CS 577: Introduction to Algorithms — Homework 5 Solutions

Problem 1

We design an algorithm that minimizes the number of employees required to attend the ticketing desk using the following greedy strategy. Starting with an empty schedule, we consider the earliest flight whose boarding period is not yet completely covered. If there is no such flight, then we are done; otherwise, we schedule a new shift as late as possible subject to covering the flight's boarding period. The new schedule covers the flight, after which we repeat until all flights are covered.

To implement this efficiently, we sort the flights in order of increasing time, and keep track of the latest end-time of a shift in the schedule. In more detail:

1. Sort the boarding times in increasing order.
2. Set $last\_shift\_end = -\infty$
3. For each boarding time t (in order),
   (a) If $t + 15$ mins $> last\_shift\_end$
      i. Let $s \leftarrow max(t - 15 \text{ mins}, last\_shift\_end)$
      ii. Staff a new employee from time s until $s + 2$ hrs.
      iii. Update $last\_shift\_end \leftarrow s + 2$ hrs

Proof of Correctness

Let S be the schedule returned by our algorithm. It is clear that S is a schedule covering all the required boarding periods. It remains to argue that S is as small as possible.

To this end, let O be an arbitrary scheduling covering all flights. We order the schedules in S and O by their ending time. Let s_j be the the end-time of the j^{th} shift in S, and o_j be the end-time of the j^{th} shift in O. We also define $s_0 = o_0 = -\infty$. Correctness of the algorithm follows from the following greedy-stays-ahead argument.

Claim 1. *For every $j = 0, 1, 2, \ldots, |S|$, it is the case that $|O| \geq j$, and $o_j \leq s_j$.*

Proof. By induction on j.

Base case: When $j = 0$, it trivially holds that $|O| \geq j$, and we have also defined $o_0 \leq s_0$.

Inductive Step: We assume that the claim holds for a particular value of $j < |S|$, and show that $|O| \geq j + 1$ and $o_{j+1} \leq s_{j+1}$.

Consider the first boarding time t which is not covered by the first j shifts in S; *i.e.*, for which $t + 15$ mins $> s_j$. (By how S is constructed and since $j < |S|$, such a shift exists.) By the inductive hypothesis, we know $o_j \leq s_j$, so this flight is also not covered by the first j flights in O. Thus there must be another scheduled shift in O; this shows $|O| \geq j + 1$.

Moreover, there must be a shift in O that covers this flight, and thus must start either at o_j or $t - 15$ mins, whichever is later, in order to ensure the entire boarding period is covered. On the other hand, the $j + 1^{th}$ schedule in S starts either at s_j or $t - 15$ mins, whichever is later; in any case, this is no earlier than the $j + 1^{th}$ shift in O. Moreover, the length of the $j + 1^{th}$ shift in O is at most the length of the $j + 1^{th}$ shift in S, as the latter is the maximum-possible two hours. It follows that $o_{j+1} \leq s_{j+1}$, completing the proof. □

Running time analysis:

Initial sorting of the boarding times takes $O(n \log n)$ time and the remainder of the algorithm runs in time $O(n)$. Hence, the overall runtime complexity of the algorithm is $O(n \log n)$.

Problem 2

We first explain how to compute the maximum possible score for the first player, and then we describe the procedure to recover the first move of the first player which gives this maximum possible score. The trick is to use two simultaneous recurrences, one for computing the maximum attainable value when it is the first player's turn, and one for computing the maximum attainable value when it is the other player's turn.

For i, j with $1 \leq i \leq j \leq n$, define $\mathrm{OPT}_1(i, j)$ to be the largest value v so that the person playing when the cards i through j are on the table can always attain value at least v, no matter how the other player plays. Similarly, $\mathrm{OPT}_2(i, j)$ is the largest value v so that the person *not* playing when the cards i through j are on the table is nevertheless guaranteed to get value at least v. The value we are interested in is $\mathrm{OPT}_1(1, n)$.

We can find $\mathrm{OPT}_1(i, j)$ and $\mathrm{OPT}_2(i, j)$ co-recursively as follows:

$$\mathrm{OPT}_1(i,j) = \begin{cases} \text{Card}[i] & : \quad i = j \\ \max \left\{ \begin{array}{ccc} \text{Card}[i] & + & \mathrm{OPT}_2(i+1, j) \\ \mathrm{OPT}_2(i, j-1) & + & \text{Card}[j] \end{array} \right\} & : \quad i < j \end{cases}$$

$$\mathrm{OPT}_2(i,j) = \begin{cases} 0 & : \quad i = j \\ \min \left\{ \begin{array}{c} \mathrm{OPT}_1(i+1, j) \\ \mathrm{OPT}_1(i, j-1) \end{array} \right\} & : \quad i < j \end{cases}$$

Proof of Correctness

The cases when $i = j$ follow straightforwardly from how OPT_1 and OPT_2 are defined. (The players have no choices to make.)

When $i < j$, $\mathrm{OPT}_1(i, j)$ is attained when the first player either takes from the left, or takes from the right. If they were to take from the left, their value would be Card$[i]$ plus whatever value they get from the leftover game on the cards $i + 1$ through j. Since it is the other player's turn next, this is exactly $\mathrm{OPT}_2(i + 1, j)$.

Similarly, if the first player were to take from the right, their value would be $\mathrm{OPT}_2(i, j - 1) + \text{Card}[j]$. Since the player gets to choose which of these two strategies to undertake, she can guarantee she gets the larger value, hence the $\max$ appearing in the definition of OPT_1.

As for OPT_2, the player's value has similar formulas (depending on whether the other player picks the leftmost or rightmost card), but since the other player gets to pick the next move, we need to use a $\min$ in order to satisfy the definition of OPT_2, and there is no contribution of Card$[i]$ and Card$[j]$.

Iterative solution

We can translate the above recurrence into an iterative approach as follows. We make two 2-D arrays $A_1[i][j]$ and $A_2[i][j]$ where $1 \leq i, j \leq n$. $A_1[i][j]$ will store $\mathrm{OPT}_1(i, j)$ for $i \leq j$, and will be undefined otherwise. A_2 is defined similarly in terms of OPT_2. The dependency between the entries in A_1 and A_2 are shown in the Figure 1.

Thus the entries in the 2-D arrays can be filled from bottom-to-top and left-to-right fashion. This can be accomplished with the following loops:

1. For i = n down to 1, For j = i to n
   (a) If i == j
      i. Set $A_1[i][j]$ = Card$[i]$
      ii. Set $A_2[i][j]$ = 0
   (b) Else
      i. Set $A_1[i][j] = \max\{A_2[i, j - 1] + \text{Card}[j], A_2[i + 1, j] + \text{Card}[i]\}$
      ii. Set $A_2[i][j] = \min\{A_1[i, j - 1], A_1[i + 1, j]\}$
2. The optimal score for the first player is stored in $A_1[1][n]$.

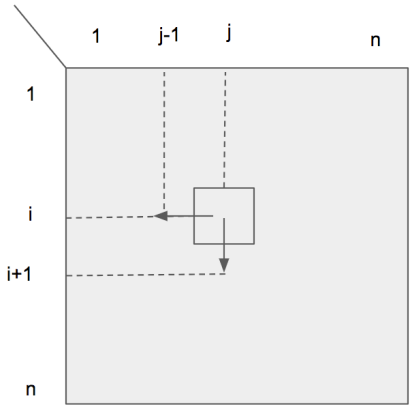

Figure 1: Dependency graph of entries in A_1 and A_2

Running Time

Using memoization, we can compute OPT_1 and OPT_2 efficiently. The total number of states is $2 \cdot \binom{n+1}{2} = O(n^2)$, and each state requires only $O(1)$ operations to compute. Hence the total time spent to compute the maximum value guaranteeable by the first player is $O(n^2)$.

Recovering the winning first move

We can recover the optimal first move of the first player by examining the two quantities contributing to the $\max$ defining $\mathrm{OPT}_1(1, n)$. If the first quantity is larger, then the first player should take the left card (card 1); otherwise, the first player should take the right card (card n).