Chapter 6

Greedy Algorithms

Exercise 6.1 *(Cormen et al. [3] Problem 16-2 p. 402)*

Suppose you are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let c_i be the completion time of task a_i, that is, the time at which task a_i completes processing. You goal is to minimize the average completion time, that is, to minimize $(1/n)\sum_{i=1}^{n} c_i$. For example, suppose there are two tasks, a_1 and a_2, with $p_1 = 3$ and $p_2 = 5$, and consider the schedule in which a_2 runs first, followed by a_1. Then $c_2 = 5$, $c_1 = 8$, and the average completion time is $(5 + 8)/2$.

a. *Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run non-preemptively, that is, once task a_i is started, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.*

b. *Suppose now that the tasks are not all available at once. That is, each task has a release time r_i before which it is not available to be processed. Suppose also that we allow preemption, so that a task can be suspended and restarted at a later time. For example, a task a_i with processing time $p_i = 6$ and release time $r_i = 1$. It means task a_i may start running at time 1 and be preempted at time 4. It can then resume at time 10 but be preempted again at time 11 and finally resume at time 13 and complete at time 15. Task a_i has run for a total of 6 time units, but its running time has been divided into three pieces. We say that the completion time of a_i is 15. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.*

Solution

a. The optimal strategy is to process tasks in order of increasing processing time. So the shortest processing task is processed first, then the second shortest one and so on and so far. Thus, the algorithm that schedules the tasks so as to minimize the average completion time is simply a sorting algorithm. This sorting algorithm will sort $\{p_i\}$ in increasing order to find the optimal schedule of S. The complexity of the current best sorting algorithm is, thereby, $O(n \log n)$.

In order to prove the optimality of this strategy, we use the exchange argument. Suppose that in the optimal order of tasks, we find task i at position k and task j at position $k + 1$ that satisfying the following condition:

$$p_i > p_j.$$

We exchange tasks i and j, i.e., task i is now in position $k + 1$ and task j is now in position k. Certainly, this exchange does not affect the completion time of the other tasks. However,

after exchanging tasks i and j, the completion time of task i is increased by p_j and one of task j is decreased by p_i. Because $p_i > p_j$, the average completion time is decreased by $\frac{p_i - p_j}{n}$. So, in the optimal order of tasks, the processing time of a task is always greater than one of the previous task. This leads to the conclusion that the increasing processing time order of tasks is the optimal one.

b. Order the tasks in the increasing order of their release time. Then process the tasks in that order. At any time t, consider all the requests released before t. The optimal strategy always processes the available (i.e., already released) task having the shortest remaining time. While processing, if a task is released and has shorter processing time than the currently processed task, then the machine preempts the currently processed task and goes on with the processing of the released one.

In order to prove the optimality of this strategy, we also use the exchange argument. Here, we do not exchange two tasks but we exchange pieces of tasks. If at the moment t the optimal schedule has two available tasks i and j where task i has the remaining time e_i and task j has the remaining time e_j with $e_i > e_j$. Suppose that task i is finished at f_i and task j is finished at f_j. Thus, the completion time of task i and j is:

$$f_i + f_j = \max\{f_i, f_j\} + \min\{f_i, f_j\}.$$

Moreover, at moment t the optimal schedule chooses to process task i instead of task j. Certainly, there are $e_i + e_j$ time units allocated to finish tasks i and j. Now, we modify the optimal schedule by taking the first e_j allocated time units of task i and j to finish task j and taking the next e_i allocated time units to finish task i. Clearly, this modification does not give any impact on the completion time of other tasks. After modification, the task i is finished at $\max\{f_i, f_j\}$. Because task j is moved backward to time t, the new completion time of task j is less than or equal to f_j. Because $e_i > e_j$ then the new completion time of task j is also less than or equal to f_i. That why the new completion time of task i and j are less than or equal to $\max\{f_i, f_j\} + \min\{f_i, f_j\} = f_i + f_j$. Thereby, the modified schedule is better or at least as good as the optimal one.

We can use heap-sort to schedule tasks. The remaining time of each job is stored in a node of the heap. Each released job is inserted in the heap and each finished job is removed from the heap. Note that when the heap is updated by a newly released task, the remaining time of the currently processed task is changed. However, the currently processed task is already at the root of the heap. Therefore, only one insertion of the newly released task is enough to update the heap. Because each job is inserted only one time and is removed only one time, overall it corresponds to n element insertions into an empty heap and n removals. As each add/delete is $O(\log n)$, the overall complexity is $O(n \log n)$.

Exercise 6.2 *(Kleinberg and Tardos [11] - Solved Exercise 2 p. 185)*

Your friends are starting a security company that needs to obtain licenses for n different pieces of cryptographic software. Due to regulations, they can only obtain these licenses at the rate of at most one per month.

Each license is currently selling for a price of \$100. However, they are all becoming more expensive according to exponential growth curves: in particular, the cost of license j increases by a factor of $r_j > 1$ each month, where r_j is a given parameter. This means that if license j is purchased t months from now, it will cost $100 \times r_j^t$. We will assume that all the price growth rates are distinct; that is, $r_i \neq r_j$ for licenses $i \neq j$ (even though they start at the same price of \$100).

The question is: Given that the company can only buy at most one license a month, in which order should it buy the licenses so that the total amount of money it spends is as small as possible?

Give an algorithm that takes the n rates of price growth $r_1, r_2, \ldots, r_n$, and computes an order in which to buy the licenses so that the total amount of money spent is minimized. The running time of your algorithm should be polynomial in $O(n)$.

Solution

The problem can be stated as finding the order $j_1, j_2, \ldots j_n$ to minimize:

$$100 \times (r_{j_1} + r_{j_2}^2 + ... + r_{j_n}^n). \tag{6.1}$$

It is equivalent to minimize:

$$r_{j_1} + r_{j_2}^2 + ... + r_{j_n}^n. \tag{6.2}$$

Because $r_j > 1$ so it is very easy to see that $r_{j_1} > r_{j_2} > ... > r_{j_n}$ is the best solution. Thus the algorithm to solve this problem is simply a sorting algorithm. Quick-sort algorithm ($O(n \log n)$) can be used to find the optimal order.

In order to prove this conclusion, we use the exchange argument. Suppose that in the optimal order there are j_u and j_{u+1} that $r_{j_u} < r_{j_{u+1}}$. If we exchange j_u and j_{u+1}, and if it is an optimal order, this exchange should lead to a smallest cost solution.

Before the exchange, the money payed for two licenses j_u and j_{u+1} is

$$r_{j_u}^u + r_{j_{u+1}}^{u+1},$$

and after the exchange, the money payed for two licenses j_u and j_{u+1} is:

$$r_{j_{u+1}}^u + r_{j_u}^{u+1}.$$

Because $r_{j_{u+1}} > r_{j_u} > 0$ and $r_{j_{u+1}} - 1 > r_{j_u} - 1 > 0$, then

$$r^u_{j_u} + r^{u+1}_{j_{u+1}} - r^u_{j_{u+1}} - r^{u+1}_{j_u} = r^u_{j_{u+1}}(r_{j_{u+1}} - 1) - r^u_{j_u}(r_{j_u} - 1) > 0.$$

After the exchange, we obtained a better solution, a contradiction.

Note that, if we do not have the condition: $r_{j_1} > r_{j_2} > ... > r_{j_n}$, then there always exists at least one pair of two consecutive indices j_u and j_{u+1} such that $r_{j_u} < r_{j_{u+1}}$.

Consequently, we conclude that $r_{j_1} > r_{j_2} > ... > r_{j_n}$ is the best solution.

Exercise 6.3 *(Cormen et al. [3] Problem 16-1 p. 402)*

Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.

a. *Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.*

b. *Suppose that the available coins are in the denominations that are powers of c, i.e., the denominations are $c^0, c^1, c^2, \ldots, c^k$ for some integers $c > 1$ and $k \geq 0$. Show that the greedy algorithm always yields an optimal solution.*

c. *Give a set of coins denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.*

d. *Give an $O(nk)$-time algorithm that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.*

Solution

a. The greedy strategy consists of always taking the largest denomination that we can at the time. We repeat this until we have the correct amount of change.

MAKE_CHANGE(M)
1. for $i = k$ down to 1
2. $\quad \mathsf{count}[i] \leftarrow \lfloor M/d_i \rfloor$
3. $\quad M \leftarrow M - \mathsf{count}[i] \times d_i$

Since for each denominator, we calculate the number of coins we can take until we would make more than the amount of change asked for, this operation takes time $O(k)$.

In the particular case of quarters, dimes, nickels and pennies, let compute a vector (Q, D, N, P) that determines the number of coins to use. The greedy strategy is to set

- $Q = \lfloor n/25 \rfloor$ and $n_Q = n - 25Q$
- $D = \lfloor n_Q/10 \rfloor$ and $n_D = n_Q - 10D$
- etc.

In this formulation, we have four subproblems:

- finding the optimal Q when making $n = 25Q + 10D + 5N + 1P$;
- finding the optimal D when making $n_Q = 10D + 5N + 1P$;
- finding the optimal N when making $n_D = 5N + 1P$;
- finding the optimal P when making $n_N = 1P$.

These problems have an optimal substructure: given a fixed set of large coins, you want to use as few small coins as possible. We prove in reverse order that the greedy choice is appropriate for each subproblem. The optimal substructure and greedy choice properties will justify the greedy algorithm.

First consider the choice of P in $n_N = 1P$. The only possible choice is $P = n_N = n_N/1$, so the greedy choice is vacuously optimal.

Next consider the choice of N in $n_D = 5N + 1P$. By the above, an optimal solution sets $P = \lfloor (n_D - 5N)/1 \rfloor$. Therefore, if we set $N < \lfloor n_D/5 \rfloor$, the optimal solution would at least 5 pennies. But no such solution can be optimal because it is more advantageous to replace them with a nickel; therefore, the greedy choice is optimal.

Now turn to the choice of D in $n_Q = 10D + 5N + 1P$ and make a similar argument. By the above, an optimal sets $N = \lfloor (n_Q - 10D)/5 \rfloor$. Therefore, if we set $D < \lfloor n_Q/10 \rfloor$, the optimal solution would include at least two nickels. Again, this is actually suboptimal, so only the greedy choice leads to an optimal choice.

The final step is a little bit difficult. A non-greedy choice, with $Q < \lfloor n/25 \rfloor$, implies that $n_D = \lfloor (n - 25Q)/10 \rfloor \geq \lfloor 25/10 \rfloor = 2$. So far, there is no contradiction, so we need to consider two cases. First, if $n_Q \geq 30$, then $D \geq 3$. But three dimes are a less efficient choice than a quarter and a nickel, so by adding a quarter we improve the solution. On the other hand, if $25 < n_Q < 30$, then $D = 2$ and $5 \leq n_D < 10$. It follows that $N = 1$ and we have two dimes and a nickel. That combination is suboptimal because it can be replaced by a quarter. So in either case the solution is suboptimal; conclude that the optimal choice for Q is the greedy choice.

b. When the coin denominations form a geometric series, the following greedy algorithm produces a multi-set on the set $\{c^0, c^1, \ldots, c^k\}$ such that

$$n = \sum_{j=0}^{k} m_j c^j.$$

We divide the problem into $k+1$ sub-problems, each of which is the choice of the number m_j of c^j-valued coins. Define $n_{k+1} = n$ and n_j as the remainder of n_{j+1} when using the maximum number of c^j coins. We next get $n_k = n_{k+1} - m_k c^k$, and so on, so that the generic expression is $n_j = n_{j+1} - m_j c^j$. After recursive substitution, it leads to $n_{j+1} = m_j c^j + m_{j-1} c^{j-1} + \cdots + m_0 c^0$.

GREEDY_GEOMETRIC_CHANGE(n)
1. **for** $j \leftarrow k$ **down to** 0 **do**
2. $m_j \leftarrow \lfloor n_{j+1}/c^j \rfloor$
3. $n_j = n_{j+1} - m_j c^j$

We show by induction on j that the greedy choice is optimal at each step. If $j = 0$, the problem is $n_1 = m_0 c_0 = m_0$. So the greedy possibility is $m_0 = n_1 = \lfloor n_1/c^0 \rfloor$.

Let j be such that $c^j \leq n < c^{j+1}$ if $n \leq c^k$, otherwise let $j = k$ if $n > c^k$. The algorithm returns a multi-set $A = \{c^j\} \cup A'$ where, by the induction hypothesis, A' is an optimal solution for $n - c^j$. Let the multi-set B be an optimal solution for n. For the greedy choice, we show that B must contain a "c^j". Suppose B does not contain a "c^j". Then B contains only elements "c^i" with $i < j$, so

$$\sum_{i=0}^{j-1} m_i c^i = n > c^j. \tag{6.3}$$

Now assume that $m_i < c$ for each $0 \leq i < j$. We get:

$$\sum_{i=0}^{j-1} m_i c^i \leq (c-1) \sum_{i=0}^{j-1} c^i = (c-1)\frac{c^j - 1}{c-1} = c^j - 1 \text{ (assuming that } c \geq 2).$$

But this contradicts (6.3), so there must be some i in the range $0 \leq i \leq j-1$ for which $m_i \geq c$. Then, we can replace the c times c^i with a single c^{i+1}, which contradicts the optimality of B. Thus B must contain a "c^j".

By induction, the greedy choice property holds at every step, so the greedy algorithm is correct.

---

b. A less abstract solution ...

Let $T(M, k)$ the minimum number of coins used to change M into coins of denominations $c^0, c^1, \ldots c^k$. We have the following inequality:

$$T(M, k) \leq T(M - c^k, k) + 1 \tag{6.4}$$

Because $T(M - c^k, k)$ corresponds to the best solution of change $M - c^k$ into coins of $c^0, c^1, \ldots c^k$, then plus one coin c^k gives us a solution (not necessarily the best) to change M into denominations $c^0, c^1, \ldots c^k$. This solution has $T(M - c^k, k) + 1$ coins and cannot be better than the best solution which consists of converting M into coins of $c^0, c^1, \ldots c^k$: $T(m, k)$. The equality is proved.

Now, we consider all possible cases of the optimal solution of converting M into coins of $c^0, c^1, \ldots c^k$.

- If there is only one coin c^k in the optimal solution, we have

$$T(M, k) = 1 + T(M - c^k, k - 1)$$

- If there are only two coins c^k in the optimal solution, we have
$$T(M,k) = 2 + T(M - 2c^k, k-1)$$
- ...
- If there are only x coins c^k in the optimal solution, we have
$$T(M,k) = x + T(M - xc^k, k-1)$$
- ...
- If there are only $\lfloor M/c^k \rfloor$ coins c^k in the optimal solution, we have
$$T(M,k) = \left\lfloor \frac{M}{c^k} \right\rfloor + T\left(M - \left\lfloor \frac{M}{c^k} \right\rfloor c^k, k-1\right)$$

Using equation (6.4), we deduce the following inequality :

$$x + T(M - xc^k, k-1) \geq x + 1 + T(M - (x+1)c^k, k-1)$$

Thus, the optimal solution will be at the maximum of x : $x = \lfloor \frac{M}{c^k} \rfloor$. In other words, we have the following recurrence relation:

$$T(M,k) = \left\lfloor \frac{M}{c^k} \right\rfloor + T\left(M - \left\lfloor \frac{M}{c^k} \right\rfloor c^k, k-1\right)$$

We can then easily see that the greedy algorithm makes use of this recurrence relation in order to calculate the optimal solution.

c. $\{1, 50, 51\}$
Greedy algorithm solution: $51 + 49 \times 1$
Optimal choice: 2×50.

d. Dynamic programming algorithm
Let n' be such that $0 \leq n' \leq n$ and a set of coins of denominations $c_1, c_2, \ldots c_k$.
Allocate two arrays:

- $v[n']$ holds the minimum number of coins needed to make n' cents
- $d[n']$ holds the denomination of the first coin to dispense when making n' cents, $n' \geq 1$.

We compute these in order of increasing n'. Initially, set $v[0] = 0$. For each n' from 1 to n, we see which of the k possible coins we should dispense first: we know $d[n']$ should be one of the k denominations. Indeed,

$$d[n'] = \arg\min_{c_i} v[n' - c_i] \quad \text{where} \quad v[n'] = 1 + v[n' - d[n']].$$

So it requires to identify $d[n']$ that minimizes this quantity and to update $v[n']$ accordingly; this takes $O(k)$ time.

Arrays are populated in $O(kn)$. We can then reconstruct solution in $O(n)$ time, dispensing one coin at a time.

Exercise 6.4

Concordia University wants to determine the minimum number of exam periods to plan. In order to answer the question, you propose the following modeling of the problem. Build an undirected graph, called conflict graph, where each node is associated with an exam, and such that there is an edge between two nodes if there is at least one student who has to take the two exams associated with the endpoints (nodes) of that edge. Next, the idea is to color the graph with the minimum number of colors, i.e., to compute the chromatic number, such that each color will be associated with a different time period.

a. *Propose a greedy algorithm in order to compute a non trivial upper bound. What is its complexity?*

b. *Propose another greedy algorithm in order to compute a non trivial lower bound. What is its complexity?*

c. *Propose a branch-and bound algorithm in order to compute the chromatic number of your conflict graph.*

d. *There are a limited number of rooms available for holding exams (let us forget about the capacity of the rooms), leading a constraint on the number of nodes that can be assigned the same color. How can you modify your branch-and-bound in order to handle that last constraint?*

Solution

Exercise 6.5 *(Final Exam - Winter 2010 - Cormen et al. [3] Problem 16.2 - 4)*

Professor Midas drives an automobile from Newark to Reno along Interstate 80. His car's gas tank, when full, holds enough gas to travel n miles, and his map gives the distances between the gas stations on his route. The professor wishes to make as few gas stops as possible along the way. Give an efficient method by which Professor Midas can determine at which gas stations he should stop and prove that your algorithm provides an optimum solution.

Solution

Since Professor Midas drives along only one road: Interstate 80, we can model this problem according to Figure 6.1.

Figure 6.1: Graph model for Professor Midas' Problem

The car starts from Newark, and there are m gas stations on his way, named as $G = \{g_1, g_2, \ldots, g_m\}$. We know the location for every gas station. Our purpose is to select a subset of gas stations $G' = \{g'_1, g'_2, \ldots, g'_t\} \subseteq G$ such that $|g'_j - g'_{j-1}| \leq n$, $2 \leq j \leq t$, $|\text{ Newark } - g'_1| \leq n$, $|\text{Reno} - g'_t| \leq n$, and the size of G' is minimum. This can be achieved using the Algorithm 7 that is described below. Next, we need to prove that G' is the optimal solution for this question. We use contradiction to prove it.

Statement 1. G' has the minimum number of gas stations that Professor Midas needs during his way to Reno.

Proof. If $G' = \{g'_1, g'_2, \ldots, g'_t\}$ is not a schedule with minimum number of gas stations, then there exists a subset of gas stations R that contains a schedule with a smaller number of gas stations. Let $R = \{r_1, r_2, \ldots, r_s\}$ such that $|\text{ Newark } - r_1| \leq n$, $|r_i - r_{i-1}| \leq n$, $|r_s - \text{ Reno }| \leq n$ and $s < t$. Let r_k and g'_k be the first different gas stations (may start from 1). Obviously, we have $r_1 = g'_1$, $r_2 = g'_2$, $\cdots$, $r_{k-1} = g'_{k-1}$. Then we have $|\text{ Newark } - r_k| < |\text{ Newark } - g'_k|$, which means the location of r_k is closer to Newark than g'_k (since $|r_{k-1} - r_k| < |g'_{k-1} - g'_k|$, according to our algorithm). This statement is true for all the gas stations from r_{k+1} to r_s. Say, $|\text{ Newark } - r_{k+1}| < |\text{ Newark } - g'_{k+1}|, \cdots, |\text{ Newark } - r_s| < |\text{ Newark } - g'_s|$. Since $s < t$, without loss of generality, let us assume that $t = s + 1$. we have that $|g'_s - g'_{s+1}| \leq n$ but $|g'_s - \text{ Reno }| > n$ (otherwise we do not need to choose g'_{s+1}). Therefore, we have $|r_s - \text{Reno}| > n$, which contradicts our assumption.

Consequently, G' should has the minimum number of gas stations.

Algorithm 7 Best Schedule to Refuel Gas

1: **Input:** The locations for gas stations G
2: **Output:** A best schedule G'
3: Set $G' \leftarrow \emptyset$
4: Step 1: Find $g'_1 = \max_{1 \le i \le m} \{g_i : |g_i - \text{ Newark }| \le n\}$, insert g'_1 into G'.
5: Step 2: Find $g'_j = \max_{1 \le i \le m} \{g_i : |g_i - g'_{j-1}| \le n\}$, $(2 \le j \le t)$, then insert g'_j into G'. Continue this step until $|g'_t - \text{ Reno }| \le n$.

Exercise 6.6

Propose a greedy algorithm that solves exactly the following location problem for mobile phone stations.

Input: *the locations of n houses along a straight line We want to place cell phone base stations along the road so that every house is within 4 miles of one of the base stations.*

Output: *a minimal set of base stations.*

Assumption: *No pair of two successive house locations are more than 8 miles apart.*

Solution

The algorithm is simple: pick the leftmost house not covered by a base station and place a base station 4 miles to the left.

We now prove that this approach is optimal. Suppose that our greedy algorithm places m base stations at the locations

$$b_1 < b_2 < \cdots < b_m,$$

and assume that this solution is not optimal.

Let then

$$\text{OPT} = b'_1 < b'_2 < \cdots < b'_k,$$

be an optimal choice of base stations, with $k < m$.

Observe that $b'_1 < b_1$. This is because we put the first base station b_1 as far to the right as possible. Maybe $b'_2 < b_2$? Suppose that this does not hold, so we have

$$b'_1 < b_1 < b_2 \leq b'_2.$$

Since we needed to place a base station at b_2, there must be a house situated at a location h such that $b_1 + 4 < h$.

Then $b_2 = h + 4$. But this implies $b'_1 + 4 < h < b'_2 + 4$, so the location h cannot be covered neither by b'_1, nor by b'_2, contradiction with the fact that OPT is a valid solution.

Thus $b'_2 \leq b_2$. Similarly $b'_3 \leq b_3$. But then there is no need to place another phone station at the location b_{k+1}, as all the houses that might be deserved by it are already covered by b'_k. Thus, our greedy strategy produces an optimal solution.

Exercise 6.7

Consider the following problem. The input consists of the lengths $\ell_1, \ell_2, \ldots, \ell_n$, and access probabilities $p_1, p_2, \ldots, p_n$, for n files $F_1, F_2, \ldots, F_n$. The problem is to order these files on a tape so as to minimize the expected access time. If the files are placed in the order $F_{s(1)}, F_{s(2)}, \ldots, F_{s(n)}$ then the expected access time is

$$\sum_{i=1}^{n} \left(p_{s(i)} \sum_{j=1}^{i} \ell_{s(j)} \right)$$

where the term $p_{s(i)} \sum_{j=1}^{i} \ell_{s(j)}$ is the probability that you access the ith file times the length of the first i files.

For each of the below greedy algorithms, either give a proof that the algorithm is correct using an exchange argument, or a proof that the algorithm is incorrect.

(i) *Order the files from shortest to longest on the tape. That is, $\ell_i < \ell_j$ implies that $s(i) < s(j)$.*

(ii) *Order the files from most likely to be accessed to least likely to be accessed. That is, $p_i < p_j$ implies that $s(i) > s(j)$.*

(iii) *Order the the files from smallest ratio of length over access probability to largest ratio of length over access probability. That is, $\ell_i/p_i < \ell_j/p_j$ implies that $s(i) < s(j)$.*

Solution

Exercise 6.8

A ski rental agency has m pairs of skis, where the height of the ith pair of skis is s_i. There are n skiers who wish to rent skis, where the height of the ith skier is h_i. Ideally, each skier should obtain a pair of skis whose height matches her/his own height as closely as possible. We would like to assign skis to skiers so that the sum of the absolute differences of the heights of each skier and her/his skis is minimized. Assuming that $m = n$, design a greedy algorithm for the problem. Prove the correctness of your algorithm.

Solution

Greedy algorithm. Assume the people and skis are numbered in increasing order by height. Match the people and their ski following those two orders.

Complexity. $O(n \log n)$ for the two sorts, and then $O(n)$ for computing the discrepancies, and matching the skiers to their skis.

Optimality proof.
If the greedy algorithm is not optimal, then there is some input $h_1, h_2, \ldots, h_n, s_1, s_2, \ldots, s_n$ for which it does not produce an optimal solution. Let the optimal solution be

$$S^\star = \{(h_1, s_{j(1)}), (h_2, s_{j(2)}), \ldots, (h_n, s_{j(n)})\},$$

and let the output of the greedy algorithm be

$$G = \{(h_1, s_1), (s_2, h_2), \ldots, (h_n, s_n)\}.$$

Beginning with h_1, compare $S^\star$ and G. Let h_i be the first person who is assigned different skis in G than in $S^\star$. Let s_j be the pair of skis assigned to h_i in $S^\star$. Create solution S' by switching the ski assignments of h_i and p_j. By the definition of the greedy algorithm, $s_i \leq s_j$. The total cost of S' is given by:

$$\text{COST}(S') = \text{COST}(S^\star) - \frac{1}{n}\left(|h_i - s_j| + |h_j - s_i| - |h_i - s_i| - |h_j - s_j|\right).$$

There are six cases to be considered. For each case, one needs to show that

$$|h_i - s_j| + |h_j - s_i| - |h_i - s_i| - |h_j - s_j| \geq 0.$$

Case 1: $h_i \leq h_j \leq s_i \leq s_j$.
$|h_i - s_j| + |h_j - s_i| - |h_i - s_i| - |h_j - s_j| = (s_j - h_i) + (s_i - h_j) - (s_i - h_i) - (s_j - h_j) = 0.$

Case 2: $h_i \leq s_i \leq h_j \leq s_j$.
$|h_i - s_j| + |h_j - s_i| - |h_i - s_i| - |h_j - s_j| = (s_j - h_i) + (h_j s_i) - (s_i - h_i) - (s_j - h_j) = 2(h_j - s_i) \geq 0.$

Case 3: $h_i \leq s_i \leq s_j \leq h_j$.
$|h_i - s_j| + |h_j - s_i| - |h_i - s_i| - |h_j - s_j| = (s_j - h_i) + (h_j - s_i) - (s_i - h_i) - (h_j - s_j) = 2(s_j - s_i) \geq 0$.

Case 4: $s_i \leq s_j \leq h_i \leq h_j$.
$|h_i - s_j| + |h_j - s_i| - |h_i - s_i| - |h_j - s_j| = (h_i - s_j) + (h_j - s_i) - (h_i - s_i) - (h_j - s_j) = 0$.

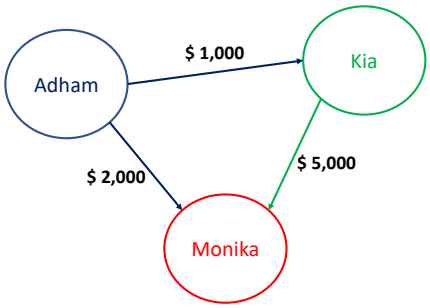
Case 5: $s_i \leq h_i \leq s_j \leq h_j$.
$|h_i - s_j| + |h_j - s_i| - |h_i - s_i| - |h_j - s_j| = (s_j - h_i) + (h_j - s_i) - (h_i - s_i) - (h_j - s_j) = 2(s_j - h_i) \geq 0$.

Case 6: $s_i \leq h_i \leq h_j \leq s_j$.
$|h_i - s_j| + |h_j - s_i| - |h_i - s_i| - |h_j - s_j| = (s_j - h_i) + (h_j - s_i) - (h_i - s_i) - (s_j - h_j) = 2(h_j - h_i) \geq 0$.

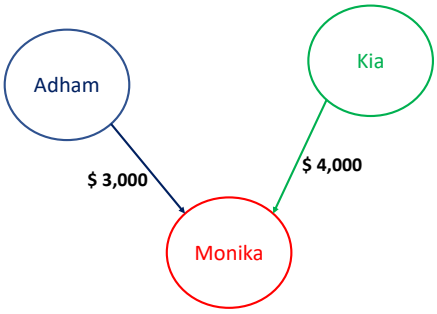
Exercise 6.9 *Cash flow problem.*

Design a greedy heuristic for the following cash flow problem, analyze its computational complexity and prove that it outputs an optimal solution.
Minimize Cash Flow among a given set of friends who have borrowed money from each other. Given a number of friends who have to give or take some amount of money from one another. Design an algorithm by which the total cash flow among all the friends is minimized.

Figure 6.2(a) shows input debts to be settled, and Figure 6.2(b) shows how debts can be settled in an optimized way.



(a) Adham has to pay $ 1,000 to Kia and $ 2,000 to Monika, Kia has to pay $ 5,000 to Monika



(b) Adham has to pay $ 3,000 to Monika, Kia has to pay $ 4,000 to Monika

Figure 6.2: Debts among friends

Solution

Greedy Algorithm

The idea is to design a greedy algorithm in which, at each step, set all the amounts of a person and repeat the operation for the $n-1$ remaining people.

How to pick the first person? To pick the first person, calculate the net amount for every person where net amount is obtained by subtracting all debts (amounts to pay) from all credits (amounts to be paid). Once net amount for every person is evaluated, find two persons with maximum and minimum net amounts. These two persons are the most creditors and debtors. The person with minimum of two is our first person to be settled and removed from list. Let the minimum of two amounts be x. We pay "x" amount from the maximum debtor to maximum creditor and settle

one person. If x is equal to the maximum debit, then maximum debtor is settled, else maximum creditor is settled.

The greedy algorithm is described below.

For every person P_i, $i = 1, 2, \ldots, n$, proceed as follows.

1. Compute the net amount for every person. The net amount for person P_i can be computed be subtracting sum of all debts from sum of all credits.

2. Find the two persons that are maximum creditor and maximum debtor. Let the maximum amount to be credited from maximum creditor be `max_Credit` and maximum amount to be debited from maximum debtor be `max_Debit`. Let the maximum debtor be P_d and maximum creditor be P_c.

3. Find the minimum of `max_Debit` and `max_Credit`. Let minimum of two be x. Debit "x" from P_d and credit this amount to P_c.

4. If x is equal to `max_Credit`, then remove P_c from set of persons and recur for remaining $n-1$ persons.

5. If x is equal to `max_Debit`, then remove P_d from set of persons and recur for remaining $n-1$ persons.

Complexity of the Greedy Algorithm

To calculate the net amount of each person, we need $O(n^2)$.
To find the min cash flow we need: $T(n) = T(n-1) + O(1)$. We can prove that $T(n)$ is $O(n)$ or $T(n) \leq cn$:

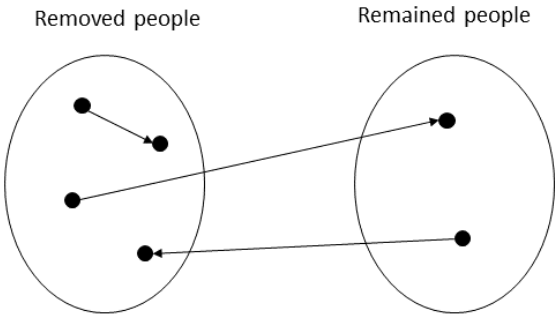
$$T(n) = T(n-1) + O(1) \leq c(n-1) + a = cn - c + a \leq cn \Leftrightarrow c \geq a, n \geq 1$$

$\Rightarrow$ Hence, the time complexity of the algorithm is $O(n^2)$

Optimality Proof of the Solutions output by the greedy algorithm

Each person has to pay or receive an amount of money which is not less than his/her net amount. Then the minimum cash flow:

$$\texttt{minCashFlow} = \frac{1}{2}\sum_{i=1}^{n} \text{NET}(P_i)$$



Let x_k be the amount of money we transferred from the debtor to the creditor at step 3, then we have the total of all x_k:

$$\sum_k x_k = \sum_{k \in \text{CREDITORS}} \text{NET}(P_k) = \sum_{k \in \text{DEBTORS}} \text{NET}(P_k) = \frac{1}{2} \sum_{i=1}^{n} \text{NET}(P_i)$$

That is the total amount of money we transferred, a.k.a. cash flow, equals the total amount of money which all creditors will receive or all debtors have to pay. In turn, this amount equals half of total net amount off all persons.