

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.

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 i th item is in that optimal answer or is not. Write the recurrence relation.
2. Write pseudocode to fill in the $C[i, j]$ matrix, use your answer from #1.

Opening Questions – Greedy Algorithms

3. Briefly explain what two properties a problem must have so that a greedy algorithm approach will work.
4. A good cashier gives change using a greedy algorithm to minimize the number of coins they give back. Explain this greedy algorithm.
5. For what types of optimization problems does optimal substructure fail?

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?

7. Probe 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.)