When can you use Dynamic Programming?

DP computes recurrences efficiently by storing partial results. Thus DP can only be efficient when there are not too many partial results to compute. There are $n!$ permutations of an n-element set - we cannot use DP to store the best solution to each sub-permutation. There are 2^n subsets of an n-element set, we cannot use dynamic programming to store the best solution for each subset.

Dynamic Programming works best on objects which are linearly ordered and cannot be rearranged, so the number of partial results is not exponential. Characters in a string, matrices in a chain, the left-to-right order of the leaves in a BST. One commonality to all the dynamic programming solutions we explored is that all the problems had some sort of ordering restriction. Here is an example that does not. Because of the constraint on the total weight limit, it is not an exponential enumerate all the subsets of an n-element set.

$$\sum_{i \text{ items taken}} w_i \leq W \&\& \sum_{i \text{ items taken}} v_i \text{ is max}$$

0-1 Knapsack Problem

Given N items and a total weight limit W, v_i is the value and w_i is the weight of item i, maximize the total value of items taken.
The dynamic programming algorithm computes entries of a matrix $C[0 \dots N, 0 \dots W]$
$C[i,j]$ = the optimal value of the items put into the knapsack from among items $\{1, 2, \dots, i\}$ with total weight $\leq j$ with $C[0,?] = C[?,0] = 0$

1. When you think about calculating $C[i,j]$ there are two options. The ith item is in that optimal answer or is not. Write the recurrence relation. If you remove an item that was in the optimal answer, then you have a new problem where the weight limit is $W - w_i$ and items $1 \dots i-1$. If you remove an item that was not in the optimal answer, then you have a new problem with items $1 \dots i-1$ with the same weight limit W.

Table 17.1:

C	0	1	2	…	W
0	0	0	0	…	0
1	0			…	
2	0			…	
3	0			⋱	
⋮	⋮	⋮	⋮	⋱	⋮
N	0			…	

2. Write pseudocode to fill in the $C[i,j]$ matrix, use your answer from #7.

Algorithm 17.1 Knapsack Problem Pseudocode

```
function KNAPSACK
    for $i \leftarrow 1 \dots N$ do
        for $j \leftarrow 1 \dots W$ do
            if $w_i \leq j$ then
                if $v_i + C[i-1, j-w_i] > C[i-1, j]$ then
                    $C[i, j] \leftarrow v_i + C[i-1, j-w_i]$
                else
                    $C[i, j] \leftarrow C[i-1, j]$
                end if
            else
                $C[i, j] \leftarrow C[i-1, j]$
            end if
        end for
    end for
end function
```

Opening Questions – Greedy Algorithms

3. Briefly explain what two properties a problem must have so that a greedy algorithm approach will work. Optimal subtructure and the Greedy Choice property.

4. A good cashier gives change using a greedy algorithm to minimize the number of coins they give back. Explain this greedy algorithm. To prove that there is optimal substructure, you can prove that there is some number of each denomination (pennies, nickels, dimes, quarters) that, with enough, will total some value k. The order of the coins doesn't matter, but the size of each denomination does. To prove that this is greedy, you need to be able to show that if you cut out a part of the optimal solution and insert the greedy choice, you will get a better value. For example, if you need 37¢ and you don't have a quarter, you could cut out 25¢ from the value and insert a quarter, removing coins but keeping the same value.

5. For what types of optimization problems does optimal substructure fail? The subproblems are not independent.

Activity Selector Problem

Given a set S of n activities each with start S_i, Finish F_i, find the maximum set of Compatible Activities (non-overlapping). (or could be jobs scheduled with fixed start and end times instead of meetings)

6. The brute force approach would be to find all possible subsets of n activities, eliminate the ones with non-compatible meetings, and find the largest subset. How many subsets are there? For every meeting, there is a binary choice determining if the meeting is being used, so 2^n.

7. Prove the Activity Selector Problem has optimal substructure (using similar proof by contradiction approach that we used for dynamic programming: assume you have an optimal answer, remove something to get to the largest sub-problem, show that the sub-problem must also be solved optimally). If you were to choose meetings based off of the time of each meeting, and start with the shortest meeting, this can be countered by the case of having 3 meetings, with the shortest meeting conflicting with the other two longer meetings, whereas those 2 meetings don't overlap. If you choose meetings by the earliest, you could have a meeting that goes all day and excludes all other meetings in the day.