Dynamic Programming I

Dynamic Programming is a general technique to design efficient algorithms for a variety of problems. The word programming in 'dynamic programming' refers to the fact that the method consists of filling in a table. We will start with a simple example of a dynamic programming algorithm, and then give discuss the general technique and setting for dynamic programming.

String Reconstruction.

We start with the following 'toy problem'. Suppose that all the blanks and punctuation marks have been inadvertently deleted from a text file, so the file looks something like "onceuponatimeinafarfarawayland...". We wish to reconstruct the file, using only an online dictionary for help. Equivalently, we can restate the problem as follows: given a sequence of n characters $x_1, \ldots, x_n$, and a dictionary of allowable words, find a decomposition of the sequence into words from the dictionary. Let us start by solving the simpler decision problem: is there a decomposition of the given sequence into words from the dictionary?

Let us start by defining $T(i)$ to be 1 if the sequence $x_1, \ldots, x_i$ can be decomposed into a sequence of words from the dictionary, and 0 otherwise. The value we are interested in computing is $T(n)$. We will show that if have already computed the values $T(1)$, ..., $T(j)$ then we easily compute $T(j+1)$ as follows:
$T(j+1) = 1$ iff there is a k, $1 \leq k \leq j+1$ such that $T(k-1) = 1$ and $x_k x_{k+1} \ldots x_{j+1}$ is a word in the dictionary. To make this definition consistent we define $T(0) = 0$. We can state the above condition in mathematical notation (identifying 1 with True and 0 with False) as follows:
$T(j+1) = \bigvee_{1 \leq k \leq j+1} [T(k-1) \wedge Dict(k, j+1)]$

The algorithm should now be clear. We fill in the entries of a table T with entries indexed from 0 to n. $T(0)$ is initialized to 0. After the first j entries have been filled in, the $j+1^{st}$ entry is filled in by looking running through the indices $1 \leq k \leq j+1$, and using the above rule. Assuming that dictionary lookup takes a single step, this algorithm takes $O(n)$ steps to fill in each entry in the table, for a grand total of $O(n^2)$ steps.

Getting back to our original goal — of finding a decomposition of $x_1, \ldots, x_n$ into a sequence of words from the dictionary — rather than just determining whether or not such a decomposition exists. This is easily accomplished as follows: in addition to the table T, we also store a table B, where if $T(i) = 1$, then $B(i)$ is the index of the beginning of the word ending at the x_i. The algorithm is modified as follows: when algorithm sets entry $T(j+1) = 1$, it sets $B(j+1) = k$, where k is an index such that $[T(k-1) \wedge Dict(k, j+1)]$.

Naturally, this program just returns a meaningless Boolean, and does not tell us how to reconstruct the text. Expanding the innermost loop (the last assignment statement) to

$\{T[i,j]$:=true, first$[i, i+d] := k$, exit for$\}$

where first is an array of pointers initialized to *nil*, gives us also the end of the first word of each substring that is indeed the concatenation of dictionary words. Notice that this improves the running time, by exiting the for loop after the first match; more optimizations are possible. This is typical of dynamic programming algorithms: Once the basic algorithm has been derived using dynamic p rogramming, clever modifications that exploit the structure of the problem speed up its running time.

Dynamic Programming versus Divide and Conquer

It is useful to compare and contrast two different paradigms for designing algorithms: divide and conquer and dynamic programming.
Divide and Conquer: starting with an instance of size n, we break it into a instances of size n/b each. Moreover we show how to 'glue' the solutions to these subproblems together in $O(n^c)$ steps to obtain a solution to our original instance. Why does this outline yield a polynomial

time algorithm? One way to see this is to analyze the recurrence relation. But if we want just a crude polynomial time bound (i.e. it might give us an $O(n^3)$ bound instead of $O(n^2)$, for instance), then we can reason as follows: the total running time of the algorithm is bounded by the total number of subproblems (as we run through the recursion) times the maximum gluing cost. Since the maximum gluing cost is $O(n^c)$, we just have to show that the total number of subproblems is bounded by some polynomial in n. Why is this the case? Let us work out, for example, the case $a = 2$ and $b = 2$. At the lowest level of recursion, we have n subproblems of size 1 each. At the next lowest level, there are $n/2$ subproblems of size 2 each... The total number of subproblems is therefore $n + n/2 + n/4 + \ldots \leq 2n$.
Exercise: Show for any constants of a, b the total number of subproblems is $O(n^d)$, where d is some constant that depends upon a, b.
Dynamic Programming: notice that in the algorithm for string reconstruction, to solve an instance of size n, we needed the solutions to instances of size $n-1$, $n-2$, ..., 1. The problem is that the sizes of the problem instances is no longer dropping by a constant factor in each iteration. Is the number of subproblems still polynomially bounded?

First let us answer that question assuming no further properties of our instance. For simplicity let us assume that each instance of size n is broken into two instances — one of size $n-1$ and the other of size $n-2$. How many subproblems do we get in all? If $S(n)$ is the number of subproblems we get when we start from an instance of size n, then we have

$$S(n) = S(n-1) + S(n-2)$$

But this gives us the Fibonacci numbers — $S(n)$ is exponentially large in n. But then why did our algorithm for string reconstruction run in polynomial time? Recall that while computing Fibonacci numbers we ran into the same problem: if we computed them recursively, it took exponential time. However, if we iteratively worked our way up or memoized, then it took polynomial time. This was because if we blindly computed recursively then to compute $F(n)$ we must compute $F(n-1)$ and $F(n-2)$. Now to compute $F(n-1)$ we must compute $F(n-2)$ and $F(n-3)$. The recursive algorithm does not notice that $F(n-2)$ is being computed twice. Expanding the tree of recursion a few more levels reveals that each number is re-computed a very large number of times (actually an exponential number of times!).

In general dynamic programming works whenever there is some special property of the problem that allows us to bound the number of subproblems. Here are a few of the typical reasons we can bound the number of subproblems:

- The input is $x_1, x_2, \ldots x_n$. A subproblem is $x_1, x_2, \ldots x_i$.

  The number of distinct subproblems is n. This was the case for the string reconstruction problem.

- The input is $x_1, x_2, \ldots x_n$ and $y_1, y_2, \ldots y_m$ A subproblem is $x_1, x_2, \ldots x_i$ and $y_1, y_2, \ldots y_j$.

  The number of distinct subproblems is nm. This is the structure for the edit distance problem, which we will see in the next lecture.

- The input is $x_1, x_2, \ldots x_n$. A subproblem is $x_i, x_2, \ldots x_j$.

  The number of distinct subproblems is $O(n^2)$. This is because there are n choices for i, and for each way of choosing i, there are at most n ways of choosing j. This is the structure for the matrix chain multiplication problem.

  The exact number of subproblems is not hard to figure out either: there are $\binom{n}{2} = n(n-1)/2$ ways of picking distinct i, j. We must add to this the n ways of choosing $i = j$ for a grand total of $n(n+1)/2$.

The moral is that to design a dynamic programming algorithm, you must look for special structure — usually a linear structure — that allows us to bound the total number of subproblems.