Dynamic Programming

Next we want to study a very powerful technique for designing algorithms known, somewhat incorrectly as *dynamic programming* (hearkening back to the days when "programming" was used to mean "optimization").

Optimization problems always ask for the *best* way to do something, in the sense of maximizing or minimizing some numerical score, with accompanying decisions.

This technique is so powerful and generally useful because it is actually just a way of using recursion properly. The basic idea is to use recursion to express the solution to a given instance of a problem in terms of solutions to sub-problems with the same structure, and then to store the solution to sub-problems in a table, rather than possibly recomputing them repeatedly.

Unlike the divide and conquer technique, which we have seen in use to get more efficient algorithms than the obvious ones, dynamic programming is typically used to get any sort of algorithm at all, other than brute force.

> This algorithm design technique is one of the most fruitful for a typical person to actually use to develop a new algorithm. So, you will get the chance in this chapter, and later on Test 2, for the first time in the course to invent a new algorithm for a given problem.

Computing the Binomial Coefficient

As a first example, let's think about computing the binomial coefficient.

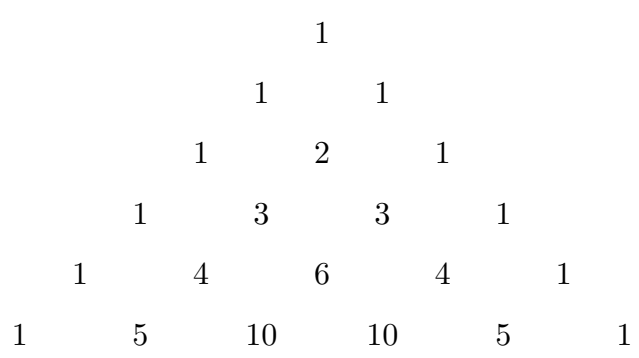
The binomial coefficient, $\binom{n}{k}$, can be defined by the formula

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!},$$

but this formula does not lead to an easy to implement, or, probably, efficient algorithm.

Instead, we observe that "Pascal's triangle" contains all the binomial coefficient values—$\binom{n}{k}$ is found in row n, column k of Pascal's triangle.

Pascal's triangle is formed by putting 1's diagonally on both edges and then adding up all pairs of numbers to give the number immediately below them:



But, if we just implement this idea by some code like this:

```
int pasTri( int n, int k ) {
  if( k==0 | k == n )
    return 1;
  else
    return pasTri( n-1, k-1 ) + pasTri( n-1, k );
}
```

it turns out to be horribly inefficient!

⇒ Get the file `Pascal.java` and run it with inputs 30 and 15. After a pause it will produce the correct result. Try it with inputs 50 and 25.

This example shows that we often need to use a chart to store all the answers to related subproblems of the original problem, or else recursion can do a vast amount of *duplicate work*.

A dynamic programming approach to the problem of computing a $\binom{n}{k}$ is simply to maintain a chart in which we store the answer for any subproblem that we compute, and then when asked to do an instance of the problem, we always look in the chart first to see if the answer is already known.

For example, if we ask for the answer to $\binom{10}{3}$, these answers will be recursively filled in:

	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4		
5	1	5	10	?		

and then the ultimate answer to $\binom{5}{3}$ can be computed as $\binom{4}{2} + \binom{4}{3} = 6 + 4 = 10$.

Often we will find it easier to fill in the chart using iteration rather than literally using recursion, but the underlying concept of any dynamic programming algorithm is recursive.

⇒ Now examine `PascalDynProg.java`. This is the original code in `Pascal.java` with some changes so that whenever a subproblem is computed, its result is stored in a chart, and when the answer to a subproblem is needed, we always look in the chart to see if it is already available.

Try this program with larger values for n and k, like 1000 and 500.

Introductory Example (the Rocks Game)

Here is a game perhaps played long ago by cave people: make two piles of smallish (so they're easy to handle) rocks. We'll call these piles "pile A" and "pile B" and say that pile A has m rocks and pile B has n rocks. Two players take turns making moves. The possible moves are:

- take one rock from pile A
- take one rock from pile B
- take one rock from pile A and one rock from pile B

The player who takes the last rock or rocks on their turn wins the game.

⇒ Let's play the game, say with 7 rocks in pile A and 8 rocks in pile B.

Now come up with an algorithm for determining which player can win when faced with two given piles (assuming they know how many rocks are in each pile). This is a perfect situation for recursion, because if we are faced with m rocks in pile A, and n rocks in pile B, then we think, if only some clever person could help us figure out the consequences of making any one of the up to three moves available to us (there might not be any rocks in one pile or the other). But, using recursion, we can be that clever person!

So, given non-negative integers m and n, we will build a chart where the cell in row j, column k represents the sub-problem:

> "what's the optimal move for a player faced with j rocks in pile A and k rocks in pile B"

where $0 \leq j <= m$ and $0 \leq k \leq n$.

Let's write in each cell information telling a player what to do when faced with the situation given by the cell, namely `A` or `B` meaning to take a rock from pile A or pile B, respectively, or `2` meaning to take a rock from both piles, or `Q` meaning to quit because you can't win.

As with all dynamic programming algorithms, we need to figure out the crucial idea: if all the relevant cells have been filled in, except for the one we are trying to fill in, how do we decide what to do—that is, what symbol to write in the current cell. If we can figure that out, and can figure out how to handle the *base cases*, we should be able to fill in the chart.

⇒ Let's draw a sketch showing how to figure out the value for row j, column k, if the necessary neighboring cells—representing smaller problems than the one we are facing—have been filled in.

⇒ Now, using these ideas, let's fill in this chart, representing a game that starts with 7 rocks in pile A and 8 rocks in pile B:

Pile B

Pile A	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									
5									
6									
7									

Now that we have done this using dynamic programming, we can actually see that the game is trivial and that we don't really need the chart at all! Note that this is not usually the case with dynamic programming—you usually have to do all the work to fill in the cells of the chart.

The 0-1 Knapsack Problem by Dynamic Programming

Now let's tackle a somewhat more challenging problem through the dynamic programming approach.

Recall (from the brute force chapter) that the 0-1 knapsack problem is the following:

> Given n items, each of which has a weight w_j and a profit p_j, and a knapsack (backpack) that can hold a weight of W, find the subset of the items that has the greatest total profit, subject to the constraint that the sum of the weights of the items in the subset is less than or equal to W.

This problem is known as the 0-1 knapsack problem to indicate that each item is either left out or put in the sack. This is the correct model if the items are indivisible, like a stove, a sleeping bag, and a knife.

> The continuous version of the problem allows us to use any desired portion of each item, which is appropriate if the item is something like 8 pounds of sugar, where we can break open the sack and use any amount of the sugar. This problem is actually trivial to solve, and we will use it toward the end of this course as a way to provide a *bound* for the 0-1 knapsack branch-and-bound algorithm.

An Example of the 0-1 Knapsack Problem

Here is an instance of the 0-1 knapsack problem, where the items are sorted in the order of profit per weight (which doesn't matter for the dynamic programming approach, but is useful for other approaches, so we'll get in the habit), and $W = 12$:

Item j	p_j	w_j
1	100	4
2	120	5
3	88	4
4	80	4
5	54	3
6	80	5

Earlier we saw how we could solve this problem using the brute force approach, so we won't repeat that here.

The *greedy approach* (this design technique will be studied in a later chapter) for this problem would be to do what we have done—sort the items into order of "value per weight" and use items in that order, as capacity allows.

For example, for this instance we would use items 1 and 2, for a weight of 9, skip items 3 and 4 because they would exceed the capacity, and then use item 5, for a final weight of 12 and profit of $100 + 120 + 54 = 274$. This turns out to be optimal.

⇒ However, the greedy approach does not always give the correct answer. Apply the greedy approach to the instance given below and compare to the optimal result produced by the brute force approach:

$W = 10$

Item j	p_j	w_j
1	36	6
2	25	5
3	20	5

A Dynamic Programming Approach to the 0-1 Knapsack Problem

What is a sub-problem of a given 0-1 knapscack instance? Well, we might be given any subset of the items, and any knapsack capacity less than or equal to the given capacity.

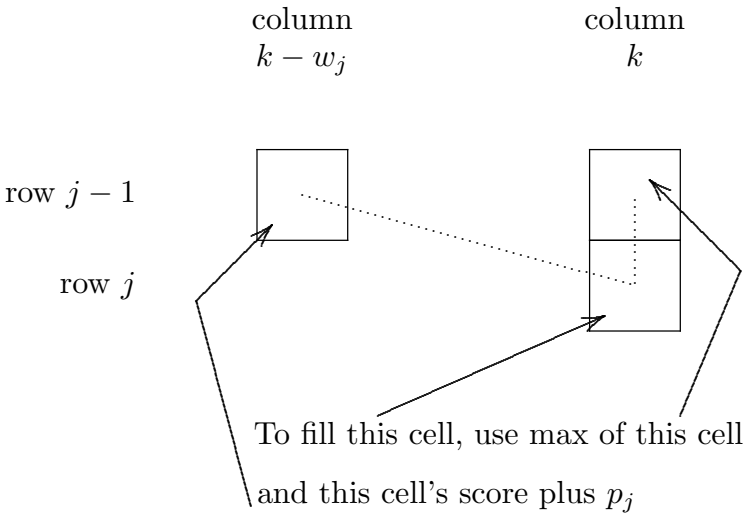
After some deep thought, we figure out that we don't need to do sub-problems with every subset of items, because the problem inherently allows not using items that are allowed, so we realize that we can determine our sub-problem as "if you are allowed to use any of items 1 through j, with some allowed capacity k, what is the optimal choice of items?"

Then we can organize a chart including cells for all possible sub-problems of this type by letting the cell in row j, column k correspond to the previously stated sub-problem.

With dynamic programming each cell has a crucial value, namely the optimal score for the sub-problem represented by the cell. As is almost always true for dynamic programming, each cell of the chart should not only hold the optimal score, but should also hold information telling us how to achieve that score. In this case, we will also put in the cell a list of items that achieves that optimal profit.

With this approach, when we are trying to fill cell $P(j, k)$, there are only two choices—we either use item j, or we don't. If we don't use item j, the optimal profit for weight k will be given by the cell above—$P(j-1, k)$. If we do use item j, the optimal profit will be in cell $P(j-1, k-w_j)$. Thus, the actual best thing to do in cell $P(j, k)$ is either to not use item j and use the items listed in cell $P(j-1, k)$, or to use item j along with the items listed in cell $P(j-1, k-w_j)$.

Here is a picture of this key recursive idea for 0-1 knapsack (for all the problems we study, you should have a similar picture):



Example of 0-1 Knapsack by Dynamic Programming

⇒ Let's demonstrate the dynamic programming algorithm for the 0-1 knapsack problem on this instance:

$W = 10$

Item j	p_j	w_j
1	60	3
2	72	4
3	80	5
4	90	6
5	48	4

We could fill in the entire chart in a sensible order, which is easiest to program, but to save wasteful effort, let's work "on demand," working back recursively from the lower-right corner cell and marking all the cells that will be needed and only filling them in.

		0	1	2	3	4	5	6	7	8	9	10
$p_1 = 60$ $w_1 = 3$	1:											
$p_2 = 72$ $w_2 = 4$	2:											
$p_3 = 80$ $w_3 = 5$	3:											
$p_4 = 90$ $w_4 = 6$	4:											
$p_5 = 48$ $w_5 = 4$	5:											

⇒ Let's also write out the cells that are filled in without using the chart, just to show how that can be done (mostly for Exercise 18).

Efficiency Analysis for the 0-1 Knapsack Algorithm

Analyzing the efficiency of this algorithm is a little confusing, because we have to consider carefully what we mean by the input size, and figure out how many bits are required to specify each of the n p_i and w_i values, as well as the W value.

It is obvious that for n items and a weight of W, this dynamic programming algorithm takes $\Theta(nW)$, because that is how many cells there are in the chart, and it takes just a few operations to compute each cell, given the cells from the row above. This looks like a trivial problem, but it is not, because for an input size N, W is like 2^N, while n is like N (assuming for simplicity that the w_i and p_i values take a fixed number of bits each), so the efficiency category is $\Theta(N2^N)$ which is viewed as computationally intensive.

For the "on demand" version of the algorithm, the cells that need to be filled can form a binary tree with n levels, resulting in efficiency in $\Theta(2^n)$, though we haven't proven that, or related it to N.

Exercise 18 [5 points] (target due date: end of Week 9)

Your job on this exercise is to demonstrate dynamic programming to solve the instance of the 0-1 knapsack problem given below, working by hand and writing out all the details. Here is the data for the instance of the 0-1 knapsack problem:

Item j	p_j	w_j
1	22	6
2	50	19
3	90	37
4	46	21
5	85	46
6	66	39

where the capacity of the knapsack is 154.

You must do the "on demand" version of the algorithm, as demonstrated in the video for the previous example.

Your cell format should be:

`(row, column): <optimal profit> <list of items that achieve that profit>`

You will want to start with the $(6, 154)$ cell, leaving empty space to fill in the optimal profit and items used. Then somewhere below write the two cells that need to be filled in in order to compute that final cell, and so on.

Note that the cells you end up needing in row 1 are trivial to compute—these are the base cases—each will either use item 1 for a profit of 22, or will not, for a profit of 0.

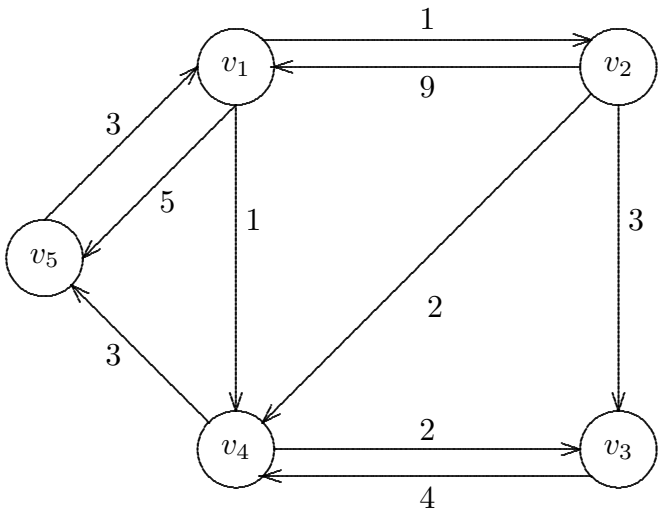
Note that some cells will not be able to do the "use item j" option because $k - w_j$ is negative.

Email me as usual with attached pictures of all the pages you write your cells on.

Floyd's Algorithm for the "All Shortest Paths in a Graph" Problem

Consider a directed graph consisting of vertices v_1 through v_n, any two of which are connected with an edge of some weight (or cost or length or other quantity, depending on the application). Some pairs of vertices don't have an edge between them, which we can model by saying that the weight of the edge is ∞).

Here is an example graph:



Here is how the graph can be implemented using a 2D array (indexed starting at 1, note):

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

The All Shortest Paths In a Graph Problem

The problem we want to solve is "given a directed graph, find the length (total weight) of the shortest path between every pair of distinct vertices."

The Recursive Idea for Floyd's Algorithm

Floyd's algorithm is based on this idea: define $D^k(i, j)$ to be the length of the shortest path from i to j using, as appropriate, intermediate vertices from v_1 up through v_k.

⇒ As with all recursion, we need the base case. For Floyd's algorithm, the base case is $D^0(i, j) = W(i, j)$, where $W(i, j)$ is the weight of the edge from i to j. Verify that this makes sense with our notation.

Next we need to figure out how to compute an arbitrary $D^k(i, j)$, for $k > 0$, in terms of already solved—and stored in a table—subproblems.

⇒ Figure out how to compute $D^k(i, j)$ from already figured out values of D^{k-1} by this idea:

> The best path from i to j, when allowed to use vertices v_1 through v_k either uses v_k or it doesn't. If the best path doesn't use v_k, then its length is simply $D^{k-1}(i, j)$. If it does use v_k, it can be broken into some best path from v_i to v_k using, as desired, vertices from v_1 through v_{k-1}, followed by the best path from v_k to v_j using, as needed, any of the vertices from v_1 through v_{k-1}.

⇒ Finally, note that $D^n(i, j)$ is the best path from i to j, and note that we can first fill in a table for D^1, then use it to fill in D^2, and so on, until we fill in D^n and are done.

Floyd's Algorithm Example

⇒ Let's demonstrate Floyd's algorithm on the previous example. We will use this large matrix so we have room to repeatedly update, so that at each stage the matrix contains $D^{(0)}$, then $D^{(1)}$, then $D^{(2)}$, and so on, finishing at $D^{(5)}$.

In each cell in row i, column j, at step k, we will write, in the upper left corner, the cost of the optimal path from vertex i to vertex j if we are allowed to use any of vertices 1 through k as intermediate vertices.

In the lower right corner of each cell we will write the intermediate vertex that was most recently used (0 if the cell uses a direct edge from i to j).

	1	2	3	4	5
1	0 / 0	1 / 0	∞ / 0	1 / 0	5 / 0
2	9 / 0	0 / 0	3 / 0	2 / 0	∞ / 0
3	∞ / 0	∞ / 0	0 / 0	4 / 0	∞ / 0
4	∞ / 0	∞ / 0	2 / 0	0 / 0	3 / 0
5	3 / 0	∞ / 0	∞ / 0	∞ / 0	0 / 0

⇒ Let's figure out how we can recursively find the shortest path from any desired starting vertex to any desired ending vertex from the final $D^{(5)}$ chart.

In case the previous example was unclear as I presented it in the video, here are the step-by-step charts (generated by the program `Code/DynProg/Floyd/Floyd.java`, which has the instance data hard-coded and produces a text file named `plainCharts` that is

most readable unless you know how to compile and view TEX documents). These charts highlight the crucial row k and column k for each step.

$D^{(0)}$

	1	2	3	4	5
1	0 0	1 0	∞ 0	1 0	5 0
2	9 0	0 0	3 0	2 0	∞ 0
3	∞ 0	∞ 0	0 0	4 0	∞ 0
4	∞ 0	∞ 0	2 0	0 0	3 0
5	3 0	∞ 0	∞ 0	∞ 0	0 0

$D^{(1)}$

	1	2	3	4	5
1	0 0	1 0	∞ 0	1 0	5 0
2	9 0	0 0	3 0	2 0	**14** 1
3	∞ 0	∞ 0	0 0	4 0	∞ 0
4	∞ 0	∞ 0	2 0	0 0	3 0
5	3 0	**4** 1	∞ 0	**4** 1	0 0

$D^{(2)}$

	1	2	3	4	5
1	0 0	1 0	**4** 2	1 0	5 0
2	9 0	0 0	3 0	2 0	14 1
3	∞ 0	∞ 0	0 0	4 0	∞ 0
4	∞ 0	∞ 0	2 0	0 0	3 0
5	3 0	4 1	**7** 2	4 1	0 0

$D^{(3)}$

	1	2	3	4	5
1	0 0	1 0	4 2	1 0	5 0
2	9 0	0 0	3 0	2 0	14 1
3	∞ 0	∞ 0	0 0	4 0	∞ 0
4	∞ 0	∞ 0	2 0	0 0	3 0
5	3 0	4 1	7 2	4 1	0 0

$D^{(4)}$

	1	2	3	4	5
1	0 0	1 0	**3** 4	1 0	**4** 4
2	9 0	0 0	3 0	2 0	**5** 4
3	∞ 0	∞ 0	0 0	4 0	**7** 4
4	∞ 0	∞ 0	2 0	0 0	3 0
5	3 0	4 1	**6** 4	4 1	0 0

$D^{(5)}$

	1	2	3	4	5
1	0 0	1 0	3 4	1 0	4 4
2	**8** 5	0 0	3 0	2 0	5 4
3	**10** 5	**11** 5	0 0	4 0	7 4
4	**6** 5	**7** 5	2 0	0 0	3 0
5	3 0	4 1	6 4	4 1	0 0

Exercise 19 [5 points] (target due date: end of Week 9)

Demonstrate Floyd's algorithm on the graph given below as a 2D array.

Note that you should draw one 2D array , which will picture $D^{(0)}$ and then successively change the values (cross out old values and write in new ones) so that the array holds the values for $D^{(1)}$, $D^{(2)}$, and so on. This will save a lot of boring copying work.

Also, note that you can and probably should run the `Floyd.java` program to check your work.

Finally, make sure that you can use your final chart to recursively determine the optimal path from each vertex to any other vertex (do a few of these to make sure). You should also draw a sketch of the original directed graph and verify that your optimal paths appear correct.

	1	2	3	4	5
1	0 0	10 0	∞ 0	∞ 0	∞ 0
2	1 0	0 0	2 0	4 0	∞ 0
3	2 0	∞ 0	0 0	3 0	6 0
4	2 0	∞ 0	3 0	0 0	4 0
5	∞ 0	1 0	∞ 0	11 0	0 0

Exercise 20 [4 points] (target due date: end of Week 9)

Now that we have explored using dynamic programming, it is time for you to practice designing an algorithm on your own. This exercise gives you the opportunity to practice that.

Consider the following simple game: given a list of positive integers v_1 through v_n, such as

$$3, 1, 7, 5, 8, 4$$

two players take turns picking either the first or the last value in the list and removing that value from the list. The goal for each player is to pick items whose values add up to the maximum possible.

Your job on this project is to design and implement, using the dynamic programming technique as detailed in the following, an algorithm to solve this problem.

⇒ Let's begin by playing the game a few times, with the given example and with new instances we make up.

Here is the idea on which you **must** base your algorithm: build a 2D chart where row j, column k holds the optimal score a player who is faced with v_j through v_k can achieve (assuming that both players play optimally, following the chart—if one player makes a sub-optimal move, then the other can move in the chart to the game they now face and proceed from there).

In addition to storing the optimal score in each cell, you must also put in either F or L, telling the player which move to make—take the First item or the Last item—when faced with this game.

> You will be asked to redo this exercise if you do not exactly follow the definition of the chart just given. The point is not to solve the problem, but rather to design the solution using dynamic programming in a specific way.

Note that some cells of this chart will not be filled in because they do not represent a valid game situation.

Your main job is to figure out precisely how you can fill the cell at (j, k) assuming that all the cells for sub-problems have been filled.

A secondary job is to figure out which cells represent bases cases, and how to fill them "from scratch."

When you understand how to fill in the chart by hand, you must implement that process with a simple Java program that takes as input the number of items, followed by the items in order, and displays on screen a nicely formatted picture of the chart, showing the optimal score achievable in each cell and F or L telling the player faced with that subproblem whether to choose the first or last item available.

Email me with all your source code files (there may well only be one) attached. The main class should be named `Ex20`.