CSC 445 Homework 3 (version 1) Fall 2022

This homework is due Thursday, October 13, at 3:30 pm. Please upload a single `PDF` file containing your submission to `Gradescope` by that time (ensuring scans of handwritten work are legible by using a free app such as `TurboScan` or `CamScanner`, and marking which pages of your submission correspond to each homework problem).

The questions are drawn from the material in class, and in Chapters 9 and 15 of the text, on finding the *kth-smallest element*, and the *dynamic programming* technique.

The homework is worth a total of 100 points. When point breakdowns are not given for the parts of a problem, each part has equal weight.

For general algorithm design questions, (i) present the ideas behind your algorithm in prose and pictures, (ii) argue that your algorithm is correct, and (iii) show your analysis that the algorithm meets the given time bound. Pseudocode is not required, but may aid in your analysis.

For questions that ask you to design a *dynamic programming* algorithm, be sure 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.

Remember to (a) start each problem on a *new page*, (b) put your answers in the *correct order*, and (c) indicate on `Gradescope` the *correspondence* between homework problems and the pages of your submission. If you can't solve a problem, state this, and write only what you know to be correct. Neatness and conciseness counts.

(1) **(Finding elements near the median)** (10 points) Given an unsorted array A of n distinct numbers and an integer k, where $1 \le k \le n$, design an algorithm that finds the k numbers in A that are *closest in value* to the median of A in $\Theta(n)$ time.

(Note: Finding the numbers that are closest in *value* to the median has no relationship in general with how these numbers are ordered in the unsorted array A.)

(2) **(Finding quantiles)** (20 points) For a set S of n numbers and an integer k, where $1 \le k \le n$, the *kth-quantiles* of S are $k-1$ elements from S whose ranks in S divide the sorted set into k groups that are of equal size (to within one unit).

Given an unsorted array A of n distinct numbers, design an algorithm that finds the kth-quantiles of A in $O(n \log k)$ time.

(Note: As an illustration, the 4-quantiles of a set of scores are the values that define the 25-, 50-, and 75-percentile cutoffs. Similarly, the 10-quantiles of a set are the values that define the 10-, 20-, 30-, ..., and 90-percentile cutoffs.)

(3) **(Answering dynamic kth-smallest queries) (bonus)** (10 points) Given a set of n numbers, the kth-smallest element can be found in $\Theta(n)$ time using the algorithm we learned in class. Suppose we have a dynamic set S of numbers that changes over time, and we want to be able to efficiently support the operations of

(a) *inserting* a new number into set S, and
(b) *finding* the kth-smallest element in the current set S, for any k.

We could support both operations by representing S as an unsorted array A. Inserting a new element would take just $\Theta(1)$ time. Finding the kth-smallest element, however, would take $\Theta(n)$ time. If there are many kth-smallest operations executed on set S, performing all of them will take a large amount of total time. We might like to speed up the time to find the kth-smallest, at a slight increase in the time to insert an element.

Design an algorithm that (a) supports inserting an element into S in $O(\log n)$ time, and (b) supports finding the kth-smallest element of S in $O(\log n)$ time as well.

(Hint: Consider modifying a balanced binary search tree data structure.)

(4) **(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 $\Theta(n^2)$ time.

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

(5) **(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.)

(6) **(Discrete knapsack)** (20 points) In the *discrete knapsack problem*, the input is a collection of n items with associated weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$, 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 $\{1, 2, \ldots, n\}$, 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 $1, 2, \ldots, n$, and consider knapsacks of all possible capacities.)

Note that Problem (3) is a *bonus* question, and is *not* required.