



CS5800: Algorithms

Week 4 – Dynamic Programming

- Intuition, Principles, and Application

Dr. Ryan Rad

Spring 2025



Northeastern
University



新年快乐





Class Average: 92.5%

HW1 Grades

Q1: 100%

Q2: 94.1% - Michelle

Q3: 95.0% - Ya

Q4: 92.4% - Michelle

Q5: 88.6% - Ya

Today

- What is Dynamic Programming?
- Dynamic Programming vs Divide & conquer
- Dynamic Programming Techniques
- Fibonacci
- Rod-Cutting Problem
- Longest Common Subsequence
- Quiz
- Brain Break
- Class Activity



What is *Dynamic Programming*?

Essence:

- It's all about breaking down complex problems into simpler, overlapping subproblems and solving each subproblem only once, storing the results for future use.

Key Idea:

- The "programming" in Dynamic Programming has nothing to do with coding; it comes from the term "mathematical programming," which means optimizing a series of decisions.

Why "Dynamic"?:

- The "dynamic" part indicates that solutions are built incrementally by considering smaller subproblems, which can lead to more efficient solutions.

Primary Application of DP

Optimization Problems:

- Each solution represents a value
- You want a solution with optimal value (maximum or minimum)

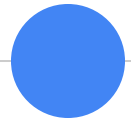


A top-down view of a wooden desk. In the top left, a portion of a silver laptop is visible. To its right is a small potted plant with green grass-like leaves. Below the laptop, a circular holder contains several pens and pencils in various colors (blue, green, yellow, red, black). A single silver pen lies vertically to the right of the holder. In the bottom left corner, a metal ruler is placed horizontally, showing markings from 0 to 21. In the bottom right corner, a white cup filled with dark coffee sits next to a small pile of green and yellow snacks. The central focus is a stack of white papers with the title text printed on them.

Dynamic Programming vs Divide & Conquer

Disjoint vs Overlapping subproblems

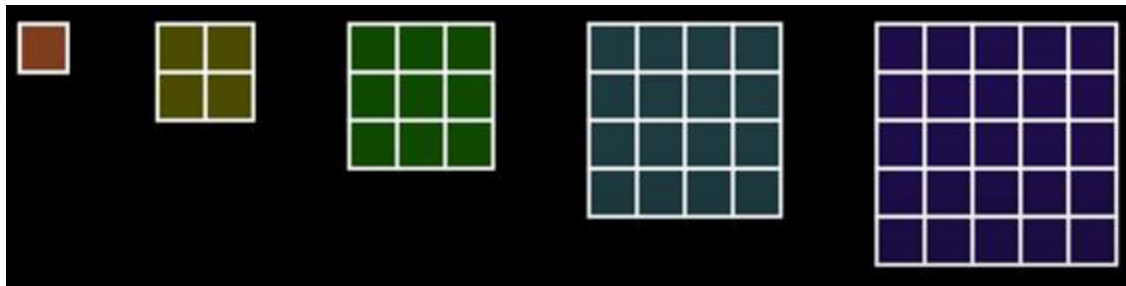




Dynamic Programming Techniques



1. **Brute Force:** Design a “brute force” recursive solution
2. **Memorized ?** Take a recursive algorithm, find the overlapping subproblems, and store the results to avoid re-computing them in future calls.
3. **Bottom-up approach:** Intellectually reorder the sub-problems

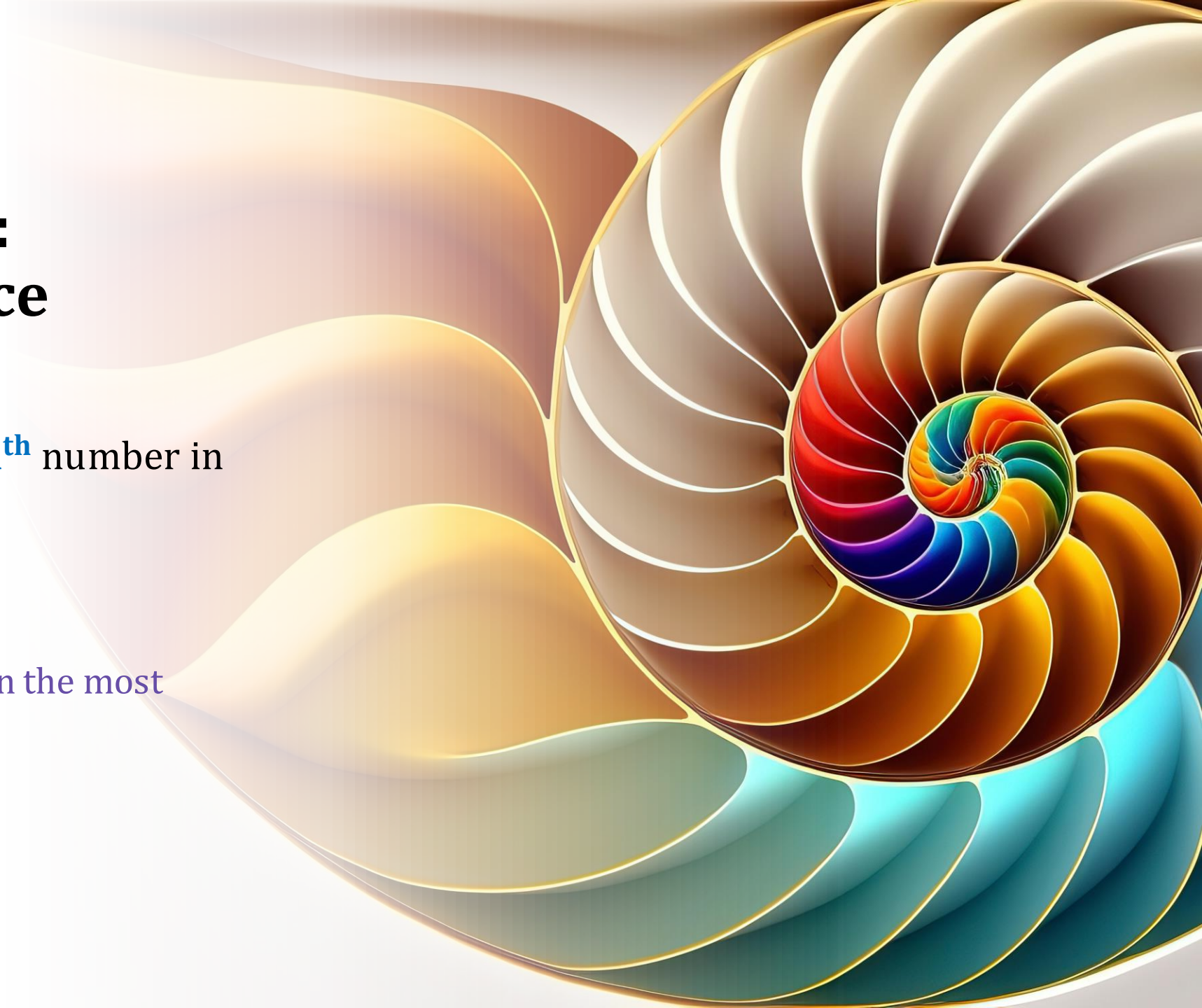


A little confusing? Don't worry, you are not alone!

Example Problem: Fibonacci Sequence

Given a number **n**, print the **nth** number in Fibonacci sequence.

How can we solve this problem in the most optimized way?

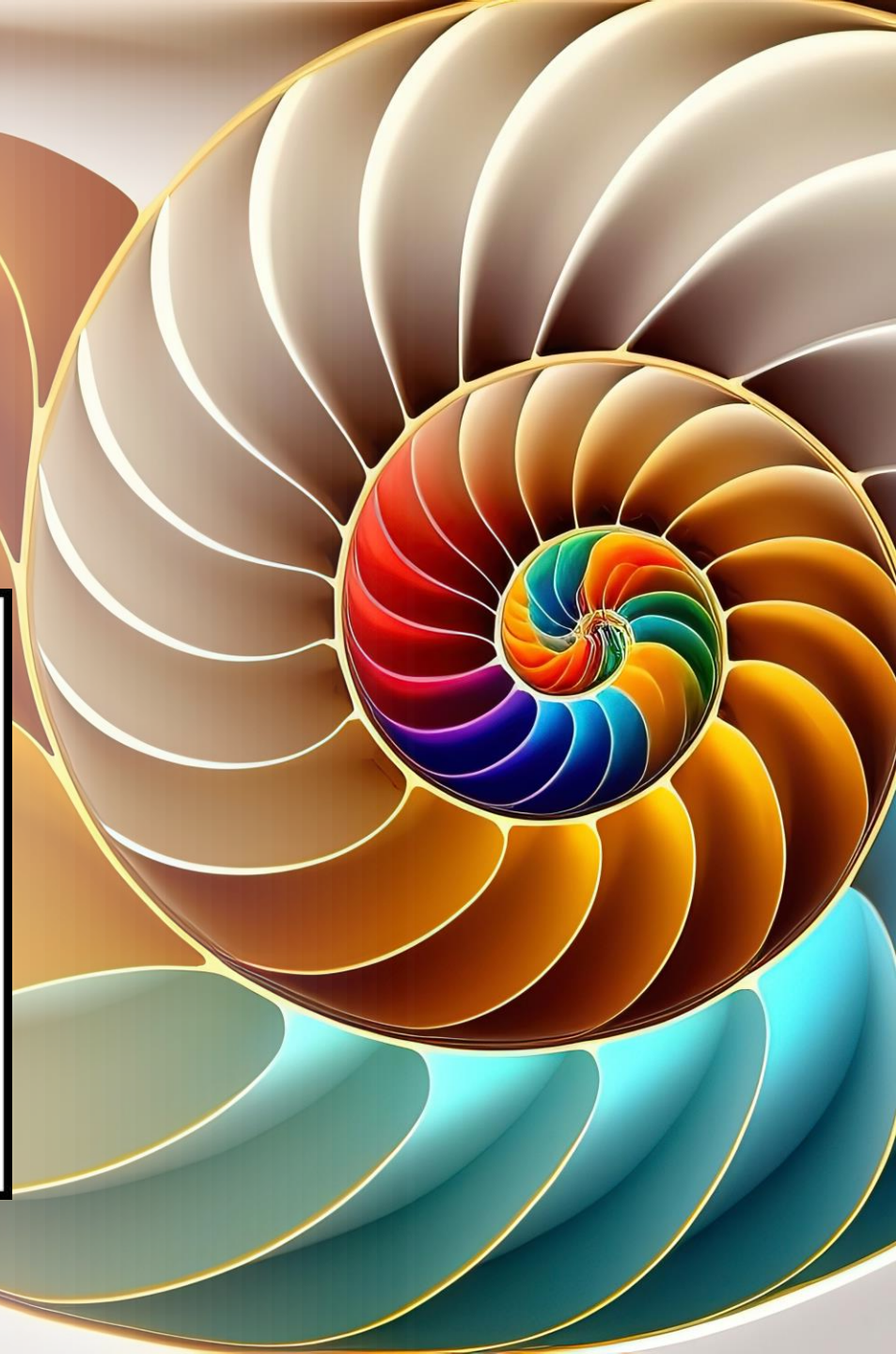


Example Problem: Fibonacci Sequence

Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987 ...

Each number is the sum of the previous two numbers.



Let's clarify

1. Can n be a non-positive number?
 - a. Depends, n can be 0, but not negative.
2. Can we use additional data structures?
 - a. Yes, assume we want the fastest overall runtime.
3. What should be the result when $n = 0$?
 - a. The result should be 0, before the first 1 in the sequence 1, 1, 2, ..., Fib(n)

Let's see some examples

Edge Case 1:

$n = 0$; Fib = 0

Output = 0

Edge Case 2:

$n = 1$; Fib = 0, 1

Output = 1

Middle Case 1:

$n = 2$; Fib = 0, 1, 1

Output = 1

Middle Case 2:

$n = 9$; Fib = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Output = 34

Approach 1: Brute Force

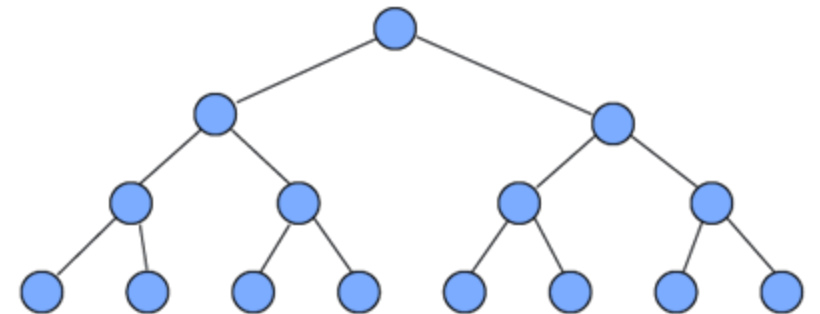
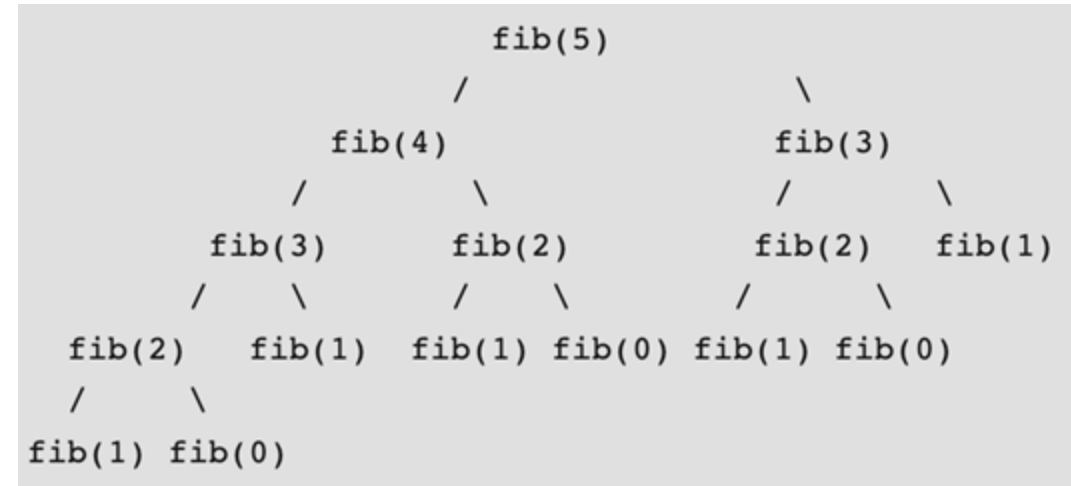
Use a recursive function to solve this:

- Starting with n and descending down, recursively return the addition of the *last* and *second last* numbers of our sequence
- End our recursion when we hit our *base case $n = 0, 1$*

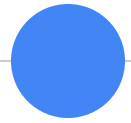


Recursive Solution

```
public static int fib(int n) {  
    if (n <= 1) {  
        return n;  
    }  
    return fib(n-1) + fib(n-2);  
}
```

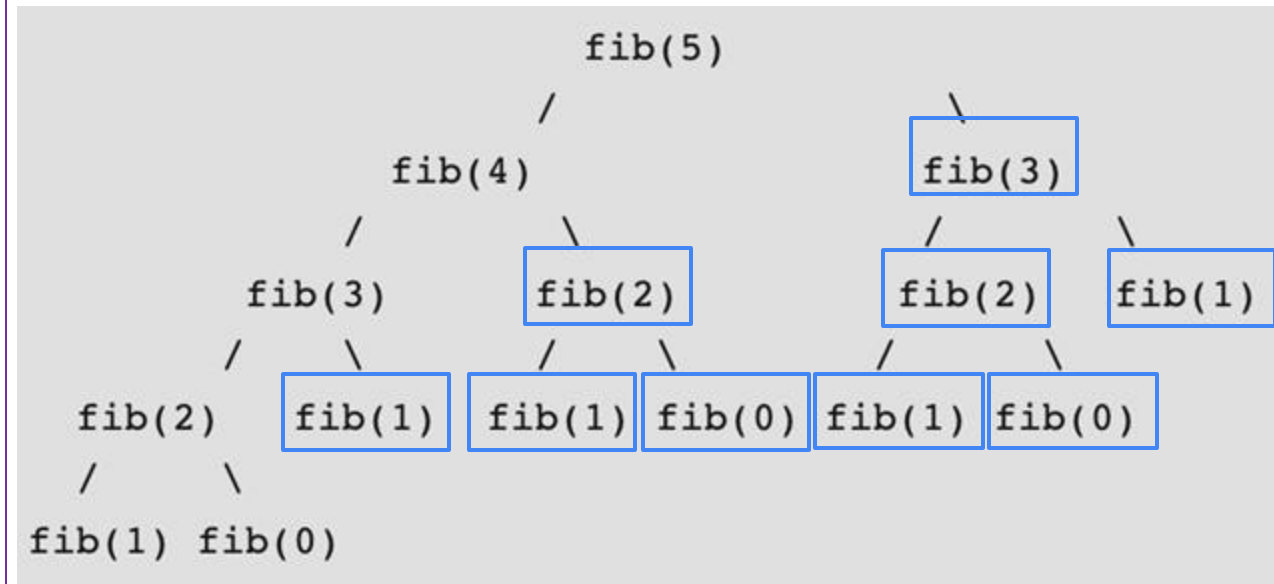


2^n nodes, each (non-leave) node makes 2 recursive calls $O(2^n)$



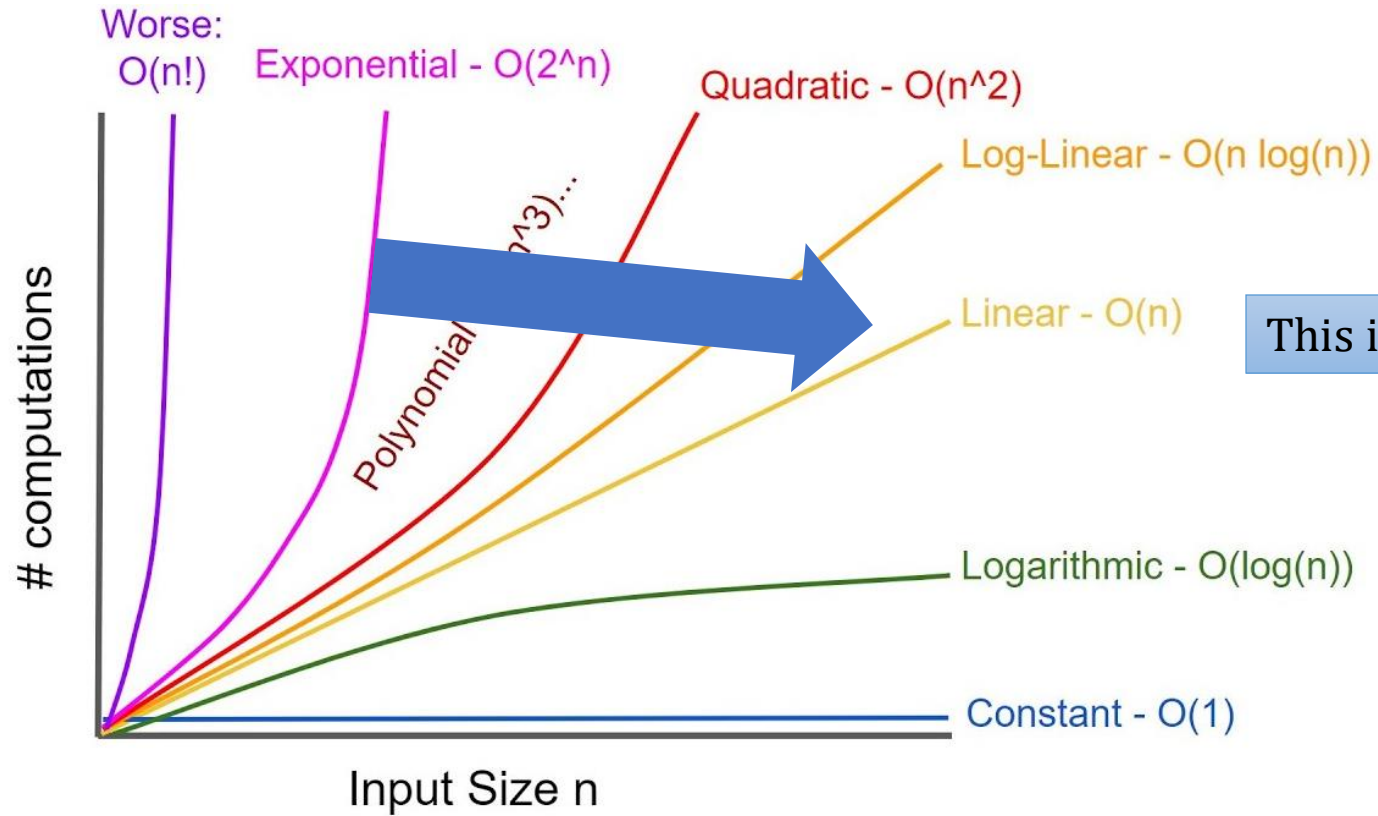
Memoized Dynamic Programming Solution

```
public int fib(int n, int[] memo) {  
    if (memo[n] != null) {  
        return memo[n];  
    } else if (n <= 1) {  
        return n;  
    } else {  
        int result = fib(n-1, memo)  
                    + fib(n-2, memo);  
        memo[n] = result;  
        return result;  
    }  
}
```



$n+1$ nodes and $2n-2$ edges gives us $O(n)$

Recap



This is a huge improvement!

Bottom-Up Solution

$\text{Memo}(i) = \begin{cases} 0 & , \text{ if } i = 0 \\ 1 & , \text{ if } i = 1 \\ \text{Memo}[1] + \text{Memo}[2] & , \text{ otherwise } \end{cases}$

$\text{Memo}(5) = \text{Memo}(4) \& \text{Memo}(3)$
 $\text{Memo}(4) = \text{Memo}(3) \& \text{Memo}(2)$
 $\text{Memo}(3) = \text{Memo}(2) \& \text{Memo}(1)$
 $\text{Memo}(2) = \text{Memo}(1) \& \text{Memo}(0)$

DP - Bottom-Up

DP - Memoized



Bottom-Up Dynamic Programming Solution



```
def fib(n):  
    f = [0] * (n + 1)  
    f[0] = 0  
    f[1] = 1  
    for i in range(2, n + 1):  
        f[i] = f[i - 1] + f[i - 2]  
    return f[n]
```

Could we do any better?

Optimized Bottom-Up – Idea

- **Create 3 variables** to hold our *second last value*, our *last value*, and our *current value* with respect to our current term in the sequence when iterating.
- Iterate in a for-loop until we hit *term n* and add our *last* & *second last values* and set them equal to our current value.
- When we exit the for-loop, we will have computed the ***Fibonacci value at n*** .

Optimized Bottom-Up – The Advantage

- Optimized with **no additional** data structure:
 - We compute the value of our current term with a fixed number of elements, $O(1)$

Reminder:

- When you are computing a value in a sequence in an interview, think about using DP!

Optimized Bottom-Up - Python Code

```
☉ def fib(n):  
    a, b = 0, 1  
    if n == 0:  
        return a  
    for i in range(2, n + 1):  
        c = a + b  
        a, b = b, c  
    return b
```

LeetCode Submission - Python Code

LeetCode

Explore

Problems

Interview

New

Contest

Discuss

Store

Description

Solution

Discuss (999+)

Submissions

Success

Details >

Runtime: **0 ms**, faster than **100.00%** of Java online submissions for Fibonacci Number.

Memory Usage: **39 MB**, less than **90.99%** of Java online submissions for Fibonacci Number.




Next challenges:

Climbing Stairs

Split Array into Fibonacci Sequence

Length of Longest Fibonacci Subsequence

N-th Tribonacci Number

Show off your acceptance:   

Time Submitted	Status	Runtime	Memory	Language
10/16/2022 18:01	Accepted	0 ms	39 MB	java

Java

Autocomplete

```
1 class Solution {
2     public int fib(int n) {
3         int a = 0, b = 1, c;
4         if (n == 0)
5             return a;
6         for (int i = 2; i <= n; i++) {
7             c = a + b;
8             a = b;
9             b = c;
10        }
11        return b;
12    }
13 }
```



Your previous code was restored from your local storage. [Reset to default](#)

Testcase

Run Code Result

Debugger 

Question Time – Q2 & Q3

```
def fib1(n):  
    if n<=1: return 1  
    else: return fib1(n-1)+fib1(n-2)
```

Brute Force

```
def fib2(n):  
    table = [0 for i in range(n+1)]  
    for i in range(n+1):  
        if i<=1: table[i]=1  
        else: table[i]=table[i-1]+table[i-2]  
    return table[n]
```

Bottom-Up DP

Q2:

- 1) fib1(44) ran in 275.69 seconds and fib2(44) ran in 0.000 seconds
- 2) fib1(44) ran in 0.000 seconds and fib2(44) ran in 275.69 seconds

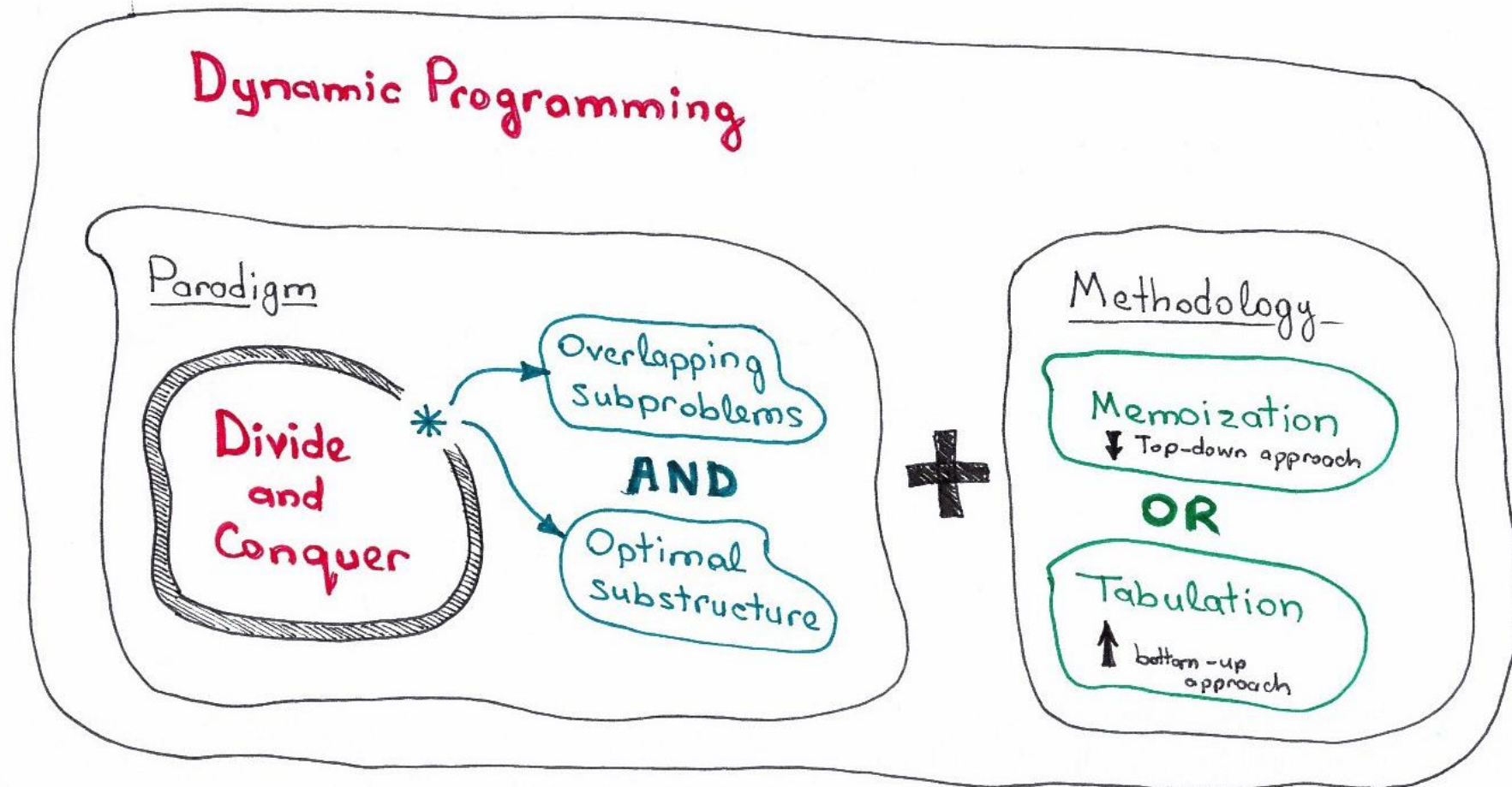
Q3:

- 1) fib1(n) is $\Theta(n^2)$ and fib2(n) is $\Theta(n)$
- 2) fib1(n) is $\Theta(n^2)$ and fib2(n) is $\Theta(n^2)$
- 3) fib1(n) is $\Theta(2^n)$ and fib2(n) is $\Theta(n)$
- 4) fib1(n) is $\Theta(2^n)$ and fib2(n) is $\Theta(n^2)$
- 5) fib1(n) is $\Theta(n!)$ and fib2(n) is $\Theta(n)$
- 6) fib1(n) is $\Theta(n!)$ and fib2(n) is $\Theta(n^2)$

Think about this!



Dynamic Programming



The “rod-cutting” problem

Given a rod of length n inches and a table of prices p_i for $i = 1, 2, 3, \dots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

If the price p_n for a rod of length n is large enough, an optimal solution might require no cutting at all.

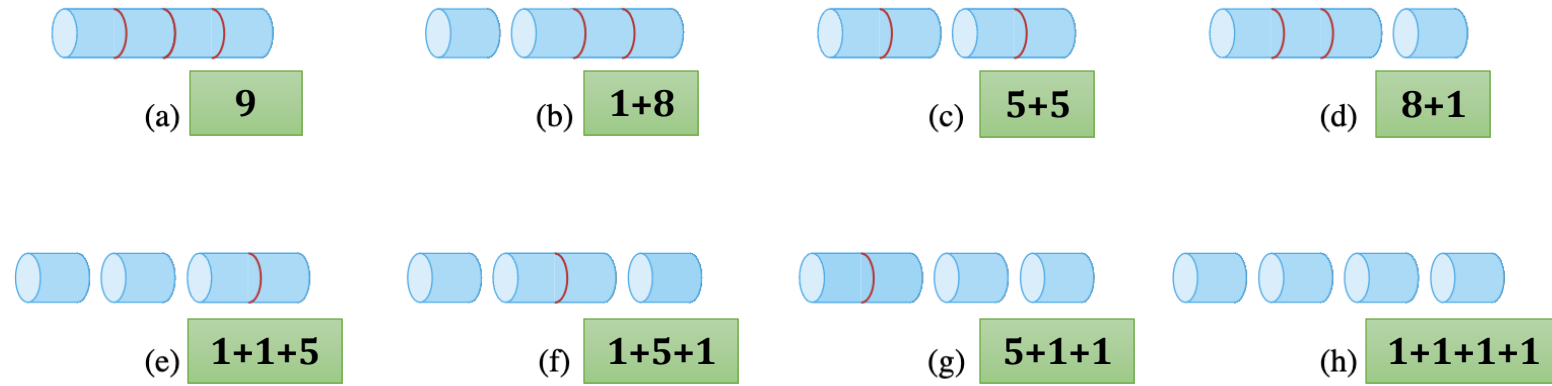
Price p_i in dollars that they charge for a rod of length i inches

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

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

The “rod-cutting” problem (continued)

- How many possible for cutting up a rod of length 4?



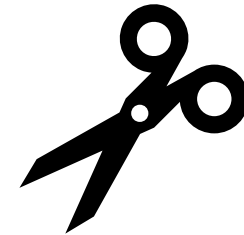
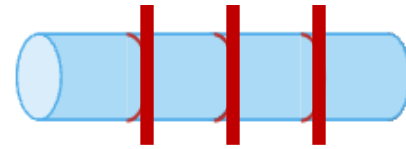
The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of the above figure.

The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

Exercise

Question

- How many possible ways for cutting up a rod of length n ?



Answer

2^{n-1} , because there are $n - 1$ places where we can choose to make cuts, and at each place, we either make a cut or we do not make a cut.

Optimal Decomposition

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

The optimal decomposition of the rod into pieces of lengths i_1, i_2, \dots, i_k provides maximum corresponding revenue $r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}$

r_1	=	1	from solution 1 = 1 (no cuts) ,
r_2	=	5	from solution 2 = 2 (no cuts) ,
r_3	=	8	from solution 3 = 3 (no cuts) ,
r_4	=	10	from solution 4 = 2 + 2 ,
r_5	=	13	from solution 5 = 2 + 3 ,
r_6	=	17	from solution 6 = 6 (no cuts) ,
r_7	=	18	from solution 7 = 1 + 6 or 7 = 2 + 2 + 3 ,
r_8	=	22	from solution 8 = 2 + 6 ,
r_9	=	25	from solution 9 = 3 + 6 ,
r_{10}	=	30	from solution 10 = 10 (no cuts) .

Basic Approach

In a related, but slightly simpler, way to arrange a recursive structure for the rod-cutting problem, let's view a decomposition as consisting of a **first piece of length i cut off** the left-hand end, and then a right-hand **remainder of length $n - i$** . Only the remainder, and not the first piece, may be further divided.

Think of every decomposition of a length- n rod in this way: **as a first piece followed by some decomposition of the remainder.**

Then we can express the solution with no cuts at all by saying that the first piece has size $i = n$ and revenue p_n and that the remainder has size 0 with corresponding revenue $r_0 = 0$. We thus obtain the following simpler version of equation:

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

Recursive top-down solution

CUT-ROD(p, n)

```
1  if  $n == 0$   
2      return 0  
3   $q = -\infty$   
4  for  $i = 1$  to  $n$   
5       $q = \max \{q, p[i] + \text{CUT-ROD}(p, n - i)\}$   
6  return  $q$ 
```

100

Figure:

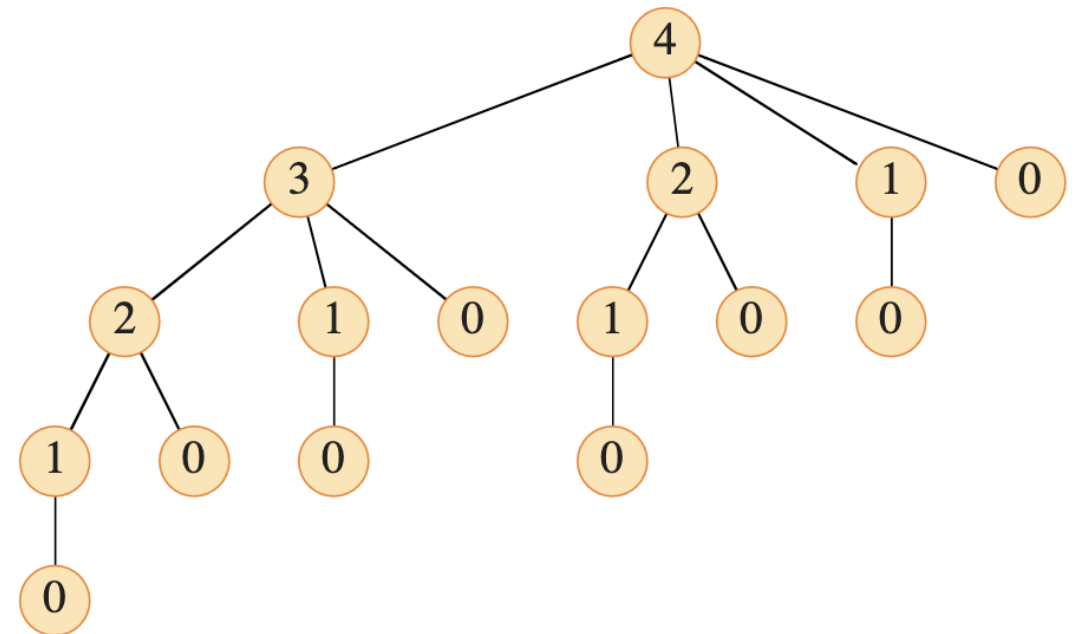
Naïve Solution (Recursive top-down)

For $n = 0$, this holds since $2^0 = 1$.

For $n > 0$, substituting into the recurrence, we have

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

- ▶ E.g., if the first call is for $n = 4$, then there will be:
 - ▶ 1 call to CUT-ROD(4)
 - ▶ 1 call to CUT-ROD(3)
 - ▶ 2 calls to CUT-ROD(2)
 - ▶ 4 calls to CUT-ROD(1)
 - ▶ 8 calls to CUT-ROD(0)



Naïve Solution (Recursive top-down)

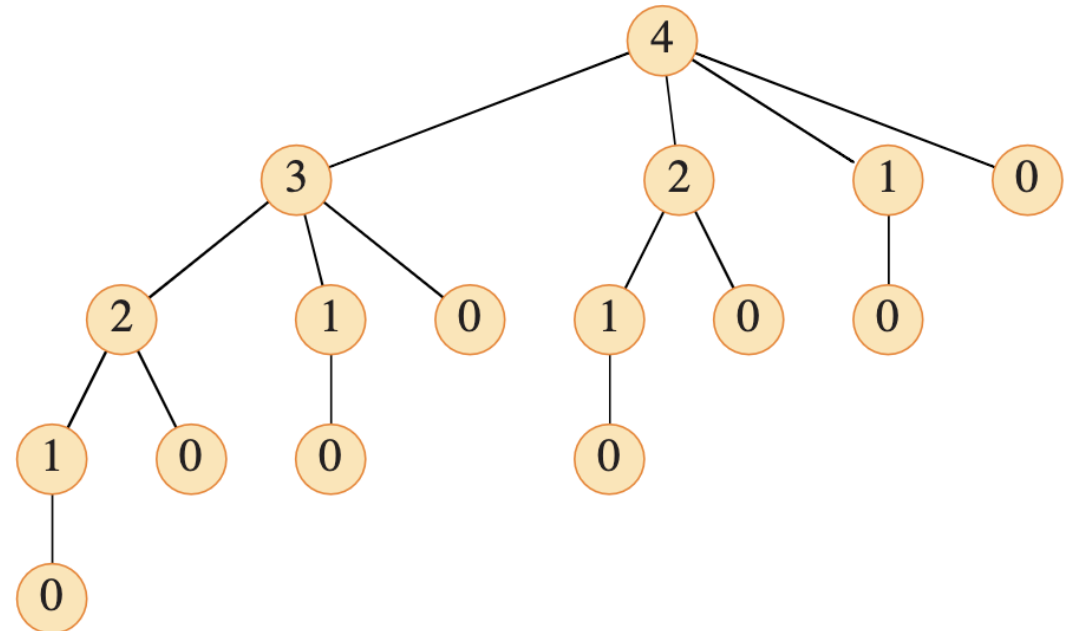
For $n = 0$, this holds since $2^0 = 1$.

For $n > 0$, substituting into the recurrence, we have

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^n T(n-i) = 1 + \sum_{j=0}^{n-1} T(j) \\ &= 1 + \sum_{j=0}^{n-1} 2^j \\ &= 1 + \frac{1 - 2^n}{1 - 2} \\ &= 2^n \end{aligned}$$

Sum of Geometric Series:

$$\sum_{i=1}^n a_1 r^{i-1} = S_n = \frac{a_1(1 - r^n)}{1 - r}, \quad r \neq 1$$



Memoization (top-down solution)

MEMOIZED-CUT-ROD(p, n)

```
1  let  $r[0 : n]$  be a new array      // will remember solution values in  $r$ 
2  for  $i = 0$  to  $n$ 
3       $r[i] = -\infty$ 
4  return MEMOIZED-CUT-ROD-AUX( $p, n, r$ )
```

MEMOIZED-CUT-ROD-AUX(p, n, r)

```
1  if  $r[n] \geq 0$                     // already have a solution for length  $n$ ?
2      return  $r[n]$ 
3  if  $n == 0$ 
4       $q = 0$ 
5  else  $q = -\infty$ 
6      for  $i = 1$  to  $n$               //  $i$  is the position of the first cut
7           $q = \max \{q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r)\}$ 
8   $r[n] = q$                         // remember the solution value for length  $n$ 
9  return  $q$ 
```

Runtime: $\Theta(n^2)$. Each subproblem is solved exactly once, and to solve a subproblem of size i , we run through i iterations of the for loop. So the total number of iterations of the for loop, over all recursive calls, forms an arithmetic series, which produces $\Theta(n^2)$ iterations in total.

Memoized top-down solution

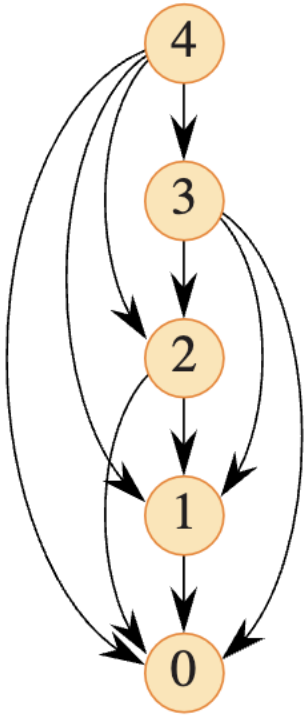


Figure: The subproblem graph for the rod-cutting problem with $n = 4$. The vertex labels give the sizes of the corresponding subproblems. A directed edge (x, y) indicates that solving subproblem x requires a solution to subproblem y . This graph is a reduced version of the recursion tree of Figure 14.3, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

In the Rod-Cutting problem, the subproblem graph has $n+1$ vertices (or nodes) and the number of Edges is given by $n+(n-1)+(n-2)+\dots+2+1$
 $= n(n+1)/2$

Here we proactively compute the solutions for smaller rods first, knowing that they will later be used to compute the solutions for larger rods.

Bottom-Up approach

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

```
1  let  $r[0:n]$  be a new array           // will remember solution values in  $r$ 
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$                        // for increasing rod length  $j$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$                    //  $i$  is the position of the first cut
6           $q = \max\{q, p[i] + r[j - i]\}$ 
7       $r[j] = q$                          // remember the solution value for length  $j$ 
8  return  $r[n]$ 
```

Reconstructing a Solution

Let's actually find the optimal way to split the rod, instead of just finding the maximum profit:

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  $n$                                 // for increasing rod length  $j$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$                                 //  $i$  is the position of the first cut
6          if  $q < p[i] + r[j - i]$ 
7               $q = p[i] + r[j - i]$ 
8               $s[j] = i$                                 // best cut location so far for length  $j$ 
9       $r[j] = q$                                         // remember the solution value for length  $j$ 
10 return  $r$  and  $s$ 

```

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

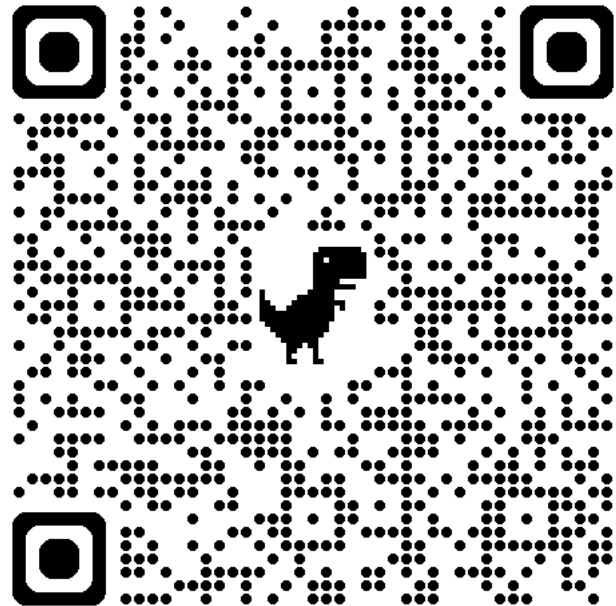
```

1   $(r, s) = \text{EXTENDED-BOTTOM-UP-CUT-ROD}(p, n)$ 
2  while  $n > 0$ 
3      print  $s[n]$            // cut location for length  $n$ 
4       $n = n - s[n]$          // length of the remainder of the rod

```

Rod Cutting Demo

- Let's see the Rod Cutting solutions in action:
 - https://colab.research.google.com/drive/14fvZPaauecU_ZyvybCM2N5qu_YDjTZg4?usp=sharing



Let's review the implementation!



10 Minutes

Longest Common Subsequence



A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, cytosine, guanine, and thymine.

For example, the DNA of one organism may be $S_1 = \text{ACCGGTCGAGTGCGGGGAAGCCGGCCGAA}$, and the DNA of another organism may be $S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

We can compare two strands of DNA to determine how “similar” the two strands are, as some measure of how closely related the two organisms are. We can, and do, define similarity in many different ways.:

- **Finding a third strand S_3** in which the bases in S_3 appear in each of S_1 and S_2 . These bases must appear in the same order, but not necessarily consecutively.

Longest Common Subsequence (LCS)

Generate all the possible subsequences & find the longest among them that is present in both strings using recursion.

Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .
2. If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y .
3. If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is an LCS of X and Y_{n-1} .

Longest Common Subsequence (LCS)

Recursive Solution:

$$\underline{c[i, j]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max \{c[i, j - 1], c[i - 1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

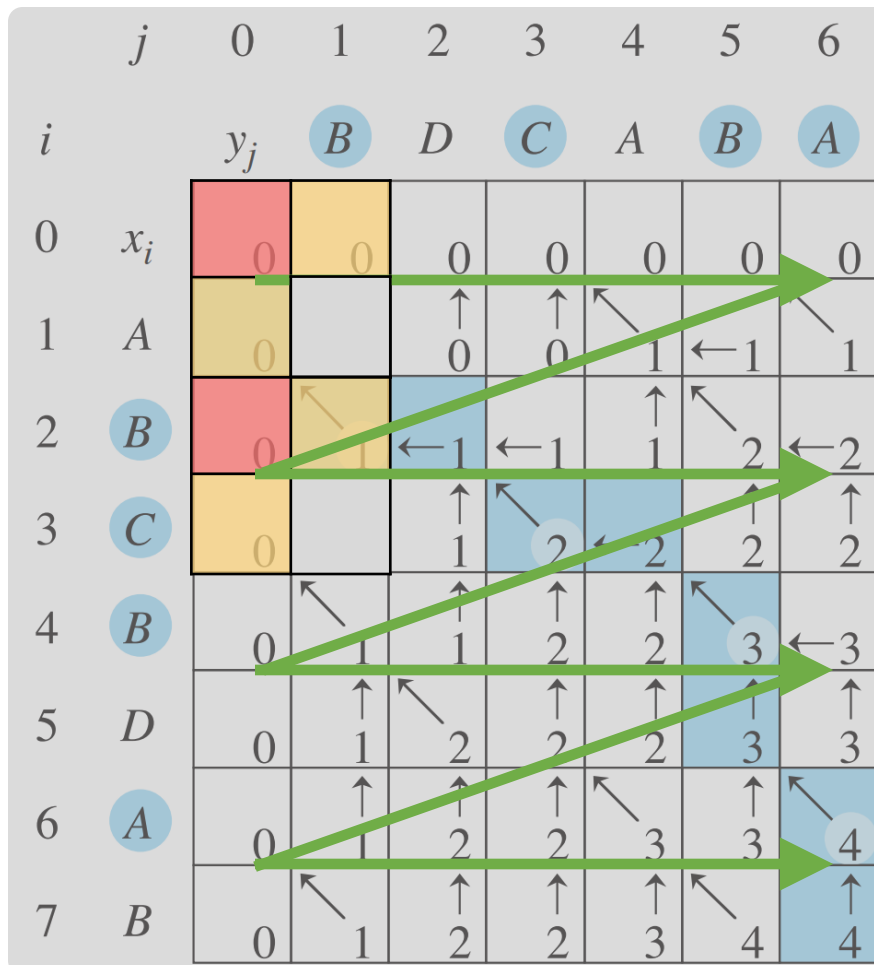
$c[i, j]$ is the lengths of the longest common subsequences between the first i elements of X and the first j elements of Y .

Proactive Solution:

	?

Longest Common Subsequence (LCS)

		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
0	x_i								
1	A								
2	B								
3	C								
4	B								
5	D								
6	A								
7	B								



The table shows the computation of the Longest Common Subsequence (LCS) between two strings X and Y. The rows represent string X (A, B, C, B, D, A, B) and the columns represent string Y (B, D, C, A, B, A). The table is filled with values from 0 to 4, representing the length of the LCS for each subproblem. Green arrows trace the path of the LCS, starting from the bottom-right cell (7,6) and moving towards the top-left, indicating the sequence of characters in the LCS: A, B, C, B.

LCS-LENGTH(X, Y, m, n)

```

1 let  $b[1 : m, 1 : n]$  and  $c[0 : m, 0 : n]$  be new tables
2 for  $i = 1$  to  $m$ 
3    $c[i, 0] = 0$ 
4 for  $j = 0$  to  $n$ 
5    $c[0, j] = 0$ 
6 for  $i = 1$  to  $m$            // compute table entries in row-major order
7   for  $j = 1$  to  $n$ 
8     if  $x_i == y_j$ 
9        $c[i, j] = c[i - 1, j - 1] + 1$ 
10       $b[i, j] = \nwarrow$ 
11    elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
12       $c[i, j] = c[i - 1, j]$ 
13       $b[i, j] = \uparrow$ 
14    else  $c[i, j] = c[i, j - 1]$ 
15       $b[i, j] = \leftarrow$ 
16 return  $c$  and  $b$ 
```

PRINT-LCS(b, X, i, j)

```

1 if  $i == 0$  or  $j == 0$ 
2   return           // the LCS has length 0
3 if  $b[i, j] == \nwarrow$ 
4   PRINT-LCS( $b, X, i - 1, j - 1$ )
5   print  $x_i$        // same as  $y_j$ 
6 elseif  $b[i, j] == \uparrow$ 
7   PRINT-LCS( $b, X, i - 1, j$ )
8 else PRINT-LCS( $b, X, i, j - 1$ )
```

LCS - Demo

Consider the two words **ALGORITHMS** and **LOGARITHMIC**.

Represent these two words as a sequence of letters. You will notice that these two sequences contain many common subsequences, such as OTHM, highlighted below:

AL**GO**RITHMS L**GO**RITHMIC

To Do:

- Go to <http://lcs-demo.sourceforge.net/>
- Make sure you set your “max size” to 11 so that you can run the program on
- ALGORITHMS (10 letters) and LOGARITHMIC (11 letters)
- The correct answer is 7 (e.g. LGRITHM or LORITHM).

Next Week: Greedy Programming

 Required Prep:

Option 1:

From the 4th edition of CLRS:

Read only Chapter 15 of the Course Textbook - Greedy Algorithms

Make sure you do a close and careful reading of Section 15.1, Section 15.2, and Section 15.4 (Offline Caching).

You are welcome to skim/skip the remaining sections of the chapter: Section 15.3

Option 2:



Just Watch: <https://youtu.be/ctcpCIMXX1w>



Questions?