Dynamic Programming – class 2

- Main approach is recursive, but holds answers to subproblems in a table so that can be used again without re-computing

- Can be formulated both via recursion and saving in a table (memoization) or saving in a table bottom-up. Typically, we first formulate the recursive solution, and then turn it into recursion plus dynamic programming via memoization, or bottom-up.

- "programming" as in tabular, not programming code

Example: Rod cutting:

We are given prices p_i for each rod 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: We are given a 4 inches rod. Best solution to cut up? We'll first list the solutions:

1. Cut into 2 pieces length 2:

$$p_2 + p_2 = 5 + 5 = 10$$

2. Cut into 4 pieces length 1:

$$p_1 + p_1 + p_1 + p_1 = 1 + 1 + 1 + 1 = 4$$

3-4. Cut into 2 pieces, length 1 and length 3 (or vice versa length 3 and then 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 (any order):

$$p_1 + p_1 + p_2 = 7; p_2 + p_1 + p_1 = 7; p_1 + p_2 + p_1 = 7$$

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!

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

- Cut rod into length i and n-i

- Only remainder n-i can be cut (recursed on) further

We'll define:

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

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

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

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

(this can be seen as recursing on n-i)

There are many possible choices of i:

$$r_n = \max \begin{Bmatrix} p_1 + r_{n-1} \\ p_2 + r_{n-2} \\ \dots \\ p_n + r_0 \end{Bmatrix}$$

Recursive (top-down) pseudo code:

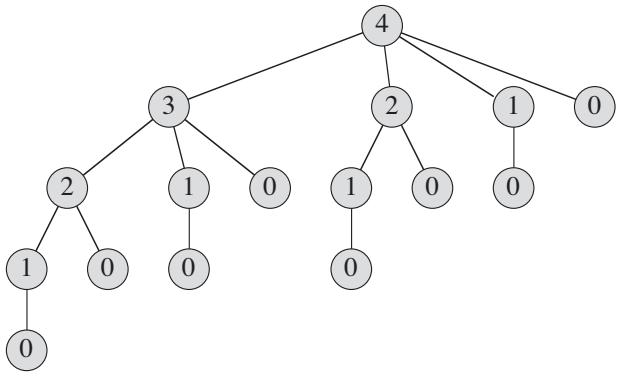
```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Problem?

Run time very slow; like brute force

Why?

Cut-rod calls itself repeatedly with the same parameter values. We can see by plotting a tree:



- Node label = size of subproblem called on

- Can see by eye that many subproblems called repeatedly. We call this a problem with subproblem overlap.

- Number of nodes exponential in n (2^n); therefore exponential number of calls to Cut-Rod

Dynamic programming approach:

- We saw that recursive solution inefficient, since repeatedly computing answer to same subproblem (overlapping subproblems)
- Instead, solve each subproblem only once and save its solution. Next time we encounter subproblem, look it up in hash table or array. We call this memoization = sub-solution has been remembered. (recursive, top-down solution)
- We'll also discuss a second, equivalently good solution, of saving the results of subproblems of increasing size (in order) in an array, each time using results from previously computed array entries (bottom-up solution).

(1) Recursive top-down solution: Cut-Rod with Memoization:

Step 1: Initialization:

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

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

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] ≥ 0
2       return r[n]
3  if n == 0
4       q = 0
5  else q = −∞
6       for i = 1 to n
7            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

(2) There's a simpler bottom-up solution, going in order, each time using previous value from array:

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4       q = −∞
5       for i = 1 to j
6            q = max(q, p[i] + r[j − i])
7       r[j] = q
8  return r[n]
```

Lines 1-2 check if value already known or memoized; Lines 3-7 compute the maximal revenue if it has not already been memoized, and line 8 saves it.

Run time: For both top-down and bottom-up versions:

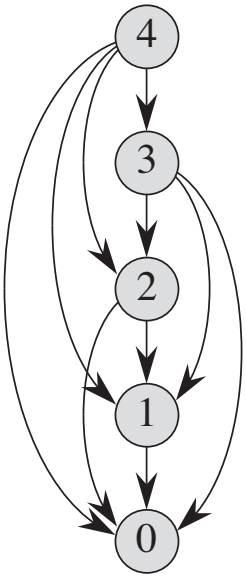
$O(n^2)$

Easiest to see for bottom-up version: doubly-nested for loop.

We can also view the subproblems encountered in graph form:

- We reduce previous tree that included all the subproblems repeatedly

- Here each vertex represents subproblem of a given size



Vertex label: subproblem size

Edge from x to y: We need a solution to subproblem y when solving subproblem x.

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 the graph.