6.3 Elements of Dynamic Programming

Dynamic programming refers to the process of recursively breaking a problem into subproblems, then combining those solutions to create a solution for the main problem. We've seen some examples of using dynamic programming, but when exactly should we choose this technique? We will discuss two key properties about a problem that can suggest that dynamic programming is a viable way to solve the problem.

6.3.1 Optimal Substructure

The first property is called **optimal substructure**. A problem exhibits optimal substructure when the optimal solution contains within it optimal solutions to subproblems. This was true for both the weighted scheduling problem and the rod cutting problem.

The following is a loose guide on how you might go about discovering that a problem has this property.

1. **Making a choice.** At some point in the algorithm there is a choice to make. Making the choice leaves one or more subproblems to be solved.
2. **Decomposable problem.** If the correct choice is made, the optimal solution can be described by combining the solutions from the induced subproblems.
3. **The subproblem solutions are optimal.** We must also be careful to check that using the solutions to subproblems directly are optimal.

Question 71. State the choices made during the weighted scheduling problem, what the subproblems are, and why the solutions to them are optimal.

Question 72. State the choices made during the rod cutting problem, what the subproblems are, and why the solutions to them are optimal.

Not every problem has this property!! Be careful that you are not misguided into using dynamic programming when it isn't applicable. Let's look at two similar looking problems to see this in action.

Problem (SHORTESTPATH)**.**

INPUT: An unweighted graph $G = (V, E)$ and the labels u, v of two vertices in G.

OUTPUT: A path from u to v consisting of the fewest edges.

Problem (LONGESTSIMPLEPATH)**.**

INPUT: An unweighted graph $G = (V, E)$ and the labels u, v of two vertices in G.

OUTPUT: A path from u to v consisting of as many edges in G as possible. The path must be simple, meaning each edge can only be used once.

Question 73. Draw example inputs and outputs for SHORTESTPATH and LONGESTSIMPLEPATH, where the graph G has 6 vertices. Use the same graph for both examples.

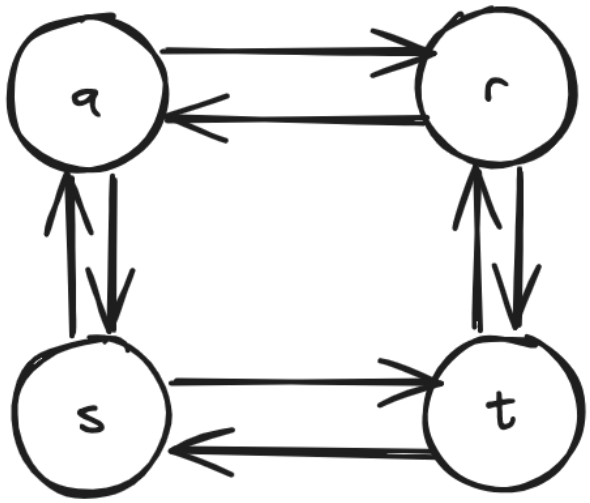
Let's first look at the SHORTESTPATH problem. Again, to identify optimal substructure, we want to make sure that the optimal solution is just a combination of optimal solutions to smaller problems.

Suppose that we have the optimal path p from u to v in our graph. To check for optimal substructure, we decompose our solution into a few pieces. In this case let's split it into two pieces by choosing a vertex w that is on this path. Let p_1 and p_2 be the paths from u to w along p and w to v along p respectively.

Question 74. Describe the two subproblems that p_1 and p_2 are solutions to. Is it possible that there is a better solution to these problems?

Now let's look at the LONGESTSIMPLEPATH problem. We want to check if this problem has optimal substructure. To reiterate, this means that the solution to full problem can be written as a composition of the solutions to subproblems.

Question 75. Consider an instance of the LONGESTSIMPLEPATH problem with the following graph as input with labels q and t. What is the solution?



Question 76. Show by counterexample that LONGESTSIMPLEPATH does not have the optimal substructure property. In other words, the optimal solution cannot be described by solutions to subproblems.

6.3.2 Overlapping Subproblems

The second property ingredient we need in an optimization problem to apply dynamic programming is having **overlapping subproblems**. When we think of breaking our problem in to subproblems, sometimes the same subproblem is solved many different times. If this is only repeated say one or two times, it doesn't cause an issue for us. However, the number of calls can easily blow up. Recall our tree that we drew in Question 69 that showed the number of overlapping calls we made to solve the rod cutting problem.

Question 77. Does the weighted scheduling problem have the overlapping subproblem property? If so, write down a scenario where the same subproblem gets called twice.

There are two main ways that people deal with overlapping subproblems. We saw these in the rod cutting example, the top-down approach (memoization) and the bottom-up approach (tabulation). In both approaches, we are keeping track of solutions to subproblems that we have solved already so that we don't waste computation time to resolve them.

Top-Down (Memoization): To implement an algorithm using memoization, first write a recursive algorithm to solve your problem. The algorithm should be clear about what the subproblems are and how we combine the solutions to the subproblems.

Question 78. Write down a recursive algorithm to solve the weighted scheduling problem.

Now to create a memoized version of the algorithm, we will use an array initialized as a global variable to store the solutions to our subproblems. In this step, it is important to know how many subproblems we will have to deal with.

Question 79. For the weighted scheduling problem, how many possible subproblems are there? Remember, this problem asks "for all events 1 through i, what is the maximum value we can achieve?" for an input of i.

Once the memo is defined, we split our function into two parts.

1. IF this subproblem has been solved: return the solution stored in the memo.
2. ELSE: solve the problem just like we did in the recursive version.

Question 80. Write down an algorithm to solve the weighted scheduling problem using memoization.

Bottom-Up (Tabulation): In the bottom up approach, instead of using a recursive algorithm, we will instead iterate through our table from the bottom and build up the solutions using the cells we have filled out so far. This was the way we solved the weighted scheduling problem in the first place.

In practice, the bottom-up method outperforms the top-down approach especially when you have a problem where every subproblem has to be solved at least once. First of all, function calls can be expensive depending on your system, so you can accumulate overhead from your recursive calls. The bottom-up approach also doesn't need a large external array to store the intermediate solutions. However, if your problem doesn't require looking at every subproblem for most inputs, then the top-down approach can be faster in practice.

6.3.3 Pieces to identify

The following components are also extremely helpful when defining a dynamic programming algorithm.

1. **Subproblem domain:** The space of possible subproblems to consider.
2. **Memo table definition:** The items we will be storing in our tables. Alternatively, the name of solutions to subproblems.
3. **Goal:** The location of our solution in the table.
4. **Initial values:** Solutions to trivial subproblems to start building on.
5. **Recurrence:** A compact representation of the recursive function.

Question 81. Identify the above pieces in the weighted scheduling problem.

6.4 Truck Loading Problem

Problem (TRUCKLOADING)**.**

INPUT: A truck that can hold boxes up to weight W, and a list $\{w_1, w_2, \ldots, w_n\}$ of weights of n boxes to load.

OUTPUT: Which boxes to load to achieve the maximum possible weight without exceeding W.

Question 82. Write down an example instance of the truck loading problem. That is, write down a viable input output pair.

In the weighted scheduling problem and rod cutting, there was one way to generate subproblems from the main problem. There are two ways to do so here, because there are two main parameters to the problem. One way to create subproblems is like the weighted scheduling problem, where we consider the first i items instead of all n items. The other way is to reduce the capacity of the truck to something smaller and see what happens.

It's important to consider both, because we can think of choosing to add the last item to a truck to induce a new subproblem which uses the remaining items, and a smaller truck instead.

Question 83. Reason about whether or not this problem has the optimal substructure property.

- What are a set of easy top level choices to make, which break the problem into multiple subproblems? Remember, a subproblem must also be a valid instance of the main problem!

- If someone tells you the solution to this first choice, does the correct answer to the subproblems build into the correct solution of the full problem?

Question 84. Using the above analysis, can you define a recursive algorithm that solves the problem? This algorithm should **not** use a memo and will likely be inefficient.

Question 85. Given your answer to the previous problem, does the truck loading problem also have overlapping subproblems? If yes, write down a scenario where the same subproblem gets called twice.

Question 86. Identify the pieces of the dynamic programming algorithm.

Subproblem domain: The space of possible subproblems to consider.

Memo table definition: The items we will be storing in our tables. Alternatively, the name of solutions to subproblems.

Goal: The location of our solution in the table.

Initial values: Solutions to trivial subproblems to start building on.

Recurrence: A compact representation of the recursive function.

The following is an example of a table you might have at the end of a dynamic programming algorithm to solve TRUCKLOADING, with an instance where $n = 3$ with $w_1 = 6$, $w_2 = 2$, $w_3 = 5$ and $W = 7$.

Remember, each cell `memo[i, j]` stores what the optimal solution is to an instance where we use the first i items, with a truck that can have up to weight j.

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0							
2	0							
3	0							

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0							
2	0							
3	0							

Question 87. What is the runtime of the dynamic programming algorithm?

6.5 Matrix Chain Multiplication

Question 88. Suppose we have two matrices A with size p by q and B with q by r. How many multiplications do we need to compute their product AB?

Question 89. Suppose we have three matrices such that the dimensions are

$$A : p \times q$$
$$B : q \times r$$
$$C : r \times s.$$

What is the total number of multiplications we need if we compute (AB) first? What if we compute (BC) first?

Problem (MATRIXCHAIN)**.**

INPUT: A sequence of matrices $M_1, M_2, \ldots, M_n$ and a list of dimensions $d_0, d_1, d_2, \ldots, d_n$ such that M_i is size d_{i-1} by d_i.

OUTPUT: What is the most efficient way to group the matrices to minimize the total number of multiplications?

Question 90. Reason about whether or not this problem has the optimal substructure property.

- What are a set of easy top level choices to make, which break the problem into multiple subproblems? Remember, a subproblem must also be a valid instance of the main problem!

- If someone tells you the solution to this first choice, does the correct answer to the subproblems build into the correct solution of the full problem?

Question 91. Using the above analysis, can you define a recursive algorithm that solves the problem? This algorithm should **not** use a memo and will likely be inefficient.

Question 92. Given your answer to the previous problem, does the MATRIXCHAIN problem also have overlapping subproblems? If yes, write down a scenario where the same subproblem gets called twice.

Question 93. Identify the pieces of the dynamic programming algorithm.

Subproblem domain: The space of possible subproblems to consider.

Memo table definition: The items we will be storing in our tables. Alternatively, the name of solutions to subproblems.

Goal: The location of our solution in the table.

Initial values: Solutions to trivial subproblems to start building on.

Recurrence: A compact representation of the recursive function.

The following is an example of a table you might have at the end of a dynamic programming algorithm to solve MATRIXCHAIN, for the following instance.

Each cell `memo[i, j]` stores what the optimal solution is to an instance where we consider the sequence starting from matrix i and ending at matrix j.

- $A_1 : 10 \times 15$
- $A_2 : 15 \times 5$
- $A_3 : 5 \times 60$
- $A_4 : 50 \times 100$
- $A_5 : 100 \times 20$
- $A_6 : 20 \times 40$
- $A_7 : 40 \times 47$

1	2	3	4	5	6	7	
0 -	750 1	3750 2		41750 2	46750 2		1
	0 -	4500 2	37500 2	41500 2	47000 2	56925 2	2
		0 -	30000 3	40000 4	44000 5	53400 6	3
			0 -	120000 4		214000 5	4
				0 -	80000 5	131600 5	5
					0 -	37600 6	6
						0 -	7

Question 94. Write down a bottom-up algorithm to solve the MATRIXCHAIN problem.

Question 95. What is the runtime of the dynamic programming algorithm?