



Dynamic Programming



Introduction

- Divide and conquer
 - Partition the problem into independent subproblems
 - Solve the subproblems recursively
 - Combine the solutions to solve the original problem.
 - **Dynamic programming** solves every subproblem just once and stores the answer in a table.
- **Main approach:** recursive, holds answers to a sub problem in a table, can be used without recomputing.
 - It follows a bottom-up technique by which it start with smaller and hence simplest subinstances.

Dynamic Programming

- A dynamic-programming algorithm **solves each subsubproblem just once and then saves its answer** in a table, thereby avoiding the **work of recomputing** the answer every time it solves each subsubproblem.
- The technique of storing sub-problem solutions is called **memoization**
- Typically the Dynamic programming is applied to optimization problems (**Problems can have many possible solutions**)
- **Steps to develop the Dynamic-programming algorithm**
 - Characterize the structure of an optimal solution.
 - Recursively define the value of an optimal solution.
 - Compute the value of an optimal solution, typically in a bottom-up fashion.
 - Construct an optimal solution from computed information.

Example: Rod cutting

- Dynamic programming to solve a simple problem in deciding where to cut steel rods.
 - Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free.

Serling Enterprises to know the best way to cut up the rods

We are given prices p_i and rods of length i :

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Question: We are given a rod of length n , and want to **maximize** revenue, by cutting up the rod into pieces and selling each of the pieces

Example: Rod cutting

We'll first list the solutions:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

1.) Cut into 2 pieces of length 2: $p_2 + p_2 = 5 + 5 = 10$

2.) Cut into 4 pieces of length 1: $p_1 + p_1 + p_1 + p_1 = 1 + 1 + 1 + 1 = 4$

3-4.) Cut into 2 pieces of length 1 and 3 (3 and 1): $p_1 + p_3 = 1 + 8 = 9$ $p_3 + p_1 = 8 + 1 = 9$

5.) Keep length 4: $p_4 = 9$

6-8.) Cut into 3 pieces, length 1, 1 and 2 (and all the different orders)

$$p_1 + p_1 + p_2 = 7 \quad p_1 + p_2 + p_1 = 7 \quad p_2 + p_1 + p_1 = 7$$

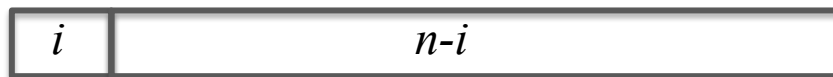
Total: 8 cases for $n = 4$ ($= 2^{n-1}$). We can slightly reduce by always requiring cuts in non-decreasing order. **But still a lot!**

Example: Rod cutting

Note: We've computed a brute force solution; all possibilities for this simple small example. **But we want more optimal solution!**

One solution:

recurse on further



What are we doing?

- Cut rod into length i and $n-i$
- Only remainder $n-i$ can be further cut (recursed)

We need to define:

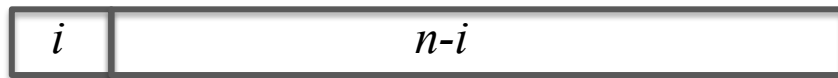
a.) **Maximum revenue** for log of size n : r_n (that is the solution we want to find).

b.) **Revenue (price)** for single log of length i : p_i

Example: Rod cutting

Example: If we cut log into length i and $n-i$:

recurse on further



Revenue: $p_i + r_{n-i}$

Can be seen by recursing on $n-i$

What are we going to do?

There are many possible choices of i :

$$r_n = \max \left\{ \begin{array}{c} p_1 + r_{n-1} \\ p_1 + r_{n-2} \\ \dots \\ p_n + r_0 \end{array} \right\}$$

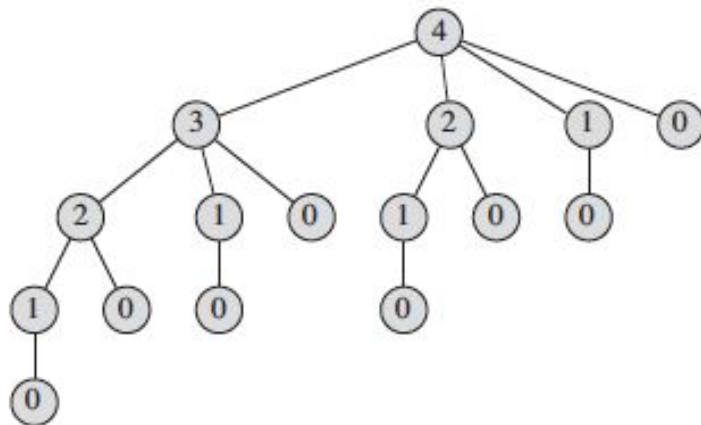
Example: Rod cutting

Recursive (top-down) pseudo code:

```
CUT-ROD( $p, n$ )  
1  if  $n == 0$   
2    return 0  
3   $q = -\infty$   
4  for  $i = 1$  to  $n$   
5     $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$   
6  return  $q$ 
```

Problem? **Slow runtime** (it's essential brute force)!

But why? Cut-rod calls itself repeatedly with the same parameter values (tree):



Example: Rod cutting

- Recursive solution is inefficient, since it repeatedly calculates a solution of the same subproblem (overlapping subproblem).
- Instead, solve each subproblem only once AND save its solution. Next time we encounter the subproblem look it up in a hashtable or an array
- (**Memoization**, recursive top-down solution).
- We save the solution of subproblems Of increasing size (i.e. in order) in an array. Each time we will fall back on solutions that we obtained in previous steps and stored in an array (bottom-up solution).
-

Recursive top-down solution: **Cut-Rod with Memoization**

- **Step 1 Initialization:**

Creates array for holding memoized results, and initialized to minus infinity. Then calls the main auxiliary function.


MEMOIZED-CUT-ROD(p, n)

1 let $r[0..n]$ be a new array

2 for $i = 0$ to n

3 $r[i] = -\infty$

4 return **MEMOIZED-CUT-ROD-AUX**(p, n, r)



Step 2: The main auxiliary function, which goes through the lengths, computes answers to subproblems and memoizes if subproblem not yet encountered:

MEMOIZED-CUT-ROD-AUX(p, n, r)

```
1  if  $r[n] \geq 0$ 
2      return  $r[n]$ 
3  if  $n == 0$ 
4       $q = 0$ 
5  else  $q = -\infty$ 
6      for  $i = 1$  to  $n$ 
7           $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$ 
8   $r[n] = q$ 
9  return  $q$ 
```

Bottom-up solution: **Bottom Up Cut-Rod**

Each time we use previous values from arrays:

BOTTOM-UP-CUT-ROD(p, n)

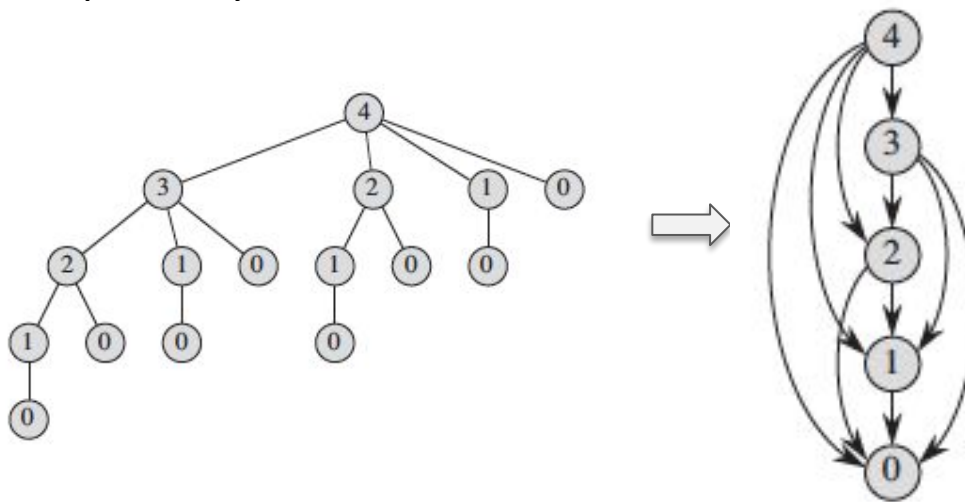
```
1  let  $r[0..n]$  be a new array } Check if value already known or memoized
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$ 
6           $q = \max(q, p[i] + r[j - i])$  } Compute maximum revenue
7       $r[j] = q$  } if it hasn't already been
8  return  $r[n]$  } Saving value computed.
```

Run time:

Run time: For bottom-up and top-down approach
 $O(n^2)$

We can also view the subproblems encountered in graph form.

- We reduce the previous tree that included all the subproblems repeatedly



- each vertex represents subproblem of a given size
- Vertex label: subproblem size
- Edges from x to y : We need a solution for subproblem x when solving subproblem y



Run time:

Run time: Can be seen as number of edges $O(n^2)$

Note: Run time is a combination of number of items in table (n) and work per item (n).
The work per item because of the max operation (needed even if the table is filled
And we just take values from the table) is proportional to n as in the number of edges
in graph.

– Example-2: Fibonacci numbers

- Divide problem into two subproblems to calculate (n-1)th and (n-2)th Fibonacci numbers

Problem:

Let's consider the calculation of **Fibonacci** numbers:

$$F(n) = F(n-2) + F(n-1)$$

with seed values $F(1) = 1, F(2) = 1$

or

$$F(0) = 0, F(1) = 1$$

What would a series look like:

0, 1, 1, 2, 3, 4, 5, 8, 13, 21, 34, 55, 89, 144, ...

<https://www.cs.usfca.edu/~galles/visualization/DPFib.html>

Divide and Conquer

Recursive Algorithm:

```
Fib(n)
{
    if (n == 0)
        return 0;

    if (n == 1)
        return 1;

    Return Fib(n-1)+Fib(n-2)
}
```

Recurrence relation:

$$T(n) = T(n-1) + T(n-2) + o(1)$$

Time Complexity: $O(2^{(n/2)})$

- Using **Dynamic Programming** approach with memoization:
 - Use same recurrence relation and store the results of the problem that solved. i.e Memorize these result in an array

```
void fib () {  
    fibresult[0] = 1;  
    fibresult[1] = 1;  
    for (int i = 2; i<n; i++)  
        fibresult[i] = fibresult[i-1] + fibresult[i-2];  
}
```

fib(20) = 10946

0	1	14	610
1	1	15	987
2	2	16	1597
3	3	17	2584
4	5	18	4181
5	8	19	6765
6	13	20	10946
7	21		
8	34		
9	55		
10	89		
11	144		
12	233		
13	377		



References

- <https://www.cs.usfca.edu/~galles/visualization/DPFib.html>