

Chapter 15: Dynamic Programming

Dynamic programming is a general approach to making a sequence of interrelated decisions in an optimum way. While we can describe the general characteristics, the details depend on the application at hand. Most fundamentally, the method is *recursive*, like a computer routine that calls itself, adding information to a stack each time, until certain stopping conditions are met. Once stopped, the solution is unraveled by removing information from the stack in the proper sequence.

Here is an overview of the method:

1. Define a small part of the whole problem and find an optimum solution to this small part.
2. Enlarge this small part slightly and find the optimum solution to the new problem using the previously found optimum solution.
3. Continue with Step 2 until you have enlarged sufficiently that the current problem encompasses the original problem. When this problem is solved, the stopping conditions will have been met.
4. Track back the solution to the whole problem from the optimum solutions to the small problems solved along the way.

While this sounds new, you in fact already know how to solve a problem by dynamic programming: Dijkstra's shortest route algorithm is classic dynamic programming! The "small part of the problem" at each stage is simply to determine the next closest node to the origin. You enlarge this problem slightly at each stage by appending all of the unsolved nodes and arcs that are directly attached to a solved node. The stopping conditions are met when the next closest node is the destination node. Now you recover the solution by tracing back along the arcs in the arc set from destination node back to the origin.

There are several important characteristics of dynamic programming, as described next.

The problem can be divided into stages. In the shortest route problem, each stage constitutes a new problem to be solved in order to find the next closest node to the origin. In some dynamic programming applications, the stages are related to time, hence the name *dynamic* programming. These are often dynamic control problems, and for reasons of efficiency, the stages are often solved backwards in time, i.e. from a point in the future back towards the present. This is because the paths that lead from the present state to the future goal state are always just a subset of all of the paths leading forward from the current state. Hence it is more efficient to work backwards along this subset of paths. We will be dealing with static examples in which the direction is immaterial, but for form's sake we will also work backwards so that it will not be a surprise when you need to do so for a time-related dynamic control problem. This is known as *backwards recursion*.

Each stage has a number of states. Most generally, this is the information that you need to solve the small problem at a stage. For the shortest route problem, the state is the set of solved nodes, the arcs in the arc set, and the unsolved arcs and nodes directly connected to solved nodes.

The decision at a stage updates the state at the stage into the state for the next stage. In the shortest route problem the decision is the arc to add to the arc set and the corresponding unsolved node that is added to the solved set. This obviously affects the sets of solved and unsolved nodes and arcs and the arcs in the arc set. Hence the decision updates the state for the next stage.

Given the current state, the optimal decision for the remaining stages is independent of decisions made in previous states. This is the fundamental dynamic programming principle of optimality. It means that it is okay to break the problem into smaller pieces and solve them independently. For example, in the shortest route problem, we only care about the total distance from the origin to a solved node; we don't care about the actual route from the origin to that solved node. Decisions made thereafter use only the distance information, without regard to the actual route (i.e. the previous decisions).

There is a recursive relationship between the value of decision at a stage and the value of the optimum decisions at previous stages. In other words, the optimum decision at this stage uses the previously found optima. In a recursive relationship, a function appears on both sides of the equation. In words, the shortest route recursive relationship looks like this:

$$\begin{aligned} & \text{(length of shortest route from origin to node } i) \\ & = \min_{i,j} \{ \text{(length of shortest route from origin to solved node } j) \\ & \quad + \text{(length of arc from solved node } j \text{ to unsolved node } i) \} \end{aligned}$$

Note how the “length of shortest route” appears on both sides of the equation: it is recursive. All dynamic programming recursive relationships show the optimum function on both sides of the equation: the new optimum is always derived from the old optimum along with some local value. Note that the relationship need not be addition as it is here. It could be anything: multiplication or some other abstract relationship.

To formulate a dynamic programming solution, you must answer the following questions:

- What are the states in the solution?
- How is the state defined at a stage?
- What kind of decision must you make at a stage?
- How does the decision update the state for the next stage?
- What is the recursive value relationship between the optimum decision at a stage and a previous optimum decision?

Students are expected to write out these items as part of their problem formulation.

It's time for an example to clarify all of this theory. For some reason, dynamic programming seems to be one of the less intuitive optimization methods and students seem to learn best by being shown several examples, hence that is what we will do next.

An Equipment Replacement Problem

A bicycle courier must obviously have a bicycle at all times that he can use for his deliveries; he can't be without one. Hence he wants to determine the cheapest way to buy and maintain bicycles over the 5-year timeframe that he is currently contemplating working as a bicycle courier before finishing his graduate program and moving on to other career options. He has collected some information about costs. A new Acme bicycle costs \$500. Maintenance costs vary with the age of the bike: \$30 in the first year, \$40 in the second year, and \$60 in the third year. Given the amount of use they get, bicycles will last a maximum of three years before they are sold. He figures he can get \$400 if he sells a bicycle after one year of use, \$300 after two years of use, and \$250 after three years of use. He has no bicycle at the moment. When should he buy bicycles and how long should he keep them to minimize his total costs over the 5-year timeframe?

Let's formulate this as a dynamic programming problem by answering the required questions. The most important items are the stages and the states. See if you can figure out what they might be before looking below. How might we divide this problem into small stages that can be built upon to reach the final answer?

- *Stages*: The problem at time t is to find the minimum cost policy from time t to time 5. Hence we are subdividing the problem by time in this case. And we will be doing a backwards recursion, so the first problem we solve will be trivial: what is the best thing to do from time 5 until time 5? This is just the same as when we start a shortest route solution by labeling the origin with 0, indicating that the distance from the origin to itself is zero. The next stage will solve time 4 to time 5, the next time 3 to time 5, then time 2 to time 5, then time 1 to time 5, and finally we will have enlarged the problem enough to solve the original problem of what to do from time 0 (i.e. now) until time 5.
- *State at a stage*: The state at a stage is fairly simple in this case: just how many years remain until time 5.
- *Decision at a stage*: Since we are solving the problem anew at each stage, the courier has no bicycle now and must buy a bicycle to get started. The decision then, is how long to keep the bicycle he purchases.
- *Decision update to the state*: Given that he buys a bicycle at the start of the current stage, and makes a decision to keep it for a certain number of years, then the state (how many years until time 5) is updated accordingly. For example if the state is currently 4 (i.e. we are at time 1 and hence it is 4 years until time 5), and the decision is to buy a bicycle now and keep it for 3 years, then the state at the next decision point will be $4-3=1$, i.e. we will now be at time 4 with just one year to go until time 5.
- *Recursive value relationship*: First, let's define a few functions and variables:
 - $g(t)$: minimum net cost from time t until time 5, given that a new bicycle is purchased at time t .
 - c_{tx} : net cost of buying a bicycle at time t and operating it until time x , including the initial purchase cost, the maintenance costs, and the salvage value when it is sold.

- The recursive value relationship is then: $g(t) = \min_x \{c_{tx} + g(x)\}$ for $t=0,1,2,3,4$.

Note how $g(\cdot)$ appears on both sides of the recursive relationship. The optimum from time t to time 5 depends on two things: the value of the current decision c_{tx} , and the value of a previously found optimum $g(x)$.

Note that c_{tx} depends only on how long we keep the bicycle (1-3 years only), so we can work out its possible values in advance to save a little calculation later. Each calculation takes into account the initial purchase price, the maintenance costs, and the salvage value:

- Keep bicycle 1 year: $c_{01} = c_{12} = c_{23} = c_{34} = c_{45} = 500 + 30 - 400 = \130
- Keep bicycle 2 years: $c_{02} = c_{13} = c_{24} = c_{35} = 500 + (30 + 40) - 300 = \270
- Keep bicycle 3 years: $c_{03} = c_{14} = c_{25} = 500 + (30 + 40 + 60) - 250 = \380

Before starting the solution, it is interesting to do some simple analysis to see which length of ownership is the most economical. Is it generally better to keep a bicycle for 1, 2, or for 3 years? The average annual cost of ownership is: for 1 year $\$130/1 = \130 , for 2 years $\$270/2 = \135 , for 3 years $\$380/3 = \126.67 . So it is cheapest to keep each bicycle for 3 years, but if that is not possible, then for just 1 year. Given that we have a 5 year timeframe, we should expect to see some combination of 3 year and 1 year ownerships. Now back to dynamic programming to find the optimum solution...

We also define $g(5) = 0$. No costs are incurred after the timeframe ends. Given that we are using a backwards recursion, we start at time 5 and work backwards. The optimum choice (lowest cost) at each stage is highlighted in bold.

- **$g(5) = 0$** . [This is the trivial first stage.]
- **$g(4) = c_{45} + g(5) = 130 + 0 = 130$** . [Also a trivial stage since there is only one possible decision.]
- $g(3) = \text{minimum over: [finally a real decision]}$
 - **$c_{34} + g(4) = 130 + 130 = 260$**
 - $c_{35} + g(5) = 270 + 0 = 270$
- $g(2) = \text{minimum over:}$
 - $c_{23} + g(3) = 130 + 260 = 390$
 - $c_{24} + g(4) = 270 + 130 = 400$
 - **$c_{25} + g(5) = 380 + 0 = 380$**
- $g(1) = \text{minimum over:}$
 - **$c_{12} + g(2) = 130 + 380 = 510$ [tie]**
 - $c_{13} + g(3) = 270 + 260 = 530$
 - **$c_{14} + g(4) = 380 + 130 = 510$ [tie]**
- $g(0) = \text{minimum over:}$
 - **$c_{01} + g(1) = 130 + 510 = 640$ [tie]**
 - $c_{02} + g(2) = 270 + 380 = 650$
 - **$c_{03} + g(3) = 380 + 260 = 640$ [tie]**

At this point we have hit the stopping conditions: the small problems have been expanded until the entire original problem has been solved. At this point we know that the cheapest policy has a total cost of \$640. What we *don't* yet know is how to achieve this result. Here is where we have to unravel the information on the “stack” by tracing back through the solution from time 0 to time 5. We know that we have to buy a bicycle at time 0, but there is tie for the best length of time to keep it: either 1 or 3 years. Let's follow the 1-year ownership first: that leads us to year 1, where we buy another bicycle and again to a tie for the best length of time to keep it, again either 1 year or three years. Following the one year ownership again takes us to year 2, where we buy another bicycle, and this time keep it for 3 years until time 5. So at least one solution that gives a minimum total cost of ownership of \$640 is to buy bicycles at times 0, 1, and 2.

Following up the various ties in the solution, we find that there are 3 different solutions that give the same minimum total cost of ownership: buy at 0, 1, 2; buy at 0, 1 and 4; and buy at 0, 3, and 4. This is not surprising given that our initial cursory analysis showed that keeping a bicycle for 3 years is most economical, followed by keeping bicycles for 1 year. The 3 equivalent solutions are made of all the possible combinations of 3-year and 1-year ownership patterns within a 5-year timeframe.

This problem doesn't seem at all like a shortest route problem does it?

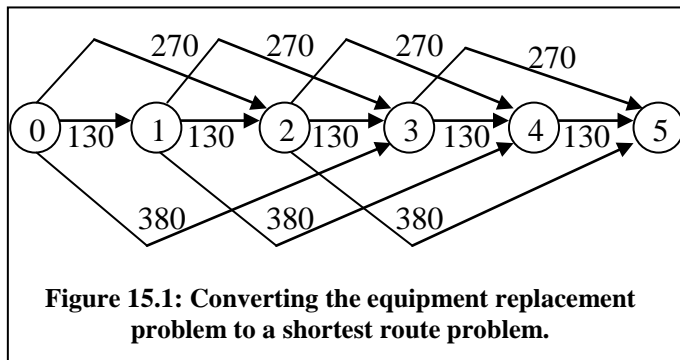


Figure 15.1: Converting the equipment replacement problem to a shortest route problem.

However it is surprisingly easy to transform an equipment replacement problem into a shortest route problem. Just construct a network in which the nodes represent points in time (time 0, time 1, etc.) and the connecting arcs are labeled with the cost of keeping a bicycle for the appropriate amount of time. The equivalent network is shown in Figure 15.1: just find the shortest route from time 0 to time 5!

In this very special case, linear programming could actually be applied to this network formulation. But not all dynamic programming problems can be turned into linear programs so easily.

A Knapsack Problem

Knapsack problems typically involve an upper limit on some sort of capacity (e.g. the weight capacity that can be carried in a knapsack), and a set of discrete items that can be chosen, each of which has a value, but also a cost in terms of the capacity (e.g. it has some weight). The goal is to find the selection of items that has the greatest total value while still respecting the limit on the capacity. The classic textbook example is a hiker filling a knapsack with items to take on a hike, each with a certain value and a given weight, but the formulation applies to many other applications such as selecting discrete cargo items for a ship or airplane, or even which weapons to load into a nuclear submarine for an extended cruise.

Let's start with our hiker and her knapsack. She is trying to determine how to choose among several items that she can pack, and how many of each item to take. She can carry up to 10 kg in

her knapsack. To keep it simple, let's consider just 3 possible items, whose weights and values are summarized in the table below.

item	weight (kg)	value
1. food pack	3	7
2. large bottle of water	4	8
3. tent	6	11

She is going to a scout meeting with her troop, so it is possible that she can take more than one of each item on behalf of the troop. How many of each item should she take to maximize the total value while not exceeding her 10 kg carrying capacity?

Before even starting, let's try to guess the solution for this tiny problem, based on the value/weight ratios for the items. These are $7/3=2.33$ for food packs, $8/4=2$ for bottles of water, and $11/6=1.83$ for tents. Hence we would prefer to take as many food packs as possible since they have the greatest value/weight ratio. It's a pretty good bet that the final solution will include some food packs, but it's hard to determine exactly what the final solution will be. We'll have to set up and solve the dynamic program to find out.

But first, note that this problem can be restated as:

$$\begin{aligned} &\text{Maximize } 7x_1 + 8x_2 + 11x_3 \\ &\text{subject to } 3x_1 + 4x_2 + 6x_3 \leq 10 \\ &\text{where } x_1, x_2, x_3 \geq 0 \text{ and integer.} \end{aligned}$$

Do you know a way to solve this problem without using dynamic programming? It can be solved as an integer program via branch and bound in this case. As we will see in the next example though, there are many dynamic programming problems that cannot be turned into more familiar forms such as linear programs (as for the equipment replacement problem) or integer programs solved by branch and bound.

Let's formulate this knapsack problem for a dynamic programming solution.

- *Stages*: The natural breakdown in this case is by item. And since we are practicing backwards recursions, we will work in the order tents, bottles of water, food packs. We will first consider just tents, then bottles of water and tents, then food packs, bottles of water and tents.
- *State at a stage*: The state at a stage is the amount of carrying capacity remaining in the knapsack.
- *Decision at a stage*: The hiker must decide how many copies to take of the item being considered at the current stage.
- *Decision update to the state*: The number of copies of the item taken reduces the carrying capacity for future stages in an obvious way.
- *Recursive value relationship*: First, let's define a few functions and variables:
 - t is the current stage (1, 2, or 3, indicating food packs, bottles of water, or tents)

- d_t is the carrying capacity remaining when you reach stage t .
- x_t is the decision, i.e. the number of copies of item t that the hiker decides to take
- v_t is the value of one copy of item t
- $f_t(d_t)$ is the maximum value obtainable when you enter stage t with a remaining carrying capacity of d_t , considering only stages $t, t+1, \dots, 3$.
- The recursive value relationship is: $f_t(d_t) = \max_{x_t} \{v_t x_t + f_{t+1}(d_t - w_t x_t)\}$ where $0 \leq x_t \leq d_t/w_t$ and integer.

Note how $f(\cdot)$ appears on both sides of the recursive relationship. The optimum for stage t to stage 3 depends on two things: the value of the current decision at stage t (i.e. $v_t x_t$), and the value of the previously found optimum $f_{t+1}(\cdot)$. Note also that $f_{t+1}(\cdot)$ is calculated with a remaining carrying capacity of $d_t - w_t x_t$ meaning that the weight of the items taken at stage t has reduced the carrying capacity d_t with which you entered stage t .

A difference from the equipment replacement problem is immediately apparent: you can enter a stage with in different states. When you are working with stage 3 by itself (the first stage to be calculated), you have no idea what the stage might be by the time stages 1 and 2 are considered. Hence for each stage we have to take into account all possible states. This means we need to use a table representation instead of the simple list we used for equipment replacement. Here goes:

We start with stage 3, the tents. You will see that in many dynamic programming problems, the first and last stages considered are simpler than the intermediate stages. That is the case here, where we are considering *only* tents at this time. So the recursive value relationship reduces to $f_3(d_3) = \max_{x_3} \{11x_3\}$ where $0 \leq x_3 \leq 1$. The hiker can take just 0 or 1 copies of the tent since it weighs 6 kg and her carrying capacity is just 10 kg.

The fully expanded table for stage 3 looks like this, where the best choice in each row is shown in bold:

d_3	$x_3 = 0$	$x_3 = 1$	$f_3(d_3)$
0	0	-	0
1	0	-	0
2	0	-	0
3	0	-	0
4	0	-	0
5	0	-	0
6	0	11	11
7	0	11	11
8	0	11	11
9	0	11	11
10	0	11	11

The first column indicates the possible remaining carrying capacity when we finally reach stage 3: it could be anything between 0 and 10. The two centre columns show the value of making the decision indicated at the top of the column: note that “-“ indicates that the particular decision is

not possible. For example, look at the $d_3=2$ line in the table. If you have only 2 kg of carrying capacity left, then you obviously can't take a 6 kg tent, hence the entry in the $x_3 = 1$ column is “-“. The rightmost column just summarizes the best possible thing to do in each row (i.e. the largest value you can attain given your remaining carrying capacity upon entry).

For stage 3, the most important breakpoints are those associated with the weight of a tent (6 kg). If you have less than 6 kg carrying capacity remaining, then you obviously can't take a tent; 6 kg or more left and you can. For this reason, the table for stage 3 can be compressed quite a bit:

d_3	$x_3 = 0$	$x_3 = 1$	$f_3(d_3)$
0-5	0	-	0
6-10	0	11	11

Stage 2 will encompass items 2 and 3, and hence will make reference to the table for stage 3. The recursive relationship is now $f_2(d_2) = \max_{x_2} \{8x_2 + f_3(d_2 - 4x_2)\}$ where $0 \leq x_2 \leq 2$ (since, at a weight of 4 kg per bottle of water, we can take at most 2 bottles). The corresponding table is:

d_2	$x_2 = 0$	$x_2 = 1$	$x_2 = 2$	$f_2(d_2)$	$d_3 = d_2 - 4x_2$
0	0	-	-	0	0
1	0	-	-	0	1
2	0	-	-	0	2
3	0	-	-	0	3
4	0	8	-	8	0
5	0	8	-	8	1
6	11	8	-	11	6
7	11	8	-	11	7
8	11	8	16	16	0
9	11	8	16	16	1
10	11	19	16	19	6

This table may be a little counterintuitive. Let's look at cell for row $d_2=10$ and column $x_2=1$, which has a value of 19. How did it get that value? $d_2=10$ means that we enter stage 2 and 3 (which is what is represented in this table) with 10 kg of carrying capacity remaining, and $x_2=1$ means that the hiker has decided to take one copy of item 2 (i.e. a bottle of water). This has a value of 8 and a weight of 4 kg, leaving 6 kg of carrying capacity for stage 3. Now enter the table for Stage 3 on the $d_3=6$ line, and we see right away that the best thing to do has a value of 11. Hence the total value is 8 (for a bottle of water) plus 11 (for a tent), for a total of 19. The other cells are calculated in the same way, following the recursive relationship.

Stage 1 is again simpler, but for an interesting reason. Now that we have expanded the problem enough to encompass the entire original problem, we know *exactly* what d_1 , the carrying capacity as we start the problem, is: it is 10 kg! Hence the table for stage 1 (which encompasses stages 2 and 3 as well), has just a single row for $d_1=10$, as shown below. The recursive relationship is also slightly simplified because d_1 is replaced by a numeric value: $f_1(10) = \max_{x_1} \{7x_1 + f_2(10 - 3x_1)\}$.

d_1	$x_1 = 0$	$x_1 = 1$	$x_1 = 2$	$x_1 = 3$	f_1	$d_2 = 10 - 3x_1$
10	19	18	22	21	22	4

The values in this last table are calculated using only the information about item 1 *and the table for stage 2*. We don't need the table for stage 3 at this point, because stage 2 already incorporates stage 3. As an example, consider the value in the $x_1=1$ column. The value of 18 is obtained (just as in the recursive relationship) from 1 copy of item 1 (value 7), which has a weight of 3 kg, leaving 7 kg carrying capacity, plus the best thing to do in the stage 2 table when there is 7 kg of carrying capacity left, and that has a value of 11, for a total of 18.

Now at this point we can see that the maximum value combination of items that the hiker can carry has a value of 22, but we still need to unravel the actual solution. To do this we trace back through the tables from stage 1 to stage 3. In stage 1, the best value comes from setting $x_1=2$, leaving a carrying capacity of 4 kg for the next stage (as summarized in the rightmost column). Entering the table for stage 2 on the line for $d_2=4$, the best value is for $x_2=1$, leaving a carrying capacity of 0 as summarized in the rightmost column. Entering the table for stage 3 on the line for $d_3=0$, the best value is for $x_3=0$. Hence the solution associated with the maximum value of 22 is to take 2 food packs (item 1), plus one large bottle of water (item 2), and no tents (item 3).

There are a few things to notice about this example. First, the number of rows in the tables can become quite large if there are many states. The number of columns can also become large if there are many possible decisions, so dynamic programming can be crippled by combinatorial explosion for large problems. Second, each cell calculation involves only the item considered at the current stage and the optimum results from just the previous table (since that table encapsulates all the data from all of the previous tables). This is helpful for efficiency. Finally, we need all of the tables to recover the actual solution, and not just its value.

A More General Example: Simultaneous Failure

A data storage company maintains banks of disk drives in three separate locations for security reasons, with probabilities of failure that depend on the locations. For additional security, the company can also provide backup disk drives in each location. There are two backup disk drives available, and the company wishes to determine where to install them to minimize the overall probability that all three locations will fail simultaneously. The estimated probabilities of failure are summarized in the following table.

		Location		
		A	B	C
backup	0	0.20	0.30	0.40
drives	1	0.10	0.20	0.25
assigned	2	0.05	0.10	0.15

For example, if no backup drives are assigned, then the overall probability of simultaneous failure of all three locations is $0.20 \times 0.30 \times 0.40 = 0.024$.

This example is still discrete and deterministic, but it is different in that the recursive relationship that will result is not additive, but multiplicative. The objective is to distribute the 2 available

backup disk drives so that the overall probability of failure is minimized. This can be formulated for a dynamic programming solution. Again there is no implied direction in this problem, but we will choose a backwards recursion.

- *Stages*: The natural breakdown in this case is by location. For a backwards recursion we will first consider location C, then locations B and C, then locations A and B and C.
- *State at a stage*: The state at a stage is the number of backup disk drives left to distribute.
- *Decision at a stage*: How many backup disk drives should be assigned to this location? The number could be anything between 0 and 2, depending on the state.
- *Decision update to the state*: The number of backup disk drives assigned to a location reduces the number of backup disk drives available for assignment.
- *Recursive value relationship*: First, let's define a few functions and variables:
 - t is the current stage (A, B, or C, indicating the location)
 - d_t is the number of backup disk drives remaining for assignment when we enter stage t .
 - x_t is the decision, i.e. the number of backup disk drives assigned to this location
 - $p_t(x_t)$ is the probability of failure at stage t given that x_t backup disk drives are assigned to this location
 - $f_t(d_t)$ is the minimum overall probability of failure for stages $t+1, \dots, 3$ obtainable when you enter stage t with d_t backup disk drives available for allocation
 - The recursive value relationship is: $f_t(d_t) = \min_{x_t} \{p_t(x_t) \times f_{t+1}(d_t - x_t)\}$ where $0 \leq x_t \leq 2$.

As usual, the first table (for location C) is relatively simple. The recursive value relationship is just $f_C(d_C) = \min_{x_C} \{p_C(x_C)\}$ because we not have to worry about stages beyond stage C. The resulting table is:

d_C	$x_C=0$	$x_C=1$	$x_C=2$	$f_C(d_C)$
0	0.4	-	-	0.4
1	0.4	0.25	-	0.25
2	0.4	0.25	0.15	0.15

The second stage (covering locations B and C) has the recursive relationship $f_B(d_B) = \min_{x_B} \{p_B(x_B) \times f_C(d_B - x_B)\}$. As you can see, the minimum function $f(\cdot)$ appears on both sides of the relationship as we expect. The stage B table is:

d_B	$x_B=0$	$x_B=1$	$x_B=2$	$f_B(d_B)$	$d_C=d_B-x_B$
0	0.12	-	-	0.12	0
1	0.075	0.080	-	0.075	1
2	0.045	0.050	0.040	0.040	0

Let's make sure we understand this table. Consider the row in which $d_B=1$ and the column in which $x_B=0$, which has a cell value of 0.075. Where does that value come from? Well, if we have one backup disk drive left to allocate (i.e. $d_B=1$) and we decide not to allocate it to location B (i.e. $x_B=0$), then the probability of failure of location B is 0.30, but we still have one backup disk drive left to allocate, so now we enter the stage C table on the $d_C=1$ line, and the best decision in that line has a value of 0.25, so our recursive relationship for stage B gives a lowest probability of failure under these conditions of $0.30 \times 0.25 = 0.075$ for stage B and C. This is the value in that cell. See if you can see why the other cells have the values they do.

Now we come to the last stage, stage A, which will encompass stages B and C as well and will then encompass the entire original problem, our signal to stop. The recursive relationship for stage A is $f_A(d_A) = \min_{x_A} \{p_A(x_A) \times f_B(d_A - x_A)\}$. The stage A table is again somewhat simpler than the others since we know the number of backup disk drives available when we start the problem: it is exactly 2. We don't have to worry about intermediate numbers as we do for stages B and C.

d_A	$x_A=0$	$x_A=1$	$x_A=2$	$f_A(d_A)$	$d_B=d_A-x_A$
2	0.0080	0.0075	0.0060	0.0060	0

Let's review again where the numbers in this table come from. Let's consider the optimum solution in the $x_A=2$ column, which has a value of 0.0060. That results from the stage A value of assigning 2 backup disk drives to stage A (i.e. 0.05) multiplied by the value of the best action given that you go forward to the next table with 0 backup disk drives left to distribute: the $d_B=0$ row in the stage B table has a best value of 0.12, so the final result is $0.05 \times 0.12 = 0.0060$. Note that we don't have to look at the stage C table at all to make this calculation. That's because the stage B table already incorporates all the information from the stage C table. This is a general property of dynamic programming: you only need to look at the current stage and one other optimum solution. There is not very much savings from this property in this tiny example, but there can be a huge savings when you have a large problem with many stages.

So at this point we know that the optimum solution has a minimum probability of failure of 0.0060, but we still need to unwind the recursion to find the actual assignment of backup drives to locations that achieves this result (pretty simple in this example). It goes like this:

- From the stage A table we see that of the 2 backup drives available, 2 should be assigned to stage A.
- So we enter the stage B table with 0 backup drives, and the optimum decision in this case is to assign 0 backup drives to stage B.
- So we enter the stage C table with 0 backup drives, and the optimum decision in this case is to assign 0 backup drives to stage C.

This example shows some of the generality of dynamic programming. The recursive relationship is multiplicative in this case, so there is no possibility of some kind of conversion to linear programming. Plus it also shows the value of redundancy in designing failsafe systems!

Final Comments on Dynamic Programming

We have dealt thus far with the simplest possible case for dynamic programming: discrete and deterministic problems in which the direction of solution really didn't matter. Let's discuss each of these cases.

First, there are problems which *must* be done by backwards recursion. For example, consider the employee scheduling problem in which we must choose the number of employees working in each month over the next year. For each month we have forecasts of the number needed to do the work. However skilled workers are hard to hire, so layoffs are to be avoided since they incur a cost and we may not get the workers back. On the other hand, excess employees are costly. The problem is to determine how many employees to hire and lay off every month to minimize overall costs. It is hugely more efficient to solve this problem working backwards in time since we know the numbers of employees needed each month. If we work forwards in time, then we have to keep track of a huge number of possible staffing levels that *might* be needed at future times instead of the small number of known staffing levels.

Continuous models (e.g. water level behind a dam) can be handled by finding meaningful break points in the continuum (e.g. empty, minimum level for water supply, minimum level for hydroelectricity generation, full). Probabilistic versions of dynamic programming are also possible; the goal is then to minimize or maximize the expected value of the recursive value relationship.

Finally it's important to understand that dynamic programming, though tedious to carry out by hand, is actually quite efficient compared to a brute force listing of all possible combinations to find the best one. For example, if you are finding your way through a graph which has 5 possible nodes to go to at each of 6 stages (and is fully connected between each stage), then there are $5^6=15625$ possible paths if they are all enumerated, and each of these requires 5 addition operations for 15,625 total operations. However an analysis of a Dijkstra's algorithm dynamic programming solution shows that it takes just 105 operations. Hence the dynamic programming solution requires just $105/15625 \approx 0.007$ of the work!