Review

Dynamic Programming

- Divide into sub-problems
- Solve sub-problems first, then combine to solve larger problem
- The sub-problems are overlapping
  - Divide and conquer will solve the same sub problem again and agaain **(Recursion)**
  - Dynamic programming will solve each sub-problem once, and remembers the answer **(Memorize)**
- Trades space (to save sub-problem solutions) to save time
- Usually used to solve **'optimization'** problems (similar to greedy)

Coin Change Problem

$$D_n = D_{n-1} + D_n n - 3 + D_{n-4}$$

Base Cases

$$\begin{aligned}
&D_0 = 1 \\
&D_1 = 1 \Rightarrow \text{`}1' \\
&D_2 = 1 \Rightarrow \text{`}1 + 1' \\
&D_3 = 2 \Rightarrow \text{`}1 + 1 + 1' \text{ or } \text{`}1 + 2'
\end{aligned}$$

Rod Cutting

Give a rod of length n with $n-1$ cutting points, as well as revenue for each length:

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

Find the best cutting for the rod that maximizes the revenue.

For a rod of length 4, what's the best cut?

- 1, 3
- 2, 2

How many possible cuttings for rod with length n?

- Exponential $\rightarrow 2^{n-1} = 2^3 = 8$

Each cutting point is a random variable (0 or 1)

- We have $n-1$ cutting points
- The total possibilities is 2^{n-1}

Naive Approach

- Try all possibilities and select the max
- Best choice: two pieces each of size 2 $\Rightarrow$ $revenue = 5+5 = 10$

Optimizaiton Problem

- Rod cutting is an optimizaiton problem (maximize profit).
- Has optimal substructure property:
- You must have the optimal cut for each sub-problem to get the global optimal.
- Has recursive exponential solution
- Has polynomial dynamic programming solution

Define the cost (revenue): r_i is the max revneue for a rod of length i.

i	r_i	optimal
1	1	1
2	5	2
3	8	3
4	10	2, 2
5	13	2, 3
6	17	6
7	18	1, 6
8	22	2, 6
9	25	3, 6
10	30	10

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

Basic Approach: Recursive

First, cut a piece off the left of the rod, and sell it. Then, find the optimal way to cut the remainder of the rod.

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

Leaves only one subproblem to solve rather than two subproblems.

```
Cut-Rod(p, n)
if n == 0
    return 0
q = -infinity
for i = 1 to n
    q = max(q, p[i] + Cut-Rod(p, n-i))
return q
```

Similar disadvantages to fibonacci recursive solution.

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n > 0 \end{cases}$$

Memorizing

```
Memoized-Cut-Rod(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = -infinity
return Memoized-Cut-Rod-Aux(p, n, r)
```

```
Memoized-Cut-Rod-Aux(p, n, r)
if r[n] >= 0
    return r[n]
if n == 0
    q = 0
else
    q = -infinity
    for i = 1 to n
        q = max(q, p[i] + Memoized-Cut-Rod-Aux(p, n-i, r))
r[n] = q
return q
```

- Solves each subproblem only once
- Solves subproblems for sizes $0, 1, 2, \ldots, n$
- To solve subproblem of size n, the for loop iterates n times
- Overall recursive calls, the total number of iterations $= 1 + 2 + \ldots$
- $\Theta(n^2)$ time

```
Bottom-Up-Cut-Rod(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -infinity
```

```
    for i = 1 to j
        q = max(q, p[i] + r[j-i])
    r[j] = q
return r[n]
```

- Nested loops, $1 + 2 + 3 + \ldots + n$
- $\Theta(n^2)$ time

Bottom up is *probably* easier to code.
These algorithms tell you the optimal revenue, but not *how to get it*.