An exponential-time algorithm

Recall our friend the Fibonacci function:

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \qquad \text{fib}(0) = \text{fib}(1) = 1$$

We gave a simple recursive algorithm for calculating $\text{fib}(n)$:

$$\text{def fib}(n)\ [(n \le 1) \to 1 \mid T \to \text{fib}(n-1) + \text{fib}(n-2)]$$

But unfortunately this algorithm is very inefficient; let's analyze it. We can write a recurrence for $C(n)$, the number of function calls to calculate $\text{fib}(n)$:

$$C(0) = C(1) = 1 \qquad C(n) = 1 + C(n-1) + C(n-2)$$

Since $C(n)$ is strictly increasing, we have $1 + C(n-2) \le C(n-1)$. Substituting into the recurrence above, $C(n) \le 2C(n-1) \le 4C(n-2) \le 2^k C(n-k)$ so long as $n - k \ge 2$. The latter implies $C(n) \le 2^{n-2} C(2) = 2^{n-2} \cdot 3 \le 2^n$. That is, $C(n) \in O(2^n)$.

> *In general, we say an algorithm takes exponential time if its runtime is $2^{O(n^d)}$ for some $d \ge 1$.*

Dynamic programming

The reason this algorithm is so inefficient is that it recalculates $\text{fib}(k)$ multiple times for each $k \in 2..(n-1)$. This seems wasteful; can we avoid it?

Yes: we saw earlier that we can sweep k upward from 2 to n, calculating $\text{fib}(k)$ just once for each k. A simple way to organize the computation is with an array A:

$$\begin{array}{l} A(0) \leftarrow 1 \\ A(1) \leftarrow 1 \\ \text{with } (k \leftarrow 2) \\ \text{while } (k \le n) \\ \text{do } A(k) \leftarrow A(k-1) + A(k-2),\ k \leftarrow k+1 \\ \text{return } A(n) \end{array}$$

> *Note that we haven't defined array types yet: they require two tools we haven't provided, namely assignments (for modifying array elements in place) and dependent*

types (types that take parameters, like the array size). That means we won't be able to prove all possible theorems about programs that use arrays; but we still have enough tools to prove many interesting properties.

This program works because all recursive calls from $\mathrm{fib}(k)$ have to use arguments that are strictly smaller than k — which we will have already computed in previous iterations and stored in A. This strategy is called *dynamic programming*.

In fact, dynamic programming works for *any* pattern of recursive calls. That is, when calculating for argument k, we could call recursively with $\lfloor \frac{k}{2} \rfloor$ or $\lfloor \sqrt{k} \rfloor$ or any other function whose output is strictly smaller than k.

We can think of dynamic programming as trading memory for time: whenever we need $\mathrm{fib}(k)$, we could either recalculate it (which takes time) or retrieve it from a previous calculation (which takes memory). In this case there's a clear advantage for storing and retrieving the calculated values: we get a huge reduction in time for a modest increase in memory.

The tradeoff is actually even better than we said above: because of the definition of Fibonacci numbers, we actually only need to store $\mathrm{fib}(k)$ and $\mathrm{fib}(k-1)$ at any given time. (From these values we can calculate $\mathrm{fib}(k+1)$, discard $\mathrm{fib}(k-1)$, and increment k.) In general, though, dynamic programming might need to store all previous values of k. At least on an initial design, it's easiest to do it this way, so that we don't have to worry about the pattern of recursive calls; once we've figured out the initial design, we can worry about optimizations.

In general, though, the tradeoff could go either way. For example, if we had a function $R(n)$ that recursively called $R(\lceil \frac{n}{2} \rceil)$ and $R(\lfloor \frac{n}{2} \rfloor)$, we'd only have $O(n)$ total recursive calls in the naive version; sweeping k from 1 to n would take just as much time, and would require extra memory for the table of function values.

It can be subtle to compare memory use between the simple recursive approach and the dynamic programming approach, since simple recursion tends to use more memory on the call stack (to store its deeper sequence of recursive calls), while dynamic programming tends to use more memory on the heap (to store its table of function values). Typically the call stack uses only a small amount of memory, but we still have to take this memory into account to make a fair comparison.

Multiple arguments

If we have a recursive function with more than one argument, we can apply

dynamic programming by sweeping some or all of the arguments. And, we can store each result only temporarily or keep it in a table for later. These are design choices: depending on our particular pattern of reuse, it might or might not be worth applying dynamic programming to each argument, and it might or might not be worth storing a particular result for later.

If decide to make a table for two arguments, we need a 2d array to store the function values. For example, Pascal's triangle looks like this:

$$\begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \end{array}$$

Each number is the sum of the two numbers above it. We can write a recurrence for Pascal's triangle as follows: if $P(i,j)$ is the jth number in the ith row, then

$$P(i,j) = P(i-1,j) + P(i-1,j-1)$$

$$P(i,1) = P(i,i) = 1$$

The recursive algorithm for calculating $P(i,j)$ takes exponential time. But we can calculate all the entries in the nth row in time $O(n^2)$ with dynamic programming: we sweep the row index i from 1 to n, and within each row we sweep j from 1 to i.

Similarly, with three arguments to index, we would need a 3d array to store our table of function values, and so forth. Dynamic programming can take a lot of time if we need to sweep lots of arguments, and can take a lot of memory if we need to store a high-dimensional table — but it can still be worth it if it avoids a naive recursion that takes exponential time.

For example, we can use dynamic programming to parse a string according to a context-free grammar; one common method for this problem (the CKY chart parser) takes $O(n^3 g)$ time and uses a table of size $O(n^2)$ for a string of length n and a grammar of size g. The dynamic program sweeps four variables: three indices into the string and one index into the grammar. But the inner two loops produce results that only need to be stored temporarily, so the table indices correspond only to the outer two loops.

Memoizing

To implement dynamic programming, we had to change the structure of our recursive function — we had to calculate in order of increasing k instead of decreasing k. Sometimes it's convenient to being able to switch to dynamic programming while still keeping almost the same code. To do this, we'll use a strategy called *memoizing*. (For contrast, we'll call the previous dynamic programming strategy *sweeping*.)

We use an array A just as before to store our calculated function values. For concreteness we'll write two indices, $A[i,j]$, but everything below works for one index or for three or more indices.

We initialize $A[i,j]$ to a *sentinel* such as F for all i,j. The important property of a sentinel is that it should be a value that our function can't possibly return.

Whenever we calculate our function for a new pair of arguments i,j, we store it in $A[i,j]$. This is called a *memo*, since it is a reminder for us of the value of $A[i,j]$. Finally, as the first line in our function, we check for memos: if $A[i,j]=F$ there is no memo, and we continue as before. On the other hand, if $A[i,j]\neq F$, we have a stored memo; so, we return $A[i,j]$ instead of making any recursive calls.

Because of the check for memos, we will compute $A[i,j]$ at most once for each i,j — just like in our sweeping strategy. But unlike sweeping, we compute $A[i,j]$ lazily and as needed; we'll touch pairs of indices i,j in an order that is determined by the computation, instead of precomputed.

Both memoizing and sweeping count as dynamic programming. In most cases there's not a big benefit to doing it one way or the other; the decision is mostly based on ease of implementation.

Two places where there can be a difference that matters:

- For some functions (like Fibonacci) we might be able to discard some entries of A once they are no longer relevant. In these cases we would need less memory for sweeping, since we can ensure the values are computed in a favorable order.
- On the other hand, in some cases we might not need to touch some elements of A at all; this can happen if recursive calls skip down to smaller

indices like $\frac{i}{2}, \frac{j}{2}$ instead of proceeding from i to $i-1$ to $i-2$ and so forth. In these cases memoizing can use less memory.

To take advantage of skipping some indices, we'd need to replace the array of memos A by a dictionary: a data structure that can store and look up $A[i,j]$ for a sparse set of index pairs i, j.

But even in these cases, the difference between sweeping and memoizing can be small enough that other considerations outweigh it.

DP example: longest common subsequence

A *subsequence* of a string S is a sequence of characters from S that are in the same order as S but not necessarily contiguous. For example, if S is `XYZZY`, then `XYZ` and `YY` are both subsequences, but `ZYX` is not.

Given two strings X, Y, the LCS problem is:

> Find a string Z which is a subsequence of both X and Y, and which is as long as possible.

This problem sometimes comes up in computational biology: we have a database of DNA sequences, and we want to check whether a given sequence fragment matches anything in the database. Since the sequences can have errors including insertions, deletions, and changes, and since the sequences can be changed by mutations, we want to find the database entry that has the longest common subsequence with our query.

A naive recursive algorithm for the LCS problem works as follows: each recursive call $\mathrm{LCS}(i,j)$ finds the longest common subsequence of $X[1:i]$ with $Y[1:j]$.

- If $X[i]=Y[j]$, then the last character of $\mathrm{LCS}(i,j)$ will be the common value, say q. We can recursively call $LCS(i-1,j-1)$ and append q.
- If $X[i]\neq Y[j]$, then we must discard either $X[i]$ or $Y[j]$. We can recursively call $LCS(i-1,j)$ and $LCS(i,j-1)$ and return the longer one.
- If either sequence is empty (i.e., if $i=0$ or $j=0$) then the LCS is empty.

The naive algorithm takes exponential time, since it makes two recursive calls that each reduce i or j by only 1. But we can get a much more efficient algorithm with dynamic programming: we keep an array $A[i,j]$ of function

values. We can either memoize or sweep; either way, we calculate each $A[i, j]$ once, at $O(1)$ cost given the previous values. If X and Y have lengths m and n respectively, there are mn elements that we have to calculate, so the whole process takes $O(mn)$ time — way better than exponential. (We do need an extra $O(mn)$ memory to store A.)

Exercise: use this algorithm to calculate the LCS of `XYZZY` and `XYZXYZ`.

DP example: shortest paths

One place that dynamic programming shows up in machine learning is in reinforcement learning, in the so-called *Bellman equation*. We'll analyze a simplified version of this recursive equation: we'll consider only deterministic problems where we are given a model of our environment, instead of the general case where we can have randomness and uncertainty.

More specifically, our goal is to compute all-source single-destination shortest paths in a weighted directed graph. That is, we have a graph (V, E) with edge costs $c_e \in \mathbb{R}$ for $e \in E$. Our goal is to find the lowest cost paths from all possible starting vertices $s \in V$ to a single marked goal vertex $t \in V$. Negative costs are allowed, so long as there are no negative-cost cycles: if there were, we could go around one forever and get a path with total cost equal to $-\infty$.

To this end, we want to find the *cost-to-go* or *value* function $J(v)$ for $v \in V$: this function tells us the best possible cost of a path from v to the goal t. We'll see below that knowing J lets us read off the shortest path from any vertex.

We can recursively define J as:

$$J(t) = 0 \qquad J(v) = \min_{(v,w) \in E} [c_{v,w} + J(w)]$$

That is, we can split the shortest path from v into two pieces: the first edge (v, w) and the rest of the path from w to t. The total cost of the path is the sum of the costs of the two pieces, and the best path is the one that minimizes this cost.

This recursive definition suggests a recursive algorithm: to compute $J(v)$ we iterate over all edges (v, w), recursively call $J(w)$, and set $J(w)$ to the best cost we find. This algorithm is not a good one, since it can turn into an infinite loop if

there's a cycle in our graph.

But, this recursive definition and algorithm do show us how to use J once we know it: the shortest path from v is the one that goes through the best w. So, if we're at v, we can find the best path cheaply if we know J: we loop over all outgoing edges, pick the best one, move to w, and repeat.

We can get a better recursive algorithm by including a limit on the number of edges we'll explore: define $J(v, n)$ to be the cost of the best path from v to t that contains at most n edges. Then

$$J(t, 0) = 0 \qquad J(v, 0) = \infty \quad \forall v \neq t$$

$$J(v, n) = \min_{(v,w)\in E} [c_{v,w} + J(w, n-1)] \quad \forall v, \forall n \geq 1$$

This algorithm still takes exponential time, but at least it doesn't take infinite time!

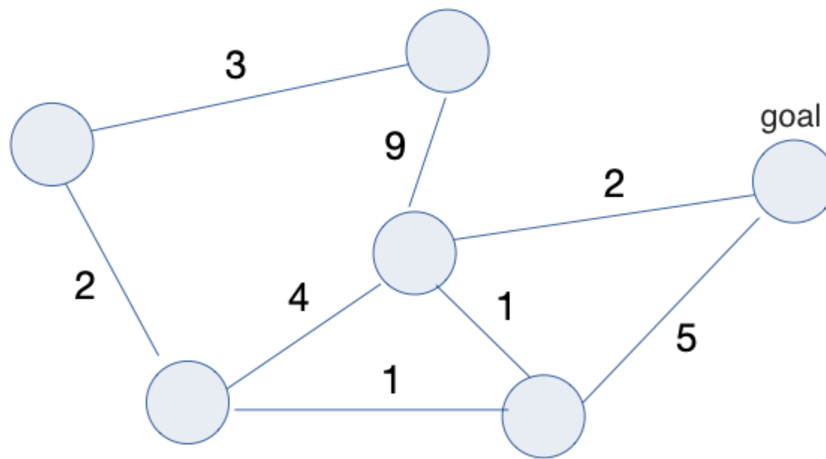
We can improve our algorithm by using dynamic programming: we sweep the path length n up from 0 to $|V| - 1$, since no cycle-free path can have more than $|V| - 1$ edges. We initialize $J(v, 0)$ to ∞ for all v, except for $J(t, 0) = 0$. Then for $n = 1, 2, \ldots$ we sweep over all $v \in V$ and calculate

$$J(v, n) = \min\left[J(v, n-1),\ \min_{(v,w)\in E} [c_{v,w} + J(w, n-1)]\right]$$

We can optimize by dropping $J(\cdot, n-1)$ once we've computed $J(\cdot, n)$. And, if we sort E so that we group edges that share a starting vertex, then it takes time $O(|E|)$ to compute $J(v, n)$ for all v.

The result is the *Bellman-Ford* algorithm. With the optimizations above, its total runtime is $O(|V|\,|E|)$ and its total memory use is $O(|V|)$: again, dynamic programming gives us a huge win over the original exponential-time algorithm.

Exercise: use Bellman-Ford to calculate least-cost paths for the graph below.



Note that this graph is undirected. That means we can traverse each edge in either direction: if $(v, w) \in E$, then $(w, v) \in E$ as well (and the cost is the same in both directions).

Resources: notes on dynamic programming from Avrim Blum; Bellman-Ford on Wikipedia