6.006
Rivest
#L19.1
11/6/08

Admin: Quiz 2 next Wed 11/12 in room 1-190, 7-9 pm

Readings: CLRS 15.1-15.4

Outline: Dynamic Programming (1/4)

- ☐ "Rod-cutting" problem
- ☐ Dynamic programming approaches:
  - ☐ top-down + memoization
  - ☐ bottom-up (smallest subproblems first)

Rod Cutting:

We have a supplier who gives a rod of length, say, n.
(n may vary from rod to rod).
We want to cut rod into pieces & sell pieces for maximum revenue.
(Pieces have integer sizes 1, 2, ...)
Market determines price: let p_i = price for a piece of length i
(Assume cost of doing cuts is negligible)

e.g.

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

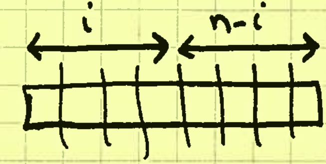
Let r_n = maximum revenue we can get by cutting up a piece of size n:

$r_1 = 3$ (no cutting possible $= p_1$)
$r_2 = 6$ $(3+3 = r_1 + r_1)$
$r_3 = 10$ (p_3)
$r_4 = 13$ $(p_1 + p_3)$
$r_5 = 16$ $(r_1 + r_4 = p_1 + p_1 + p_3)$
$r_6 = 20$ $(p_3 + p_3)$
$r_7 = 23$ $(p_3 + p_3 + p_1)$

Given: $p_1, p_2, \ldots$

Compute: $r_1, r_2, \ldots$

How?



Not too hard:

$$r_n = \max\left(p_n,\ r_1 + r_{n-1},\ r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right)$$

cut into i & $n-i$
& recurse on pieces

Suffices to consider only leftmost cut.

$r_0 = 0$

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

for leftmost piece

only recurse on right-hand piece, since we are considering left-most cut position

Features to note: (characteristics of dyn. prog. problems in general):

- many related subproblems (computing $r_1, r_2, \ldots$) all of same type
- computing optimal solution for one (e.g. r_n) requires consideration of optimal solution of smaller subproblems: "optimal substructure"
- "overlapping subproblems": both r_6 and r_7 benefit from knowing solution to r_3, for example.
- a bit like divide & conquer, except there is some optimization (i.e. maximization or minimization) involved; we are trying to find best way to divide problem up into subproblems, rather than just taking a fixed division.

First cut at python code:

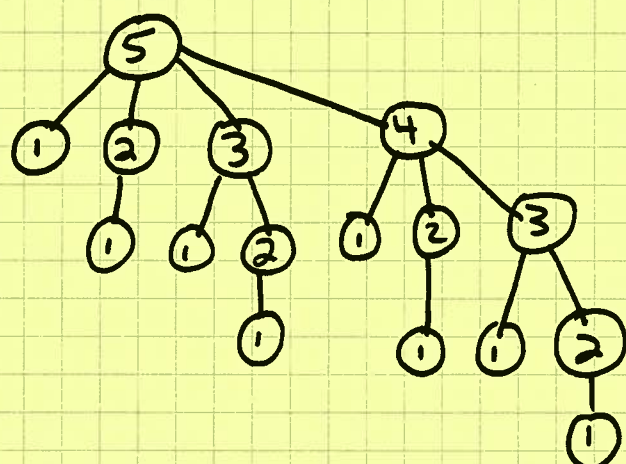
```
def r(p,n):
    if n==0: return 0
    ans = p[n]
    for i in range(1,n):
        ans = max(ans, p[i]+r(p,n-i))
    return ans
```

Gives correct answers, but...

slow (r(p,27) takes ≈ 1 minute...) (dies for n ≥ 30...)

why?

Consider r_5:



(n) ≡ r called with argument n

complexity $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + \sum_{1 \le i < n} T(i) & \text{if } n > 1 \end{cases} = \#$ calls

n	1	2	3	4	5	
T(n)	1	2	4	8	16	...

$T(n) = 2^n$ (prove it!)

But: we are recomputing same things over and over... there are only n distinct problems here: $r_1, \ldots, r_n$!

Only compute each subproblem once:

Two ways of doing this, both save previously computed answers:

$\left.\begin{array}{l} \text{① Top-down + "memoization"} \\ \text{② Bottom-up + array of pre-computed values} \end{array}\right\}$ dynamic programming!

① Top down + memoization

```
memo = {}
def r(p,n):
    if n in memo: return memo[n]
    if n==0: return 0
    ans = p[n]
    for i in range(1,n):
        ans = max(ans, p[i]+r(p,n-i))
    memo[n] = ans
    return ans
```

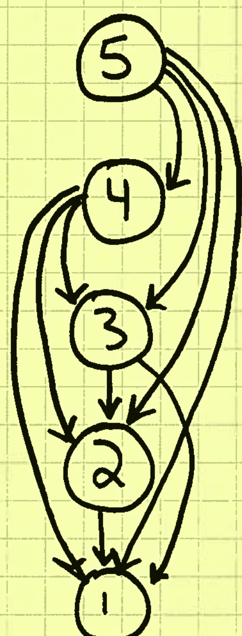
Complexity = $\Theta(n^2)$ to compute r_n

② Bottom-up: compute r_1, then r_2, ...

```
r = [0]*(n+1)  # r is array now
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
```

time $= \Theta(n^2)$

"Subproblem dependence graph"



vertices = subproblems to solve

edges = dependence between subproblems

Ⓐ → Ⓑ

solving A requires solving B first...

Note:

- Top-down + memoization is really just DFS on subproblem dependence graph

  ($\therefore$ time $= O(V+E)$)

- Bottom-up method is just solving problems in order determined by reverse of topological sort

  (i.e. solve subproblems in order of increasing size...)

  (time is still $O(V+E)$)

Setting up subproblem dependence graph is good first step in working on D.P. problem.

Exercise: how to reconstruct optimal solution?

D.P. buzzwords to remember:

- subproblem dependence graph
- optimal substructure
- overlapping subproblems