Consider the following greedy algorithm for the Activity-Selection problem. Select the activity $a_i$ with the shortest duration $d_i = f_i - s_i$. Commit to scheduling $a_i$. Let $S'_i$ consist of all activities $a_j$ which do not overlap with $a_i$: namely, either $f_j \leq s_i$ or $f_i \leq s_j$. Recursively solve Activity-Selection on $S'_i$, scheduling the resulting activities together with $a_i$.

(a) (2 pts) Give a simple example of the input where the proposed greedy algorithm fails to compute the correct optimal solution.

Solution: **************** INSERT YOUR SOLUTION HERE ************

(b) (4 pts) Consider the following attempt to nevertheless justify the correctness of this algorithm using the “Greedy Always Ahead” method. Given a solution $Z$, let $F_i(Z)$ be the sum of the $i$ shortest activities scheduled by $Z$, and $\infty$ if $|Z| < i$. Try to tell exactly where the claim “For any $i$ and $Z$, $F_i(Z) \geq F_i(\text{greedy})$” fails. Is it base of induction? Or, if in the inductive step, for what smallest $i$ does the transition from $i$ to $i+1$ fails? Justify your answer.

Solution: **************** INSERT YOUR SOLUTION HERE ************

(c) (4 pts) Consider the following alternative attempt to justify the correctness of this algorithm using the “Local Swap” method. Given a solution $Z$, try to argue that it is always safe to substitute one activity in $Z$ by the first activity of the greedy algorithm, and then argue that the resulting recursive subproblems are the same for the “perturbed” opt and the greedy. Where does your argument run into problem? Is it the swap, or the recursive subproblem part?

Solution: **************** INSERT YOUR SOLUTION HERE ************

(d) (Extra credit) (6 pts) Let $t_{opt}$ be the size of the optimum solution and $t_{bogus}$ be the size returned by the bogus greedy algorithm discussed. Argue that $t_{bogus} \geq t_{opt}/2$. To do this, argue that the first activity $a$ scheduled by the bogus greedy algorithm overlaps at least one and at most two activities scheduled by the correct optimum algorithm.

In case it overlaps two activities $a_1$ and $a_2$, argue that the recursive subproblem resulting from scheduling $a$ has the optimum value at least as large as the one resulting from excluding $a_1$ and $a_2$ from opt. (Then use induction on the size of $t_{opt}$.)

In the the case of only one activity $a_1$, argue that you can find at most one more activity $a_2$ in opt such that the recursive subproblem resulting from scheduling $a$ has the optimum value at least as large as the one resulting from excluding $a_1$ and $a_2$ from opt. (Then use induction on the size of $t_{opt}$.) How to you find $a_2$ if you need it? (This is tricky.)

Solution: **************** INSERT YOUR SOLUTION HERE ************
You have \( n \) CDs in your library labeled 1, \ldots, \( n \). You would like to arrange them on your CD rack in linear order \( \pi(1), \ldots, \pi(n) \), according to some permutation \( \pi \) on \( n \) numbers. After such an arrangement is finalized, the cost to access CD \( i \) is equal to \( \pi(i) \), as you need to scan through the first \( \pi(i) \) CDs until CD \( i \) is found. Given a sequence of \( k \) CD requests \( r_1, \ldots, r_k \in \{1 \ldots n\} \), your job is to figure out the permutation \( \pi \) (i.e., an array \( A \) such that \( A[i] = \pi(i) \)) having the smallest total access cost \( \sum_{j=1}^{k} \pi(r_j) \).

(a) (4 pts) Describe in English a greedy algorithm for this problem.

Solution: ****************** INSERT YOUR SOLUTION HERE ****************** 

(b) (4 pts) Prove the correctness of your algorithm using the local-swap argument.

Solution: ****************** INSERT YOUR SOLUTION HERE ****************** 

(c) (4 pts) Write the pseudocode for implementation which runs in time \( O(n + k) \). (Hint: At some point use counting sort.)

Solution: ****************** INSERT YOUR SOLUTION HERE ******************
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 pts) Arithmetic: $f_i = 10 + i$, for $i = 0 \ldots 7$.

Solution: ******************* INSERT YOUR SOLUTION HERE *******************

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

Solution: ******************* INSERT YOUR SOLUTION HERE *******************
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 in 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) (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 $n_1 \ldots n_k$ 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!

Solution: ***************** INSERT YOUR SOLUTION HERE ************

(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 $n_1 \ldots n_k$, 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.

Solution: ***************** INSERT YOUR SOLUTION HERE ************

(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: ***************** INSERT YOUR SOLUTION HERE ************

**** INSERT YOUR NAME HERE ****, Homework 9, Problem 4, Page 1
(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: **************** INSERT YOUR SOLUTION HERE ***************

(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: **************** INSERT YOUR SOLUTION HERE ***************

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