CS270 Combinatorial Algorithms & Data Structures **Spring 2003**

Lecture 12: 2.27.03

Lecturer: Papadimitriou *Scribes: Brian Fields and Soni Mukherjee*

12.1 Overview of Dynamic Programming

In the context of dynamic programming, the "programming" part means "planning," "scheduling," which is the meaning the work had in the 1950's —just like in "linear programming". Another thing these two have in common is that they serve as ultimately powerful tools, "sledgehammers" in the algorithm designer's toolbox. While in the case of *linear* programming many problems can be reduced to it, dynamic programming is broadly applicable due to being a very general technique. In fact, dynamic programming is so general and powerful, that perhaps all computation can be represented through it (in the sense that all computation can be reduced to the evaluation of a sequential circuit, as we shall note in relation to NP-completeness, and this is just like a DAG).

Dynamic programming is similar to the divide-and-conquer approach in that the solution of a large problem depends on previously obtained solutions to easier subproblems. The significant difference, however, is that dynamic programming permits subproblems to overlap. By overlap, we mean that the subproblem can be used in the solution of two different subproblems. In contrast, the divide-and-conquer approach creates subproblems that are completely separate and can be solved independently.

An illustration of the difference is shown in Figure 12.1. Here, the problem to be solved is shown as the root of a tree, where children are easier subproblems. The leaves of the trees are trivial subproblems that can be solved directly — in D.P., these leaves are often the input to the algorithm. The primary difference between divide and conquer and D.P. is clear: subproblems in divide and conquer do not interact, while in D.P., they might.

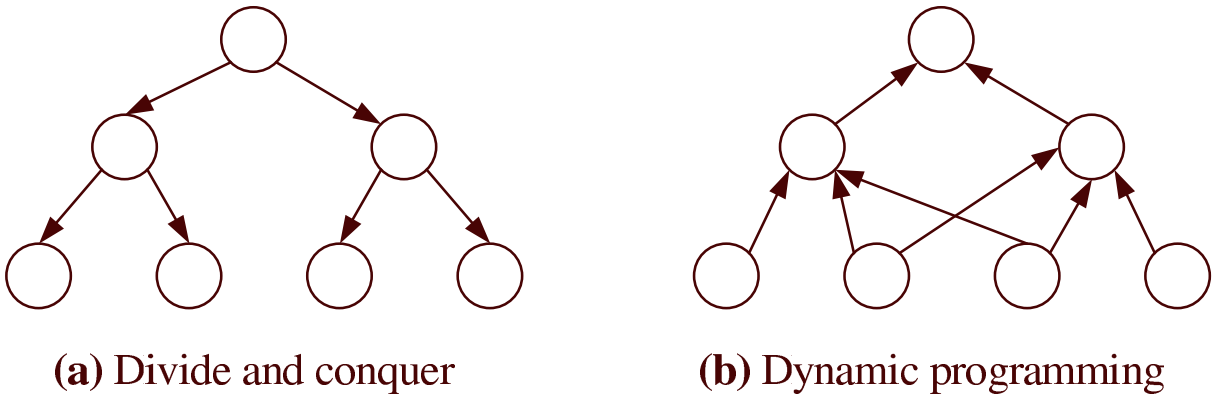

Figure 12.1: **Contrasting divide and conquer and dynamic programming.** The primary difference is that the subproblems of the divide-and-conquer approach are independent, while in dynamic programming they interact. Also, dynamic programming solves problems in a "bottom-up" manner as opposed to divide-and-conquer's "top-down" approach.

A second difference is also illustrated by the figure: while the divide-and-conquer approach is recursive, top-down,

D.P is best thought as bottom up. It will become clearer how this works as we work through several examples. We will start off with a very simple application of dynamic programming: computing Fibonacci numbers.

12.2 Example 1: Computing Fibonacci numbers

For the first time in this class, we will work with Fibonacci numbers. It is remarkable that we got this far without them, considering the level of attention paid to them by computer theorists. To see how important they are to some people, look up the latest issue of Fibonacci Quarterly, a publication of the Fibonacci Association [WEB].

We'll start with a straightforward recursive algorithm for computing Fibonacci numbers, given below.

Algorithm 1 Fib(n)

```
1: if (n ≤ 2) then
2:     return 1
3: else
4:     return Fib(n − 2) + Fib(n − 1)
```

The running time of this algorithm is described by the following recursion.

$$T(0) = T(1) = 1 \tag{12.1}$$

$$T(n) = T(n-1) + T(n-2) \tag{12.2}$$

The solution of this recursion is very exponential. Clearly this is not a very efficient algorithm for computing Fibonacci numbers.

For finding better solutions, the key point to notice is that the inefficiency comes from recomputing solutions to subproblems which have already been computed before. We can greatly improve running time by remembering previous solutions, e.g. in a hash table, and recalling them when the result is needed again. Only when the solution to a subproblem is not in the hash table does it need to be computed from scratch.

The new algorithm is as follows:

Algorithm 2 Fib(n)

```
1: if (Fib(n) solution in hash table) then
2:     return hash_table[n]
3: if (n ≤ 2) then
4:     return 1
5: else
6:     sol ← Fib(n − 2) + Fib(n − 1)
7:     store sol in hash table as solution to Fib(n)
8:     return sol
```

This approach to solving the problem is known as **memoization**, which is closely related to dynamic programming. By removing the recursion, we can show the algorithm in a manner more typical of dynamic programming:

Algorithm 3 Fib(n)

```
1: F [1] ← F[2] ← 1
2: for i ← 2 to n do
3:     F[i] ← F[i − 1] + F[i − 2]
4: return F[n]
```

Note that F[1] and F[2] are the trivial subproblems of this algorithm. In other words, they are the leaves of the D.P. DAG of Figure 12.1. The rest of the (nontrivial) subproblems are solved efficiently by directly using the solution to simpler subproblems. It is easy to see from this algorithm that dynamic programming is a bottom-up technique: the simpler subproblems are solved first, then those solutions are employed for solving more sophisticated subproblems.

It is often useful to draw a DAG for dynamic programming algorithms, where the nodes are the subproblems and the edges are references between the subproblems. We did this at a conceptual level in Figure 12.1. A specific DAG for the fibonacci algorithm is shown below.

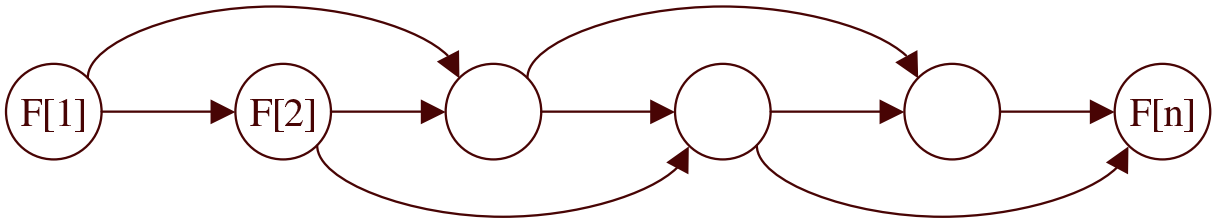

Figure 12.2: **Fibonnaci DP graph.** This DAG is a specific instantiation of the DP tree illustrated in Figure 12.1. The complexity of the algorithm can be determined by counting the number of edges.

We can determine the complexity of dynamic programming algorithms by counting the number of edges in a DAG such as this one (although it is sometimes easier to simply examine the loop nestings). In this graph, there are $2n$ edges, since there are n nodes and each node (except the first two) has an in degree of 2. Thus, the complexity is O($2n$) = O(n).

12.3 Example 2: Cutting a rod example

The goal of the algorithm is to make n cuts through the rod at pre-specified locations with minimal effort, or cost. The cost of each cutting operation is proportional to the length of the rod that is to be cut. In the real world, you could consider the cost to be the effort to lift the portion of the rod onto the cutting machine. The rod and the input parameters are illustrated in Figure 12.3.

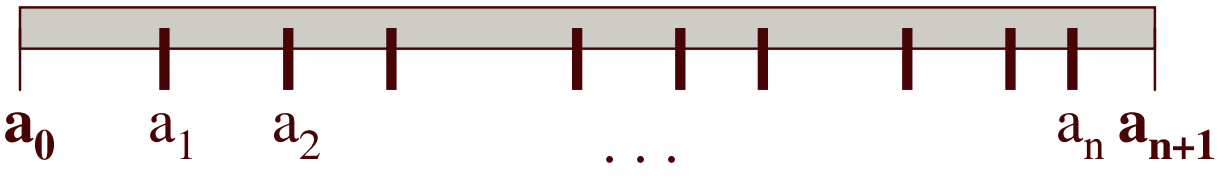

Figure 12.3: **Cutting the rod example.** Each of a_1 through a_n are prespecified points where the rod is to be cut. a_0 and a_{n+1} denote the two ends of the rod.

To apply dynamic programming to a problem, the first step is always to define an appropriate subproblem. Before we do that, however, let's start with a couple of definitions. Let A be the set of all cuts that must be made to the rod, where $A = a_1, a_0, a_1, ..., a_n$. Thus, A is input to the algorithm. Also, let $Cost(i, j)$, where $j > i$ be the minimum possible cost to make all necessary cuts of the rod between a_i and a_j.

The elementary, trivial, subproblem is $Cost(i, i + 1)$, which is obviously equal to zero — since no cuts need to be made. The question is how should we define a subprobem for $j > i + 1$ in terms of simpler subproblems?

Well, first, we know that at least one cut will have to be made, so $Cost(i, j)$ will be at least $a_j - a_i$. (We have to lift the rod onto the table at least one time.) The rest of the cost will depend upon where this first cut is made. To correctly

compute the minimum cost, we need the placement of the cut, call it k, to minimize the cost function for each of the two remaining pieces, $Cost(i,k)$ and $Cost(k,j)$. This gives us the following formula:

$$Cost(i,j) \leftarrow a_j - a_i + \min_{i<k<j}\{Cost(i,k) + Cost(k,j)\} \tag{12.3}$$

Once we have this formula, it is easy to convert it to a dynamic programming algorithm. The strategy is to ensure a bottom-up progression, solving the simpler subproblems first. The algorithm is shown below:

Algorithm 4 Cost(A)

```
1:  for i ← 0 to n + 1 do
2:      C[i, i + 1] ← 0
3:  for h ← 2 to n + 1 do
4:      for i ← 0 to n + 1 − h do
5:          C[i, i + h] ← a_j - a_i + min_{i<k<j} C[i, k] + C[k, j]
6:  return C[0, n + 1]
```

Here, the loop on line 3 grows the size of the rod for which the cost function is to be computed from 2 to $n+1$. When $h = 2$, the costs can be computed from the trivial subproblems, where the rod size is 1; and as h increases, the increasingly more sophisticated subproblems can be solved from the solutions to the already computed, simpler subproblems.

The complexity of the algorithm is easily determined to be O(n^3) by examining the number of nested loops, remembering the implementation of the **min** function would be a third loop.

An alternative way to determine complexity, as in the Fibonacci example above, is to count the number of edges in the D.P. tree. For this algorithm, there are n^2 subproblems, corresponding to computing $Cost(i,j)$ for all necessary i's and j's. Each subproblem is a node in the tree. The in degree at each of these nodes is equal to the number of possible cuts, i.e., choices for k. Thus, the total number of edges is $O(n^3)$.

Although it is not shown in the algorithm above, a real implementation would maintain, for each C[i, j], the k value that resulted in the minimum cost. That way, once the algorithm has finished, a binary tree is created where each node represents a portion of the rod; and the children of each node are the two pieces which remain after the cut at k.

12.4 Example 3: Substring matching

Suppose there is a database A of length m, for example, a list of all the computer science professors. Each string in the database is not delimited in any way from the others, so it would look something like: 'AIKENBODIKCULLER...'. Our goal is to find the closest match in the database for an input string b of length n, for example b could be 'BABADIMITIOS'. More specifically, we want to find the (contiguous) substring in A for which the edit distance is the smallest, where *edit distance* is defined as the number of edits necessary to convert the substring of A to b. Possible edits include inserting a character, deleting a character, and overwriting a character with another character.

As with most dynamic programming problems, the hardest part is defining the subproblem. In this case, the subproblem is finding the edit distance between a prefix of b and a substring of A. More specifically, $ed[i,j]$ is the minimum edit distance between some contiguous string ending at $A(i)$, where $A(i)$ is the ith letter of A, and the string $b(1)b(2)...b(j)$.

For our elementary (trivial) subproblems, we have

$$ed[i,0] \leftarrow 0 \tag{12.4}$$

because if b is empty, then no edits are needed for the empty string in A to be indentical to b, and

$$ed[0, j] \leftarrow j \tag{12.5}$$

because if we must start with the empty string from A, then the quickest way to get to $b(1)b(2)...b(j)$ is to make j inserts.

To gain intuition about the dynamic programming solution to this problem, we will track the values of $ed[i, j]$ in a table, where i increases as we move down and j increases as we move from left to right. After solving the elementary subproblems, the table is as follows.

		B	A	B	A	D...
	0	1	2	3	4	5...
A	0					
I	0					
K	0					
E	0					
N	0					
B	0					

Now we will attempt to solve more sophisticated subproblems by using results from simpler subproblems.

There are four ways to get to $ed[i, j]$ from smaller (and previously solved) subproblems. Each of these ways is specified by determining what the last step will be when editing a contiguous string ending at $A(i)$ to get $b(1)...b(j)$. Then, we determine what needs to be done before the last step (which represents a simpler subproblem), and putting these two parts together, we get an equation for $ed[i, j]$.

1. If the last step is an insert, then first edit a contiguous string ending at $A(i)$ until we get $b(1)...b(j-1)$, and then insert $b(j)$.

$$Result : ed[i, j] \leftarrow ed[i, j-1] + 1 \tag{12.6}$$

In the table, this corresponds to moving one position to the right. For example, consider $ed[2, 2]$. First we edit a string ending at $A(2) \rightarrow I$ to get $b(1) \rightarrow B$. This takes $ed[2, 1] \rightarrow 1$ edit. Then, we insert $b(2) \rightarrow A$, to the end to get $b(1)b(2) \rightarrow BA$. This takes another edit, for a total of 2 edits. Therefore, $ed[2, 2] \leftarrow 2$.

		B	A	B	A	D...
	0	1	2	3	4	5...
A	0	1	1			
I	0	**1**				
K	0					
E	0					
N	0					
B	0					

$\longrightarrow$

		B	A	B	A	D...
	0	1	2	3	4	5...
A	0	1	1			
I	0	**1**	**2**			
K	0					
E	0					
N	0					
B	0					

2. If the last step is a delete, then first delete $A(i)$ from a contiguous string ending at $A(i)$. Then, edit the remaining string until we get $b(1)...b(j)$.

$$Result : ed[i, j] \leftarrow 1 + ed[i-1, j] \tag{12.7}$$

In the table, this corresponds to moving one position down. Again, consider $ed[2, 2]$. This time, our first step is deleting $A(2) \rightarrow I$ from a string ending at $A(2)$. Then, we edit a string ending at $A(1)$ to get $b(1)b(2) \rightarrow BA$, which takes 1 edit, for a total of 2 edits. Therefore, $ed[2, 2] \leftarrow 2$.

		B	A	B	A	D...
	0	1	2	3	4	5...
A	0	1	**1**			
I	0	1				
K	0					
E	0					
N	0					
B	0					

$\longrightarrow$

		B	A	B	A	D...
	0	1	2	3	4	5...
A	0	1	**1**			
I	0	1	**2**			
K	0					
E	0					
N	0					
B	0					

3. If $A(i)$ is identical to $b(j)$, then nothing is done in the last step. In this case, $A(i)$ is unchanged, and the only edits are those to convert the substring ending at $A(i-1)$ to $b(1)...b(j-1)$.

$$Result : ed[i,j] \leftarrow ed[i-1,j-1] \tag{12.8}$$

In the table, this corresponds to moving diagonally, one position down and to the right. For example, consider $ed[1,2]$. First we edit a string ending at $A(0)$, which is the empty string, to get $b(1) \rightarrow B$. This takes $ed[0,1] \rightarrow 1$ edit. Since $A(1)$ is identical to $B(2)$, the next step is doing nothing, which takes 0 edits. Therefore, the total number of edits is 1, so $ed[1,2] \leftarrow 1$.

		B	A	B	A	D...
	0	**1**	2	3	4	5...
A	0	1				
I	0					
K	0					
E	0					
N	0					
B	0					

$\longrightarrow$

		B	A	B	A	D...
	0	**1**	2	3	4	5...
A	0	1	**1**			
I	0					
K	0					
E	0					
N	0					
B	0					

4. If $A(i)$ and $b(j)$ are not identical, then the last step can be to overwrite $A(i)$ with $b(j)$. For this case, first edit a substring ending at $A(i-1)$ to get $b(1)...b(j-1)$, and then overwrite $A(i)$ with $b(j)$.

$$Result : ed[i,j] \leftarrow ed[i-1,j-1] + 1 \tag{12.9}$$

In the table, this also corresponds to moving diagonally, one position down and to the right. To see this, consider $ed[2,2]$ one last time. First, we edit a string ending at $A(1) \rightarrow A$ to get $b(1) \rightarrow B$, which takes $ed[1,1] \rightarrow 1$ edit. Then, we overwrite $A(2) \rightarrow I$ with $b(2) \rightarrow A$, for a total of 2 edits. Therefore, we again have $ed[2,2] \leftarrow 2$.

		B	A	B	A	D...
	0	1	2	3	4	5...
A	0	**1**	1			
I	0	1	**2**			
K	0					
E	0					
N	0					
B	0					

$\longrightarrow$

		B	A	B	A	D...
	0	1	2	3	4	5...
A	0	**1**	1			
I	0	1	**2**			
K	0					
E	0					
N	0					
B	0					

Since we want to use the minimum number of edits, the formula for $ed[i,j]$ is

$$ed[i,j] = min \begin{cases} ed[i,j-1]+1 \\ 1+ed[i-1,j] \\ dist[A(i),b(j)] + ed[i-1,j-1] \end{cases} \tag{12.10}$$

where $dist$ returns 1 if its two arguments are identical and 0 otherwise, giving us a clever way to combine possibilites **3** and **4** above. The algorithm is shown below:

Algorithm 5 EditDist(A, b)

1: **for** $i \leftarrow 0$ **to** m **do**
2: $ed[i, 0] \leftarrow 0$
3: **for** $j \leftarrow 1$ **to** n **do**
4: $ed[0, j] \leftarrow j$
5: **for** $i \leftarrow 1$ **to** m **do**
6: **for** $j \leftarrow 1$ **to** n **do**
7: (Eq. 12.10)
8: **return** $\min_{0<i<m+1} ed[i, n]$

The reason we return the minimum over i in line 8 is because we want to allow the substring we are editing to end at any place in A, not just the last character.

The resulting DAG has a node for each pair (i, j), so it has a total of mn nodes. Each node has one incoming edge for each of the four possibilities. Therefore, the time complexity of this algorithm is $O(4mn)$, or $O(mn)$.

One problem is that the algorithm only returns the minimum number of edits needed, we may also want to get the substring in A that is the closest match to b. To do this, we can keep a record of what we add, delete, or insert at each step, and then go backwards from b to form the original substring.

Also, instead of actually using an $m * n$ table, we can just use an n-array, and remember only the last row of the table. (But then remembering the closest match becomes tricky...)

12.5 Example 4: Knapsack problem

We have a knapsack that can only hold a certain weight, which we will call $maxwt$, and we want to maximize the total value of the items in the knapsack. We have a group of n items to choose from, each one with a positive weight $weight[i]$ and a positive value $val[i]$, and there is only one copy of each item.

This problem is NP-hard, but we can get the best possbile algorithm by using dynamic programming.

First we must define a subporblem. Our parameters are $maxwt$ and a collection of items, so a subproblem would be filling a knapsack with a smaller max weight and only being able to choose from a smaller collection of items. Since we want to maximize the value in the knapsack, we will use the following function:

$val[w, i]$ is the max value possible in a knapsack with max weight w and choosing from the first i items of a collection.

Our elementary subproblems are

$$val[w, 0] \leftarrow 0 \tag{12.11}$$

because if we have no items to choose from we cannot get any value, and

$$val[0, i] \leftarrow 0 \tag{12.12}$$

because if the knapsack cannot have any weight we cannot put any items in it.

Now we must use subproblems to solve other subproblems.

To solve $val[w, i]$, consider the following two possibilities:

1. We don't choose the ith item. Since adding the ith item to the collection doesn't change anything, we have

$$val[w, i] \leftarrow val[w, i-1] \tag{12.13}$$

2. We choose the ith item. Then, because we get the value of the ith item, but we lose its weight from the max weight of the knapsack that we have to use, we have

$$val[w, i] \leftarrow value[i] + val[w - weight[i], i-1] \tag{12.14}$$

Therefore, $val[w, i]$ is

$$val[w, i] = max \begin{cases} val[w, i-1] \\ value[i] + val[w - weight[i], i-1] \end{cases} \tag{12.15}$$

This way, we can keep solving subproblems until we find $val[maxwt, n]$ for $n =$ total number of items. The algorithm is shown below:

Algorithm 6 Knapsack($maxwt$, n)

```
1: for w ← 0 to maxwt do
2:     val[w, 0] ← 0
3: for i ← 1 to n do
4:     val[0, i] ← 0
5: for w ← 1 to maxwt do
6:     for i ← 1 to n do
7:         (Eq. 12.15)
8: return val[maxwt, n]
```

For the D.P. DAG, we need a node for every pair, (w, i), so the total number of nodes is $n * maxwt$. The in degree of each node is two, one for each possibility. Therefore, the running time of the algorithm is $O(2 * n * maxwt)$, or $O(n * maxwt)$. Since $maxwt$ is an integer, the algorithm is not polynomial, but it is the best that we have.

References

[WEB] Fibonacci Quarterly Home Page. *The Fibonacci Association*. http://www.sdstate.edu/ wcsc/http/fibhome.html.