Math 182 Lecture 11

Chapter 6 Dynamic Programming

Recall: previously learned divide-and-conquer

- d&c typically solves disjoint subproblems

In contrast, dynamic programming involves overlapping subproblems

E.g. Fibonni(n) $\Theta(\phi^n)$ exponential

Fib(5) calls Fib(4), Fib(3)

Fib(4) calls fib(3), Fib(2)

overlap.

FibFast(n) $\Theta(n)$

We computed $F_0, F_1, \ldots, F_n$ storing the results as we go.

Dynamic programming often used for optimization problems

(1) Characterize what an optimal solution looks like
(2) Recursively define value of optimal sol.
(3) Compute value of optimal solutions
(optional) (4) Reconstruct a solution which is optimal.

§6.1 Rod cutting

We have rods of length n inches cut up rod into smaller rods which we can sell at prices determined by table.

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

10 inches → \$30

9+1 → 24+1 = 25

2+2+2+2+1+1 → 5·4 + 2 = 22

$n = 5$

5 → \$10

2+2+1 → 5+5+1 = \$11

Goal: maximize revenue.

Rod-cutting problem: Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtained by cutting up the rod and selling the pieces.

Q: How many ways to cut up rod of length n?

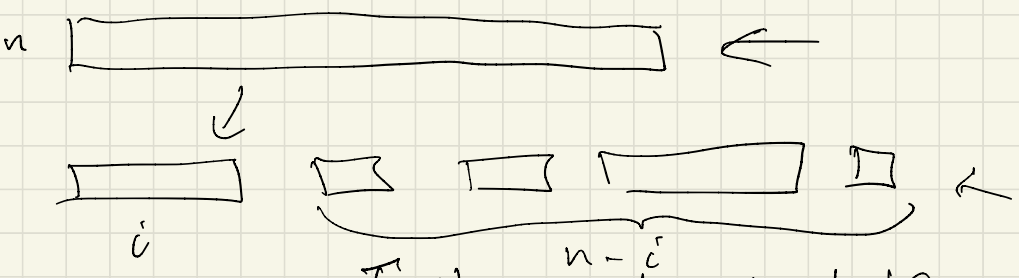
A: 2^{n-1}

for each of n-1 potential cut spots, choose yes or no.

Even if we take into account permutations, then the number will be the partition-number of n, which is still expaential..

What can we say about optimal sol?

Sps we have an optimal sol.



Let r_n be max revenue for length n.

$$\begin{cases} r_0 = 0 \\ r_n = \max\{p_i + r_{n-i} : 1 \le i \le n\} \end{cases}$$

recurrence for r_n.

Recursive top-down implementation

$p = p[0..n]$ prices
n = length of rod.

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

Analyze running time:
$T(n)$ = # of calls to Cut-Rod for n

$T(0) = 1$

$n \geq 1$ $T(n) = 1 + \sum_{i=1}^{n} T(n-i) = 1 + \sum_{i=0}^{n-1} T(i)$

Claim $T(n) = 2^n$ thus Cut-Rod is $\Omega(2^n)$.

Pf: Induction $n = 0$ ✓

Assume $n \geq 1$, $T(m) = 2^m$ for $m < n$

Then $T(n) = 1 + \sum_{i=0}^{n-1} T(i) = 1 + \sum_{i=0}^{n-1} 2^i$

$= 1 + \frac{1-2^n}{1-2} = 1 + (2^n - 1) = 2^n$ ✓ ∎

Using dynamic programming

Idea: After first time solving subproblem, store value of solution. In the future first check to see if answer already stored.

This gives us a time-memory trade-off

Two ways to do dynamic programming

Method 1: top-down memoization

Memo = memorandum "to be remembered"

idea: do same thing as recursive algorithm except store values first time sol to subproblem is solved, in future don't do recursive call if already have sol.

```
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)
```

← Create array r[0..n]

| ← initialize it to <-∞, -∞, ..., -∞>

```
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
```

| check if sol already known

| stores sol to r[n] in line 8.

Method 2 Bottom-up-method

Idea: solve subproblems in natural order of increasing size/complexity respecting the dependencies between subproblems.

```
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]
```

create array; set $r[0] = 0 = r_0$

solve subproblems of increasing size

solves subproblem of size j

return $r_n = r[n]$.

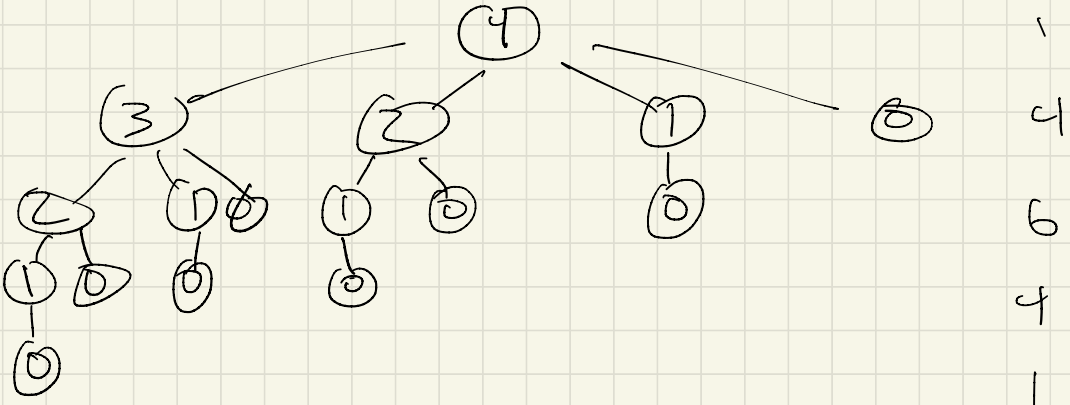
Running time of both is $\Theta(n^2)$

Get triangular sum $\rightarrow \sum_{i=1}^{n} n - i = \Theta(n^2)$

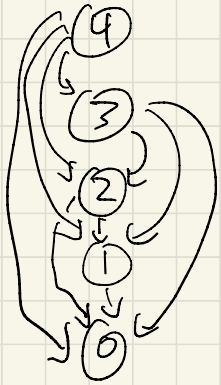
Subproblem graphs

Consider Cut-Rod(p, 4)

Recursion-tree



Subproblem graph



Reconstructing a solution

Can store value of ~~length~~ length of first cut of optimal solution. for each possible length.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[1..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

s[j] length of first cut to make which is part of optimal sol.

Example $n = 8$, $r_8 = r[8] = 22$

$s[8] = 2 \rightsquigarrow 8 = 2 + 6$

$s[6] = 6 \rightsquigarrow 2 + 6$ optimal for 8.

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$		1	2	3	2	2	6	1	2	3	10

this traces backwards through s, printing out segs of cuts needed to make.

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n − s[n]
```

§6.2 Matrix-chain multiplication

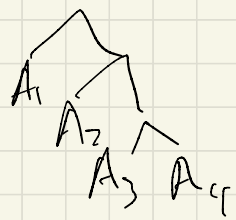
Consider matrices $A_1, A_2, \ldots, A_n$

want to compute

$$A_1 A_2 \cdots A_n.$$

How to evaluate? different possible groupings. E.g. $n = 4$

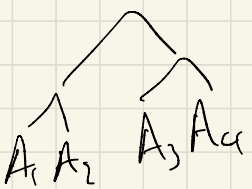
$$(A_1 (A_2 (A_3 A_4)))$$



want grouping which minimizes # of scalar multiplications

$$(A_1 ((A_2 A_3) A_4))$$

$$((A_1 A_2) (A_3 A_4))$$



$$((A_1 (A_2 A_3)) A_4)$$

$$(((A_1 A_2) A_3) A_4)$$

MATRIX-MULTIPLY(A, B)

```
1  if A.columns ≠ B.rows
2        error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4        for i = 1 to A.rows
5              for j = 1 to B.columns
6                    c_ij = 0
7                          for k = 1 to A.columns
8                                c_ij = c_ij + a_ik · b_kj
9        return C
```