**Dynamic Programming**

Dynamic Programming is an algorithm design technique for *optimization problems:* often minimizing or maximizing. Like divide and conquer, DP solves problems by combining solutions to sub-problems. Unlike divide and conquer, sub-problems are not independent.

The term Dynamic Programming comes from Control Theory, not computer science. Programming refers to the use of tables (arrays) to construct a solution. In dynamic programming we usually reduce time by increasing the amount of space.

We solve the problem by solving sub-problems of increasing size and saving each optimal solution in a table (usually). The table is then used for finding the optimal solution to larger problems. Time is saved since each sub-problem is solved only once.

***Algorithm Design***

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution in a bottom up fashion.

4. Construct an optimal solution from computed information.

***Example: Rod Cutting***

* You are given a rod of length n ≥ 0 (n in inches).
* A rod of length i inches will be sold for pi dollars.
* Cutting is free (simplifying assumption).

**Problem**: given a table of prices *pi* determine the maximum revenue *rn* obtainable by cutting up the rod and selling the pieces.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Length *i* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Price *pi* | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 30 |

**Greedy approach:**

Select the length that has the maximum price/length.

Reduce to problem to a smaller sub-problem.

Use the greedy approach for length = 4

Greedy solution is {3, 1} with revenue 9

Optimal solution is {2, 2} with revenue 10

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Length *i* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Price *pi* | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 30 |
| *Pi / i* | 1 | 2.5 | 2.67 | 2.25 | 2 | 2.83 | 2.43 | 2.5 | 2.67 | 3 |

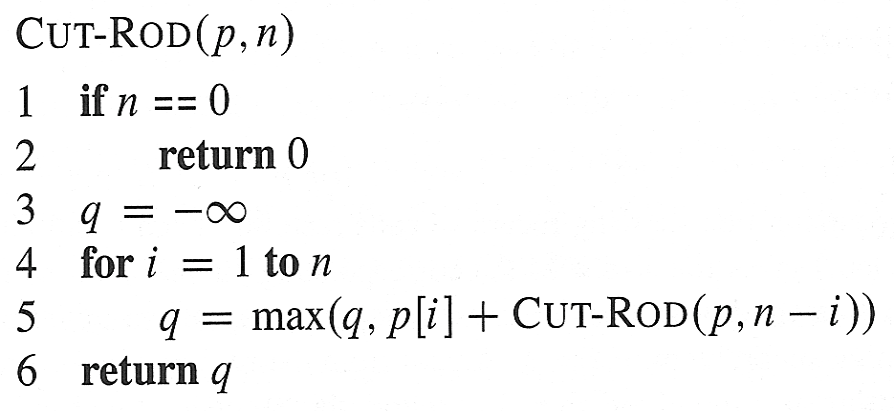
**Recursive solution:**

rn = max(pn, r1 + rn-1, r2 + rn-2, …, rn-1 + r1)

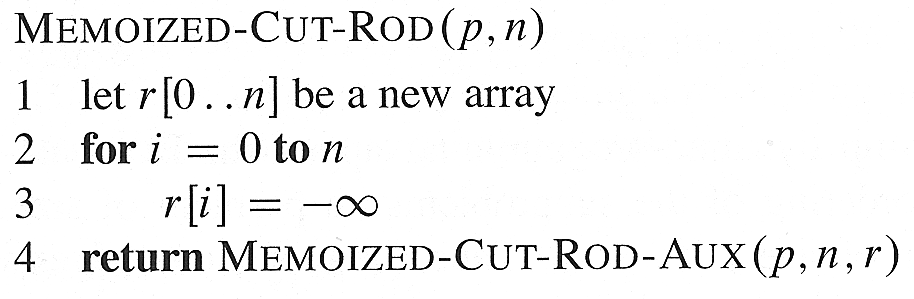
A slightly different way of stating the same recursion, which avoids repeating some computations, is

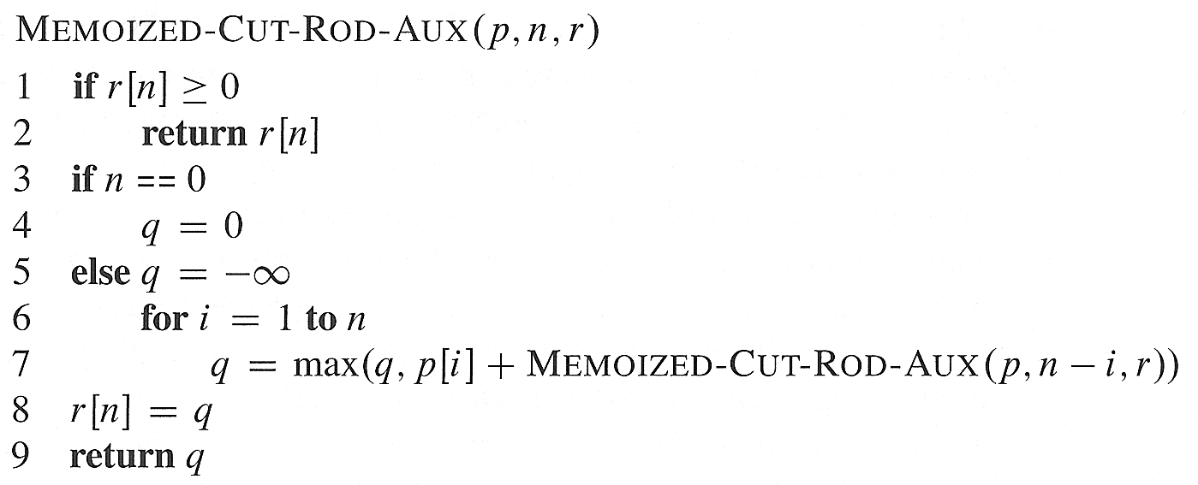
rn = max1≤i≤n(pi + rn-i)

And this latter relation can be implemented as a simple top-down recursive procedure:

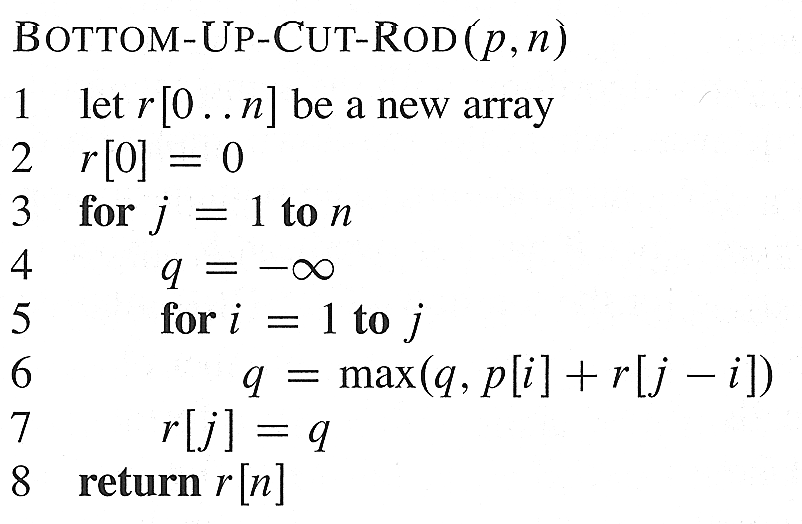


***Top-down with memorization:***

******

****

***Bottom-up:***

****

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Length *i*** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **Price *pi*** | **0** | **1** | **5** | **8** | **9** | **10** | **17** | **17** | **20** | **24** | **30** |
| **Rev *Ri*** | **0** | **1** | **5** | **8** | **10** | **13** | **17** | **18** | **22** | **25** | **30** |

We begin by constructing (by hand) the optimal solutions for i = 1…….10:

r1 = 1 (no cuts)

r2 = 5 (no cuts)

r3 = 8 (no cuts)

r4= 10 (2+2)

r5 = 13 (2+3)

r6 = 17 (no cuts)

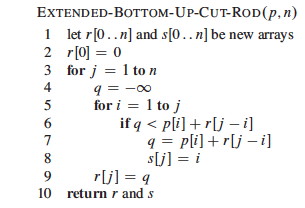
r7 = 18 (2+2+3)

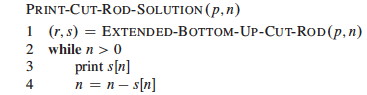
r8 = 22 (2+6)

r9 = 25 (3+6)

r10 = 30 (no cuts)

***Reconstructing a Solution***

******

******

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***Length i*** | ***0*** | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** | ***7*** | ***8*** | ***9*** | ***10*** |
| ***Price pi*** | ***0*** | ***1*** | ***5*** | ***8*** | ***9*** | ***10*** | ***17*** | ***17*** | ***20*** | ***24*** | ***30*** |
| ***Rev Ri*** | ***0*** | ***1*** | ***5*** | ***8*** | ***10*** | ***13*** | ***17*** | ***18*** | ***22*** | ***25*** | ***30*** |
| ***Si*** | ***0*** | ***1*** | ***2*** | ***3*** | ***2*** | ***2*** | ***6*** | ***1*** | ***2*** | ***3*** | ***10*** |