Lecture Notes for CSCI 311: Algorithms

Set 8-Dynamic Programming Formalism

Professor Talmage

February 24, 2025

---

1 Outline

Having seen a couple of examples of how Dynamic Programming can convert a simple but prohibitively expensive recursive solution into a much more efficient solution, we will step back and consider the general pattern of Dynamic Programming and discuss how you can apply it to a new problem.

First, Dynamic Programming applies to *optimization problems*. These are problems with many possible "solutions", among which we need to find the best, according to some given measure.

- There may be many equally-good best solutions
- We often care more about finding the value of the best solution than the actual solution. Dynamic Programming does this, but we can typically modify the code fairly easily to give the solution as well as its value.

1.1 How-To

To solve a problem with Dynamic Programming, we follow this outline:

1. Find the recursive structure of the problem
    - Figure out how an optimal solution is related to an optimal solution to a smaller problem. Break the problem into two pieces, split off the first step, etc., and consider how we can build an overall solution from solutions to the remaining pieces.
    - From the book: "Characterize the structure of an optimal solution."
    - This is known as finding *optimal substructure* and gives us the number of subproblems and number of choices among subproblems.
2. Write the recurrence for the optimal value.
    - Take your understanding of the problem breakdown and write it formally.
    - From the book: "Recursively define the value of an optimal solution."
    - This tells us what the space of subproblems looks like and how we compute each one. Pay attention to how many parameters you have and what range of values each can take on.
    - Use this to figure out the order in which you will solve subproblems.
    - Start thinking about *overlapping subproblems*: Will we get the same recursive calls in multiple places?
3. Write your code.

- Start by building a recursive function around your recurrence from the previous step. Check for repeated work, if so memoize or convert to bottom up for efficiency.
- From the book: "Compute the value of an optimal solution"
- Top-Down (Memoization): Compute in dependency order, saving results. Correctness follows from simple recursive algorithm.
- Bottom-Up: Reverse dependency order. Complexity is typically size of subproblem table times cost per entry.

4. (Optional) Save the choices that led to the optimal solution.
   - From the book: "Reconstruct an optimal solution based on step 3."
   - Will typically need to add subsolution-saving to code which previously only saved subsolution values.

Notes:

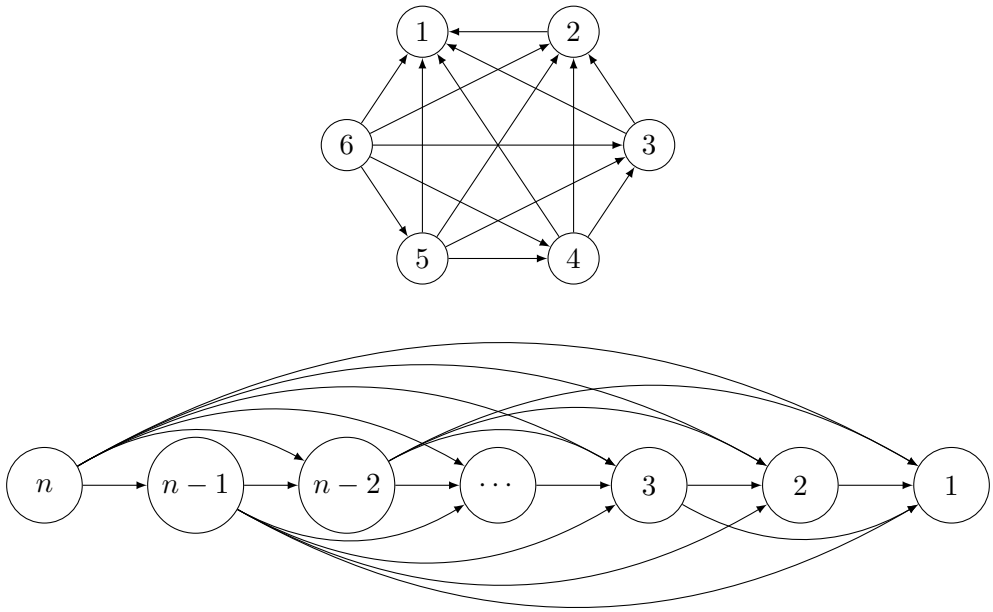
- Dynamic Programming can be applied to any recursive optimization algorithm (optimal substructure), but is only useful when there are *Overlapping Subproblems*.
- Storing sub-solutions trades space for time. If you have k parameters or dimensions of your subproblem space, you can expect to need something like $\Theta(n^k)$ space.
- Choose how to balance time and space costs on a per-problem basis.

2 Subproblem Graphs

We most often represent the space of possible subproblems with a table. This is good for memory, since we get $O(1)$ indexing to access subproblem solutions. However, tables do not store general relationships between different subproblems, which can make it harder to understand the order in which we need to compute subproblems.

Another approach is to represent the space of subproblems with a graph. We will treat graphs more thoroughly later, but for now recall that a graph is a collection of objects, called nodes or vertices, and a set of relationships between nodes, which we call edges. We will let each subproblem be a node, and have a directed edge from each subproblem node to all others which it calls recursively. This is much like a collapsed version of a recursion tree, where each subproblem appears only once, but may have many edges pointing to it, if it appears many times in the tree. We can then express the runtime of a dynamic programming solution as the number of nodes times the non-recursive cost per node. This is analogous to what we did before when we counted the number of non-trivial recursive calls–those which found an empty cell in the subproblem table and had to actually do the work to compute that value.

Example: The subproblem graph for the rod-cutting (resource-allocation) problem has each node dependent on all smaller-numbered nodes. We here give the subproblem graph for $n = 6$, and sketch the graph arranged in a different shape for general n.



How do these graph relate to Dynamic Programming?

- Top-Down solutions are Depth-First traversals from node n.
- Bottom-Up solutions start at base cases (nodes with no out-edges), and work up, only solving nodes with edges to previously-solved nodes.
- The time to solve a particular subproblem depends on its out-degree, which is the number of recursive calls it would make in the simple recursive algorithm.

Exercise: Draw the subproblem graph for $Fibonacci(6)$, then sketch the structure for general n.

- How many edges from each node? To each node?
- How do these counts relate to the runtimes of the simple recursive algorithm and the dynamic programming algorithms?

Exercise: In the below subproblem graph, give one possible top-down and one possible bottom-up order of the nodes: