CS5050 Advanced Algorithms

Fall 2023

Assignment 5: Algorithm Analysis

Due Date: 11:59:59 p.m., *Thursday, Nov 16, 2023*

Total Points: 80

Note: For each of the following problems, you need to design a dynamic programming algorithm. When you describe your algorithm, you are asked to clearly explain the **subproblems** and the **dependency relation** of your algorithm.

1. **(20 points)** The knapsack problem we discussed in class is the following. Given a knapsack of size M and n items of sizes $\{a_1, a_2, \ldots, a_n\}$, determine whether there is a subset S of the items such that the sum of the sizes of all items in S is exactly equal to M. We assume M and all item sizes are positive integers.

   Here we consider the following *unlimited version* of the problem. The input is the same as before, except that there is an unlimited supply of each item. Specifically, we are given n item sizes $a_1, a_2, \ldots, a_n$, which are positive integers. The knapsack size is a positive integer M. The goal is to find a subset S of items (to pack in the knapsack) such that the sum of the sizes of the items in S is exactly M and each item is allowed to appear in S multiple times.

   For example, consider the following sizes of four items: $\{2, 7, 9, 3\}$ and $M = 14$. Here is a solution for the problem, i.e., use the first item once and use the fourth item four times, so the total sum of the sizes is $2 + 3 \times 4 = 14$ (alternatively, you may also use the first item 2 times, the second item one time, and the fourth item one time, i.e., $2 \times 2 + 7 + 3 = 14$).

   Design an $O(nM)$ time dynamic programming algorithm for solving this unlimited knapsack problem. For simplicity, you only need to determine whether there exists a solution (namely, you answer is just "yes" or "no"; if there exists a solution, you do not need to report an actual solution subset).

2. **(20 points)** Here is another variation of the knapsack problem. We are given n items of sizes $a_1, a_2, \ldots, a_n$, which are positive integers. Further, for each $1 \leq i \leq n$, the i-th item a_i has a positive value $value(a_i)$ (you may consider $value(a_i)$ as the amount of dollars the item is worth). The knapsack size is a positive integer M.

   Now the goal is to find a subset S of items such that the sum of the sizes of all items in S is **at most** M (i.e., $\sum_{a_i \in S} a_i \leq M$) and the sum of the values of all items in S is **maximized** (i.e., $\sum_{a_i \in S} value(a_i)$ is as large as possible). Note that each item can be used at most once in this problem.

   Design an $O(nM)$ time dynamic programming algorithm for the problem. For simplicity, you only need to report the sum of the values of all items in the optimal solution subset S and you do not need to report the actual subset S.

3. **(20 points)** In class, we studied the longest common subsequence problem. Here we consider a similar problem, called *maximum-sum common subsequence problem*, as follows. Let A be an array of n numbers and B another array of m numbers (they may also be considered as two sequences of numbers). A *maximum-sum common subsequence* of A and B is a common subsequence of the two arrays that has the maximum sum among all common subsequences of the two arrays (see the example given below). As in the longest common subsequence problem studied in class, a subsequence of elements of A (or B) is not necessarily consecutive but follows the same order as in the array. Note that some numbers in the arrays may be **negative**.

Design an $O(nm)$ time dynamic programming algorithm to find the maximum-sum common subsequence of A and B. For simplicity, you only need to return the sum of the elements in the maximum-sum common subsequence and do not need to report the actual subsequence.

Here is an example. Suppose $A = \{36, -12, 40, 2, -5, 7, 3\}$ and $B = \{2, 7, 36, 5, 2, 4, 3, -5, 3\}$. Then, the maximum-sum common subsequence is $\{36, 2, 3\}$. Again, your algorithm only needs to return their sum, which is $36 + 2 + 3 = 41$.

4. **(20 points)** Given an array $A[1 \dots n]$ of n distinct numbers (i.e., no two numbers of A are equal), design an $O(n^2)$ time dynamic programming algorithm to find a *longest monotonically increasing subsequence* of A (see the definition below). Your algorithm needs to report not only the length but also the actual longest subsequence (i.e., report all elements in the subsequence).

   Here is a formal definition of a *longest monotonically increasing subsequence of* A (refer to the example given below). First of all, a *subsequence* of A is a subset of numbers of A such that if a number a appears in front of another number b in the subsequence, then a is also in front of b in A (i.e., the subsequence follows the same order as in A). Next, a subsequence of A is *monotonically increasing* if for any two numbers a and b such that a appears in front of b in the subsequence, a is smaller than b. Finally, a *longest monotonically increasing subsequence of* A refers to a monotonically increasing subsequence of A that is the longest (i.e., has the maximum number of elements) among all monotonically increasing subsequences of A.

   For example, if $A = \{20, 5, 14, 8, 10, 3, 12, 7, 16\}$, then a longest monotonically increasing subsequence is $5, 8, 10, 12, 16$. Note that the answer may not be unique, in which case you only need to report one such longest subsequence.