Problem 1 (25 Points)

You are given n jobs, where each job j has its processing time p_j and weight w_j. We need to process all of them on one machine. We use C_j to denote j's completion time in a schedule. Our goal is to find the best schedule that minimizes the average weighted completion time (i.e., $\min \frac{\sum_{j=1}^{n} w_j C_j}{n}$). Design a greedy algorithm for it.

Solution. First, for each job j, we define ratio r_j as $\frac{p_j}{w_j}$. Then, we sort all the jobs in ascending order according to r_j and schedule these jobs in this order.

Correctness. Suppose the optimal schedule is $a_1, a_2, \ldots, a_n$. For the sake of contradiction, we assume the output of our algorithm is not optimal and there exists two adjacent jobs a_i and a_{i+1} in the optimal solution such that $\frac{p_{a_i}}{w_{a_i}} > \frac{p_{a_{i+1}}}{w_{a_{i+1}}}$. Then, we consider a new schedule by switching a_i and a_{i+1}, i.e., $a_1, a_2, \ldots, a_{i+1}, a_i, \ldots, a_n$. The completion time of a_1 to a_{i-1} and a_{i+2} to a_n keep the same. The completion time of a_i increase $p_{a_{i+1}}$ and the completion time of a_{i+1} decrease p_{a_i}. Therefore, the change in the total weighted completion time is

$$p_{a_{i+1}} \cdot w_{a_i} - p_{a_i} \cdot w_{a_{i+1}}.$$

Notice that $\frac{p_{a_i}}{w_{a_i}} > \frac{p_{a_{i+1}}}{w_{a_{i+1}}}$. We have that the change is always smaller than 0. Thus, it is a better schedule, which is a contradiction.

Time Complexity. Obviously $O(n \log n)$.

*This solution is provided by **Jiaxin Song** and **Yuhao Zhang**.* □

Problem 2 (20 points)

You are given n prices $p_1, p_2, \cdots, p_n$, where p_i represents the i-th day price of a stock. On the i-th day, you are allowed to do one of the following operations:

- Buy one unit of stock and pay p_i. Your stock will increase by one.
- Sell one unit of stock and get p_i if your stock is at least one. Your stock will decrease by one.
- Do nothing.

How to maximize the profit (the money left after n days)?

Professor Tao thinks the question is easy and designs the following greedy algorithm: We enumerate from day 1 to n and maintain a min-heap. In each iteration, we do two operations:

1. We insert p_i into the heap.
2. Let q be the minimized price in the heap. If $q < p_i$, we increase $profit$ by $p_i - q$, pop q from the heap, and insert p_i into the heap.

Finally, Professor Tao claims $profit$ is the maximized profit we can get. Do you think Professor Tao is correct? If yes, prove the correctness. Otherwise, give a counterexample.

Solution. The algorithm is correct.

We first define *strategy* and *local improvement*. Strategy x is a mapping from $\{1, \ldots, n\}$ to $\{-1, 0, 1\}$, where x_i represents the strategy at day i and $-1, 0, 1$ mean buying, passing, and selling respectively. The profit can be written as $\sum_{i=1}^{n} x_i p_i$.

A strategy is said to be locally improvable for pair (i, j) where $i > j$, if the profit can be improved by either the following two rules to change the strategy at day i and day j.

- Rule 0: $x_i \leftarrow x_i - 1$ and $x_j \leftarrow x_j + 1$, if $x_i \geq 0$ and $x_j \leq 0$.
- Rule 1: $x_i \leftarrow x_i + 1$ and $x_j \leftarrow x_j - 1$, if $x_j \geq 0$, $x_i \leq 0$, and $\forall i \leq k \leq j, \sum_{l=1}^{k} x_i \leq 0$.

Here we provide an intuitive understanding for the two rules. In Rule 0, if we sell or pass at day i and buy or pass at day j, yet day j has a higher price than day i, we can change our strategy to buy at day i and sell at day j to get extra $v_j - v_i$ profit. In Rule 1, if we buy at day i and sell at day j, yet day j has a higher price than day i (which means the transaction makes a loss), we can pass in both days. Note that in this case, we need to maintain at each day, the amount we buy does not exceed the amount we sell, so that the strategy is valid.

A strategy is said to be *unimprovable* if $\sum_{i=1}^{n} x_i = 0$ and it is not locally improvable.

Lemma 1. A strategy is optimal if it is unimprovable.

Proof. Assume strategy x is not optimal, and we want to prove it is not unimprovable. We consider two cases. First, if $\sum x_i \neq 0$, it is trivial that the strategy is not unimprovable.

Second, we assume $\sum x_i = 0$. Then, there exists an optimal solution x' such that $\sum_{i=1}^{n} (x'_i - x_i) p_i > 0$. Since x' is optimal, we have $\sum_{i=1}^{n} x'_i = 0$. We use c to denote $x'_i - x_i$, where $c_i \in \{-2, -1, 0, 1, 2\}$, and we have $\sum c_i = 0$ and $\sum c_i p_i > 0$. We Then, there must exist a pair of (i, j) where $i < j$ and either $c_i \geq 0, c_j \leq 0, c_i p_i - (-c_j p_j) > 0$ or $c_j \geq 0, c_i \leq 0, c_j p_j - (-c_i p_i) > 0$ holds. In the latter case, we can update x accordingly and it is the same as

Rule 0. In the former case, if $\sum_{i=1}^{k} x_i \leq 0, \forall i \leq k \leq j$, it is the same as Rule 1. If the condition does not hold, since x' is valid, there exists (i', j') such that $c'_i < 0$ and $c'_j > 0$ and either $p_j - p'_i > 0$ or $p'_j - p'_i$ holds Hence, x is not unimprovable.

We turn to Tao's algorithm. In Tao's algorithm, if an element p_i is not in the heap, $x_i = -1$. If it is in the heap once, $x_i = 0$ and if twice, $x_i = 1$.

Lemma 2. The strategy by Tao's algorithm is unimprovable.

Proof. We prove this by induction.

- Base step: At day 1, the strategy by Tao's algorithm is $x_1 = 0$, which is unimprovable.
- Inductive hypothesis: Assume at day $i - 1$, Tao's algorithm is unimprovable.
- Inductive step: We consider day i. We further consider two operations by Tao.
  - Passing: $x_i = 0$. In this case, $\forall k < i$ where $x_k \geq 0$, $p_k > p_i$. Rule 0 will decrease the profit. Rule 1 is not applicable because if we update x_i to -1, it will violate the condition.
  - Selling: $x_i = 1$. Assume we buy at day j. In this case, Rule 0 is not applicable because $x_i > 0$. We assume the strategy can be improved by Rule 1, then there exists $j < k < i$ such that (k, i) forms a pair and $x_k + 1, x_i - 1$ will make a higher profit. If $x_k = 0$, since we do not sell at day k, then by Tao's algorithm, there does not exist a day $k' < k$ such that $x'_k \geq 0$ and $p'_k < p_k$. So $x_k + 1$ is invalid. If $x_k = -1$, assume it is sold at day i', then $i' < j$ because otherwise at day i', it will buy from day j. Then, $x_k + 1$ is invalid at day i'. Hence, Rule 1 is not applicable either.

Moreover, in both cases, $\sum_{k=1}^{i} x_k = 0$. Hence, at day i, the strategy is unimprovable.

From the above two lemmas, we can conclude that Tao's algorithm returns an optimal solution.

□

*This solution is provided by **T.A. Xiaolin Bu**.*

Problem 3 (30 points)

There are n players who need cakes, whose requests are $x_1, x_2, \cdots, x_n$. You are asked to buy one piece of cake and then cut it to satisfy everyone's request. However, The "cut" operation is lossy. Whenever you make a "cut" operation to size W cake, you will lose the p fraction of the cake. (E.g., if you cut a cake at the median with size 2 and $p = 0.5$, you will get two pieces of cake with size 0.5, and the loss is 1. When you cut the 0.5 size cake again, you will lose 0.25 again.)

1. There are four players where $x_1 = x_2 = x_3 = x_4 = 1$, and $p = 0.5$. what is the minimum size? Write down how you cut the cake. You don't need to prove it.(5 points)
2. There are four players where $x_1 = x_2 = 1$, $x_3 = x_4 = 3$, and $p = 0.5$. What is the minimum size? Write down how you cut the cake. You don't need to prove it.(5 points)
3. Given n players, their requests, and the loss factor p, design an efficient algorithm to find the minimum size of cake you need to buy. (20 points)

Solution. 1. The minimum size is 16. We cut the cake as follow:

- Cut 16 at the median into 4 and 4.
- Cut one 4 at the median into 1 and 1.
- Cut the other 4 at the median into 1 and 1.

2. The minimum size is 32. We cut the cake as follow:

- Cut 32 into 12 and 4.
- Cut 12 at the median into 3 and 3.
- Cut 4 at the median into 1 and 1.

3. Consider this problem in a reversed but equivalent way. Every time we choose two pieces of cake $w_{k,x}, w_{k,y}$ and merge them into one piece $\frac{1}{1-p}(w_{k,x} + w_{k,y})$. Finally we get a whole cake $S = w_n$. We need to find a merge order to minimize S.

   We can view this process as constructing a binary tree. The weight of parent node p is determined by its children l and r, namely $w_p = \frac{1}{1-p}(w_l + w_r)$. So the size of cake is $S = w_{root}$. An example is shown in Figure 1 where $\{x_n\} = \{1, 2, 3, 7\}$ and $p = \frac{1}{2}$.

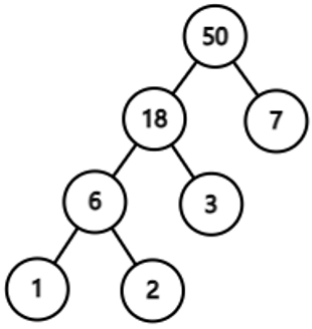

图 1: Merge Equivalent Binary Tree

Consider the contribution of every leaf node. Suppose the depth of root is 0. The leaf node x_i with depth d_i contributes $\frac{x_i}{(1-p)^{d_i}}$. Hence, we can reformulate the expression:

$$S = \sum_{k=1}^{n-1} \frac{1}{1-p}(w_{k,x} + w_{k,y}) = \sum_{i=1}^{n} \frac{x_i}{(1-p)^{d_i}} = \sum_{i \in C} \frac{w_i}{(1-p)^{d_i}}$$

Note that the contribution of a sub-tree can be replaced by the contribution of its root w'_{root}, so C is an arbitrary cover of the binary tree, which is consistent with the slides of the course. Then we have following properties:

- For any nodes w_i and w_j such that $d_i = d_j$, S remains the same if we swap w_i and w_j, which is obvious from the above expression.
- For any nodes w_i and w_j such that $w_i < w_j$ and $d_i < d_j$, we have $\frac{1}{(1-p)^{d_i}} < \frac{1}{(1-p)^{d_j}}$. By Rearrangement Inequality [**?**], we have

  $$\frac{w_i}{(1-p)^{d_i}} + \frac{w_j}{(1-p)^{d_j}} > \frac{w_j}{(1-p)^{d_i}} + \frac{w_i}{(1-p)^{d_j}}$$

  If we swap w_i and w_j, S will become S'. Thus, following inequality holds:

  $$S = \sum_{k \in C} \frac{w_k}{(1-p)^{d_k}} > \frac{w_j}{(1-p)^{d_i}} + \frac{w_i}{(1-p)^{d_j}} + \sum_{k \in C \setminus \{i,j\}} \frac{w_k}{(1-p)^{d_k}} = S'$$

  Therefore, S can be reduced by swapping such w_i and w_j.

According to properties above, S will be minimized after finite steps of swapping, where nodes with smaller weight have larger depth, which promises that the minimum S can be figured out if we always merge the nodes with smallest weight.

A corresponding greedy algorithm is described in following pseudo-codes:

Algorithm 1 Cake Size Minimization

Input: Set of requests $X = \{x_n\}$ and loss factor p

Output: Minimum cake size S^*

```
Construct a minimum heap H
for x_i ∈ X do
    Push x_i into H
for k ← 1 to n − 1 do
    w_x ← minimum element in H
    Pop w_x from H
    w_y ← minimum element in H
    Pop w_y from H
    Push (w_x + w_y)/(1 − p) into H
S* ← minimum element in H
return S*
```

Consider its running time. Building the heap is $O(n)$. Fetching and popping the minimum element in the heap is $O(\log n)$, which happens no more than $2n$ times. Therefore, its time complexity is $O(n \log n)$.

*This solution is provided by **T.A. Panfeng Liu** and **Stu. Xiangyuan Xue**.* □

Problem 4 (25 points)

You are given a fractional number p/q, where p and q are two positive integers and $p < q$. Can you design a greedy algorithm to decompose it into the following form: $p/q = 1/a_1 + 1/a_2 + \cdots + 1/a_k$, where $a_1 < a_2 < a_3 < \cdots < a_k$ are all positive integers? (For example, $2/3 = 1/2 + 1/6$ and $2/3 = 1/2 + 1/7 + 1/42$ are both valid decomposition, while $2/3 = 1/3 + 1/3$ is not.) Remember to analyze the time complexity. Your algorithm should at least terminate for every input. Notice that if you succeed, you prove the decomposition always exists for all fractional numbers.

Solution. See Algorithm 2.

Algorithm 2 Decompose fractional number

Input: a fractional number $\frac{p}{a}$
Output: a list of decomposed fractional number

1: $result$ is initialized as an empty list
2: $result \leftarrow$ DECOMPOSE($\frac{p}{a}, result$)
3: **function** DECOMPOSE($\frac{q}{p}, result$)
4: $\quad s \leftarrow \left\lfloor \frac{q}{p} \right\rfloor$
5: $\quad r \leftarrow q - s \times p$
6: $\quad result.pushback(\frac{1}{s+1})$
7: $\quad p \leftarrow p \times (s+1)$
8: $\quad q \leftarrow q - r$
9: $\quad$ **if** $r = 0$ **then**
10: $\quad\quad$ **return** $result$
11: $\quad$ **else**
12: $\quad\quad$ **return** DECOMPOSE($\frac{p}{q}, result$)

Correctness: We have $q \div p = s...r$. Since $\frac{1}{s+1} < \frac{p}{q} \leq \frac{1}{s}$, we greedily subtract $\frac{1}{s+1}$ from $\frac{p}{q}$ and get $\frac{p-r}{q\times(s+1)}$. If $r = 0$, then the algorithm finishes decomposition, else the numerator decreases by at least 1, so the algorithm terminates after at most p calls.

Time complexity: In the worst case, the function is called for p times, each time costs $O(1)$. Thus $O(p)$ totally.

*This solution is provided by **T.A. Kangrui Mao** and **Stu. Yiyao Yang**.* □