Lecture 5: DYNAMIC PROGRAMMING [*]

10-607 COMPUTATIONAL FOUNDATIONS FOR MACHINE LEARNING

1 Dynamic Programming

Recall the Fibonacci function:

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

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

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

Unfortunately this algorithm is very inefficient; let's analyze it. We can write a recurrence for $C(n)$, the number of function calls required 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 $1 + C(n-2) \leq C(n-1)$. Substituting into the recurrence above, $C(n) \leq 2C(n-1) \leq 4C(n-2) \leq 2^k C(n-k)$ so long as $n - k \geq 2$. The latter implies $C(n) \leq 2^{n-2}C(2) = 2^{n-2} \cdot 3 \leq 2^n$. That is, $C(n) \in O(2^n)$: the algorithm takes *exponential* time! (In general, we say an algorithm takes exponential time if its runtime is $2^{O(n^d)}$ for some $d \geq 1$.)

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

The answer is yes. Intuitively, the approach is to iterate over or *sweep* k from 2 to n, calculating $\text{fib}(k)$ just once for each k, reusing previously computed values. A simple way to organize the computation is with an *array* A:

$A(0) \leftarrow 1$
$A(1) \leftarrow 1$
with $(k \leftarrow 2)$
while $(k \leq n)$
do $A(k) \leftarrow A(k-1) + A(k-2),\ k \leftarrow k+1$
return $A(n)$

[*]Compiled on Wednesday 4th October, 2023 at 17:07

Note that we haven't formally defined array types yet: they require some tools we haven't provided, which 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 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*. Crucially, dynamic programming works for any pattern of recursive calls.

We can think of dynamic programming as trading memory for time: whenever we need 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 fib(k) and fib($k-1$) at any given time. (From these values we can calculate fib($k+1$), discard fib($k-1$), and increment k.) In general, though, dynamic programming might need to store all previous values of k.

However, this 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.

1.1 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 2-D array to store the function values. For example, Pascal's triangle looks like this:

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

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 3-D 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 naïve 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.

1.2 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 be 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* or placeholder value, F, for all pairs 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 pre-computed.

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.

Knowledge check

1. **Select one:** What is the fundamental tradeoff between a naïve recursive implementation of a function and dynamic programming?
   - A Dynamic programming is easier to implement but the resulting code is less interpretable.
   - B Dynamic programming results in fewer recursive calls but is not applicable to all recursive functions.
   - C Dynamic programming reduces the number of arguments to a recursive function but can occasionally return wrong answers.
   - D Dynamic programming reduces the runtime of computing a recursive function but increases the amount of memory required.
   - **Answer**: D; dynamic programming requires storing previously computed values in memory so as to not have to recompute them.

1.3 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 often arises 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 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 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.

Knowledge check

1. **Select one:** If X and Y have lengths m and n respectively, what is the big O runtime of the dynamic programming solution to the LCS problem?
   A $O(mn)$
   B $O(m^2n)$
   C $O(mn^2)$
   D $O(m^2n^2)$
   - **Answer**: A; there are mn elements that we have to calculate, so the whole process takes $O(mn)$ time, which is way better than exponential! Of course, we do need an extra $O(mn)$ memory to store A.