CS161 Handout 14

Summer 2001 August 13, 2001

Solution Set 6

Problem 1: Renting Skis (20 points)

A ski rental agency has m pairs of skis, where the height of the i^{th} pair of skis is s_i. There are n skiers who wish to rent skis, where the height of the j^{th} skier is h_j. Ideally, each skier should obtain a pair of skis whose height matches his on height as closely as possible. Design an efficient algorithm to assign skis to skiers so that the sum of the absolute differences of the heights of each skier and his skis is minimized.

The key observation: there is no advantage to cross matching — reversing the height order of skiers and skis. That is, if $s_1 < s_2$ and $h_1 < h_2$, there is no reason to match $s_1 \leftrightarrow h_2$ and $s_2 \leftrightarrow h_1$. To show this, we first look at all the possible relationships of the 4 heights, and show that the cost of matching the shorter of two skiers to the shorter of two skis is always at least as good as the cost of cross matching them. With the loss of generality, we can assume that s_1 is the smallest height ($s_1 \le h_1$).

Case 1: $s_1 \le h_1 < h_2 \le s_2$

The sum of the absolute differences of the matched ski/skier heights is:

- If $s_1 \leftrightarrow h_1$ and $s_2 \leftrightarrow h_2$, $h_1 - s_1 + s_2 - h_2 = (s_2 - s_1) - (h_2 - h_1)$.
- If $s_1 \leftrightarrow h_2$ and $s_2 \leftrightarrow h_1$, $h_2 - s_1 + s_2 - h_1 = (s_2 - s_1) + (h_2 - h_1)$.

Since $(h_2 - h_1) > 0$, the first matching costs less than the second.

Case 2: $s_1 \le h_1 \le s_2 \le h_2, s_1 < s_2, h_1 < h_2$.

The sum of the absolute differences of the matched ski/skier heights is

- If $s_1 \leftrightarrow h_1$ and $s_2 \leftrightarrow h_2$, $h_1 - s_1 + h_2 - s_2 = (h_2 - s_1) - (s_2 - h_1)$.
- If $s_1 \leftrightarrow h_2$ and $s_2 \leftrightarrow h_1$, $h_2 - s_1 + s_2 - h_1 = (h_2 - s_1) + (s_2 - h_1)$.

Since $(s_2 - h_1) \ge 0$, the first matching costs no more than the second.

Case 3: $s_1 < s_2 \le h_1 < h_2$.

Verified similarly.

To show that there is always an optimal solution with no cross matching, let S be an optimal matching. If S has no cross matches, we are done. Otherwise, consider two skiers and two skis involved in a cross match, and reverse their matching so that the shorter of the two skiers has the shorter of the two skis. We saw above that this change cannot increase the cost of the match, so this revised solution is at least as good as S.

Algorithm:
If m = n, simply sort the skiers by height, sort the skis by height, and match the ith skier with the ith pair of skis. Correctness: any other assignment would have a cross match, so this solution is optimal. Running time is $O(n \lg n)$.

If $m \neq n$, start as before by sorting the skiers and skis (because there's no advantage to matching them up out of order). Then use dynamic programming. Notice that dynamic programming is an appropriate technique.

- If you try all ways of matching up the skiers and the skis, there will be lots of repeated subproblems. Imagine that you've matched $s_i \leftrightarrow h_j$, and then you decide to match $s_k \leftrightarrow h_l$. You're left with the same subproblem as when you first try to match $s_k \leftrightarrow h_l$ and then select $s_i \leftrightarrow h_j$ next.
- Solutions have optimal substructure. Imagine that some subset of an optimal matching is not optimal — that there's a better way to match that subset of skiers with that subset of skis. Substituting that better subset match in the original solution improves the original solution, so it couldn't have been optimal in the first place.

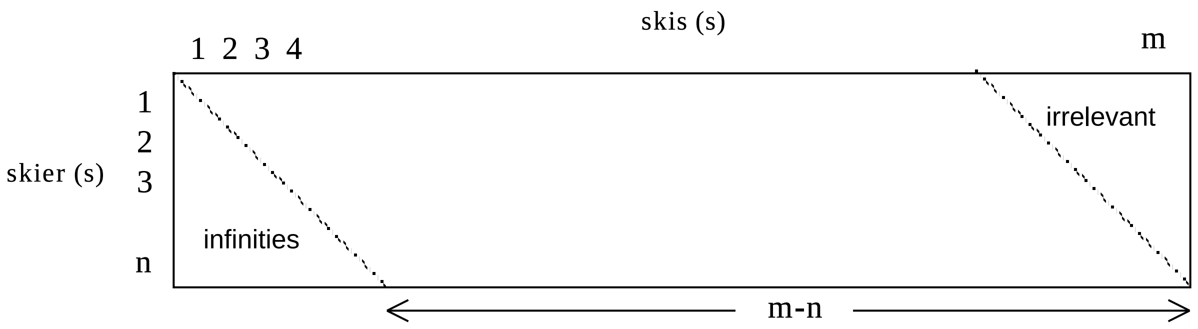
Assume that $n \leq m$ (there are more skis than skiers). (This is okay because the problem is symmetrical: If $m < n$, just interchange the skiers with the skis in the algorithm. In any case, you want to match up all of whichever things there are fewer of.)
After sorting the skiers and skis, consider how to match the first i skiers with the first j pairs of skis. Let $A[i,j]$ be the optimal cost (sum of the absolute differences of heights) for matching the first i skiers with the first j pairs of skis so that all i skiers have skis. The solution we seek is $A[n,m]$. We can define A as follows:

$$A[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \min\left(A[i,j-1], A[i-1,j-1] + |h_i - s_j|\right) & \text{if } 1 \leq i \leq j \\ \infty & \text{if } i > j \geq 1 \end{cases}$$

This is correct because

- If there's an optimal match not using s_j, it matches the first i skiers to the first j - 1 pairs of skis.
- If there's an optimal solution using s_j, it can assign $h_i \leftrightarrow s_j$, because all skiers (the h's) get matched up, and any other way of using both h_i and s_j would either be a cross match or would involve another skier of height h_i or another pair of skis of height s_j (and would thus have the same cost as this match.)
  The rest of this match must be an optimal match of the first i - 1 skiers with the first j - 1 skis.

Since $A[i,j]$ depends on a value in the previous row of A and an earlier value in the same row of A, calculate A in a row-major order (top row to bottom row, left to right within rows.)



Note that only the part of A between the diagonal lines needs to be computed, because the triangle at the bottom left just has infinities and the triangle in the upper right never actually contributes to the final $A[n,m]$ value.

To find an actual assignment of skiers to skis, we can trace a path from $A[n,m]$ to see where each value came from. At each step,

- If $A[i,j] = A[i,j-1]$, continue tracing at $A[i,j-1]$.
- Otherwise, match $h_i \leftrightarrow s_j$ and continue tracing from $A[i-1,j-1]$.

Time: $\Theta(n \lg n)$ to sort the skiers + $\Theta(m \lg m)$ to sort the skis + $\Theta((m-n)n)$ to compute A. Total time: $\Theta(n \lg n + m \lg m + (m-n)n)$. Notice that when m = n, this reduces to $\Theta(n \lg n)$, like the simple algorithm shown above for the m = n case.

Problem 2: Building Towers (20 points)

Professor Babylonia wants to construct the tallest tower possible out of building blocks. She has n types of blocks, and an unlimited supply of blocks of each type. Each type-i block is a rectangular solid with linear dimensions $\langle x_i, y_i, z_i \rangle$. A block can be oriented so that any two of its three dimensions determine the dimensions of a base and the other dimension is the height. In building a tower, one block may be placed on top of anything block as long as the two dimensions of the upper block's base are each strictly smaller than the corresponding base dimensions of the lower block. (So, for example, blocks oriented to have equal-sized bases cannot be stacked.) Design an efficient algorithm to determine the tallest tower that the professor can build. (You must account for the fact that the blocks can be reoriented.

First of all, note that although there are an unlimited supply of blocks, at most 3 blocks of each type — one with x_i as a height, one with y_i as the height, and one with z_i as the height — can appear in the tower, because a surface can only be placed on a strictly larger surface, so a block can't be used twice with the same nase surface. So we can transform out problem to one with 3n blocks. (In fact, only two blocks of each type can possibly be used in the tower, but we don't know in advance which two.) For each block type i, we have three blocks: one of height z_i and base dimensions x_i and y_i, one with height y_i with base dimensions x_i and z_i, and one with height z_i with base dimensions y_i and z_i. Let's call the dimensions of the block **height**, **width** (the smaller base dimensions) and **length** (the larger base dimension). Then block a can be placed on block b if and only if length[a] < length[b] and width[a] < width[b].

The dynamic programming approach is as follows:
First sort the blocks in reverse order by width. In the sort, width[B_i] < width[B_j] only if i > j, so a block can be placed only on blocks that appear earlier in the list.
Let H_i be the height of the tallest tower that can be built with B_i on top. That tower consists of B_i on top of the tallest tower that can be built with some tower B_j on top that can support B_i, which is some B_j with j < i, length[B_j] > length[B_i], and width[B_j] > width[B_i]. (The width must be checked, because some of B_j may have the same width as B_i.) Define the set

$$\text{bigger}_i = \{ j : j < i, \text{length}[B_j] > \text{length}[B_i], \text{length}[B_j] > \text{length}[B_i] \}$$

so that

$$H_0 = 0$$
$$H_i = \text{height}[B_i] + \max_{j \ \text{bigger}_i} H_j$$

Since the H value depends only on earlier H values, we can compute a table H from left to right. The height of the tallest tower is then $\max_{1 \le i \le 3n} H[i]$. To find the blocks in the tallest tower, we can keep track of which blocks are placed on which. That is, when the max j is chosen to place i on, keep a pointer from i to j.

The chain of pointers starting at the i for which H[i] is maximum shows the blocks that made up that H[i]-height tower.

The run time is

- $O(n \lg n)$ to sort the 3n blocks.
- $O(n^2)$ to compute the $O(n)$ entries in H if the max is found by scanning H (because computing H[i] must scan i entries, so the entries scanned for all i form an arithmetic series)
- $O(n)$ to find the maximum height in H and trace the chain of blocks.

for a total of $O(n^2)$.

Problem 3: Multiple Lecture Halls (15 points, courtesy of CLR, Exercise 17.1-2)

Suppose we have a set of activities to schedule among a large number of lecture halls. We wish to schedule all of the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall. There is an obvious algorithm that runs in $O(n^2)$time, but there is even better algorithm whose running time is bounded by the time is takes to sort the events.

Let S be the set of n activities.

The obvious solution of using Greedy-Activity-Selector to find a maximum-size set S1 of compatible activities from S for the first lecture hall, then using it again to find a maximum-size set S2 of campatible activities from S - S1 for the second lecture hall, (and so on until all the activities are assigned), requires $O(n^2)$ time in the worst case.

There is a better algoritm, whose aymptotic running time is just the time is takes to sort the activities by time — $O(n \lg n)$ time for arbitrary times, or possibly as fast as $O(n)$ if the times are small integers. The general idea is to go through all the activities in order of start time, assigning each to any hall that is available at that time. To do this, move through the set of events consisting of all activities starting and activities finishing, in the order of event time. Maintain two lists of lecture halls: Halls that are busy at the current event-time t (because they have been assigned an activity i that started at $s_i \le t$ but won't finish until $f_i > t$) and halls that are free at time t. (As in the activity-selection problem in Section 17.2, we are assuming that activity time intervals are half-open — i.e. that $s_i \ge f_j$, activities I and j are compatible.) When t = the start time of some activity, assign that activity to a free lecture hall and move the hall from the free list to the busy list. When t = the finish time of some activity, move the activity's hall from the busy list to the free list. (The activity is certainly in some hall, because event times are processed in order and the activity must have started before its finish time, hence must have been assigned to a hall.)

To avoid using more halls than necessary, always pick a lecture hall that has had an event previously assigned to it, if possible, before picking a new lecture hall. (This can be done by always working with the front of the free-halls list — putting freed halls onto the front of the list and taking halls from the front of the list — so that a new hall doesn't come to the front of the list and get chosen if there are previously-used halls available.)

This guarantees that the algorithm uses as few lecture halls as possible: The algorithm will terminate with a schedule requiring $m \le n$ lecture halls. Let i be the first activity scheduled in lecture hall m. The reason that i was put in the mth lecture hall was is that the first m - 1 lecture halls were busy at time s_i. So at this time, there are m activities occuring simultaneously. Therefore, any schedule must use at least m lecture halls, so the schedule returned by the algorithm is optimal.

Running time:

- Sort the 2n activity-starts / activity-ends events. (In the sorted order, an event-ending event should precede an event-starting event at the same time.) $O(n \lg n)$ time for arbitrary times, possibly $O(n)$ if the times are restricted to a small ordered domain.
- Process the events in $O(n)$ time: Scan the 2n events, doing $O(1)$ work for each (moving a hall from one list to another and possibly associating an activity with it.)

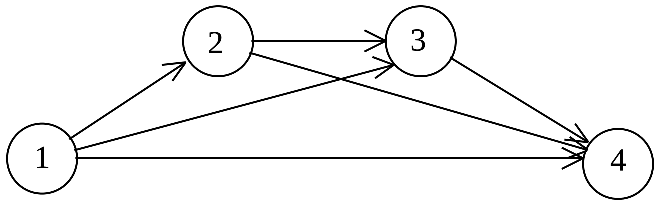
Total: $O(n + \text{time to sort})$

Problem 4: Topological Sorting Issues (15 points, courtesy of CLR, Exercise 23.4-2)

There are many different orderings of the vertices of the vertices of a directed graph G that are topological sorts of G. `Topological-Sort` (as presented on pp. 485-487 of the text) produces an ordering that is the reverse of the depth-first finishing times. Show that there exists a graph for which two distinct adjacency list representations yield the same topological sort.

This problem was considerably easier, since the first part of it was cut from the assignment. We were left with the task of presenting two adjacency list representations of the same graph which yield the same topological sort.

Consider a directed acyclic graph that represents the total ordering (not just a partial ordering) of the first four natural numbers.



There is only one topological sort (1 2 3 4), but any of the 12 different adjacency list representations is constrained to yield the same output.