

This homework is due Thursday, March 26, in class. Please place it in the folder for homework at the start of class. The questions are drawn from the material in class and in Chapter 15 of text on dynamic programming.

The homework is worth a total of 100 points. If a point breakdown is not given for each part of a problem, each part has equal weight.

For questions that ask you to design a *dynamic programming* algorithm, remember to use the four-part framework:

- (1) *characterize* the recursive structure of an optimal solution,
- (2) *derive* a recurrence equation for the value of an optimal solution,
- (3) *evaluate* the recurrence bottom-up in a table, and
- (4) *recover* an optimal solution from the table of solution values.

You will be graded on each part. Be sure to analyze the time for Parts (3) and (4) of your algorithm.

Please write on just one side of a page, do not use scrap paper, put your answers in the correct order, and staple your pages together. If you can't solve a problem, state this, and write only what you know to be correct. Neatness and conciseness count.

- (1) **(Matrix chain multiplication)** (10 points) Find the optimal multiplication order for an instance of Matrix-Chain Multiplication on 6 matrices whose sequence of row- and column-dimensions is (5, 10, 3, 12, 5, 50, 6). Show your work by presenting the filled-in dynamic programming table.

- (2) **(Longest common subsequence length)** (15 points) Show how to compute just the *length* of a longest common subsequence of two strings of m and n characters, where $m \leq n$, using only $O(m)$ additional space beyond the input strings.

(Note: Filling in the entire dynamic programming table takes $\Theta(mn)$ space, which exceeds the space bound for this problem. It is actually possible to both compute the length and *find* the longest common subsequence itself in $O(m)$ space and $O(mn)$ time, though this is much more complicated.)

- (3) **(Longest increasing subsequence)** (20 points) Given a string S of numbers, an *increasing subsequence* of S is any subsequence T of S such that the numbers in T , read left-to-right across T , are strictly increasing. For example, if $S = (3, 1, 6, 2, 5, 4)$, an increasing subsequence of S is $T = (1, 2, 4)$.

Design a dynamic programming algorithm that finds the *longest* increasing subsequence of a string S of length n in $O(n^2)$ time.

(Note: Do *not* solve this problem by reducing it to the longest common subsequence problem. Design a dynamic programming algorithm from first principles.)

- (4) **(Editing strings)** (30 points) Given two strings $A[1 : m]$ and $B[1 : n]$, the *edit distance* between A and B is the minimum cost of a script that edits A into B . A script is a series of edit operations, each edit operation has a non-negative cost, and the cost of a script is the sum of the costs of its operations.

The allowed edit operations in a script are:

- *copy*, which leaves a character unchanged, and has cost 0,
- *substitute*, which replaces a character a with another character b , and has cost c_{sub} ,

- *insert*, which adds a character a into a string, and has cost c_{ins} ,
- *delete*, which removes a character a from a string, and has cost c_{del} , and
- *transpose*, which replaces two adjacent characters ab in a string by the characters ba , and has cost c_{tra} .

Design a dynamic programming algorithm to compute the edit distance between A and B and recover the corresponding edit script in $\Theta(mn)$ time. You may assume that an optimal script never edits a given character more than once. The costs of operations are part of the input to your algorithm.

(Hint: Order the operations in an edit script so they occur left-to-right across string A , and then examine the possible ways in which an optimal script could end.)

- (5) **(Discrete knapsack)** (25 points) In the *discrete knapsack problem*, the input is a collection of n items with associated weights w_1, w_2, \dots, w_n and values v_1, v_2, \dots, v_n , and a capacity k . Item i has weight w_i and value v_i . All weights w_i and the capacity k are positive integers. The output is a subset S of the items $\{1, 2, \dots, n\}$, called a knapsack, such that the total weight of all the items in knapsack S is at most k , and the total value of all the items in S is maximum. In other words, a solution to the discrete knapsack problem is an optimal knapsack S of items that does not exceed the weight capacity k while having the greatest possible value.

Design a dynamic programming algorithm that solves the discrete knapsack problem in $\Theta(nk)$ time.

(Hint: Examine the items in the order $1, 2, \dots, n$, and consider knapsacks of all possible capacities.)