Dynamic Programming

- Like divide-and-conquer, DP solves problems by combining solutions to subproblems
- We use DP when the subproblems overlap
    - Subproblems share subproblems
    - Solutions to already-solved problems are stored so they don't need to be recalculated
        - Time-memory tradeoff

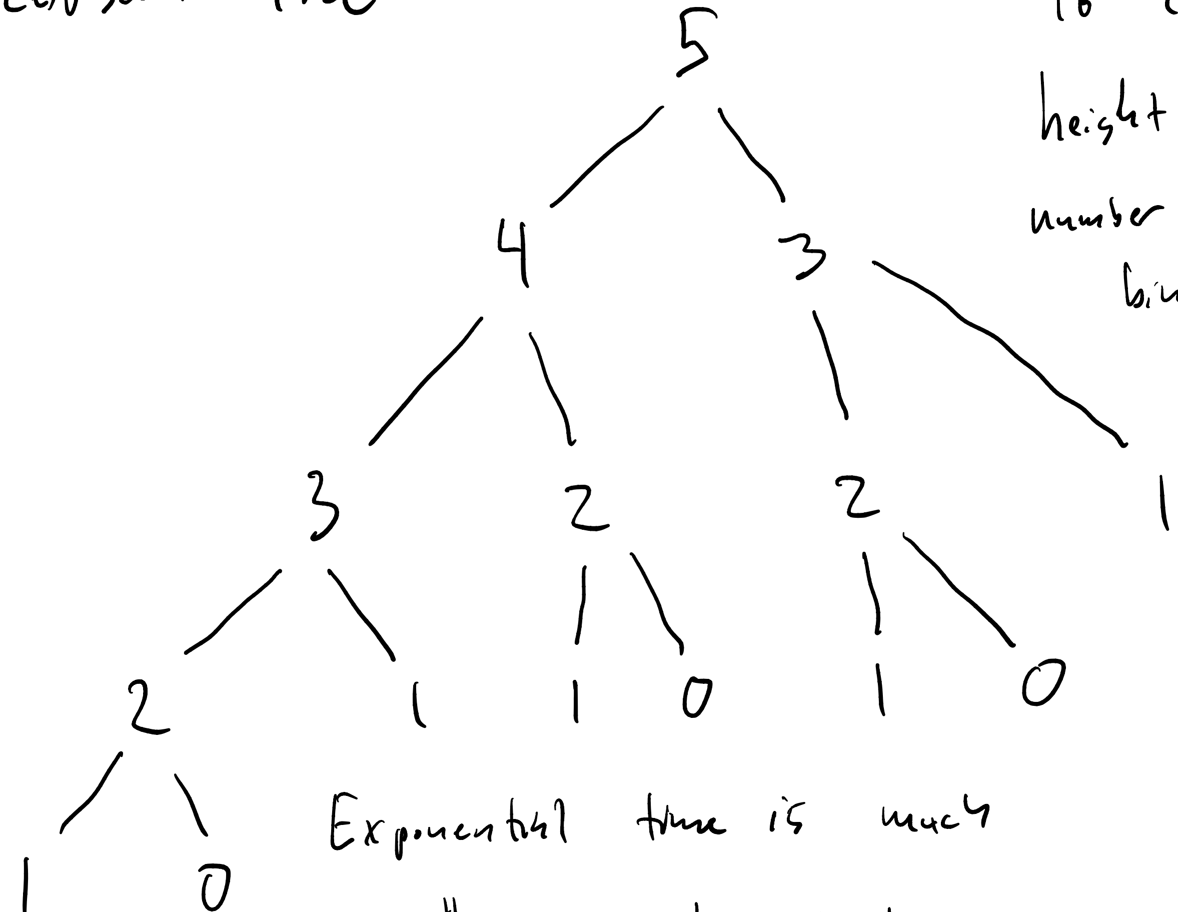
- Memoization
    - Used with recursive DP
    - Save the results of recursive calls in an array or hash table

- Recursive Fibonacci

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

```
fib(n)
    if n < 2:
        return n
    else
        return fib(n-1) + fib(n-2)
```

Recursion tree



16 calls

height is $n-1$

number of nodes in a binary tree is $\leq 2^{h+1}-1$

$O(2^n)$

Tree grows exponentially <u>very bad</u>

Exponential time is much worse than any polynomial time

- DP approach
  - store the nth fibonacci number when it is calculated
  - If a fib. number is already calculated, use the stored value instead of a recursive call

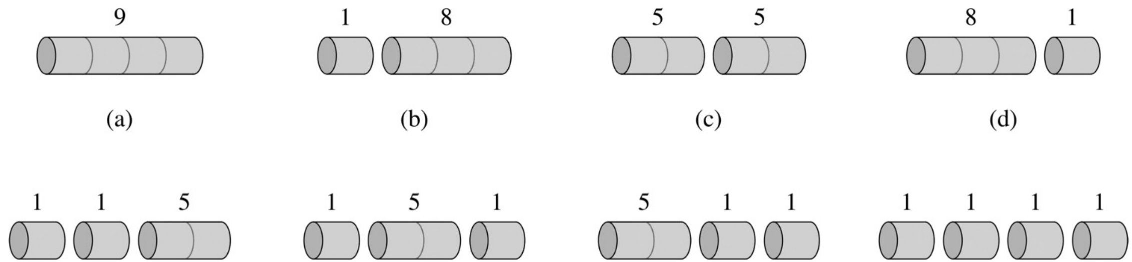
- Using DP for optimization problems

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution, typically in a bottom up fashion
4. Construct an optimal solution from computed information

- Rod cutting
  - A rod of integer length n can be cut into smaller rods, each with an integer length
  - A rod of length i can be sold for price p_i
  - A table stores all the prices of possible rod sizes $\leq n$
  - What are the optimal cuts to get the best revenue r_n?

$n = 4$ $\qquad$ 2^{n-1} ways to cut a rod of length n

can't do brute force for longer rods



(a) (b) (c) (d) (e) (f) (g) (h)

000
100
010
001
110
101
011
111

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

- Optimal substructure
  - Say we initially cut a piece of length i
  - If we don't cut that piece smaller, the optimal r_n we can achieve is $p_i + r_{n-i}$ where r_{n-1} is the optimal revenue we can get by cutting a rod of length $n-i$

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

This is now recursively defined

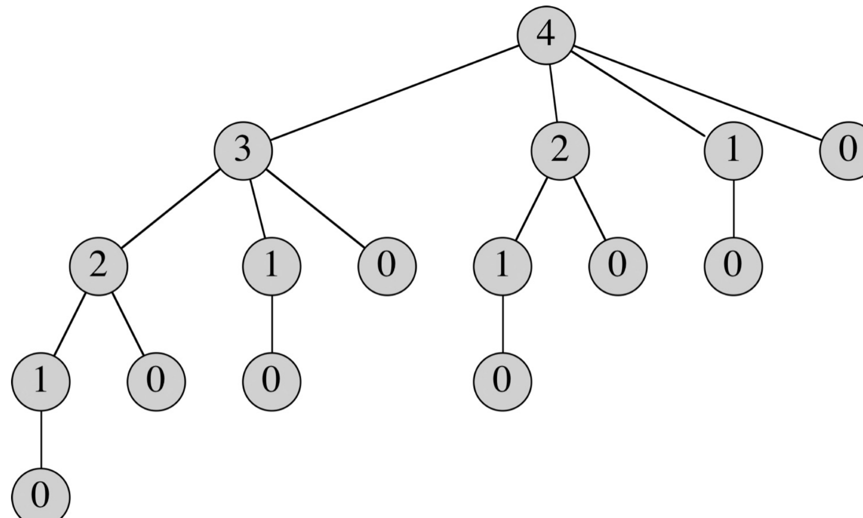
CUT-ROD(p, n)

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

recursion tree shows this

is brute force

returns r_n



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

Store already-calculated subproblem solutions in r

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0
      return r[n]
  if n == 0
      q = 0
  else q = −∞
      for i = 1 to n
          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
  r[n] = q
  return q
```

- Bottom-up solution
  - Works when a problem only ever needs solutions to smaller subproblems, not larger
  - Solve the smallest subproblem first, then work up to larger ones

$n = 4$

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

```
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

0	1	5	8	10
0	1	2	3	4

j	i	q
4	4	16

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

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

```
let r[0..n] and s[1..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
return r and s
```

PRINT-CUT-ROD-SOLUTION(p, n)

```
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n - s[n]
```