individual makespan.

1. Prove that an optimal solution to the load balancing problem has makespan at least $\frac{1}{m}\sum_{i=1}^{k} t_i$
2. Prove that the greedy load balancing algorithm produces a solution which has makespan at most $2 \cdot O$, where O is the optimal makespan. (Hint: at the moment the last job is inserted, consider what the state of the algorithm must be.)

Answers:

1. Any solution schedules $\sum_{i=1}^{k} t_i$ total time worth of jobs across m machines. If every machine had load strictly less than $\frac{1}{m}\sum_{i=1}^{k} t_i$, the total load would be strictly less than $\sum_{i=1}^{k} t_i$, a contradiction. Hence in every solution, and in particular in an optimal one, some machine has load at least $\frac{1}{m}\sum_{i=1}^{k} t_i$.
2. We show that the greedy algorithm never raises the load of a machine above $2 \cdot O$. Consider the moment some job j, of length t_j, is assigned to a machine M. The algorithm chooses the machine with the smallest current load, so the load of M is at most the average load, which is at most $\frac{1}{m}\sum_i t_i \leq O$ by part 1. Moreover $t_j \leq O$, since the optimal solution must place job j somewhere. Thus after the assignment the load of M is at most $O + O = 2 \cdot O$. Since every machine's final load was set by such an assignment, the greedy makespan is at most $2 \cdot O$.

Problem 3. In this problem, you will prove the relationship between Huffman Coding and Stone Merging. For reference, in Stone Merging, you are given n piles of stones with sizes $a_1, \cdots, a_n$, and must merge them into a single pile by merging two piles at a time. The cost of merging two piles is equal to the sum of the sizes of those piles, and you seek to merge them all with the lowest total cost.

A *merge tree* for a series of merges is a rooted binary tree built as follows: Each initial pile is a leaf of the tree, and each merge creates a new internal node which has two edges to the piles it is merging.

1. Draw the merge tree for the piles $2, 3, 5, 7, 11$ using the "always merge the two smallest piles" strategy, recording each move and the total cost.
2. A **full binary tree** is a binary tree such that every internal node has exactly two children. Show that every full binary tree with n leaves corresponds to a merge sequence, and that every merge sequence produces a full binary tree.
3. For a tree T and a leaf v in T, let $d_T(v)$ be the depth of leaf v in T, where the root has depth 0. Prove that the total cost of a merge sequence is equal to $\sum_v a_v \cdot d_T(v)$, summed over the leaves v of the corresponding merge tree.
4. Let $x_1, \ldots, x_n$ be the symbols of an alphabet, where symbol x_i occurs with frequency $a_i > 0$ in a text to be compressed. A **binary code** assigns to each symbol x_i a binary string $c(x_i)$, called its *codeword*; a text is encoded by concatenating the codewords of its symbols. The code is a **prefix code** if no codeword is a prefix of another codeword. For example, $\{0, 10, 11\}$ is a prefix code, but $\{0, 01, 10\}$ is not, since 0 is a prefix of 01. The *cost* of a prefix code is the total length of the encoded text, $\sum_{i=1}^{n} a_i \cdot |c(x_i)|$, and an **optimal prefix code** is one of minimum cost.

   (i) Suppose the code is not a prefix code, say $c(x_1) = 0$, $c(x_2) = 01$, and $c(x_3) = 10$. Give an encoded string that can be decoded in more than one way. Then explain why, for a prefix code, an encoded string can always be decoded unambiguously by reading it from left to right.

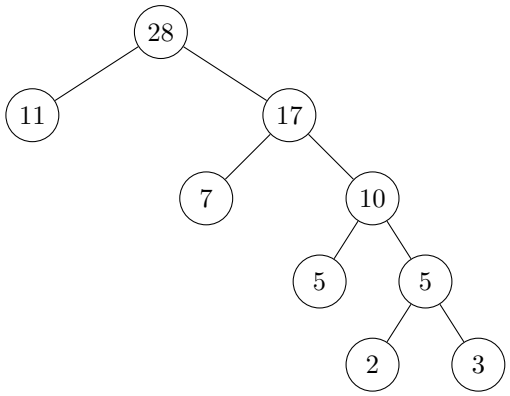
   (ii) Consider four symbols x_1, x_2, x_3, x_4 with frequencies $a_1 = 45$, $a_2 = 13$, $a_3 = 12$, $a_4 = 30$, and the prefix code $c(x_1) = 0$, $c(x_2) = 110$, $c(x_3) = 111$, $c(x_4) = 10$. Compute the cost of this code. How does it compare to the cost of the fixed-length code that assigns every symbol a two-bit codeword?

   A prefix code can be represented by a binary tree whose leaves are the symbols: each left edge is labeled 0 and each right edge 1, and the codeword of a symbol is the sequence of labels on the path from the root to its leaf. For example, a leaf reached by going left, then right, then left has codeword 010. Since every symbol is a leaf, no codeword is a prefix of another, and the length of a codeword equals the depth of its leaf.

   (iii) Using the previous parts, devise an algorithm that computes an optimal prefix code for given frequencies $a_1, \ldots, a_n$, and justify its correctness. Check that it produces a code of the same cost as the one in part (ii).

Answers:

1. The merges are $2+3=5$ (cost 5), then $5+5=10$ (cost 10), then $7+10=17$ (cost 17), and finally $11+17=28$ (cost 28), for a total cost of $5+10+17+28=60$. The merge tree is shown below, with each internal node labeled by the size of the merged pile.



2. Consider the merge tree of a merge sequence. Every internal node corresponds to a merge of two piles and therefore has exactly two children, so the merge tree is a full binary tree.

   Conversely, given a full binary tree with n leaves, assign the piles to the leaves and process the internal nodes bottom-up, merging the two piles corresponding to the children of each internal node. Since every internal node has exactly two children, each step merges exactly two piles, and this yields a merge sequence whose merge tree is the given tree.

3. We argue by induction on the number of leaves. A single pile has merge cost 0 and its leaf has depth 0, so the claim holds. Now let T be the merge tree of a sequence whose final merge combines the piles corresponding to subtrees T_1 and T_2. By induction, the cost of the merges within T_i is $\sum_{v\in T_i} a_v \cdot d_{T_i}(v)$. The final merge costs the sum of all leaf sizes, $\sum_v a_v$, and every leaf is one level deeper in T than in its subtree, so

$$\text{cost}(T) \;=\; \sum_{v\in T_1} a_v \cdot d_{T_1}(v) + \sum_{v\in T_2} a_v \cdot d_{T_2}(v) + \sum_v a_v \;=\; \sum_v a_v \cdot d_T(v).$$

4. (i) The string 010 can be decoded as $0\,|\,10 = x_1x_3$ or as $01\,|\,0 = x_2x_1$, so the code is ambiguous.

   For a prefix code, read the encoded string from left to right. As soon as the bits read so far form a codeword $c(x_i)$, the intended codeword must be $c(x_i)$: any other codeword consistent with these bits would have $c(x_i)$ as a prefix, which the prefix property forbids. Hence each codeword is determined as soon as it is complete, and the whole string decodes in exactly one way.

   (ii) The cost is $45\cdot 1+13\cdot 3+12\cdot 3+30\cdot 2=45+39+36+60=180$. The fixed-length code assigns two bits to each symbol, for a cost of $2\cdot(45+13+12+30)=200$. The prefix code saves 20 bits by giving the most frequent symbol the shortest codeword.

   (iii) *Algorithm.* Treat the frequencies $a_1,\ldots,a_n$ as pile sizes and run the greedy merging strategy of part 1, always merging the two smallest piles. Take the resulting merge

tree as the code tree: label left edges 0 and right edges 1, and read off each symbol's codeword as the path from the root to its leaf. (This is Huffman's algorithm.)

Correctness. First, some optimal prefix code corresponds to a full binary tree: if a node of a code tree has only one child, contracting that node shortens the codewords below it without changing any other codeword or the prefix property, and does not increase the cost. By part 2, every full binary tree with n leaves is the merge tree of some merge sequence, and conversely. By part 3, the cost of a prefix code, $\sum_i a_i \cdot |c(x_i)|$, equals the cost of the corresponding merge sequence, since $|c(x_i)|$ is the depth of leaf x_i. Therefore an optimal prefix code corresponds to a minimum-cost merge sequence, and the greedy strategy, which is optimal for stone merging, produces an optimal prefix code.

Check. On the frequencies $45, 13, 12, 30$ the greedy merges are $12+13 = 25$, $25+30 = 55$, and $45 + 55 = 100$, with total cost $25 + 55 + 100 = 180$, matching the cost of the code in part (ii). Indeed, the resulting tree is exactly the tree of that code.