

COMP3600/COMP6466 in 2008 – Tutorial Two Solutions

Question 1.

Give an $O(n^2)$ -time dynamic programming algorithm to find a longest decreasing subsequence of a sequence of n numbers.

Let x_1, x_2, \dots, x_n be the given sequence. The key property is: for any j , the longest decreasing sequence ending with x_j either consists only of x_j or it consists of a longest decreasing sequence ending with some earlier x_i that is larger than x_j .

This gives a recurrence. Define $D(j)$ to be the length of the longest decreasing sequence ending with x_j , for $1 \leq j \leq n$. Then, for $1 \leq j \leq n$,

$$D(j) = \max_{i < j \ \& \ x_i > x_j} \{1, D(i) + 1\}.$$

Use the recurrence to compute $D(1), D(2), \dots, D(n)$, in that order. Then the largest $D(j)$ is the answer to the problem.

Question 2.

Show how to reconstruct an LCS from the completed c table and the original sequence $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ in $O(m + n)$ time, without using the b table.

The LCS recurrence tells us to choose between 3 cases, and the b array is used to remember which of those cases we chose. However we can deduce that information by inspecting the c array and the two input strings. If $x_i = y_j$, $b[i, j]$ is the diagonal arrow. Otherwise, $b[i, j]$ is a horizontal or vertical arrow depending on whether $c[i-1, j] = c[i, j]$ or $c[i, j-1] = c[i, j]$. At least one of these must be true (by the recurrence); if both are true then both horizontal and vertical arrows are valid.

Question 3.

Consider the problem of making change for n cents using the least number of coins. Does the obvious greedy algorithm work for coin denominations 100, 50, 20, 10, 5? What if the coin denominations were 17, 10, 4, 2, 1? Give an algorithm that always works.

The “obvious greedy algorithm” is to repeatedly choose the largest coin that is not too large.

The greedy algorithm works for 100, 50, 20, 10, 5. First note that an optimal choice never has more than one 50 (since two 50s could be replaced by one 100), no more than two 20s, one 10, or one 5. Thus, any amount bigger than 105 requires use of the 100 coin. All values 105 or less are easily considered one at a time.

For 17, 10, 4, 2, 1 the greedy algorithm fails. For example, making 20 from 10+10 is few coins than 17+2+1 or 17+1+1+1.

To make a general algorithm, note that removing any coin from an optimal solution leaves an optimal solution for the smaller amount. Define $M(n)$ be the minimum number of coins needed to make n cents. Then $M(0) = 0$ and, for $n > 0$, $M(n)$ is the smallest of

$$M(n-17) + 1, M(n-10) + 1, M(n-4) + 1, M(n-2) + 1, M(n-1) + 1,$$

where values with negative arguments are ignored.

This gives an $\Theta(n)$ algorithm for any fixed set of coin denominations.

Question 4.

Suppose you are given two sets A and B , each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the i th element of set A , and let b_i be the i th element of set B . You then receive a payoff of $\prod_{i=1}^n a_i^{b_i}$. Given an algorithm that will *maximize* your payoff. Prove that your algorithm maximizes the payoff and state its running time.

With a little trial and error, we can conjecture that the b_i 's and a_i 's should be ordered consistently. That is, the smallest b_i appears with the smallest a_i , and so on. If this is true, we can achieve the optimum by sorting $a_1 \leq a_2 \leq \dots \leq a_n$ and $b_1 \leq b_2 \leq \dots \leq b_n$.

Suppose on the contrary that $a_1^{b_1} a_2^{b_2} \dots a_n^{b_n}$ has the greatest value, but for some $i \neq j$ we have $a_i < a_j$ and $b_i > b_j$. Now replace the terms $a_i^{b_i}, a_j^{b_j}$ by $a_i^{b_j}, a_j^{b_i}$. We have

$$\frac{\text{new value}}{\text{old value}} = \frac{a_i^{b_j} a_j^{b_i}}{a_i^{b_i} a_j^{b_j}} = \left(\frac{a_j}{a_i}\right)^{b_i - b_j} > 1.$$

That is, the new value is bigger than the old value, showing that $a_1^{b_1} a_2^{b_2} \dots a_n^{b_n}$ is not optimal (contradicting our assumption).

The running time is the time for sorting: $O(n \lg n)$.

Question 5.

Explain how to implement a CHANGE-KEY procedure in a max-heap. That is, given an element in the heap and a new key value, change the key of the element to the new value. The new key value might be either larger or smaller than the old key value.

If the key becomes larger, it might be larger than the key in its parent. This can be fixed by swapping upwards, as in HEAP-INCREASE-KEY. If the key becomes smaller, it might be smaller than the key in one or both of its children. This can be fixed by swapping downwards, as in MAX-HEAPIFY.