Design and Analysis of Algorithms

Practice sheet on Dynamic Programming

1. **Monotonically increasing subsequence**

   Given a sequence $A = a_1, \ldots, a_n$, a subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ is said to be monotonically increasing if $a_{i_j} < a_{i_{j+1}}$ for all $1 \leq j < k$. Design an $O(n^2)$ time algorithm to compute the longest monotonically increasing subsequence of sequence A.

   Hint: Can you relate this problem to some problem discussed in the lectures ?

2. **Box stacking**

   Box Stacking. You are given a set of n types of rectangular 3-D boxes, where the ith box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

   (a) Design a dynamic programming based algorithm for the problem.

   (b) Formulate the problem in terms of a suitable problem on a directed acyclic graph problem and solve it using topological ordering.

   Hint for (a): Let us consider 3 orientations of each box. So there are total $3n$ distinct boxes each with fixed orientation. For each pair of boxes, we can determine whether one box can be placed on the other one. Let $b_1, \ldots b_{2n}$ be the sequence of the boxes arranged in the increasing order of their base area. Let $H(i)$ be the height of the tallest stack of boxes we can form with the box b_i at the bottom. Proceed along these lines ...
   Hint for (b): Create the DAG as follows: For each box, create 3 nodes corresponding to its orientations. Add edges between a pair of nodes (and assign direction suitably) if some *obvious* condition is satisfied. Argue why the resulting graph is a DAG. Assign weight to each node suitably. What will be the formulation of the problem on the DAG ?

3. **Bridges across a river**

   Consider a 2-D map with a horizontal river passing through its center. There are n cities on the southern bank with x-coordinates $a(1)...a(n)$ and n cities on the northern bank with x-coordinates $b(1)...b(n)$. Note that there is no order among $a(i)$'s. Similarly, you can't assume any order among $b(i)$'s. You want to connect as many north-south pairs of cities as possible with bridges such that no two bridges cross. When connecting cities, you can only connect city i on the northern bank to city i on the southern bank. Design a polynomial time algorithm to compute the maximum number of non-crossing bridges that can be built.

   Hint: Can you formulate it in terms of some problem discussed in the lectures ?

4. **Maximum wright independent set**

There is an undirected graph $G = (V, E)$ consisting of n vertices $v_1, \ldots, v_n$. For each $1 \leq i < n$, there is an edge between v_i and v_{i+1}. Hence there are only $n - 1$ edges. Vertex v_i has a weight w_i which is a positive real number. A subset $S \subseteq V$ is said to be an independent set if for each $x, y \in S$, $(x, y) \notin E$. Weight of an independent set is the sum of weights of its vertices. Our aim is to compute the maximum weight independent set. You need to design a Dynamic Programming based algorithm for this problem along the following lines.

(a) Introduce the recursive term and define it formally and completely in the following box.

(b) Define the base case suitably in the following box.

(c) Give recursive formulation of the term in the following box.

(d) Give justification for the recursive formulation in the following box.

(e) How many total subproblems you need to solve in order to get the solution of the problem ?

5. **Edit Distance**

   Given two text strings A of length n and B of length m, you want to transform A into B with a minimum number of operations of the following types: delete a character from A, insert a character into A, or change some character in A into a new character. The minimal number of such operations required to transform A into B is called the edit distance between A and B. Design a polynomial time algorithm to compute edit distance between A and B.

   Hint: Define a term $T(i, j)$ suitably. What is the minimum number of steps to transform A into $B[1..j]$ if A is empty ? If ith character of A has to be retained in transforming $A[1..i]$ to $B[1..j]$, how will the optimal transformation look like ? Likewise, what if the ith character of A has to be deleted or changed ?

6. **Matrix chain multiplication**

   There is a sequence of matrices $M_1, M_2, \ldots, M_n$ storing numbers. For each $1 < i \leq n$, the number of rows of M_i is identical to the number of columns of M_{i-1}. So the product $M_1 \times M_2 \times \ldots M_n$ is well defined. We also know that matrix multiplication is associative. That is, $(M_1 \times M_2) \times M_3 = M_1 \times (M_2 \times M_3)$. However, the number of arithmetic operations required may vary in these two possible ways. For example, let M_1 be 10×100, M_2 be 100×5, and M_3 be 5×50.

   - If we multiply according to $((M_1 \times M_2) \times M_3)$, we perform $10 \cdot 100 \cdot 5 = 5000$ multiplication operations to compute the 10×5 matrix $(M_1 \times M_2)$ and then $10 \cdot 5 \cdot 50$ multiplication operations to multiply this matrix with M_3. So a total of 7500 multiplication operations are carried out.
   - If instead we multiply according to $(M_1 \times (M_2 \times M_3))$, we perform $100 \cdot 5 \cdot 50 = 25000$ scalar multiplications to compute 100×50 matrix $(M_2 \times M_3)$ and then another $10 \cdot 100 \cdot 50 = 50000$ multiplication operation to multiply M_1 by this matrix. Hence a total of 75000 multiplication operations are carried out.

   Our aim is to compute $M_1 \times M_2 \times \cdots \times M_n$ using least number of multiplication operations. Design an algorithm based on dynamic programming to solve this problem in polynomial time. Here is some additional explanation which should help you understand the problem better.

   The order in which we compute $M_1 \times M_2 \times \cdots \times M_n$ is defined by parenthesizing the expression of n matrices. For example, if $n = 4$, then there are the following 4 ways to compute the product:

   - $(((M_1 \times M_2) \times M_3) \times M_4)$
   - $((M_1 \times M_2) \times (M_3 \times M_4))$
   - $(M_1 \times ((M_2 \times M_3) \times M_4))$
   - $(M_1 \times (M_2 \times (M_3 \times M_4)))$

   The number of ways to parenthesize the expression is exponential (it is catalan number). Of course, we can not afford to try out each possible way to multiply the n matrices.

Hint 1: A better understanding of the algorithm for optimal triangulation of convex polygon might help.
Hint 2: Consider that parenthesized expression that corresponds to the least number of multiplication operations. If you multiply the matrices according to this parenthesization, the number of matrices will reduce by 1 each time. Finally we will be left with only 2 matrices. How do these last two matrices which are multiplied look like ? What can you infer from it ? Use it to come up with a recursive formulation.

7. **More on Bellman Ford algorithm**

   Consider the following algorithm which is a variant of the algorithm we discussed in the lecture.

```
1  L(s) ← 0;
2  for v ∈ V\{s} do
3  |  L(v) ← ∞
4  end
5  for i = 1 to n − 1 do
6  |  foreach (u, v) ∈ E do
7  |  |  if L(u) + wt(u, v) < L(v) then
8  |  |  |  L(v) ← L(u) + wt(u, v)
9  |  |  end
10 |  end
11 end
```

   Prove that $L(v)$ store distance from s to v in the graph at the end of the above algorithm. This is the actual Bellman Ford algorithm.

8. **Finding a negative weight cycle**

   Given a directed graph $G = (V, E)$ on n vertices and m edges, our aim is to detect if there is any negative weight cycle in G. Design an $O(mn)$ time algorithm to compute one such cycle, if exists.

   Hint: Revisit the beginning of Lecture 17 where we argued that if there is a negative weight cycle reachable by a path of length i from the source, then for each $j > i$, the following assertion holds:

   The label of at least one vertex of the cycle will change during jth iteration.
   Make use of the proof of this assertion to design the algorithm to detect the negative cycle.

9. **Shortest walks consisting of at most k edges**

   Suppose $G = (V, E)$ be a directed graph with potentially negative weights on the edges, and even negative weight cycle. Recall that, unlike in a path, a vertex as well as an edge may appear multiple times on a walk. The length of a walk is the sum of the length of each edge lying on the walk. Design an efficient algorithm to compute shortest walks from s to all the vertices with at most k edges. Note that k is a parameter. The time complexity of the algorithm may depend on k.

Hint: Revisit Bellman Ford algorithm. Perhaps, this algorithm is doing exactly what is required for this problem ...

10. **Is Bellman Ford algorithm really needed ?**

    Consider the problem of shortest paths in directed graph with real weights but no negative cycle. An attentive student argues that there is no need of Bellman-Ford algorithm. We could just transform the given graph with some negative edges into a graph where all edge weights are positive and then apply Dijkstra's algorithm. The transformation is the following :

    Let w be the weight of the least weight negative edge in the given graph. Add $|w|$ to the weight of each edge. Let G' be the resulting graph.

    The student claims that shortest path from s to v in G is identical to the shortest path from s to v in G' for each vertex v. Do you agree with the student ? Either provide a proof or provide a counterexample.

    Hint: This problem was discussed in the lecture as well. You should not agree with the student. So give a counterexample.

11. **Bellman Ford algorithm**

    Let $G = (V, E)$ be a directed graph on n vertices and m edges where each edge has a weight which is a real number. Show that there exists an order among the vertices such that if we process the vertices according to that order in the inner For loop of the Bellman-ford algorithm, then just after one iteration, $D[v]$ will store the distance from s to v for each $v \in V$.