# Chapter 47

# Introduction

Dynamic programming is an inductive algorithm-design technique. Similar to divide-and-conquer, it solves larger instances of a problem in terms of smaller ones. The main difference is that many of the smaller instances are shared among recursive calls, making it worthwhile to save the partial solutions so they can be reused. Dynamic programming is usually inherently parallel, but taking advantage of the parallelism is a bit more tricky because the sharing of solutions need to be properly coordinated.

**Origins of "Dynamic Programming".** Unlike many of the other algorithmic techniques, the name "dynamic programming" sheds little light on how the technique works. We could blame the bad name on the person who invented the idea, Richard Bellman. However, as the following quote from Bellman indicates, it has more to do with the US government during the McCarthy years, 1950-1957, an era when many people where jailed for expressing their constitutional rights of free speech.

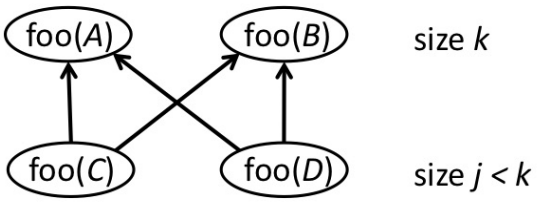
> "An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as

> an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. This, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities".
>
> Richard Bellman ("Eye of the Hurricane: An autobiography", World Scientific, 1984)

The Bellman-Ford shortest path algorithm covered in Chapter 57 is named after Richard Bellman and Lester Ford. That algorithm is a dynamic-programming algorithm, when looked at the right way.

**It is all about sharing.** When using divide-and-conquer, we reduce the problem instance that we want to solve into multiple smaller instances and assume that the smaller instances are solved recursively and independently. When calculating total work, we therefore add up the work from each recursive call. In some cases, the smaller instances are simply not independent, because solving two instances may both involve solving a smaller shared instance. For example, two instances of size $k$ may both need the solution to the same instance of size $j < k$. The idea behind dynamic programming is to take advantage of such "sharing" and instead of solving the smaller instance twice, to solve it once and share the result, effectively re-using the result as needed. In general such sharing can make significant, as much as exponential, improvement in work.

**Example 47.1.** Two smaller instance of the problem (function call) *foo* being shared by two larger instances.

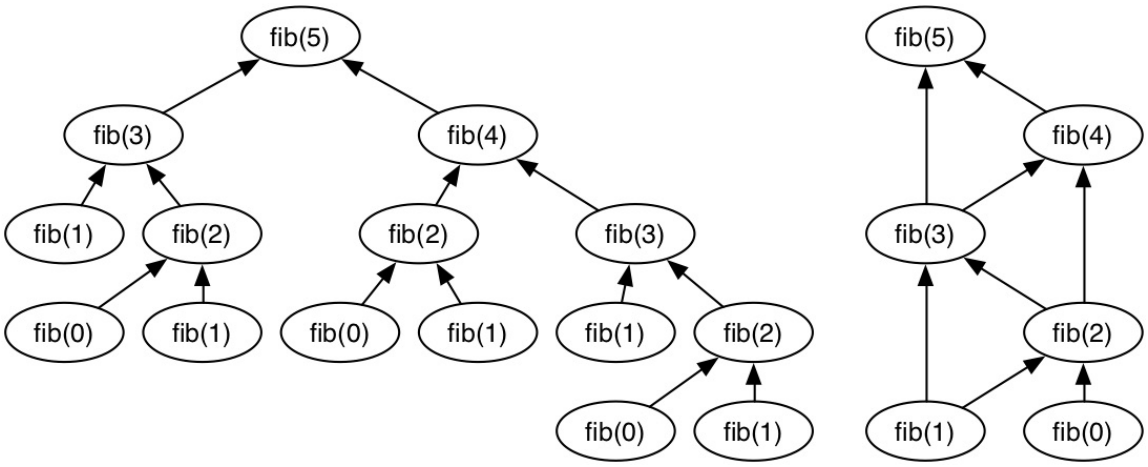



Although sharing the results in this example makes at most a factor of two difference in work, in general sharing the results can make an exponential difference in the work performed.

**Example 47.2.** Consider the following algorithm for calculating the Fibonacci numbers.

$fib(n) =$
`if` $(n \leq 1)$ `then` $1$
`else` $fib(n-1) + fib(n-2)$

This recursive algorithm takes exponential work in $n$ as indicated by the recursion tree below on the left for $fib(5)$. If the results from the instances are shared, however, then the algorithm only requires linear work, as illustrated below on the right.




Here many of the calls to *fib* are reused by two other calls. Note that the root of the tree or DAG is the problem we are trying to solve, and the leaves of the tree or DAG are the base cases.

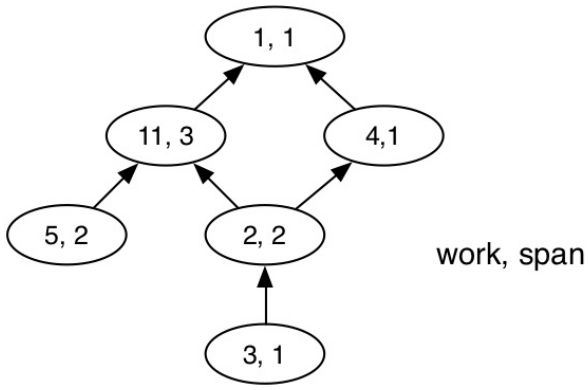
**Representing Sharing with DAGs.** With divide-and-conquer the composition of a problem instance in terms of smaller instances is typically described as a tree, and in particular the so called recursion tree. With dynamic programming, to account for sharing, the composition can instead be viewed as a Directed Acyclic Graph (DAG). Each vertex in the DAG corresponds to a problem instance and each edge goes from an instance of size $j$ to one of size $k > j$—i.e. each directed edge (arc) is directed from a smaller instances to a larger instance that uses it. The edges therefore represent dependencies between the source and destination (i.e. the source has to be calculated before the destination can be). The leaves of this DAG (i.e. vertices with no in-edges) are the base cases of our induction (instances that can be solved directly), and the root of the DAG (the vertex with no out-edges) is the instance we are trying to solve. More generally we might actually have multiple roots if we want to solve multiple instances.

**Work and Span.** Abstractly dynamic programming can therefore be best viewed as evaluating a DAG by propagating values from the leaves (in degree zero) to the root (out degree zero) and performing some calculation at each vertex based on the values of its in-neighbors. Based on this view, calculating the work and span of a dynamic program is relatively straightforward. We can associate with each vertex a work and span required for that vertex. We then have

- The *work* of a dynamic program viewed as a DAG is the sum of the work of the vertices of that DAG, and
- the *span* of a dynamic program viewed as a DAG is the heaviest vertex-weighted path in the DAG—i.e., the weight of each path is the sum of the spans of the vertices along it.

Whether a dynamic programming algorithm has much parallelism (work over span) will depend on the particular DAG. As usual the parallelism is defined as the work divided by the span. If this is large, and grows asymptotically, then the algorithm has significant parallelism. Most dynamic programs have significant parallelism but some do not.

**Example 47.3.** Consider the following DAG:




where we have written the work and span on each vertex. This DAG does $5 + 11 + 3 + 2 + 4 + 1 = 26$ units of work and has a span of $1 + 2 + 3 + 1 = 7$.

The challenging part of developing a dynamic programming algorithm for a problem is in determining what DAG to use. The best way to do this is to think inductively: how can we solve an instance of a problem by composing the solutions to smaller instances? After we formulate an inductive solution, we then think about whether the solutions can be shared and how much savings can be achieved by sharing. As with all algorithmic techniques, being able to come up with solutions takes practice.

**Definition 47.1** (Optimization and Decision Problem)**.** Most problems that can be tackled with dynamic programming are optimization or decision problems. An ***optimization problem*** requires finding a solution that optimizes some criteria (e.g., finding a shortest path, or finding the longest contiguous subsequence sum).

Sometimes we want to enumerate (list) all optimal solutions, or count the number of such solutions. A ***decision problem*** is one in which we are trying to find if a solution to a problem exists. Again we might want to count or enumerate the valid solutions.

When solving an optimization or an enumeration problem, we usually solve many smaller instances of the same problem and therefore may improve total work by sharing common solutions via dynamic programming.

**Coding Dynamic Programs.** Although dynamic programming can be viewed abstractly as a DAG, in practice we need to implement (code) the dynamic program. There are two common ways to do this, which are referred to as the top-down and bottom-up approaches. The *top-down* approach starts at the root(s) of the DAG and uses recursion, as in divide-and-conquer, but remembers solutions to subproblems so that when the algorithm needs

to solve the same instance many times, only the first call does the work and the remaining calls just look up the solution. Storing solutions for reuse is called *memoization*. The *bottom-up* approach starts at the leaves of the DAG and typically processes the DAG in some form of level order traversal—for example, by processing all problems of size 1 and then 2 and then 3, and so on.

Each approach has its advantages and disadvantages. Using the top-down approach (recursion with memoization) can be quite elegant and can be more efficient in certain situations by evaluating only those instances actually needed. The bottom up approach (level order traversal of the DAG) can be easier to parallelize and can be more space efficient, but always requires evaluating all instances. There is also a third technique for solving dynamic programs that works for certain problems, which is to find the shortest path in the DAG where the weighs on edges are defined in some problem specific way.

**Summary.** In summary the approach to coming up with a dynamic programming solution to a problem is as follows.

1. Is it a decision or optimization problem?
2. Define a solution recursively (inductively) by composing the solution to smaller problems.
3. Identify any sharing in the recursive calls, i.e. calls that use the same arguments.
4. Model the sharing as a DAG, and calculate the work and span of the computation based on the DAG.
5. Decide on an implementation strategy: either bottom up top down, or possibly shortest paths.

*It is important to remember to first formulate the problem abstractly in terms of the inductive structure, then think about it in terms of how substructure is shared in a DAG, and only then worry about coding strategies.*

*Remark* (Problems with Efficient Dynamic Programming Solutions). There are many problems with efficient dynamic programming solutions. Here we list just some of them.

1. Fibonacci numbers
2. Using only addition compute ($n$ choose $k$) in $O(nk)$ work
3. Edit distance between two strings
4. Edit distance between multiple strings
5. Longest common subsequence
6. Maximum weight common subsequence
7. Can two strings S1 and S2 be interleaved into S3
8. Longest palindrome

9. longest increasing subsequence
10. Sequence alignment for genome or protein sequences
11. Subset sum
12. Knapsack problem (with and without repetitions)
13. Weighted interval scheduling
14. Line breaking in paragraphs
15. Break text into words when all the spaces have been removed
16. Chain matrix product
17. Maximum value for parenthesizing x1/x2/x3.../xn for positive rational numbers
18. Cutting a string at given locations to minimize cost (costs $n$ to make cut)
19. All shortest paths
20. Find maximum independent set in trees
21. Smallest vertex cover on a tree
22. Optimal BST
23. Probability of generating exactly $k$ heads with $n$ biased coin tosses
24. Triangulate a convex polygon while minimizing the length of the added edges
25. Cutting squares of given sizes out of a grid
26. Change making
27. Box stacking
28. Segmented least squares problem
29. Counting Boolean parenthesization – true, false, or, and, xor, count how many parenthesization return true
30. Balanced partition – given a set of integers up to k, determine most balanced two way partition
31. Largest common subtree