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.

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:

$$\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}$$

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 is 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.

> Now examine `PascalDynProg.java`. This is the original code 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 200 and 100.

A simpler dynamic programming approach to this problem is simply to fill in the chart iteratively up to the cell that we want to know:

	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	1	
5	1	5	10	10	5	1

We will see this in our work with dynamic programming: usually we will fill in the chart containing answers to all the subproblems, rather than literally doing recursion. Still, the algorithms will always be based on the key recursive idea.

⇒ Project 21 [routine] Change-Making

When a cashier (human or machine) makes change, usually they want to use the smallest number of coins to make up the given amount. Inspired by this, we define the change-making problem to be:

> given integer denominations $1 = d_1 < d_2 < \ldots < d_n$ of coins in some culture, and an amount A, find the smallest number of coins of these denominations (of course allowing any number of any denomination) that add up to A.

A sample instance of this problem is

> "if $d_1 = 1$ (as always or we might not even have a way to make change for some amounts), $d_2 = 5$, $d_6 = 6$, find the smallest way to make the amount $A = 16$."

→ Try out solving this instance by using the brute force technique—draw a possibilities tree with each node showing a list of coins used and the total of those coins. Require the coins in the list to be in non-increasing order, like $5, 2, 2, 1, 1, 1$. Prune nodes as cleverly as you can (note that if you have found a way to make the target amount using, say, 5 coins, then you can prune all nodes that are going to take more than 5 coins), and circle nodes that are the solutions.

> This is a good time to introduce the *greedy approach*—an algorithm design technique that occasionally works based on "make the next decision to be locally optimal." For the change-making problem, the greedy approach is to reason "we want to use as few coins as possible, so whenever we have to decide what denomination of coin to choose next, we should always choose the largest denomination we can."
>
> It turns out that for standard coins, say in the U.S. (and probably in any culture), the greedy approach works!

→ Apply the greedy approach to the previous instance of the change-making problem (with coins 1, 5, and 6, and amount 16). Does this proposed algorithm give the correct answer?

→ How about with amount 10?

Now let's use the dynamic programming design technique to invent an algorithm for the change-making problem that is more efficient than brute force, but, unlike the greedy approach, correct!

The idea is to assume that you have a chart giving an optimal (smallest number of coins) way to create every amount smaller than A. Specifically, suppose that the cell with index j contains the optimal number of coins to produce the amount j, and has a list of coins (in increasing order, say) that achieves that amount.

With that comforting chart in hand, your only decision—to figure out how to fill in cell A—is "which of the possible coins should I choose, to be followed by an optimal list of coins from the chart for the corresponding sub-problem."

→ So, imagine that you have the chart for N, with every cell filled except for $N[16]$. Realizing that your choices are to choose 1, 5, or 6, figure out *which cells* of the chart you need to use, and say exactly how you would use the contents of those cells to determine what should go in cell 16.

N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
																	?

→ Formalize your algorithm (for these three denominations) by writing down a formula suitable for putting in code:

$N[j] =$

where on the right-hand side you can use any values of $N[k]$ for $k < j$.

For notational convenience, assume that $N[k] = \infty$ for any k that does not actually exist, which is $k < 0$ for this chart.

→ Apply this formula to iteratively fill in the entire chart (up to 16). Note that as usual with recursion, you will need to solve the base case problem(s) from scratch, since they can't be solved by using recursive calls to simpler problems. Put in each cell the optimal number of coins at the top, followed by a vertical (just so it fits) list of that many coins that produce the amount for the cell.

N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

⇒ Project 22 [routine] The Rocks Game

This Project will give you the opportunity to apply the dynamic programming design technique to the problem of playing a simple game, known as 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.

→ Play the game a few times to get a feel for it. We'll use Lego bricks instead of rocks, but the game is the same. Use 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 as follows.

Given non-negative integers m and n, 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 $0 \leq k \leq n$.

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

→ To start, fill in this chart, representing a game that starts with 7 rocks in pile A and 8 rocks in pile B:

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

Once you have figured out the key ideas, you can actually see that the game is trivial and you 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

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

Recall that the 0-1 knapsack problem is the following:

> Given n items, each of which has a weight w_i and a profit p_i, 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.

Example 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 dynamic programming, 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 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

We could approach the 0-1 knapsack problem similarly to change-making: we could create a one-dimensional chart, say P, where the cell at index j would hold the optimal profit for weight j, and we could figure out how to fill cell j by realizing that cell j can use a previously filled cell with another item added in. Thus, for each item k, we would look back at the cell $P(j - w_k)$. We would first have to see whether the list of items in that cell already contained item j, and if it did, we would reject it, because we only have one of each item. If it does not contain item j, then we would use the profitability in cell $P(j - w_k)$ plus p_j as our candidate profit. Doing this for all the items, we would find the winning candidate.

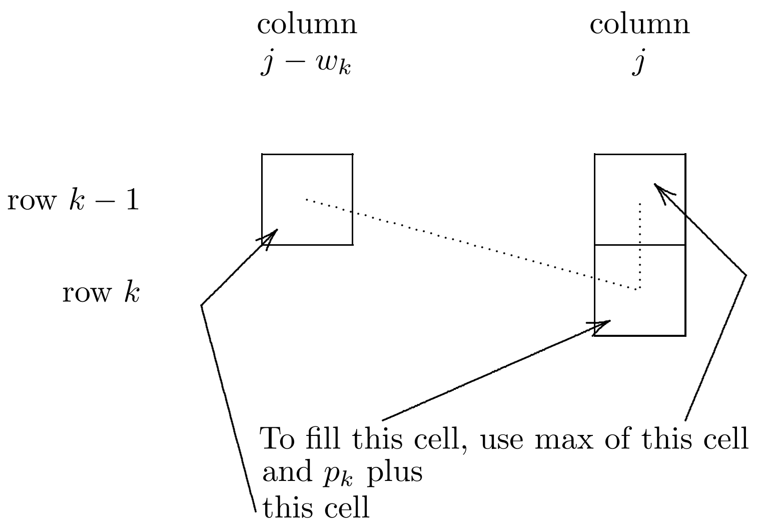
This algorithm is fine, but we will use a slightly different one that is somewhat easier to follow. Instead of worrying about all n items at each cell, we structure the sub-problems by letting row k be the row of the 2D chart when we are allowed to choose item k. WIth this approach,

> the cell in row k, column j holds the best profit achievable using items 1 through k, adding up to a weight of at most j.

And, as usual we put in the cell not only the optimal profit, but also a convenient list of items that achieves that profit.

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

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):



⇒ Project 23 [routine] 0-1 Knapsack by Dynamic Programming

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

You can fill in the entire chart in a sensible order, or work "on demand," working back recursively from the lower-right corner cell and marking all the cells that will be needed and only filling in them.

	0	1	2	3	4	5	6	7	8	9	10
1:											
2:											
3:											
4:											
5:											

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.

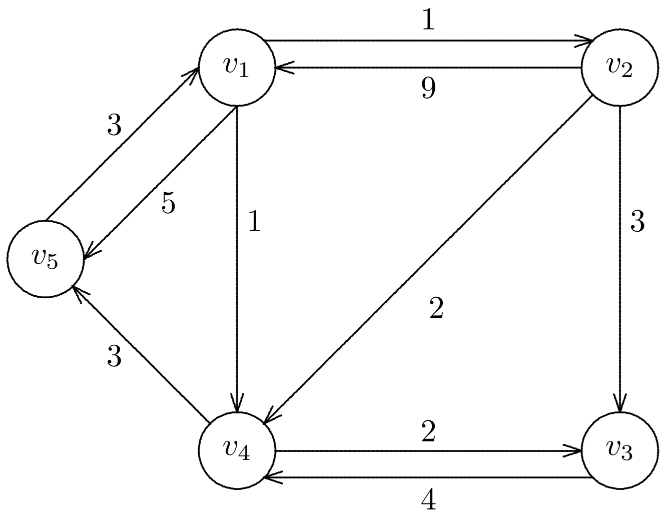
It is obvious that for n items and a weight of W, this dynamic programming algorithm takes $\Theta(nW)$, because that is how many cells there are in the chart, and it takes just a few steps to compute each cell, given the cells from the row above. This looks like a trivial problem, but it is not, because 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.

Floyd's Algorithm for the "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 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 any 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 desired 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

Here are charts for our previous example, generated by a program similar to what you're supposed to do for the next Project:

$D^{(0)}$

	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

⇒ Note that when the value in a cell is improved by using an allowed intermediate vertex, we should write down that vertex number in the cell along with the length of the optimal path. This allows the user to construct any desired optimal path from the final chart.

⇒ Project 24 [routine] Implementing Floyd's Algorithm

⇒ Make sure that you can perform Floyd's algorithm by hand—such a problem will almost certainly be on Test 2.

Then, implement Floyd's algorithm in a Java application exactly following our ideas (please, of course, do not download source code for Floyd's algorithm and turn it in as your own work—that would officially be cheating!). If your work is unnecessarily complicated or does not use precisely the input format specified, or does not produce exactly the output requested, I will not accept it.

Test your application on some sample directed graphs.

Your program must get from the keyboard the defining information for a directed graph, in precisely this order: number of vertices, followed by the weights, row by row and column by column, where -1 is used to indicate that there is no edge between the two vertices. You must use 0 for the diagonal weights (leaving out this meaningless weightt is a bad idea because it messes up the formatting).

I will actually use a data file for my test cases and redirect input. Here is the correct data file for the example:

```
5
0 1 -1 1 5
9 0 3 2 -1
-1 -1 0 4 -1
-1 -1 2 0 3
3 -1 -1 -1 0
```

Then the program must display, preferably nicely formatted, the successive tables D^0, D^1, …, D^n, with ∞ indicated as a dash, and each cell containing both the optimal length and the intermediate vertex, if any, that produces it (if the optimal length is just an edge, use 0 as the intermediate vertex).

When you are satisfied with your work, email me with one file as an attachment, with all group members CC'd as usual.

Your main class should be named `Floyd`. Be careful of going overboard on unnecessarily complex object-orientation—this assignment can trivially be done using one class, or at most two.

If your entire application consists of `Floyd.java` send me that. Or, if your application uses more than one class, attach them all. If you use an IDE, be sure to remove any non-portable extra stuff.

⇒ Project 25 [routine] Designing an Algorithm by Dynamic Programming

Now that you have explored using dynamic programming to solve some relatively simple problems, it is time for you to practice designing the algorithm on your own. This project gives you the opportunity to do 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, using the dynamic programming technique as detailed in the following, an algorithm to solve this problem.

⇒ Begin by playing the game a few times, with the given example and with new instances you 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 project 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 program 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 are ready to have your work checked, I will give you a new instance of the problem for which you will be asked to create the chart.

Remember, this is an exercise in applying the dynamic programming technique given the key recursive idea, so I will not accept work that does not follow the suggested approach!

The Traveling Salesperson Problem—Approached by Dynamic Programming

Now we will study a famous problem, the traveling salesperson problem (TSP), known in the sexist days when it was first defined as the "traveling salesman problem," and tackle it by using the dynamic programming algorithm design technique.

Here is the problem. Given a directed, weighted graph, find the total weight of the shortest/cheapest/best (actually, if we call the cost of going along an edge the "weight," the correct English term would be "lightest," but nobody says that) path that starts at v_1 and visits each other vertex exactly once, ending up back at v_1.

> The choice of v_1 is arbitrary—any of the vertices could be viewed as the starting point. For example, the path $v_1 \to v_3 \to v_5 \to v_2 \to v_4 \to v_1$ has the same cost as $v_5 \to v_2 \to v_4 \to v_1 \to v_3 \to v_5$.

Such a path is known as a "Hamiltonian circuit" of the graph, or a "tour." The TSP is to find the best Hamiltonian circuit.

Earlier you solved this problem by brute force. This brute force approach had efficiency in $\Theta((n-1)!)$. The dynamic programming approach will have efficiency in $n^2 2^n$, which is still horrible, but way better than $(n-1)!$.

⇒ Run the program `BruteForceTSP` to see the brute force method in action. Click the mouse at some points to specify a *Euclidean Traveling Salesman Problem* instance, where the weight of the edge between two points is the distance between them, and then press `s` to solve, followed by the space bar to see each permutation, or `o` to automatically find the optimal solution.

Now we want to develop a dynamic programming approach to this problem, based on this chart: let $D[A][j]$ be the length of a shortest path from v_j to v_1 that passes through each vertex in A exactly once, where A is a subset of $\{v_2, \ldots, v_n\} - \{v_j\}$.

Given this definition of D, we need to figure out the answers to all the standard dynamic programming questions to create an algorithm.

First we note that if we have all the cells of this chart filled in, then we can solve TSP by considering each $j > 1$, and computing $w_{1j} + D[\{v_2, v_3, \ldots, v_n\} - \{v_j\}][v_j]$. This quantity is the length of the best path that goes from v_1 to v_j and then passes through all the other vertices exactly once before going back to v_1. So, we can just take the best of these quantities to find the best tour.

The base cases in the chart are obviously the cells for subsets $A = \{v_k\}$ with the best value for such cells being $w_{jk} + w_{k1}$, because the only choice is to go from v_j to v_k and then go from v_k to v_1.

Now we need to figure out the crucial recursive relationship that will allow us to fill a cell assuming that related simpler cells have been filled. To do this, we just need to consider all the possible first edges from v_j to a vertex in A. For each $v_k \in A$, $w_{jk} + D[A - \{v_k\}][j]$ is the cost of the best path that goes from v_j to v_k and then passes through each other vertex in A before going to v_1. So, we simply take the best of these values as the value of $D[A][j]$. Note that this idea is really the same as the idea behind finding the best tour once we have filled the chart.

The efficiency category for this algorithm is $\Theta(n^2 2^n)$ because the chart will have a row for each subset of the $n-1$ vertices other than v_1, will have $n-1$ columns, and computing the value of a cell will take, at worst, $O(n)$ additions.

⇒ Project 26 [routine] Solving TSP by Dynamic Programming

Consider this instance of TSP:

	1	2	3	4	5
1	0	8	3	4	7
2	9	0	2	5	10
3	3	11	0	4	9
4	7	8	12	0	5
5	12	9	6	7	0

Create the dynamic programming chart for this instance, as described previously, and use it to solve this instance.

The entire algorithm is based on the definition of $D[A][j]$, which is the cost of the best path that starts at vertex j, passes through each vertex in A exactly once, and ends up at vertex 1.

For example, let's work through computing $D[\{2,3,5\}][4]$:

We have three choices for where to go first from vertex 4, namely 2, 3, or 5. For each of these choices, we can find the cost, almost, from cells in the chart that have already been filled in. For example, the best path if we go from 4 to 2 costs

$$w_{41} + D[\{3,5\}][2],$$

because $D[\{3,5\}][2]$ is the cost of the best way to start at 2, pass through 3 and 5 exactly once, and then go to 1. We can do the same calculation for first going to 3, or first going to 5, and pick the best of these three paths to give the value for $D[\{2,3,5\}][4]$.

If you get frustrated, you can use `DynProgTSP` to help—but be sure you can do these chart values by hand.

⇒ Project 27 [optional] The Sequence Alignment Problem

In this project you will be asked to develop a dynamic programming approach to a new problem, given the key observations.

Note that this Project will require you to create a Java application that solves the sequence alignment problem. All of the following material is suggested pencil-and-paper work to do to prepare yourself to create the program.

Consider the problem of aligning two strings so that they match as well as possible. This problem arises in genetics, where a sequence of molecules named A, C, G, and T form a piece of DNA. As time goes on and various mutations happen, causing symbols to change, to be removed, or to be inserted, two formerly identical sequences in the population of a species become different, but not all that different.

> Apparently (according to Dr. Liu of our Biology Department) the model developed here is actually used by biologists.

As an example instance of the problem, suppose we have the sequences `AACAGTTACC` and `TAAGGTCA` and we want to align them so that as many symbols as possible match up. We could simply shift them against each other and try to find the shift so that as many symbols as possible match up, but the biology suggests that instead we should allow ourselves to insert any number of *gaps*, denoted by `-`, in each sequence, and try in this way to get lots of pairs of symbols to match up. We need a score for any given pair of sequences, with gaps inserted, so we can decide which is better, so we say that if two actual symbols don't match, we are penalized 1 point, and if an actual symbol is matched with a gap, then we are penalized 2 points.

For example, if we insert some gaps in our sequences, like so:

```
- A A C A G T T A C C
T A A - G G T - - C A
```

we would be penalized 8 points for the 4 actual symbols that are paired up with gaps, and 2 points for the 2 pairs of actual symbols that don't match, for a score of 10.

Or, if we inserted gaps like this

```
A A C A G T T A C C
T A A G G T - - C A
```

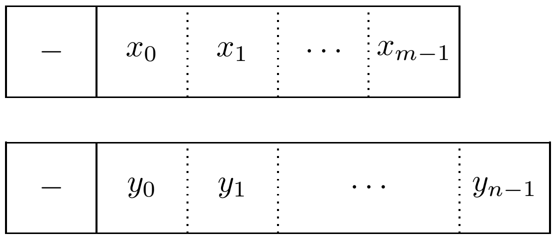
we would have a penalty of 4 for the two actual symbols that are paired with gaps, and a penalty of 4 for the four pairs of mismatched actual symbols, for a score of 8, so this alignment beats the first one.

We can now state the sequence alignment problem: given two strings of actual symbols, find a way to insert gaps that produces the smallest score.

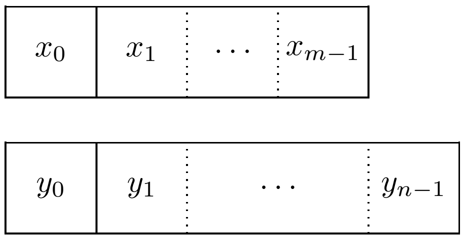
Now we will state the key recursive connection between the original problem and subproblems with the same structure, and you will be asked to figure out the algorithm from this idea.

Consider strings $x_0x_1x_2\ldots x_{m-1}$ and $y_0y_1y_2\ldots y_{n-1}$ for positive integers m and n. The key recursive idea is simply to look at all possible decisions for the first pair of symbols, and then recursively figure out the optimal way to arrange the rest.

First, we might put gaps in both strings in the first position. This would be silly, making no progress—leaving us with the same problem we started with as the sub-problem to be solved:

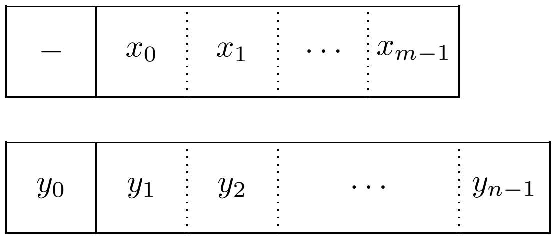


Or, we might chose to pair up x_0 and y_0:



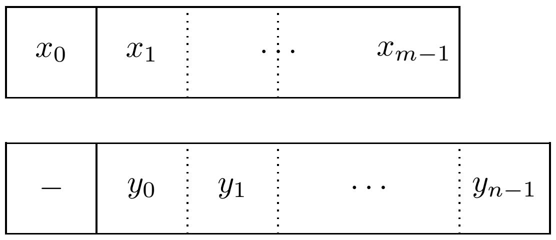
This would produce a penalty of 0 or 1, depending on whether $x_0 = y_0$. Then, we would recursively decide how to best insert gaps in $x_1x_2\ldots x_{m-1}$ and $y_1y_2\ldots y_{n-1}$, and add whatever best penalty is obtained for that subproblem to 0 or 1.

Or, we could chose to insert a gap in the first string, matching it with y_0:



for a penalty of 2, leaving us to recursively solve the sub-problem of optimally inserting gaps in $x_0x_1\ldots x_{m-1}$ and $y_1y_2\ldots y_{n-1}$.

Similarly, we could chose to insert a gap in the second string, matching it with x_0:



for a penalty of 2, and recursively solve the sub-problem of optimally inserting gaps in $x_1x_2\ldots x_{m-1}$ and $y_0y_1\ldots y_{n-1}$.

Now, we could simply implement this algorithm using recursion directly, but as usual, this would lead to sub-problems being solved repeatedly and a high degree of inefficiency, so as usual we look to build a table holding solutions to all the appropriate sub-problems. To do this, we simply note that the sub-problems we are talking about above, and the original problem, all have the same form, namely "find the optimal way to insert gaps in $x_i\ldots x_{m-1}$ and $y_j\ldots y_{n-1}$, where $0 \le i < m$ and $0 \le j < n$.

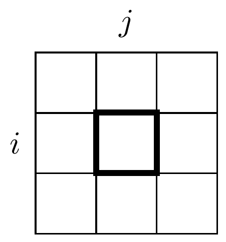
So, we create a grid of cells, one for each pair of values i and j, like so, for the given problem:

	T	A	A	G	G	T	C	A	-
A									
A									
C									
A									
G									
T									
T									
A									
C									
C									
-									

where we have added an extra gap at the end of each string to provide a convenient way to put the base cases in the chart. For consistency we say that matching two gaps has a penalty of 0.

Now you need to figure out how to compute the value in each cell in terms of already filled-in cells.

→ Here is a diagram showing the cell to be filled in, namely $A(i, j)$ ("A" for alignment?), and some neighboring cells. Figure out a precise description (formula, algorithm, or whatever) for how to compute $A(i, j)$ using the values in relevant neighboring cells, which as always with dynamic programming represent simpler sub-problems than $A(i, j)$.



→ Now, use your description to fill in all the cells in the chart below. Note that you will need to fill in base case cells "from scratch," that is, not using the same number of cells as the general case. In addition to writing the score in each cell, you must draw an arrow or otherwise indicate which of the three possible decisions is made to produce the optimal score for the cell.

	T	A	A	G	G	T	C	A	-
A									
A									
C									
A									
G									
T									
T									
A									
C									
C									
-									

→ Once the chart is filled in, write down the optimal way of aligning the two sequences, and verify that the penalty is correct.

→ Demonstrate (draw your own tidy chart) your dynamic programming algorithm for the instance of the sequence alignment problem with the sequences `GACATATTAC` and `AACGTAGAC`. Fill in the chart and clearly state the optimal insertion of gaps that gives the optimal score.

All of the preceding work was preparation for the actual work of this Project, which is to create a Java application that will solve the sequence alignment problem.

Specifically, create a Java application that will ask the user to enter two strings, will generate and fill in the dynamic programming chart and display it, and will then display in a nice way exactly how the gaps should be optimally inserted and report the optimal score.

Submit your work by email as usual, with all your Java source files attached.