Recursion is awesome!

It’s a way of thinking that helps us solve a big problem by breaking it down into a smaller problem. The structure of the solution often matches the structure of the problem, especially for:

- math problems
- inductive data structures (i.e., built up from smaller, similarly structured pieces)

Avoiding redundant work

Algorithmic techniques that avoid redundant work:

- **Dynamic programming**: Reorder the computation so that all subproblems are solved before their results are needed.
- **Memoization**: Reuse the results of previously computed answers.

A recipe for efficient algorithms:

1) Write a straightforward, recursive algorithm.
2) Analyze the algorithm, looking for costly inefficiencies, especially redundant work.
3) If you find redundant work, re-write the algorithm to avoid that work.

Dynamic programming

A dynamic-programming algorithm is “bottom-up”: the algorithm computes the results of subproblems first. It’s typically iterative (i.e., it uses a loop) and trades space for time, by storing the results of subproblems in a table.

A dynamic-programming template

1) Write a straightforward, recursive algorithm that has redundant work.
2) Design the table:
   1) What kind of information is stored in a cell (i.e., what is a subcomputation)?
   2) How many cells will there be? (i.e., how many different recursive calls / subproblems are needed to solve the full problem?)
   3) Which cells are easy to fill in? (i.e., what are the base cases?)
   4) How does a cell’s value depend on the value of other cells? (i.e., how do the recursive cases work?)
   5) Which cell contains the answer?
3) Write the code:
   1) Create a table of the appropriate size.
   2) Write code to fill in the base-case cells.
   3) Write code (usually a loop) to fill in the remaining values.
   4) Return the value in the result cell.
**Example problems**

**Making change**

Given some coin denominations, we want to find the **fewest number of coins** that will total a given amount. For example, given the denominations 1¢, 5¢, 10¢, and 25¢, find the fewest number of coins that makes 42¢. You can assume you have an infinite number of each kind of coin.

*Alternative version:* compute the list of coins (rather than the number of coins) that is required to make change.

**Shortest path in a directed, weighted, acyclic graph**

You are given a directed, weighted, acyclic graph, where a node corresponds to a city, and an edge corresponds to a road between two cities, and a weight describes the length in minutes that it takes to traverse that road. Find the **shortest (i.e., minimum-weight) path** from a given city to another, in the graph.

*Alternative version:* compute the list of cities (rather than the minimum weight) that are visited on the path from one city to another. The path should include the initial and final city.

**Knapsack**

Given some items of various weights and a knapsack that can carry N pounds, we want to find the **maximum total weight** of items that can go in the knapsack. For example, given the items of weight 5, 10, 18, 23, 30, and 45 find the most weight you can carry in a backpack that can accommodate 42 pounds. You should assume that you have only one of each kind of item.

*Alternative version:* compute the list of items (rather than the maximum weight) that can fit in the knapsack.

**Longest common subsequence (LCS)**

Given two strings s1 and s2, find the **length of the longest string** that is a non-consecutive substring of both s1 and s2. For example, the longest common subsequence of human and chimpanzee has length 4 (hman).

*Alternative version:* compute the LCS itself (rather than the length of the LCS).

**Edit distance**

Given two strings s1 and s2, find the **minimum number of modifications** it takes to turn s1 into s2, where a modification can be one of the following:

- substitute one letter for another in one of the strings
- delete a letter from one of the strings
- insert a letter into one of the strings

For example, the edit distance of cat and hat is 1.

*Alternative version:* compute the list of modifications (rather than the number of modifications).

*Next time: more dynamic programming*