Contents

[Dynamic Programming 2](#_Toc385862864)

[Properties 2](#_Toc385862865)

[Overlapping Sub problems 2](#_Toc385862866)

[Optimal Substructure 2](#_Toc385862867)

[Ways to Store the Values so that the values can be reused. 2](#_Toc385862868)

[1. Memorization (Top Down) 2](#_Toc385862869)

[2. Tabulation (Bottom Up) 2](#_Toc385862870)

[Recursion vs Dynamic Programming 3](#_Toc385862871)

[Backtracking vs Dynamic Programming 3](#_Toc385862872)

[Brute-Force/Naïve vs Dynamic Programming 3](#_Toc385862873)

[Divide and Quaker vs Dynamic Programming 3](#_Toc385862874)

[Reference Links 3](#_Toc385862875)

[1. Longest Increasing Subsequence 4](#_Toc385862876)

[2. Longest Common Subsequence 4](#_Toc385862877)

[3. Min Cost Path 4](#_Toc385862878)

[4. Coin Change 4](#_Toc385862879)

[5. Matrix Chain Multiplication 4](#_Toc385862880)

[6. Binomial Coefficient 4](#_Toc385862881)

[7. 0-1 Knapsack Problem 4](#_Toc385862882)

[8. Egg Dropping Puzzle 4](#_Toc385862883)

# Dynamic Programming

Dynamic Programming is mainly used when solutions of same sub-problems are needed again and again.

We can also say Dynamic Programming = Careful Brute-Force.

In DP computed solutions to subproblems are stored in a table so that these don’t have to recompute. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn’t have common subproblems. If we take example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

<http://20bits.com/article/introduction-to-dynamic-programming>

## Properties

1. Overlapping Sub problems
2. Optimal Substructure

## Overlapping Sub problems

fib(5)

fib(4) + fib(3)

fib(3) + fib(2) + fib(2) + fib(1)

fib(2) + fib(1) + fib(1) + fib(0) + fib(1) + fib(0) + fib(1)

fib(1) + fib(0) + fib(1) + fib(1) + fib(0) + fib(1) + fib(0) + fib(1)

## Optimal Substructure

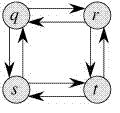
A problem is said to have Optimal Substructure if the globally optimal solution can be constructed from locally optimal solutions to sub problems

E.g. The shortest path problem has following optimal substructure property: If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v.

The standard All Pair Shortest Path algorithms like Floyd–Warshall and Bellman–Ford are typical examples of Dynamic Programming.

On the other hand the Longest path problem doesn’t have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following unweighted graph given in the CLRS book.

There are two longest paths from q to t: q -> r ->t and q ->s->t. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q->r->t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q->s->t->r.

[](http://geeksforgeeks.org/wp-content/uploads/LongestPath.gif)

## Ways to Store the Values so that the values can be reused.

### Memorization (Top Down)

### Tabulation (Bottom Up)

|  |  |  |
| --- | --- | --- |
| No. | Memorization | Tabulation |
| 1 | Top-Down | Bottom-Up |
| 2 | Is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions.  Whenever we need solution to a subproblem, we first look into the lookup table.  If the pre computed value is there then we return that value.  Otherwise we calculate the value and put the result in lookup table so that it can be reused later. | The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table. |
| 3 | Table is filled on demand. | Starting from the first entry, all entries are filled in table one by one. |
| 4 | All entries of the lookup table are not necessarily filled. | All entries of the lookup table are necessarily filled. Need to verify this. |
| 5 |  |  |
|  |  |  |

## Recursion vs Dynamic Programming

Dynamic programming and recursion have a fundamental difference, one is a top down approach and the other is a bottom up approach.

Dynamic programming gives u a better running time and a better space utilization compared to recursion.

Moreover, we can use memorization with recursion to save some space and time but still DP wins.

It makes possible to count the number of solutions without visiting them all.

Imagine backtracking values for the first row – what information would we require about the remaining rows, in order to be able to accurately count the solutions obtained for each first row value?

## Backtracking vs Dynamic Programming

## Brute-Force/Naïve vs Dynamic Programming

It takes far less time than naive methods that don't take advantage of the sub-problem overlap (like depth-first search).

The idea to solve a given problem, we need to solve different parts of the problem (sub-problems), then combine the solutions of the subproblems to reach an overall solution.

When using a more naive method, many of the subproblems are generated and solved many times.

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": 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 as a function of the size of the input.

Dynamic programming algorithms are used for optimization (for example, finding the shortest path between two points, or the fastest way to multiply many matrices).

A dynamic programming algorithm will examine all possible ways to solve the problem and will pick the best solution.

Therefore, we can roughly think of dynamic programming as an intelligent, brute-force method that enables us to go through all possible solutions to pick the best one.

## Divide and Quaker vs Dynamic Programming

DAQ goes top-down to store the solutions.

DP goes top-down and bottom-up to store the solutions.

## Reference Links

<http://en.wikipedia.org/wiki/Dynamic_programming>

<http://stackoverflow.com/questions/19610071/naive-way-to-find-largest-block-in-a-rectangle-of-1s-and-0s/19610652#19610652>

<http://www.bing.com/search?q=scan+line+algorithms&form=IE11TR&src=IE11TR&pc=SNJB>

<http://www.informatik.uni-ulm.de/acm/Locals/2003/html/judge.html>

<http://stackoverflow.com/questions/1540848/a-simple-example-for-someone-who-wants-to-understand-dynamic-programming>

<http://www.geeksforgeeks.org/tag/dynamic-programming/page/3/>

<http://www.cs.mun.ca/~kol/courses/2711-w08/dynprog-2711.pdf>

### Longest Increasing Subsequence

### Longest Common Subsequence

### Min Cost Path

### Coin Change

### Matrix Chain Multiplication

### Binomial Coefficient

### 0-1 Knapsack Problem

### Egg Dropping Puzzle