6.006 Lecture 19: Dynamic Programming I

- ☐ Rod-Cutting Problem
- ☐ Top-down & Bottom-up DP
- ☐ General characteristics of ~~DP~~ problems that DP can solve

CLRS 15.1 - 15.4

Rod-Cutting Problem

- A metal rod of length n
- We can cut it into integer-size pieces

$n=9$

- A piece of size i sells for p_i dollars
- cutting costs us nothing

i	1	2	3	4	5	6	7	...
p_i	3	4	10	11	7	15	15	...

- Goal: maximize revenue r_i from a rod of size n (by cutting it up)

$r_1 = 3 \quad (= p_1)$

$r_2 = 6 \quad (p_1 + p_1)$

$r_3 = 10 \quad (p_3)$

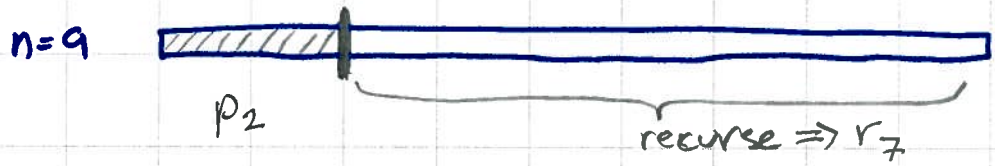
$r_4 = 13 \quad (p_1 + p_3)$

$r_5 = 16 \quad (p_3 + p_1 + p_1 = r_4 + p_1)$

$r_6 = 20 \quad (p_3 + p_3)$

Algorithm Idea

- Consider the leftmost ~~cut~~ piece & recurse



- iterate over all possible sizes for leftmost piece:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

leftmost piece

max revenue from leftover piece could be empty, so

$r_0 = 0$

Possible implementation:

```
def r(p,n):
    ans = p[n]          # no cutting ; r_n ≤ p_n + r_0 (= 0)
    for i in range(1,n):
        ans = max(ans, p[i] + r(p,n-i))
    return ans
```

Correct but slowwwww!!!

why is this slow?

$T(n) = \#$ calls to $r(p,n)$

$T(1) = 1$ never makes a recursive call

$$T(n) = 1 + \sum_{i=1}^{n-1} T(i)$$

$r(p,n)$ $r(p,n-i)$ in the loop

n	1	2	3	4	5	6	7	...
$T(n)$	1	2	4	8	16	32	64	...

$T(n) = 2^n$

Inefficient because we recompute same r_i's!

Solution: memorize (or build up)

```
memo={}
def r(p,n):
    if n in memo: return memo[n]
    ans=p[n]
    for i in range(1,n):
        ans= max(ans, p[i]+r(p,n-i))
    memo[n]= ans
    return ans
```

"top-down"

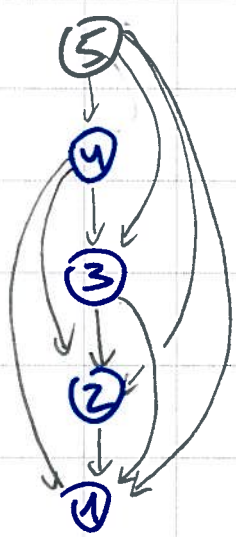
running time = $\Theta(n^2)$

OR:

```
r = [0]*(n+1)   # a list!
for k in range(1, n+1)
    ans = p[k]
    for i in range(1, k):
        ans = max(ans, p[i]+r[k-i])
    r[k] = ans
```

"bottom-up", also $\Theta(n^2)$

Subproblem dependence graph



vertices: ~~subp~~ subproblems r_i

edges: $A \longrightarrow B$ if solving A requires solving B first

both $\Theta(V+E)$ but don't build the graph!

Top-down (+ memorization): DFS on this graph

Bottom-Up: solve in reverse topological order

← too much memory to represent edges.

Characteristics of Dynamic Programming:

- Many related subproblems

optimal substructure
- Optimal solution for one (sub) problem ~~requires~~ contains optimal solution of smaller subproblems

overlapping subproblems
- Different subproblems benefit from knowing solution to <u>same</u> subproblem (so memorize)