To set about developing an algorithm based on dynamic programming, one needs a collection of subproblems derived from the original problem that satisfies a few basic properties.

1. There are only a polynomial number of subproblems.
2. The solution to the original problem can be easily computed from the solutions to the subproblems. (For example, the original problem may actually be one of the subproblems.)
3. There is a natural ordering on subproblems from “smallest” to “largest,” together with an easy-to-compute recurrence that allows one to determine the solution to a subproblem from the solutions to some number of smaller subproblems.

Last lecture, we discussed the Weighted Interval Scheduling problem. We decided that computing an optimal solution involves solving two sub-problems, as shown in the recurrence equation below.

$$OPT(j) = \max(v_j + OPT(p(j)), OPT(j - 1))$$

Then, we defined three algorithms to solve the problem. An inefficient brute-force algorithm, one using the memoization method, and the other user the bottom-up approach.

Today, we’re discussing the Knapsack problem, where each request $i$ has both a value $v_i$ and a weight $w_i$. The goal in this more general problem is to select a subset of maximum total value, subject to the restriction that its total weight not exceed $W$. The name knapsack refers to the problem of filling a knapsack of capacity $W$ as full as possible (or packing in as much value as possible), using a subset of the items 1, ..., $n$.

1) [20 minutes] Can you define the problem as a recurrence of its subproblems? How?
2) [20 minutes] Write an algorithm that solves the Knapsack problem.
3) [10 minutes] Analyze the algorithm’s running time complexity.