

CS 170 Spring 2008 - Solutions to Homework #8

Problem 6.1 (12 points)

Subproblems: Define an array of subproblems $D(i)$ for $0 \leq i \leq n$. $D(i)$ will be the largest sum of a (possibly empty) contiguous subsequence ending exactly at position i .

Algorithm and Recursion: The algorithm will initialize $D(0) = 0$ and update the $D(i)$'s in ascending order according to the rule:

$$D(i) = \max\{0, D(i-1) + a_i\}$$

The largest sum is then given by the maximum element $D(i)^*$ in the array D . The contiguous subsequence of maximum sum will terminate at i^* . Its beginning will be at the first index $j \leq i^*$ such that $D(j-1) = 0$, as this implies that extending the sequence before j will only decrease its sum.

Correctness: The contiguous subsequence of largest sum ending at i will either be empty or contain a_i . In the first case, the value of the sum will be 0. In the second case, it will be the sum of a_i and the best sum we can get ending at $i-1$, i.e. $D(i-1) + a_i$. Because we are looking for the largest sum, $D(i)$ will be the maximum of these two possibilities.

Running Time: The running time for this algorithm is $O(n)$, as we have n subproblems and the solution of each can be computed in constant time. Moreover, the identification of the optimal subsequence only requires a single $O(n)$ time pass through the array D .

Problem 6.2 (12 points)

Subproblems: Define subproblem $D(i)$ to be the minimum total penalty starting from 0 and ending at a_i . Since for any a_i , there must be a stop at some a_j ($a_i - a_j \leq 200$) and the corresponding penalty is $D(j) + [200 - (a_i - a_j)]^2$. Minimizing the penalties corresponding to all j 's gives $D(i)$.

Algorithm and Recursion: We need to minimize the total penalty among all the possibilities and therefore we have the following recursion:

$$D(i) = \min_{j: a_i - a_j \leq 200} \{D(j) + [200 - (a_i - a_j)]^2\},$$

with base condition $D(i) = (200 - a_i)^2, \forall a_i \leq 200$. To recover the optimal itinerary, we can keep track of a maximizing j for each $D(j)$ and use this information to backtrack from $D(n)$.

Correctness: To solve $D(i)$ we consider all possible hotels j we can stay at on the night before reaching hotel i : for each of these possibilities, the minimum penalty to reach i is the sum of the cost of a one-day trip from j to i and the minimum penalty necessary to reach j . Because we are interested in the minimum penalty to reach i , we take the minimum of these values over all j 's.

Running Time: The running time is $O(n^2)$, as we have n subproblems and each takes time $O(n)$ to solve, as we need to compute the minimum of $O(n)$ values. Moreover, backtracking only takes time $O(n)$.

Problem 6.13 (16 points total: 4 points part (a), 12 points part (b))

- (a) Consider the sequence $(2, 9, 1, 1)$. Then, the best available card at the start has value 2, but this leads only to a score of 3, as the opponent picks the card of value 9 in the next turn. Instead, choosing the 1 at the start yields a final score of 10.
- (b) *Subproblems:* Define subproblems $A(i, j)$ for $i \leq j$, where $A(i, j)$ is the largest difference between the total score the current player (refers to the player who chooses a card now) can obtain and the corresponding score of the next player (refers to the player who chooses a card following the current player) can obtain when playing on sequence s_i, s_{i+1}, \dots, s_j . This will be positive if and only if the first player has the larger total.

Algorithms and Recursion: We can solve all the subproblem by initializing $A(i, i) = s_i$ for all i and using the update rule for $i < j$:

$$A(i, j) = \max\{s_i - A(i + 1, j), s_j - A(i, j - 1)\}$$

We can also keep track of the optimal move at each stage by simultaneously maintaining a matrix $M(i, j)$, where $M(i, j)$ will be set to **first** if $A(i, j)$ takes the value of the first element in the maximization, and to **last** otherwise.

Correctness and Running Time: The recursion is correct, as at any stage of the game there are two possible moves for the current player: either choose the first card, in which case he will gain this amount $(s_i - A(i + 1, j))$ of more scores than the other player, or the last card, in which case he will gain this amount $(s_j - A(i, j - 1))$ of more scores than the other player. Both player adopt the same strategy of maximizing the difference of the total score he himself can gain and the other player can gain. The first player who chooses card first will always win. The algorithm computes all the subproblems and the entries of M in time $O(n^2)$ as each update takes

time $O(1)$. Moreover, the player can then just read off the matrix M to learn the best strategy at any point of the game.

Problem 6.14 (12 points)

Subproblems: Define XY subproblems. For $1 \leq i \leq X$ and $1 \leq j \leq Y$, let $C(i, j)$ be the best return that can be obtained from a cloth of shape $i \times j$. Define also a function **rect** as follows:

$$\mathbf{rect}(i, j) = \begin{cases} c_k & \text{if there exists a product } k \text{ with } a_k = i \text{ and } b_k = j \\ 0 & \text{if no such product exists} \end{cases}$$

Algorithm and Recursion: Then the recursion is:

$$C(i, j) = \max\left\{\max_{1 \leq k < i}\{C(k, j) + C(i - k, j)\}, \max_{1 \leq h < j}\{C(i, h) + C(i, j - h)\}, \mathbf{rect}(i, j)\right\}$$

It remains to initialize the smallest subproblems correctly: $C(i, 0) = 0, \forall 1 \leq i \leq X$, and $C(0, j) = 0, \forall 1 \leq j \leq Y$. Therefore, the $C(i, j)$ should be calculated in order of increasing i (from 1 to X), and for each i , in order of increasing j (from 1 to Y). The final solution for this problem is the value of $C(X, Y)$.

Correctness and Running Time: For proving correctness, notice that in order to obtain the best gain $C(i, j)$ from a cloth with dimension i and j , we can try all possibilities: horizontal cuts, vertical cuts or no cut. The one which gives the best gain should be chosen. As this is performed recursively, the cloth is getting smaller and smaller. In the base case a dimension is zero, so the gain is zero. The running time is $O(XY(X + Y + n))$ as there are XY subproblems and each takes $O(X + Y + n)$ time to evaluate.

Problem 6.21 (12 points)

The subproblem $V(u)$ will be defined to be the size of the minimum vertex cover for the subtree rooted at node u . We have $V(u) = 0$ if u is a leaf, as the subtree rooted at u has no edges to cover. The crucial observation is that if a vertex cover does not use a node it has to use all its neighboring nodes. Hence, for any internal node i

$$V(i) = \min \left\{ \sum_{j:(i,j) \in E} \left(1 + \sum_{k:(j,k) \in E} V(k) \right), 1 + \sum_{j:(i,j) \in E} V(j) \right\}$$

The algorithm can then solve all the subproblems in order of decreasing depth in the tree and output $V(n)$. The running time is linear in n because while calculating $V(i)$ for all i we look at most at $2 * |E| = O(n)$ edges in total.

Problem 6.26 (12 points)

This is a simple variant of the edit distance algorithm defined in class. The recursion is modified to:

$$E(i, j) = \max\{E(i-1, j) + \delta(x_i, -), E(i-1, j-1) + \delta(x_i, y_j), E(i, j-1) + \delta(-, y_j)\}$$

The initialization has also to be modified to deal properly with the new scoring for gaps. We have, for $i, j > 0$:

$$\begin{aligned} E(0, 0) &= 0 \\ E(i, 0) &= E(i-1, 0) + \delta(x_i, -) \\ E(0, j) &= E(0, j-1) + \delta(-, y_j) \end{aligned}$$

The correctness follows by the same argument as for the edit distance algorithm. The running time is again $O(mn)$.

Problem 6.27 (12 points)

This is another modification of the standard edit distance algorithm, only more advanced. You should notice that there is no simple way to modify the existing recursion to incorporate the new kind of gap penalties. In particular, it would be necessary at every point to know whether the previous subproblem solutions were given by alignments with a deletion or insertion in their last position. Moreover, it might be that an optimal alignment for $E(i, j)$ is not an extension of the optimal for $E(i-1, j)$, $E(i-1, j-1)$, or $E(i, j-1)$ since an alignment with a smaller previous score but terminating with a gap might have a smaller gap penalty and beat all extensions of optimal alignments. This suggests that we should not keep a single matrix of subproblems, but 3. E will be the matrix of subproblems where the alignments are constrained to be a substitution or a match in their last position. E_x will be the matrix of subproblems over the alignments which have a gap in the last position of string x . E_y is defined similarly for y . Given these definitions, we just need to work out the recursion correctly case by case.

$$\begin{aligned} E(i, j) &= \max\{E(i-1, j-1), E_x(i-1, j-1), E_y(i-1, j-1)\} + \delta(x_i, y_j) \\ E_x(i, j) &= \max\{E(i-1, j) - c_0 - c_1, E_x(i-1, j) - c_1, E_y(i-1, j) - c_0 - c_1\} \\ E_y(i, j) &= \max\{E(i, j-1) - c_0 - c_1, E_x(i, j-1) - c_0 - c_1, E_y(i, j-1) - c_1\} \end{aligned}$$

The output will just be the maximum of $E(m, n)$, $E_x(m, n)$ and $E_y(m, n)$. This takes $O(mn)$ as we still have $O(mn)$ subproblems, each evaluated in constant time.