We’ll be starting shortly!

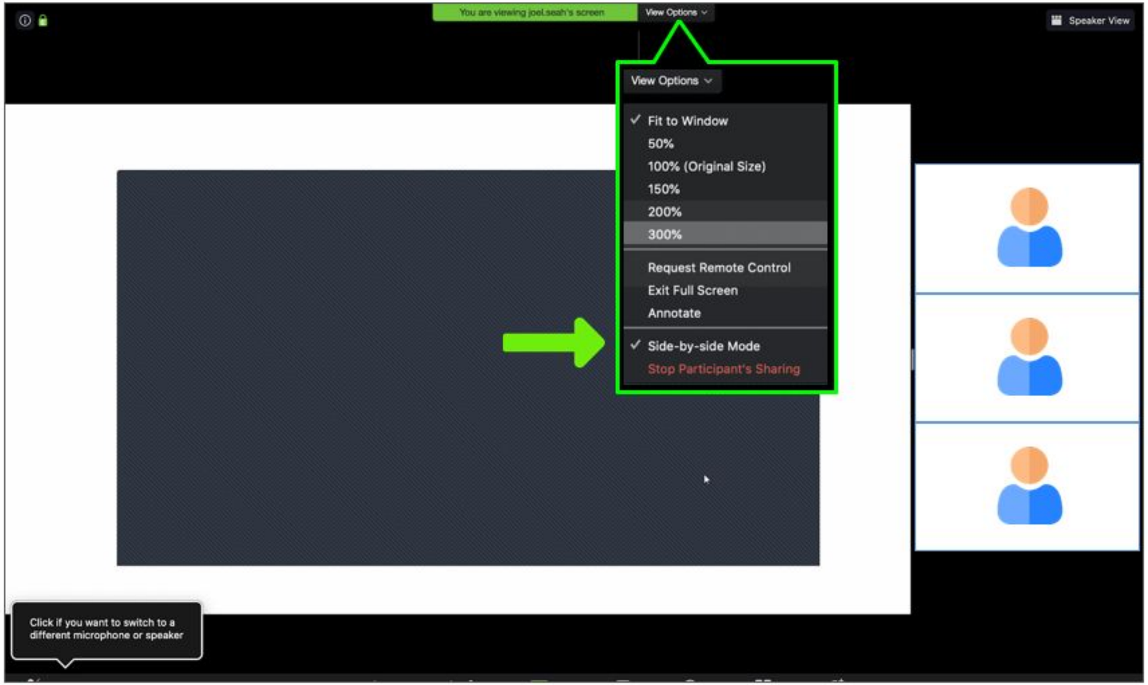
To help us run the workshop smoothly, please kindly:

- Switch off screen sharing and mute your microphone
- Submit all questions using the Q&A function
- If you have an urgent request, please use the “Raise Hand” function

Thank you!

Rocket Academy

Using Zoom: People & Slides



Side-By-Side Mode

- When sharing screen (slide share)
- With small thumbnails of people on the sidebar

STEPS:

1. View Options
2. Side-By-Side Mode

Rocket Academy

Shopee CODE LEAGUE 2020

DP: Speaker Notes

Overall Talking Points: What, Why, How

- introduce concepts
- contextualize concepts with examples
- up to **YOU** to practice! (especially for DP, this is the only way to master)

1. Overview Dynamic Programming(DP): What and Why
    a. Definition:
        i. “**Solving Problems** by incrementally solving for and **re-using** solutions of its **subproblems**”
        ii. A category of algorithms that solves certain types of problems
        iii. Two Defining Properties:
            1. **Optimal** Substructure: F(final) = G(F(small)...)
            2. **Overlapping** Subproblems: Can get **significant speed-up** from brute force
    b. Motivation:
        i. When applicable, often lead to performant solutions(correct solutions, no TLE: ‘time limit exceeded’)
        ii. goto reason #i, more weapons in your algo toolbox :D
    c. Example: Fibonacci Sequence
2. Solving Problems with DP
    a. Problem Solving Approaches (your choice, but take note of the difference):
        i. Top-Down: Recursion (exhaustive search + memoization)
        ii. Bottom-Up: Tabulation (building a table)
        iii. Top-Down vs Bottom-Up:
            1. Most the problems could be solved by either

Rocket Academy

Introduction to Dynamic Programming

Liang Qiao, **Rocket Academy**

Slides: https://tinyurl.com/y7pf7vkn
Register: https://rocketacademy.co/scl-slides

Rocket Academy

Overview

1. What is dynamic programming (DP)
2. Why/When to use dynamic programming (DP)
3. Problem solving approaches of DP
4. Problem solving strategy

Rocket Academy

Goal: Be able to solve basic DP problems

Rocket Academy

What is Dynamic Programming?

1. "**Solving Problems** by incrementally solving for and **re-using** solutions of its **subproblems**"
2. **Two** Defining Properties:
   a. **Optimal** Substructure: F(final) = G(F(small)...)
   b. **Overlapping** Subproblems

Rocket Academy

Introducing Fibonacci Sequence

$$f_n = f_{n-1} + f_{n-2}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 …

Question: how to write program to produce fib(n) where n is the index?

Rocket Academy

Introducing Fibonacci Sequence

```
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n-1) + fib(n-2)
```

What's the time complexity?
(you could answer in chat)

Rocket Academy

Naive Fibonacci: time complexity

Recurrence:

T(n) = T(n-1) + T(n-2)

>= 2*T(n-2) >= 2*2*T(n-2-2)

>= 2*2*2*T(n-2-2-2) so on and so forth ...

>= 2^(n/2)

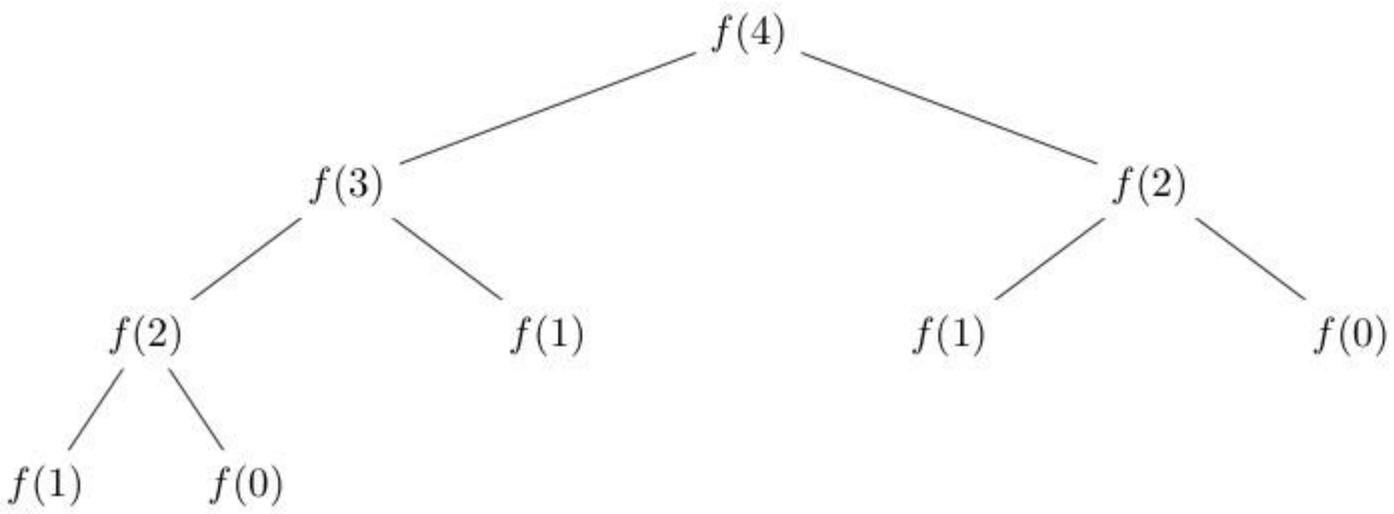
Exponential is NOT Good! Too SLOW!

Rocket Academy

Why Dynamic Programming?

Rocket Academy

Speeding up Fibonacci Sequence



Rocket Academy

Speeding up Fibonacci Sequence

```
def solve_fib(n):
  memo = {}

  def fib(n):
    if n in memo:
      return memo[n]
    if n == 0 or n == 1:
      return n
    memo[n] = fib(n-1) + fib(n-2)
    return memo[n]

  return fib(n)
```

Now it’s linear time complexity O(n)!

Rocket Academy

Dynamic Programming Fibonacci: time complexity O(n) !

T(n) = T(n-1) + T(n-2)

>= T(n-1) + O(1) **<- Why?**

>= [T(n-2)+T(n-3)] + O(1)

>= T(n-2) + O(1)*2

>= T(n-3) + O(1)*3

>= ...

>= T(n-n) + O(1)*n

>= O(n) **<- Fast Enough!**

Rocket Academy

Speeding up Fibonacci Sequence: Iterative approach

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

  return fib[n]
```

Why linear time complexity O(n)?

Rocket Academy

Speeding up Fibonacci Sequence: Iterative approach

```
def solve_fib(n):
  if n == 0 or n == 1:
    return n
  fib_n2, fib_n1 = 0, 1
  for i in range(n-1):
    fib = fib_n2 + fib_n1
    fib_n2, fib_n1 = fib_n1, fib

  return fib
```

Why linear time complexity O(n)?

Rocket Academy

Recall: What is Dynamic Programming

1. "**Solving Problems** by incrementally solving for and **re-using** solutions of its **subproblems**"
2. **Two** Defining Properties:
   a. **Optimal** Substructure: **F(final) = G(F(small)...)**
   b. **Overlapping** Subproblems

Rocket Academy

Problem Solving with Dynamic Programming

1. Problem Solving Approaches:
    a. Top-down: Recursion + Memoization
    b. Bottom-up: Tabulation
2. Problem Solving Strategy

Rocket Academy

Speeding up Fibonacci Sequence

```
def fib_topdown(n):
  memo = {}

  def fib(n):
    if n in memo:
      return memo[n]
    if n == 0 or n == 1:
      return n
    memo[n] = fib(n-1) + fib(n-2)
    return memo[n]

  return fib(n)
```

```
def fib_bottomup(n):
  if n == 0 or n == 1:
    return n

  fib_n2, fib_n1 = 0, 1
  for i in range(n-1):
    fib = fib_n2 + fib_n1
    fib_n2, fib_n1 = fib_n1, fib

  return fib
```

Rocket Academy

Problem Solving Approaches

Top-down

VS

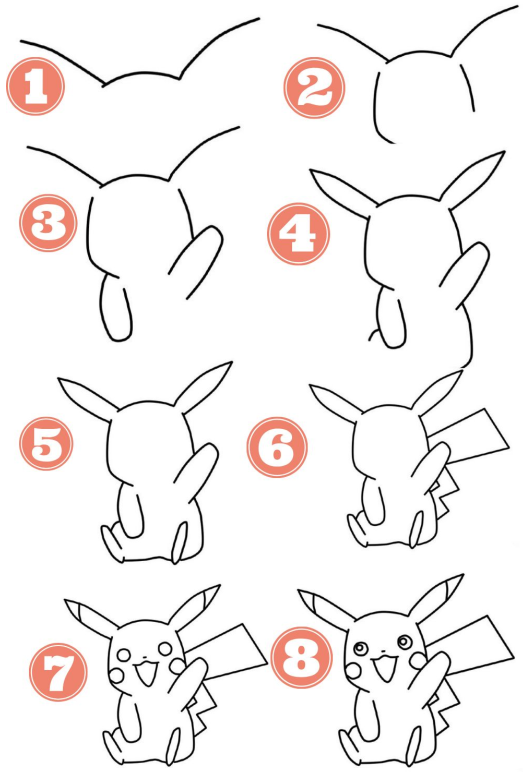
Bottom-up

More intuitive, because of **recursion + memo**

More efficient, and easier to optimize for space complexity

Rocket Academy

Problem Solving Strategy: steps to follow



Rocket Academy

Problem Solving Strategy: top down

i. Step 1: Recognize DP Problems
   1. weak signal: problems asking for min / max / total number / if solution exists
   2. strong signal: **two defining properties of DP**

ii. Step 2: Find the brute force solution (recursion)

iii. Step 3: Apply memoization (save the result for **reuse**)

Rocket Academy

Problem Solving Strategy: bottom up

i. Step 1: Recognize DP Problems
   1. weak signal: problems asking for min / max / total number / if solution exists
   2. strong signal: **two defining properties of DP**

ii. Step 2: Define Subproblem: F(small)

iii. Step 3: Define Recurrence: F(big) = G(F(small)...)
   1. If works:
      a. Define F(initialize smallest)
      b. Incrementally solve for F(bigger)
      c. Output F(final)
   2. Else: go back to Step 2 and revise

Rocket Academy

Longest Increasing Subsequence

Given input: n numbers: x1, x2, … xn
Task: find the length of longest increasing subsequence.

e.g

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

Ans: 6

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

Rocket Academy

Top-down: Longest Increasing Subsequence

arr	0	8	4	12	2	10	6	14	1	9	5	13
LIS	1	2	2	3	?							

i. Step 2: Find the brute force solution (recursion)

ii. Step 3: Apply memoization (save the result for **reuse**)

Rocket Academy

Top-down:
Longest Increasing Subsequence

```
def solve_lis(arr, i, prev):
    if i == len(arr):
        return 0

    excl_i = solve_lis(arr, i + 1, prev)
    incl_i = 0
    if arr[i] > prev:
        incl_i = 1 + solve_lis(arr, i + 1, arr[i])

    return max(incl_i, excl_i)
```

time complexity?

Rocket Academy

Top-down:
Longest Increasing Subsequence

```
def solve(arr):
    memo = {}
    def solve_lis(arr, i, prev):
        if i == len(arr):
            return 0
        if (i, prev) in memo:
            return memo[(i, prev)]

        excl_i = solve_lis(arr, i + 1, prev)
        incl_i = 0
        if arr[i] > prev:
            incl_i = 1 + solve_lis(arr, i + 1, arr[i])

        memo[(i, prev)] = max(incl_i, excl_i)
        return memo[(i, prev)]
    return solve_lis(arr, 0, 0)
```

time complexity?

Rocket Academy

Bottom-up:
Longest Increasing Subsequence

arr	0	8	4	12	2	10	6	14	1	9	5	13
LIS	1	2	2	3	?							

i. Step 2: Define Subproblem:

   LIS(i) = length of the longest increase subsequence

ii. Step 3: Define Recurrence: F(big) = G(F(small)...)

   How to express LIS(i) with LIS(i-1) or LIS(smallers)?

   1. If works:
      a. Define F(initialize smallest)
      b. Incrementally solve for F(bigger)
      c. Output F(final)
   2. Else: go back to Step 2 and revise

Rocket Academy

Bottom-up:
Longest Increasing Subsequence

arr	0	8	4	12	2	10	6	14	1	9	5	13
LIS	1	2	2	3	?							

i. Step 2: Define Subproblem:

  LIS(i) = length of the longest increase subsequence, **that end with arr(i)**

ii. Step 3: Define Recurrence: F(big) = G(F(small)...)

  How to express LIS(i) with LIS(i-1) or LIS(smallers)?

  LIS(n) = max(1+LIS(i) for i from 1..n-1)

1. If works:
   a. Define F(initialize smallest)
   b. Incrementally solve for F(bigger)
   c. Output F(final)
2. Else: go back to Step 2 and revise

Rocket Academy

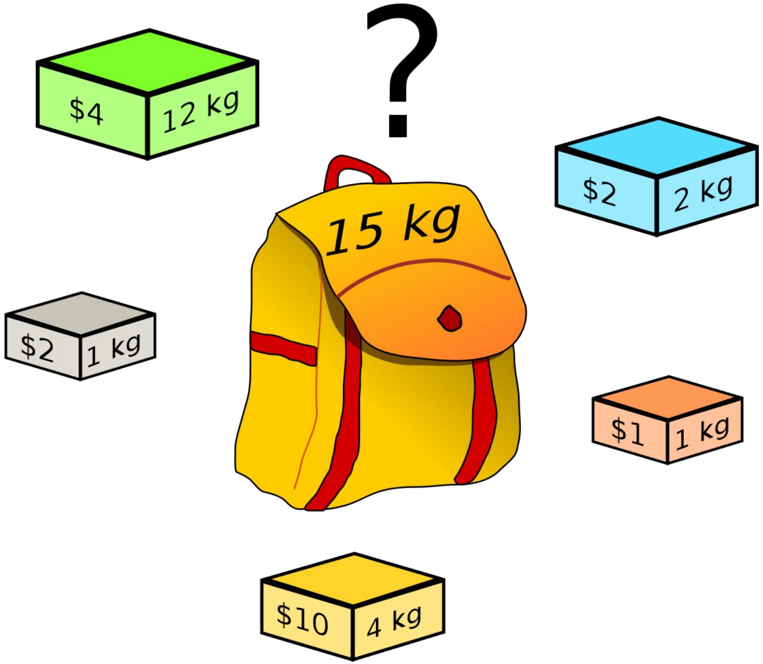
Bottom-up:
Longest Increasing Subsequence

```
def solve_lis(arr):
    n = len(arr)
    lis = [1]*n

    for i in range(1, n):
        for j in range(0, i):
            if arr[i] > arr[j]:
                lis[i] = max(lis[i], lis[j]+1)

    return max(lis)
```

time complexity?

Rocket Academy

0-1 Knapsack Problem



Longest Common Subsequence

0,1, 2, 9, 6, 1, 9, 5, 11, 7, 15, 7
0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

Ans: 0, 2, 6, 9, 11, 15.

Rocket Academy

Edit Distance

0,1, 2, 9, 6, 1, 9, 5, 11, 7, 15, 7
0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

Ans: 0, 2, 6, 9, 11, 15.

Rocket Academy

More Examples: LIS, Knapsack, LCS, Edit Distance

1. Longest Increasing Subsequence (LIS)
2. Knapsack problems
3. Longest Common Subsequence (LCS)
4. Edit Distance
5. More on https://leetcode.com/

Rocket Academy

More Learning Resources on DP

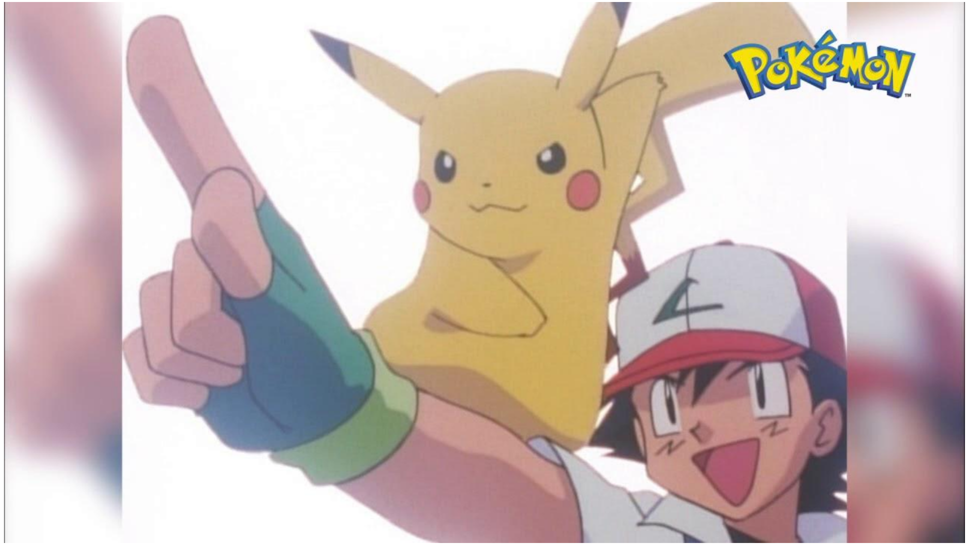
1. Introduction to Algorithm (MIT)
2. Introduction to Graduate Algorithm (Georgia Tech)
3. Dynamic Programming (Stanford)

Rocket Academy

Summary

1. **What is** dynamic programming (DP)
2. **Why/When** to use dynamic programming (DP)
3. DP problem solving **approaches** (topdown vs bottomup)
4. Problem solving strategy (steps i, ii, iii)

Now Practice Practice and Practice!



Rocket Academy

Introduction to Dynamic Programming

Liang Qiao, **Rocket Academy**

Slides: https://tinyurl.com/y7pf7vkn
Register: https://rocketacademy.co/scl-slides

Rocket Academy

Your Feedback Matters!

https://techatshopee.formstack.com/forms/shopeecodeleague_workshopfeedbackform

Get Slides: https://rocketacademy.co/scl-slides

Rocket Academy