DYNAMIC PROGRMAMING

Dynamic Programming is a very important general technique for algorithm design. It is the first technique you should try when confronted with a problem which does not at first look to be solvable in polynomial time; in many non-obvious polynomial time algorithms, it is Dynamic Programming which does the trick. [Generally, divide and conquer, the other general technique we saw enables one to improve the time for a problem which is already solvable in polynoimial time - for example, sorting, multiplication of two integers, matrices etc.]

A good starting point for a Dynamic Programming solution is to design a recursive procedure; then instead of implementing the recursion, one does it bottom-up - we see all the values of the parameters for which the recursive procedure would get executed in a recursive program (each possibly many times); then run the procedure on these values starting with the least values first (bottom-up). Whenever we are to make a recursive call, we instead fetch the result of the procedure on those values since this has already been calculated.

Example 1 : Matrix-Chain product :

Suppose we wish to compute the product $A_1 A_2 \ldots A_n$ of n matrices $A_1, A_2 \ldots A_n$, where A_i is $p_i \times p_{i+1}$. Note that the cost of multiplying a $p \times q$ matrix by a $q \times r$ matrix is pqr. [Examples, more discussion in class.] We wish to find the most efficient way of finding this product.

To first design a recursive procedure for this and all other Dynamic Programming problems, there are three points to remember :

Three Points

(1) Suppose we had on hand the optimal solution to the problem (in this case - the order in which we multiply). What is the last thing (in this case - the last multiplication) that that solution performs ?

(2) What are all the possibilities for the last thing ?

(3) Each possibility in (2) leads to a “recursive” call. What are all the sub-problems solved by a recursive procedure ?

If the last multiplication occurs in the position between k and $k+1$, then the solution has already computed the product $A_1 A_2 \ldots A_k$ and the product

$A_{k+1}A_{k+2}\dots A_n$ and the last thing it does is to multiply the two products at a cost of $p_1 p_k p_{n+1}$. Now clearly, since it is an optimal solution, it would have done the two products $A_1A_2\dots A_k$ and $A_{k+1}A_{k+2}\dots A_n$ also in the optimal manner. So, if someone magically told you that the last operation was the multiplication between k and $k+1$, then we could recursively find the optimal solution value to the problem involving $A_1, A_2, \dots A_k$ and the other problem involving the other matrices and from that get the optimal value of our problem.

But of ocurse no one would tell us this information; this seems as hard as solving the whole problem ! Here we come to our second question - What are all the possibilities for k ? Clearly k could only be 1 or 2 or 3 or $\dots n-1$, for a total of $n-1$ possibilities. We will see that instead of assuming we know k magically, we can try out all the possibilities.

Define $\text{Ans}(i,j)$ to be the minimum number of operations needed to find the product - $A_iA_{i+1}\dots A_j$ for each pair of numbers i,j with $1 \le i \le j \le n-1$. Then the above argument says that

$$\text{Ans}(i,j) = \begin{cases} 0 & \text{if } i = j \\ \text{Min}_{k:i\le k<j}\text{Ans}(i,k) + \text{Ans}(k+1,j) + p_i p_{k+1} p_{j+1} & \text{if } i < j. \end{cases}$$

This is called the Dynamic Programming Recursion. It directly leads to a recursive procedure. But it is easy to see that the reursive procedure is wasteful - it repeatedly calls itself on the same arguments !!

To avoid the waste, let us figure out for which parameter values the procedure is called. It is easy to see that the paremeters i,j can only take on values satisfying $1 \le i \le j \le n-1$, for a total of $O(n^2)$ sets of values. Now, instead of a recursive procedure, we will do the computation bottom-up filling up a table of values of $\text{Ans}(i,j)$. What is the correct order for doing this ? We must do this in an order which makes sure that whenever we are computing a particular $\text{Ans}(i,j)$, the $\text{Ans}(\cdot,\cdot)$ needed for the computation of $\text{Ans}(i,j)$ have all been already computed. It is easy to see that it suffices to do the computation in increasing order of $j-i$. It is a simple exercise now to write the program itself.

Time Analysis For computing each of the $O(n^2)$ $\text{Ans}(i,j)$'s, we have to "fetch" $O(n)$ previously computed $\text{Ans}(\cdot,\cdot)$'s, find $O(n)$ quantities - $\text{Ans}(i,k) + \text{Ans}(k+1,j) + p_i p_{k+1} p_{j+1}$, each requiring $O(1)$ operations, and take the minimum of these $O(n)$ quantities, for a grand total running time of $O(n \times n^2) = O(n^3)$.

Finding the Solution The above only finds the optimal solution value - i.e., the number of operations needed in an optimal solution. It remains to see how to construct the optimal solution; this part is usually conceptually simple once we know how to compute the optimal value as above. Here is a quick idea - it is sufficient to remember for each i, j, which k acheives the minimum in the Dynamic Programming Recursion above. One computes this information as we execute the algorithm, computing the values of $\text{Ans}(i, j)$ bottom up. Once we have finished, we can reconstruct the optimal order by looking up the best k for $1, n$ - call it k_1, then looking up the best k 's for $1, k_1$ and for k_1, n etc. You could consult any text book for details.

Example 2 : Knapsack Problem

The problem is given n items, where item i has weight a_i and benifit c_i and a knapsack of capacity b, find the subset of items which "fits" (whose total weight is at most b) and has maximum total benifit subject to this. This problem can obvioulsy be stated as :

$$\begin{aligned} \text{Max} \sum_{i=1}^{n} c_i x_i \\ \text{subject to } \sum_{i=1}^{n} a_i x_i \leq b \\ x_i \in \{0, 1\}, \end{aligned}$$

where we will assume that the a_i, c_i and b are given non-negative integers.

Let us try to answer the two questions. Suppose we had an optimal solution on hand; what is the last thing we did ? - this has to do with whether we put in or did not put in the last item into the knapsack. There are then clearly only these two possibilities for the last item. If we are magically told what we did with the last item, then it suffices to solve the problem with the remaining $n-1$ items - namley - items $1, 2, \ldots n-1$, but with a modified capacity (if we had packed the last item in.) While solving this, we have to tackle the problem with the first $n-2$ itmes with different capacities etc. Thus, a recursive program will solve the problems

What is the best benifit we can derive with the first i items, with a knapsack of capacity y ?

where y is in the range $\{0, 1, 2 \ldots b\}$. This suggests defining the quantity - $Ans(i, y)$ - which is the maximum benifit we can derive by packing a subset of the first i items into a knapsack of capacity y. The dynamic programming recursion is then (where for simplicity we take $\text{Ans}(i, y)$ to be $-\infty$ if $y < 0$)

$$\text{Ans}(i, y) = \begin{cases} 0 & \text{if } y = 0 \text{ or if } i = 0 \\ \text{Max}\,(\text{Ans}(i-1, y-a_i) + c_i\ ,\ \text{Ans}(i-1, y)) & \text{if } i, y \neq 0. \end{cases}$$

There are a total of $n(b+1)$ entries in the table to filled out bottom-up. Each entry requires at most a the maximum of a constant number of comparisons and other arithmetic operations. So the total running time is $O(nb)$. You can think about the following two questions :

The knapsack problem is known to be NP-hard, so it will only admit a polynomial time algorithm if NP=P. Does the above alg. qualify as a polynomial time algorithm ?

How does one get the actual solution ?

Example 3 : Allocation of a single Resource

There are n tasks and m units of a resource. We are given values - $A_{ij}, i = 1, 2, \ldots m; j = 1, 2, \ldots n$, where A_{ij} tells us the amount of benifit we can derive by allocating i units of the resource to task j. We are to find the allocation (how many units of resource to each task) that maximizes the total benifit. The problem then is : find $x_1, x_2, \ldots x_n$, satisfying :

$$\sum_{j=1}^{n} A_{x_i,j} \text{ is maximized subject to}$$

$$x_i \in \{0, 1, \ldots m\}, \quad \sum_{i=1}^{n} x_i \leq m.$$

The last thing we do in an optimal solution is - we allocate a certain number of units of the resource to the last task. The possible values of this number are $0, 1, 2, \ldots m$. Once we know this value, we have to solve a similar resource allocation problem on the first $n-1$ tasks.

This leads us to define $\text{Ans}(i, j)$ as the maximum benifit we can derive by an allocation of i units of the resource among the first j tasks. We have the recursion:

$$\text{Ans}(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or if } j = 0 \\ \text{Max}_{k: 0 \leq k \leq i} \text{Ans}(i-k, j-1) + A_{kj} & \text{if } i, j \neq 0. \end{cases}$$