Algorithm Design and Analysis (Fall 2022) Assignment 3

Xiangyuan Xue (521030910387)

1. We have following intuitions:

   - Job i with larger p_i should be processed later, because it will delay other jobs' completion time, thus making the result worse.
   - Job i with larger w_i should be processed earlier, because its completion time will be delayed by other jobs, thus making the result worse.

   Therefore, we define the priority of job i as $\frac{p_i}{w_i}$ and preferentially process those jobs with smaller priority. A greedy algorithm is described in following pseudo-codes:

   Algorithm 1 Average Weighted Completion Time Minimization

   Input: Set of jobs $S = \{(p_i, w_i) | i = 1, 2, \ldots, n\}$

   Output: Minimum average weighted completion time $t^* = \min \frac{1}{n} \sum_{i=1}^{n} w_i C_i$

   ```
   Sort the jobs S by priority p_i / w_i
   C ← 0, t* ← 0
   for i ∈ S do
       C ← C + p_i
       t* ← t* + w_i C
   t* ← t* / n
   return t*
   ```

   Consider its time complexity. Obviously, the real bottleneck is sort. Therefore, we claim that its time complexity is $O(n \log n)$ using merge sort or heap sort, etc.

   Now we prove its correctness. For a good arrangement, we should first agree that there is no gap between jobs. Otherwise, we can make things better by simply cancelling the gaps.

   Then consider arbitrarily two adjacent jobs k and $k+1$ such that $\frac{p_k}{w_k} > \frac{p_{k+1}}{w_{k+1}}$, or equivalently:

   $$w_{k+1} p_k > w_k p_{k+1}$$

   The total weighted completion time can be written as:

   $$S = \sum_{i=1}^{n} \left(w_i \sum_{j=1}^{i} p_j \right)$$

Now just swap k and $k+1$, and the total weighted completion time becomes:

$$\begin{aligned} S' = &\sum_{i=1}^{k-1}\left(w_i\sum_{j=1}^{i}p_j\right) + \sum_{i=k+2}^{n}\left(w_i\sum_{j=1}^{i}p_j\right) \\ &+ w_{k+1}(p_1+p_2+\cdots+p_{k-1}+p_{k+1}) \\ &+ w_k(p_1+p_2+\cdots+p_{k-1}+p_{k+1}+p_k) \end{aligned}$$

Consider the difference between S and S':

$$\begin{aligned} S - S' &= w_k(p_1+p_2+\cdots+p_k) + w_{k+1}(p_1+p_2+\dots p_{k+1}) \\ &\quad - w_{k+1}(p_1+p_2+\cdots+p_{k-1}+p_{k+1}) \\ &\quad - w_k(p_1+p_2+\cdots+p_{k-1}+p_{k+1}+p_k) \\ &= w_{k+1}p_k - w_k p_{k+1} > 0 \end{aligned}$$

Thus $S > S'$, namely S can be **reduced** by swapping such k and $k+1$. After finite steps of swapping, jobs will be sorted in ascending priority, namely the optimal solution.

Therefore, this greedy algorithm can always find the correct answer.

2. This greedy algorithm is correct. We will prove its correctness from two aspects:

- The $profit$ given by the algorithm corresponds to an available plan.

  Just re-explain this algorithm in a more straightforward way:

  - For specific day i, we are not clear whether to do something, so we first push p_i into the heap, meaning that buying at day i is to be determined.
  - Then we fetch the minimized value q in the heap. Suppose q is the price at day j. If $q < p_i$, we decide to buy at day j and sell at day i, which brings $p_i - q$ profit.
  - At the same time, we pop q to avoid buying twice and push p_i again. What is different is that this p_i means we sell at day i, and cancelling is to be determined. When we pop this p_i later, we decide to sell at day i' instead of day i. Therefore, buying cancels selling at day i and nothing happens.
  - Now the behavior of this algorithm is given practical significance. Therefore, an available plan is constructed corresponding to the $profit$ given by the algorithm.

- The $profit$ given by the algorithm is the maximized profit.

  Prove this by induction:

  - At day 1, we can't afford anything, so $profit = 0$ is certainly maximized.
  - Suppose the $profit$ is maximized at day k.
  - At day $k+1$, the $profit$ won't decrease. Assume that there is a better choice. It must be selling at day $k+1$. That required buying at some day before. However, the lowest price available has been chosen by the algorithm, so a better choice is impossible. So the $profit$ is also maximized at day $k+1$.

Thus, the $profit$ given by the algorithm is maximized.

Therefore, this greedy algorithm can always find the correct answer.

3. (a) 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.

   (b) 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.

   (c) 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 $p = \frac{1}{2}$.

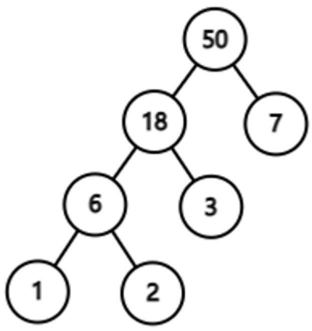

Figure 1: Merge Equivalent Binary Tree

   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[1], 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 2 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 $x_i \in X$ **do**
  Push x_i into H
for $k \leftarrow 1$ **to** $n-1$ **do**
  $w_x \leftarrow$ minimum element in H
  Pop w_x from H
  $w_y \leftarrow$ minimum element in H
  Pop w_y from H
  Push $\frac{w_x + w_y}{1-p}$ into H
$S^* \leftarrow$ 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 $O(n \log n)$.

4. Intuitively, to decompose $\frac{p}{q}$, we try to find the maximum unit fraction $\frac{1}{k}$ such that $\frac{1}{k} < \frac{p}{q}$. Suppose $\frac{p}{q} = \frac{1}{k} + \frac{p'}{q'}$. Now we just need to decompose $\frac{p'}{q'}$. This procedure is repeated until $p' = 1$, namely $\frac{p'}{q'}$ is also a unit fraction[2].

   According to this intuition, we develop following greedy algorithm described in pseudo-codes:

Algorithm 3 Unit Fraction Decomposition

Input: A fractional number p/q

Output: Denominator list $S = \{a_i | i = 1, 2, \dots, m\}$

function $\textsc{gcd}(x,\, y)$
  if $x \bmod y = 0$ **then**
    return y
  return $\textsc{gcd}(y,\, x \bmod y)$
function $\textsc{Reduce}(\frac{x}{y})$
  $x' \leftarrow \frac{x}{\textsc{gcd}(x,y)}$, $y' \leftarrow \frac{y}{\textsc{gcd}(x,y)}$
  return $\frac{x'}{y'}$
function $\textsc{Decompose}(\frac{p}{q})$
  $\frac{p}{q} \leftarrow \textsc{Reduce}(\frac{p}{q})$
  if $p = 1$ **then**
    return $\{q\}$
  $n \leftarrow \lceil \frac{q}{p} \rceil$
  $\frac{p'}{q'} = \frac{p}{q} - \frac{1}{n}$
  return $\{n\} \cup \textsc{Decompose}(\frac{p'}{q'})$
return $\textsc{Decompose}(\frac{p}{q})$

Now we prove the correctness of this algorithm from two aspects:

- **Satisfaction**: $\frac{p}{q} = \frac{1}{a_1} + \frac{1}{a_2} + \cdots + \frac{1}{a_m}$ and $a_1 < a_2 < \cdots < a_m$.

  The equality always holds as long as the algorithm terminates.

  Consider the inequality. For any fraction $\frac{p}{q}$ such that $p < q$ and $(p, q) = 1$, we have $q = sp + r$, where $s \geq 1$ and $r \leq p - 1$. We have $\frac{p}{q} = \frac{1}{s+1} + \frac{p'}{q'}$, where $\frac{p'}{q'} = \frac{p-r}{(s+1)(sp+r)}$. From the definition above we have:

  $$(s+1)p = sp + p < q + q = 2q$$

  Then $\frac{1}{s+1}$ is larger than half of $\frac{p}{q}$:

  $$\frac{1}{s+1} = \frac{p}{(s+1)p} > \frac{p}{2q}$$

  So $\frac{p'}{q'}$ is smaller than half of $\frac{p}{q}$, thus smaller than $\frac{1}{s+1}$:

  $$\frac{1}{a_{k+1}} = \frac{p'}{q'} < \frac{1}{s+1} = \frac{1}{a_k}$$

  Namely $a_k < a_{k+1}$. Since k is arbitrary, we have $a_1 < a_2 < \cdots < a_m$.

- **Termination**: For any fractional number $\frac{p}{q}$, this algorithm terminates.

  Just consider an arbitrary iteration where $\frac{p}{q} = \frac{1}{s+1} + \frac{(s+1)p-q}{(s+1)q}$. From the definition we have:

  $$0 < p' = (s+1)p - q < p$$

Thus, for the remaining term $\frac{p'}{q'}$ in every iteration, we claim that p' is strictly decreasing. And since p' is non-negative, this algorithm will terminate after no more than p iterations.

Therefore, this algorithm always correctly decompose $\frac{p}{q}$ into $\frac{1}{a_1} + \frac{1}{a_2} + \cdots + \frac{1}{a_m}$, which implies that the decomposition exists for any fractional number $\frac{p}{q}$.

Consider its running time. The proof has bounded the iteration times to p. In every iteration, we reduce the fraction $\frac{p}{q}$, which takes $O(\log \min\{p, q\})$ time (mainly for greatest common divisor). And other operations are obviously $O(1)$. Therefore, its total time complexity is $O(p \cdot \log \min\{p, q\}) = O(p \log(p + q))$.

In practice, it will run much faster than the worst case because the numerator p decays fast. Vose even proved that $\frac{p}{q}$ has a t-term decomposition where $t = O(\sqrt{\log q})$[3], which is beyond our discussion.

5. (a) It takes me about 8 hours to finish this assignment.
   (b) I prefer a 4/5 score for its difficulty.
   (c) I have no collaborators. Papers and websites referred to are listed below.

References

[1] Wikipedia. “Rearrangement inequality.” https://en.wikipedia.org/wiki/Rearrangement_inequality.

[2] Wikipedia. “Egyptian fraction.” https://en.wikipedia.org/wiki/Egyptian_fraction.

[3] Vose. “Egyptian Fractions.” Bull. London Math. Soc. 17, 21, 1985.