CS 5150/6150 Graduate (Advanced) Algorithms
Fall 2022
Assignment 5: Dynamic Programming
Total Points: 80

Submission notes

- Due at 11:59 pm on **Thursday, November 10, 2022**.
- Solutions must be typeset (not hand-written or scanned).
- You should strive to write clear, concise solutions to each problem, ideally fitting on a page or less. The easier your work is to read & follow, the easier it is for the TAs to award you points.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the grading process, so please retain these until an assignment has been returned.
- Please remember that for problem sets, *collaboration with other students must be limited to a high-level discussion of solution strategies.* If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration at the top of your homework submission. **You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.**

Instructions on How to Answer Algorithm Design Questions

Most homework questions in this semester will be algorithm design questions. Please read the following instructions on how to answer them properly (you may lose points if you do not follow the instructions).

1. **Algorithm Description** You are required to clearly describe the main idea of your algorithm.

2. **Pseudocode** Providing pseudocode for your algorithm is optional. However, pseudocode is very helpful for explaining algorithms clearly, and you are *strongly encouraged* to provide pseudocode for your algorithms. Additionally, having pseudocode available often leads to more partial credit for incorrect solutions, as the TAs can more easily identify correct ideas in your solution.

   Note: copying code from an implementation (e.g. python/java/c++) as "pseudocode" does not satisfy this requirement, as grading it is difficult, and key steps may be difficult to identify. In principle, pseudocode should be relatively easy to read while keeping all relevant operations of the algorithms. *You are encouraged to do your best to write good pseudocode, following examples given in class.*

3. **Correctness** You also need to explain/argue why your algorithm is correct, and handles all necessary cases – especially when the correctness of your algorithm is not obvious.

4. **Time Analysis** You are required to analyze the time complexity of your algorithm (i.e., provide a written justification for the claimed asymptotic runtime in terms of the algorithm's operations, not just state the time complexity). You may refer to your pseudocode in this analysis, as it often simplifies the descriptors (e.g. "the `for` loop on line 2").

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 put 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 $A[1 \ldots n]$ 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 formal 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 may report any one of such longest subsequences.