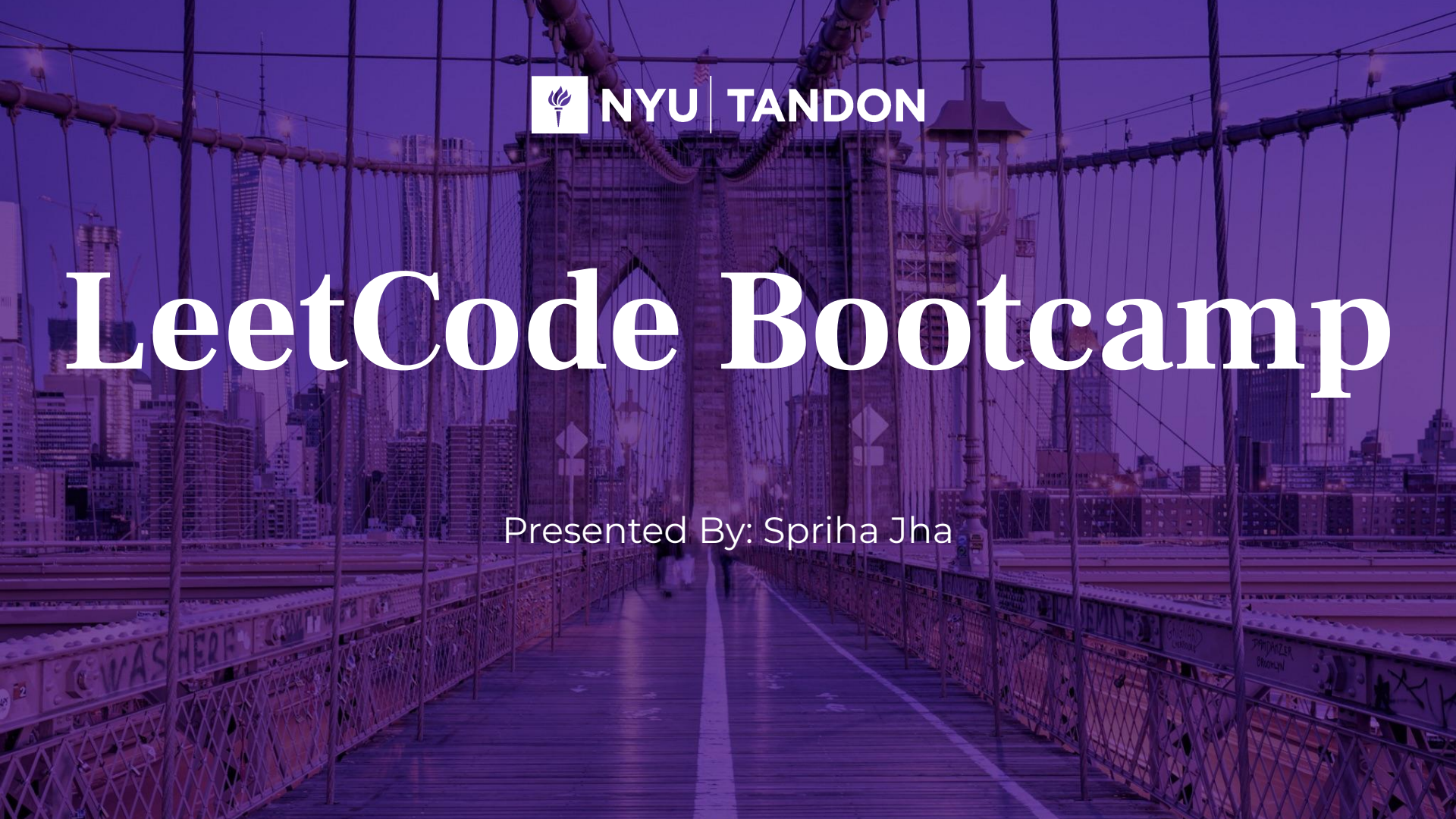

NYU TANDON
LeetCode Bootcamp
Presented By: Spriha Jha

Session Outline

01. Introduction to Dynamic Programming

02. Problem Sets

03. Debrief & Q/A

Dynamic Programming

- Dynamic programming amounts to **breaking down an optimization problem** into simpler sub-problems, and **storing the solution to each sub-problem** so that each sub-problem is only solved once.
- It's a useful technique for optimization problems, those problems that seek the maximum or minimum solution given certain constraints, because it looks through all possible sub-problems and never recomputes the solution to any sub-problem.
- Sub-problems build on each other in order to obtain the solution to the original problem.

Steps

Step 1: Identify the sub-problem in words.

Step 2: Write out the sub-problem as a recurring mathematical decision.

a. What decision do I make at every step?

b. If my algorithm is at step i, what information would it need to decide what to do in step i+1? (And sometimes: If my algorithm is at step i, what information did it need to decide what to do in step i-1?)

Step 3: Solve the original problem using Steps 1 and 2.

Step 4: Determine the dimensions of the memoization array and the direction in which it should be filled.

Steps

1. **How to recognize a DP problem?**

   Can the problem solution be expressed as a function of solutions to similar smaller problems?

2. **Identify problem variables:**

   Next, we need to express the problem in terms of the function parameters and see which of those parameters are changing.

3. **Clearly express the recurrence relation:**

   Assuming you have computed the subproblems, how would you compute the main problem?

4. **Identify the base cases:**

   A base case is a subproblem that doesn't depend on any other subproblem.

5. **Decide if you want to implement it iteratively or recursively:**

   Stack overflow issues are typically a deal breaker.

6. **Add memoization:**

- It is used for storing the results of expensive function calls and returning the cached result when the same inputs occur again.
- Store your function result into your memory before every return statement.
- Look up the memory for the function result before you start doing any other computation.

Realizing a DP problem:

Any problem has two important attributes that let us know it should be solved by dynamic programming.

- First, the question is asking for the maximum or minimum of something.
- Second, we have to make decisions that may depend on previously made decisions, which is very typical of a problem involving subsequences.

As we go through the input, each **decision** we must make is simple: is it worth it to consider this number?

Framework to solve DP:

Typically, dynamic programming problems can be solved with three main components. If you're new to dynamic programming, this might be hard to understand but is extremely valuable to learn since most dynamic programming problems can be solved this way.

First, we need some function or array that represents the answer to the problem from a given state. For many solutions on LeetCode, you will see this function/array named "dp". This array needs to represent the answer to the problem for a given state. The "state" is one-dimensional since it can be represented with only one variable - the index i.

Second, we need a way to transition between states, such as dp[5] and dp[7]. This is called a recurrence relation and can sometimes be tricky to figure out. Let's say we know dp[0], dp[1], and dp[2]. How can we find dp[3] given this information?.

The **third** component is the simplest: we need a base case.

PART 02

Problem Sets

Steps to approach the question:

Understand the problem

Take time to carefully read through the problem from start to finish is critical in finding the correct and complete solution to the problem in hand.

Code your solution

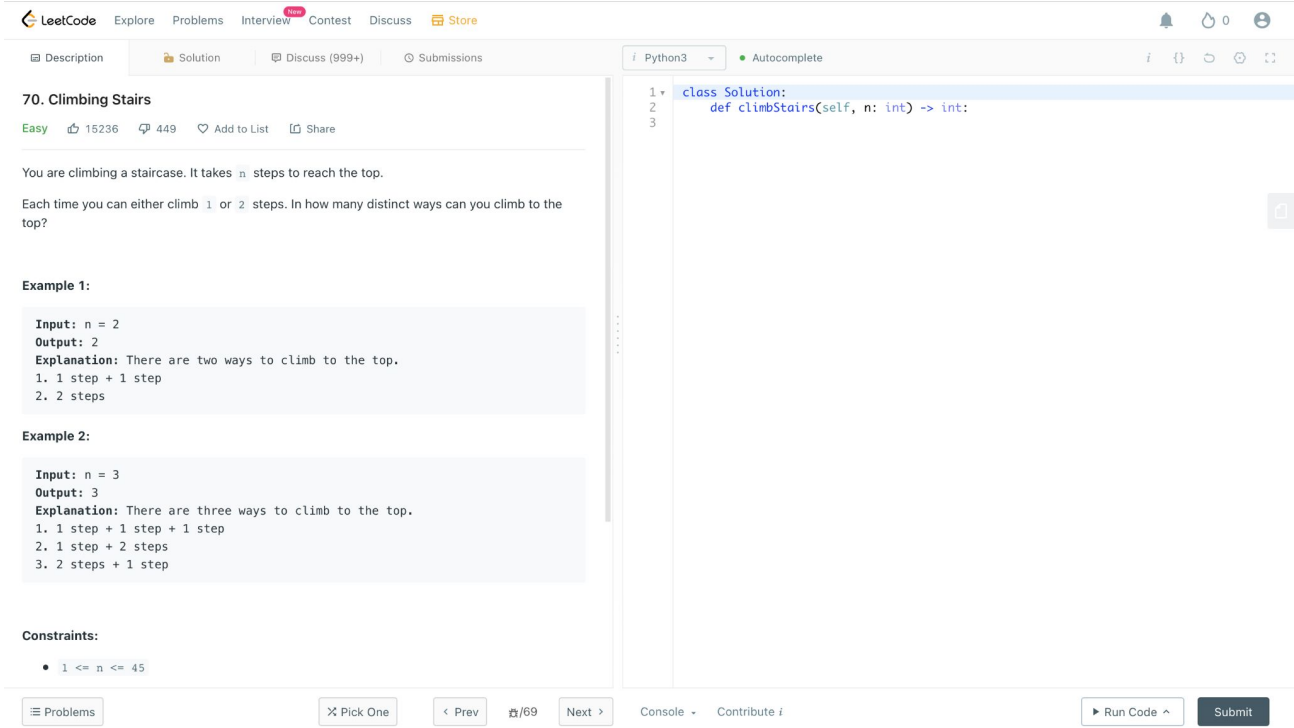
Map out your solution before you write any code. Avoid too much time trying to find the perfect solution. Validate your solution early and often.

Manage your time

Don’t forget, you have multiple questions to complete within a said time. Make sure you allocate enough time to carefully consider all problems.

PART 02

Problem 1: Climbing Stairs



NYU TANDON

Approach: Dynamic Programming

```
def climbStairs(self, n: int) -> int:
    if n <= 2:
        return n

    dp = [0]*(n+1)
    dp[1]= 1
    dp[2] = 2

    for i in range(3,n+1):
        dp[i] = dp[i-1] +dp[i-2]

    return dp[n]
```

Algorithm

As we can see this problem can be broken into subproblems, and it contains the optimal substructure property, we can use dynamic programming to solve this problem.

One can reach *i* step in one of the two ways:

1. Taking a single step from (*i*−1) step.
2. Taking a step of 2 from (*i*−2) step.

Let *dp*[*i*], denotes the number of ways to reach on *i* step:

$dp[i]=dp[i-1]+dp[i-2]$

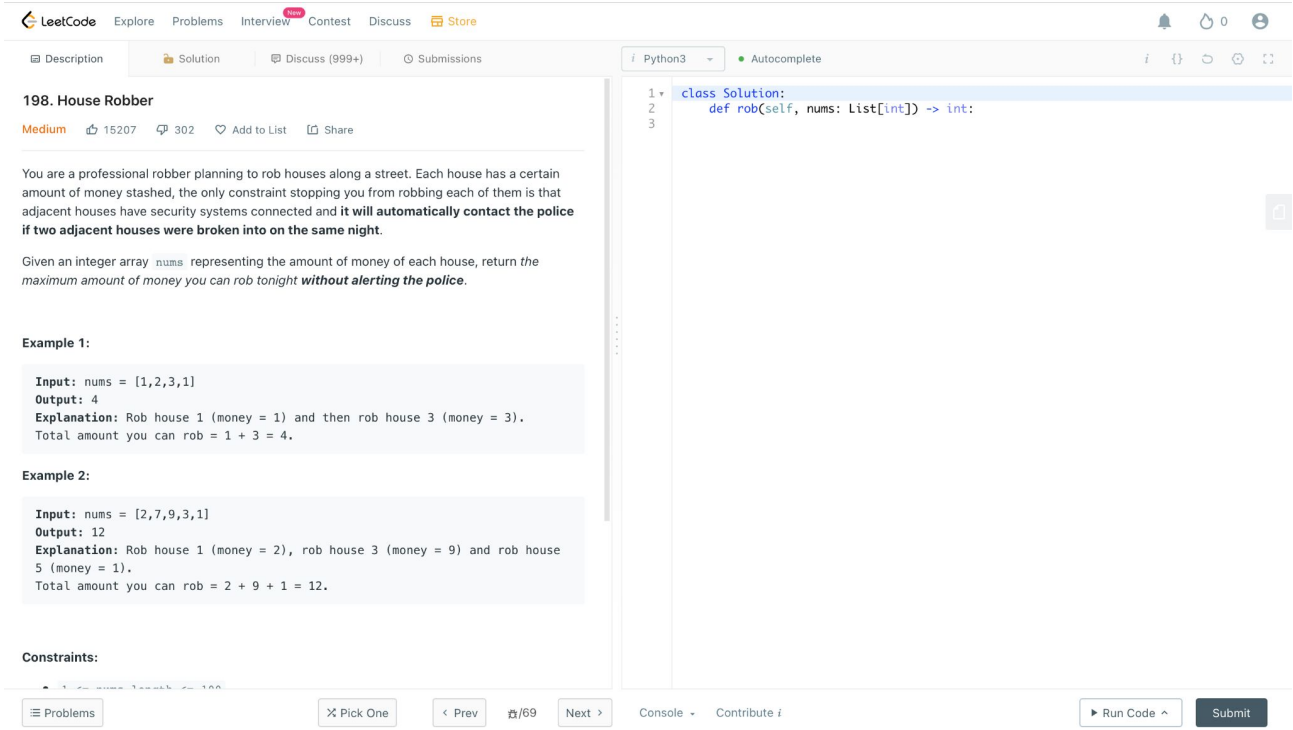
Complexity Analysis

Time complexity : $O(n)$, single loop upto n.

Space complexity : $O(n)$, dp array of size n is used.

PART 02

Problem 2: House Robber



NYU TANDON

Approach: DP

```
def rob(self, nums: List[int]) -> int:

    # Special handling for empty case.
    if not nums:
        return 0

    maxRobbedAmount = [None for _ in range(len(nums) + 1)]
    N = len(nums)

    # Base case initialization.
    maxRobbedAmount[N], maxRobbedAmount[N - 1] = 0, nums[N - 1]

    # DP table calculations.
    for i in range(N - 2, -1, -1):

        # Same as recursive solution.
        maxRobbedAmount[i] = max(maxRobbedAmount[i + 1],
maxRobbedAmount[i + 2] + nums[i])

    return maxRobbedAmount[0]
```

Algorithm

1. We define a table which we will use to store the results of our sub-problems. Let's call this table maxRobbedAmount.
2. We set maxRobbedAmount[N] to 0 since this means the robber doesn't have any houses left to rob, thus zero profit. Additionally, we set maxRobbedAmount[N - 1] to nums[N - 1] because in this case, there is only one house to rob which is the last house. Robbing it will yield the maximum profit.
3. We iterate from N - 2 down to 0 and we set maxRobbedAmount[i] = max(maxRobbedAmount[i + 1], maxRobbedAmount[i + 2] + nums[i]). Note that this is the same as the recursive formulation. The only difference is that we have *already calculated the solutions to the sub-problems and we simply reuse the solutions in O(1) time when calculating the solution to the main problem*.
4. We return the value in maxRobbedAmount[0].

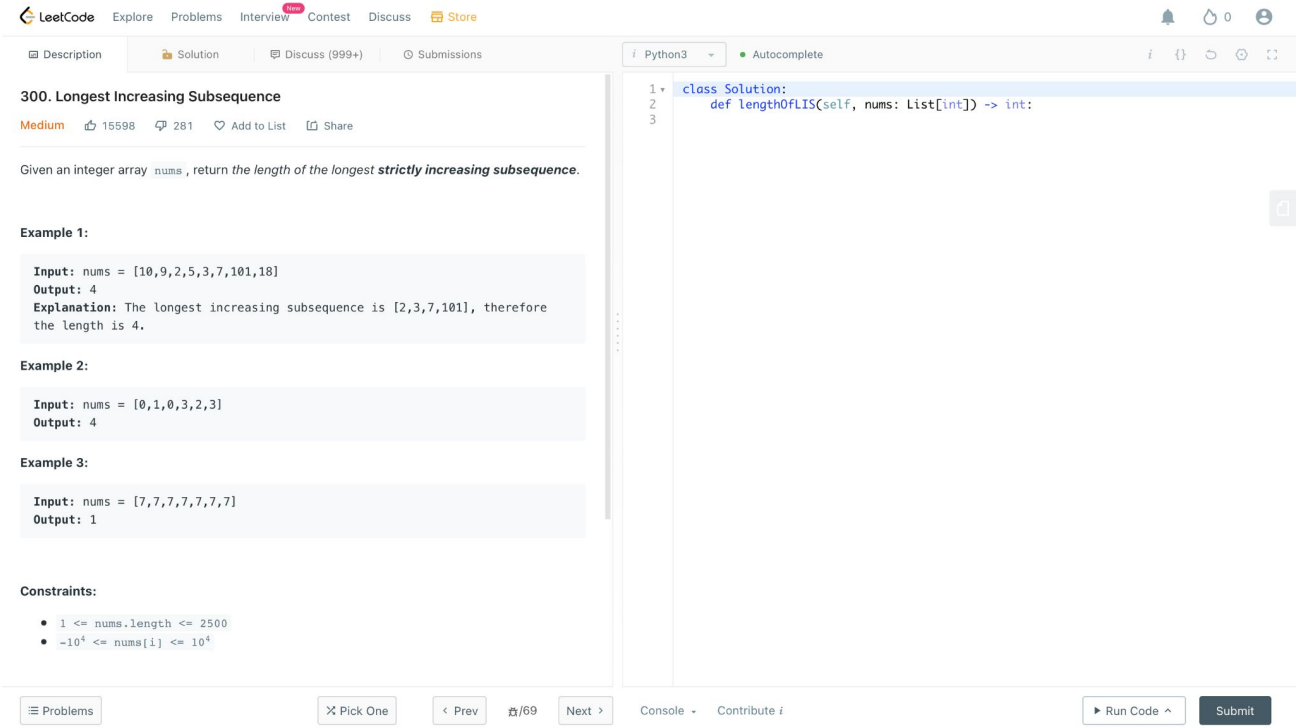
Approach: Optimal DP

```
def rob(self, nums: List[int]) -> int:
    # Special handling for empty case.
    if not nums:
        return 0

    N = len(nums)

    rob_next_plus_one = 0
    rob_next = nums[N - 1]

    # DP table calculations.
    for i in range(N - 2, -1, -1):

        # Same as recursive solution.
        current = max(rob_next, rob_next_plus_one + nums[i])

        # Update the variables
        rob_next_plus_one = rob_next
        rob_next = current

    return rob_next
```

Algorithm

1. We will make use of two variables here called robNext and robNextPlusOne which at any point will represent the optimal solution for maxRobbedAmount[i + 1] and maxRobbedAmount[i + 2]. These are the two values that we need to calculate the current value.
2. We set robNextPlusOne to 0 since this means the robber doesn't have any houses left to rob, thus zero profit. Additionally, we set robNext to nums[N - 1] because in this case there is only one house to rob which is the last house. Robbing it will yield the maximum profit.
   Note We are assuming that robNextPlusOne is the value of maxRobbedAmount[N] and robNext is maxRobbedAmount[N-1] initially.
3. We iterate from N - 2 down to 0 and set current = max(robNext, robNextPlusOne + nums[i]). Note that this is the same as the dynamic programming solution except that we are making use of our variables and not entries from the table.
4. Set robNextPlusOne = robNext.
5. Set robNext = current. Updating the two variables is important as we iterate down to 0.
6. We return the value in robNext.

Problem 3: Longest Increasing Subsequence



NYU TANDON

Approach: DP

```
def lengthOfLIS(self, nums: List[int]) -> int:
    dp = [1] * len(nums)
    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)
```

Algorithm

1. Initialize an array dp with length nums.length and all elements equal to 1. dp[i]represents the length of the longest increasing subsequence that ends with the element at index i.
2. Iterate from i = 1 to i = nums.length - 1. At each iteration, use a second for loop to iterate from j = 0 to j = i - 1 (all the elements before i). For each element before i, check if that element is smaller than nums[i]. If so, set dp[i] = max(dp[i], dp[j] + 1).
3. Return the max value from dp.

Complexity Analysis

Time complexity : $O(n^2)$, two nested loops..

Space complexity : $O(n)$.

Approach: Binary Search

```
def lengthOfLIS(self, nums: List[int]) -> int:
    sub = []
    for num in nums:
        i = bisect_left(sub, num)

        # If num is greater than any element in sub
        if i == len(sub):
            sub.append(num)

        # Otherwise, replace the first element in sub greater than or equal to num
        else:
            sub[i] = num

    return len(sub)
```

Algorithm

1. Initialize an array sub which contains the first element of nums.
2. Iterate through the input, starting from the second element. For each element num:
    a. If num is greater than any element in sub, then add num to sub.
    b. Otherwise, perform a binary search in sub to find the smallest element that is greater than or equal to num. Replace that element with num.
3. Return the length of sub.

Complexity Analysis

Time complexity : $O(n * log(n))$, binary search uses $\log(n)$ time as opposed to the $O(n)$ time of a linear scan, which improves our time complexity.

Space complexity : $O(n)$, When the input is strictly increasing, the sub array will be the same size as the input.

PART 06

Q/A


NYU TANDON
Thank you!
Upcoming: Project Presentations