# Dynamic Programming

In [mathematics](http://en.wikipedia.org/wiki/Mathematics) and [computer science](http://en.wikipedia.org/wiki/Computer_science), **dynamic programming** is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of [overlapping subproblems](http://en.wikipedia.org/wiki/Overlapping_subproblem) which are only slightly smaller and [optimal substructure](http://en.wikipedia.org/wiki/Optimal_substructure) (described below). When applicable, the method takes far less time than naive methods.

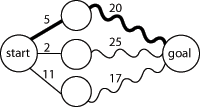
The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often, many of these subproblems are really the same. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations: once the solution to a given subproblem has been computed, it is stored or "[memo-ized](http://en.wikipedia.org/wiki/Memoization)": the next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating subproblems [grows exponentially](http://en.wikipedia.org/wiki/Exponential_growth) as a function of the size of the input.

Top-down dynamic programming simply means storing the results of certain calculations, which are later used again since the completed calculation is a sub-problem of a larger calculation. Bottom-up dynamic programming involves formulating a complex calculation as a [recursive](http://en.wikipedia.org/wiki/Recursion) series of simpler calculations.

## History

The term *dynamic programming* was originally used in the 1940s by [Richard Bellman](http://en.wikipedia.org/wiki/Richard_Bellman) to describe the process of solving problems where one needs to find the best decisions one after another. By 1953, he refined this to the modern meaning, referring specifically to nesting smaller decision problems inside larger decisions, and the field was thereafter recognized by the[IEEE](http://en.wikipedia.org/wiki/IEEE) as a [systems analysis](http://en.wikipedia.org/wiki/Systems_analysis) and [engineering](http://en.wikipedia.org/wiki/Engineering) topic. Bellman's contribution is remembered in the name of the [Bellman equation](http://en.wikipedia.org/wiki/Bellman_equation), a central result of dynamic programming which restates an optimization problem in [recursive](http://en.wikipedia.org/wiki/Recursion_(computer_science)) form.

The word *dynamic* was chosen by Bellman to capture the time-varying aspect of the problems, and because it sounded impressive. The word *programming* referred to the use of the method to find an optimal *program*, in the sense of a military schedule for training or logistics. This usage is the same as that in the phrases [*linear programming*](http://en.wikipedia.org/wiki/Linear_programming) and *mathematical programming*, a synonym for [mathematical optimization](http://en.wikipedia.org/wiki/Mathematical_optimization).



## Overview

Dynamic programming is both a mathematical optimization method and a computer programming method. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler subproblems in a [recursive](http://en.wikipedia.org/wiki/Recursion) manner. While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively; Bellman called this the "[Principle of Optimality](http://en.wikipedia.org/wiki/Bellman_equation#Bellman.27s_Principle_of_Optimality)". Likewise, in computer science, a problem that can be broken down recursively is said to have [optimal substructure](http://en.wikipedia.org/wiki/Optimal_substructure).

If subproblems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the subproblems. In the optimization literature this relationship is called the [Bellman equation](http://en.wikipedia.org/wiki/Bellman_equation).

## Dynamic programming in mathematical optimization

* [*Top-down approach*](http://en.wikipedia.org/wiki/Top-down): This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its subproblems, and if its subproblems are overlapping, then one can easily [memoize](http://en.wikipedia.org/wiki/Memoization) or store the solutions to the subproblems in a table. Whenever we attempt to solve a new subproblem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the subproblem and add its solution to the table.
* [*Bottom-up approach*](http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design): Once we formulate the solution to a problem recursively as in terms of its subproblems, we can try reformulating the problem in a bottom-up fashion: try solving the subproblems first and use their solutions to build-on and arrive at solutions to bigger subproblems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger subproblems by using the solutions to small subproblems.

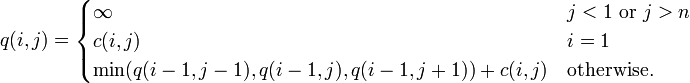
### A type of balanced 0–1 matrix

Consider the problem of assigning values, either zero or one, to the positions of an n × n matrix, with n even, so that each row and each column contains exactly n / 2 zeros and n / 2 ones. We ask how many different assignments there are for a given n. For example, when n = 4, four possible solutions are

\begin{bmatrix}
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0
\end{bmatrix} \text{ and } \begin{bmatrix}
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 \\
1 & 1 & 0 & 0
\end{bmatrix} \text{ and } \begin{bmatrix}
1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1
\end{bmatrix} \text{ and } \begin{bmatrix}
1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1
\end{bmatrix}.

q(A) = \min(q(B),q(C),q(D))+c(A) \, 

Now, let us define q(i, j) in somewhat more general terms:



The first line of this equation is there to make the recursive property simpler (when dealing with the edges, so we need only one recursion). The second line says what happens in the last rank, to provide a base case. The third line, the recursion, is the important part. It is similar to the A,B,C,D example. From this definition we can make a straightforward recursive code for q(i, j). In the following pseudocode, n is the size of the board, c(i, j) is the cost-function, and min() returns the minimum of a number of values:

### Tower of Hanoi puzzle

[](http://en.wikipedia.org/wiki/File:Tower_of_Hanoi.jpeg)

[http://bits.wikimedia.org/static-1.20wmf4/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Tower_of_Hanoi.jpeg)

A model set of the Towers of Hanoi (with 8 disks)

[](http://en.wikipedia.org/wiki/File:Tower_of_Hanoi_4.gif)

[http://bits.wikimedia.org/static-1.20wmf4/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Tower_of_Hanoi_4.gif)

An animated solution of the **Tower of Hanoi** puzzle for*T(4,3)*.

The [**Tower of Hanoi**](http://en.wikipedia.org/wiki/Tower_of_Hanoi) or **Towers of**[**Hanoi**](http://en.wikipedia.org/wiki/Hanoi) is a [mathematical game](http://en.wikipedia.org/wiki/Mathematical_game) or [puzzle](http://en.wikipedia.org/wiki/Puzzle). It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules: