1. In Lecture 13, we have seen that the Knapsack Problem (Example 1) has optimal substructure. Here, let's rewrite the Knapsack Problem in the following way: Given a knapsack of integral volume B and a set of n items $\{c_1, c_2, \dots, c_n\}$ where each item c_i has an integral size s_i and a real value v_i. Without loss of generality, let us assume that these n items are sorted by their size in non-decreasing order (in other words, $s_1 \leq s_2 \leq \cdots \leq s_n$). We are trying the select a subset S among these n items such that their total size is no larger than B (in other words, $B \geq \sum_{c_i \in S} s_i$), and our goal is to maximize the total value of the selected items (in other words, maximize $\sum_{c_i \in S} v_i$).
   a) Define a recursive relation $K(i, j)$ which represents the maximized total value of items that are selected only among items $\{c_1, c_2, \dots, c_i\}$ with total size no larger than j. Be careful of the range of i and j. (Hint: this recursive relation is similar to the one we defined for the coin changing problem.)
   b) Create a dynamic programming algorithm that uses the above recursive relation you defined to solve this problem.

2. In Lecture 13, we solved the Meeting Scheduling Problem (Example 3) by sorting the meetings by their finishing time then selecting the first meeting that is compatible with all the previously selected meetings. Create a different greedy algorithm that solves this problem optimally starting with sorting all the meetings by their starting time, then prove your algorithm can give an optimal solution.

3. Given $2n$ pairs of integers, one red and one blue in each pair: $\{a_1, b_1\}, \{a_2, b_2\}, \dots, \{a_{2n}, b_{2n}\}$, present an efficient algorithm that chooses one integer out of each pair such that exactly n red integers and exactly n blue integers are chosen, and the sum of the chosen integers is maximized.

   Here is an example with $n = 2$. Given $\{2,10\}, \{-5, -9\}, \{-1,3\}, \{5,6\}$, your algorithm should return $10, -5, 3, 5$, since exactly two red and exactly two blue integers are chosen, and their sum 13 is the maximum.

4. Create a different Topological Sort algorithm, in which we keep removing from the graph vertices with 0 in-degrees while maintaining the in-degrees of the remaining vertices. Analyze the time complexity of your algorithm.

5. An independent set of an undirected graph $G = (V, E)$ is a subset $I \subseteq V$, such that no two vertices in I are adjacent. That is, if $u, v \in I$, then $(u, v) \notin E$. A ***maximal independent set*** M is an independent set such that, if we were to add any additional vertex to M, then it would not be independent any longer. Every graph has a maximal independent set. Present an $O(|V| + |E|)$-time algorithm that computes a maximal independent set for an undirected graph G.

6. Prove or disprove the following statement:
   Let G be a simple weighted connected undirected graph. If every edge of G has a unique weight, then the minimum spanning tree of G is unique.

7. We have learned that **Dijkstra**'s algorithm and **Prim**'s algorithm are similar to each other: they both take a weighted graph G and a starting vertex s as input, and they both use a min-heap to output a spanning tree of G. The difference is that **Prim**'s algorithm outputs a minimum spanning tree of G, and **Dijkstra**'s algorithm outputs a spanning tree consisting of all the shortest paths starting from s in G.

   Since all these paths are the shortest paths, we have the following conjecture:

Given a connected simple undirected graph $G = (V, E)$ with positive edge weights: $w: E \to \mathbb{R}^+$ and arbitrary vertex $s \in V$, **Dijkstra**(G, w, s) can always output a minimum spanning tree of G.

Prove or disprove the above conjecture.

8. Let G be a simple directed graph with non-negative arc weights. We define **capacity** of a path p in G as the minimum arc weight along it:

$$cap(p) = \min_{e \in p} w(e).$$

Also, we define **volume** of a pair of vertices (u, v) as the maximum capacity among paths from u to v:

$$vol(u, v) = \max_{p \text{ is a path from } u \text{ to } v} cap(p).$$

Given graph G and a vertex s of G, present an efficient Dijkstra-like algorithm to find, for all $t \in G.V \setminus \{s\}$, the volume of (s, t).