Newton School of Technology

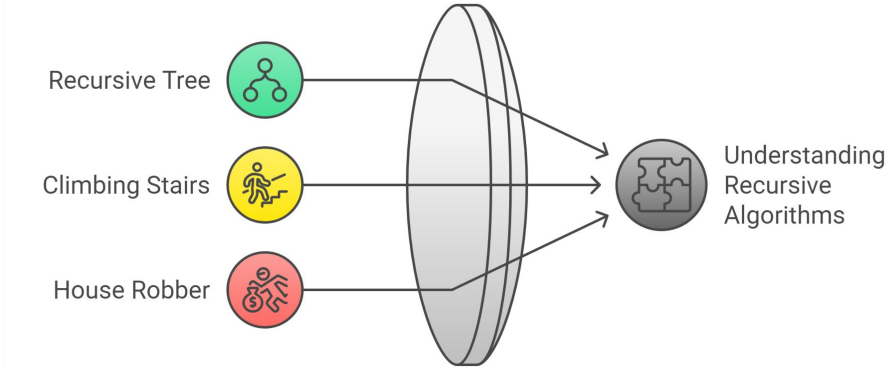
8

Recursion – Part II

by Gladden Rumao

Lecture Agenda :

1. Recursion:
    i. Recursive Tree
2. Climbing Stairs - Quick Recap
3. House Robber
4. Frog Jump



Newton School of Technology

Recap - Climbing Stairs

Description

Newton School of Technology

Problem Statement: Imagine yourself standing at the bottom (0th step) of a staircase with n steps. Your goal is to reach the top of the staircase. At each step, you have two options: you can either climb one step or two steps.

Now, you need to determine the total number of ways you can climb the stairs and successfully reach the top.

Example

Newton School of Technology

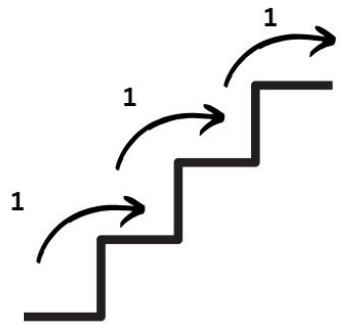
Input

N = 3

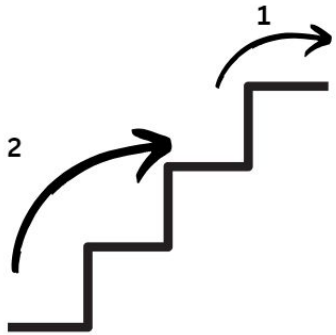
Output

Ans = 3

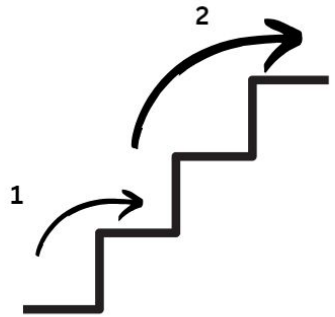
Explanation:



1 + 1 + 1 = 3



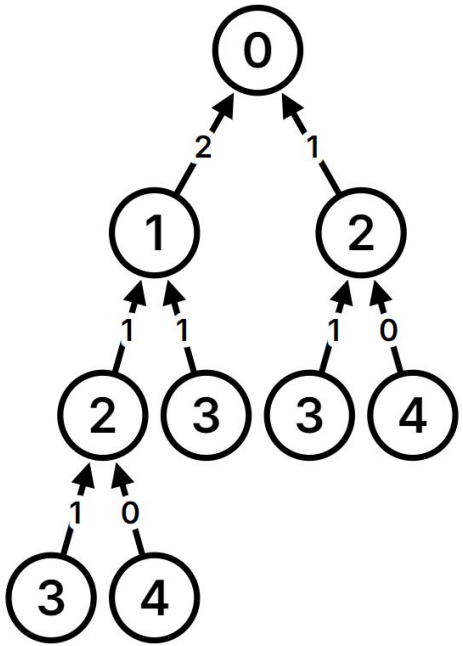
2 + 1 = 3



1 + 2 = 3

Recursive Tree

Newton School of Technology



Code

Newton School of Technology

```python
def climbStairsHelper(current, n):
    # Base case: if we reach exactly the nth step
    if current == n:
        return 1
    # Base case: if we go beyond the nth step
    if current > n:
        return 0
    # Recursive calls: climb 1 step or 2 steps
    return climbStairsHelper(current + 1, n) + climbStairsHelper(current + 2, n)

def climbStairs(n):
    return climbStairsHelper(0, n)
```

Code Analysis

Newton School of Technology

Time Complexity : O(2^N)

Space Complexity : O(N)

Recursive Tree vs Call Stack

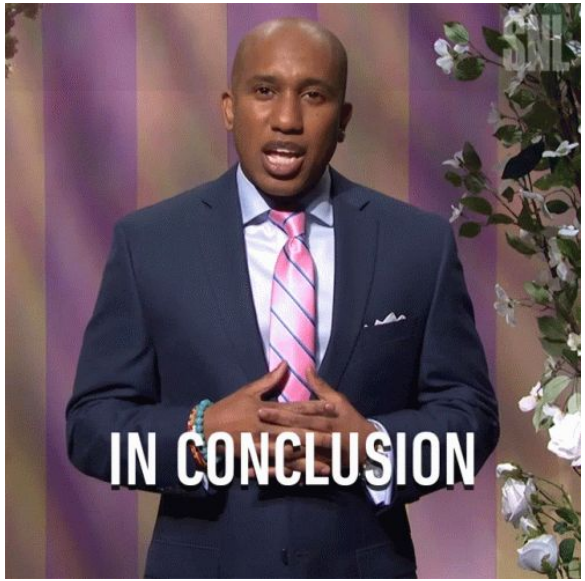
Newton School of Technology

Feature	Recursive Tree	Recursive Call Stack
Representation	A linear structure showing sequence of function calls	A branching structure showing how the problem is divided into sub-problems
Purpose	To track function call , their state , and parameters during execution	To visualize, how problem is broken down into subproblems
Visualization	Visualize function calls in a stack like structure with each call pushed and popped	Displays how each recursive calls generate further calls , branching from each node
Efficiency Analysis	Shows the actual call sequence,useful for detecting stack overflow and optimizing memory usage	Useful for identifying inefficiencies such as redundant subproblems and figure out base case of the problem

Conclusion

Newton School of Technology

In conclusion, both representations serve complementary purposes. The **recursive call stack** helps with debugging and understanding memory usage during recursion, while the **recursive tree** provides insight into problem decomposition, complexity analysis, and optimization strategies. Together, they offer a holistic view of recursive algorithms and their efficiency.



Newton School of Technology

House Robber

House Robber

Newton School of Technology

Description

Newton School of Technology

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses are broken into on the same night**.

13

Given an integer array `nums` representing the amount of money of each house, return the maximum amount of money you can rob tonight **without alerting the police**.

Example

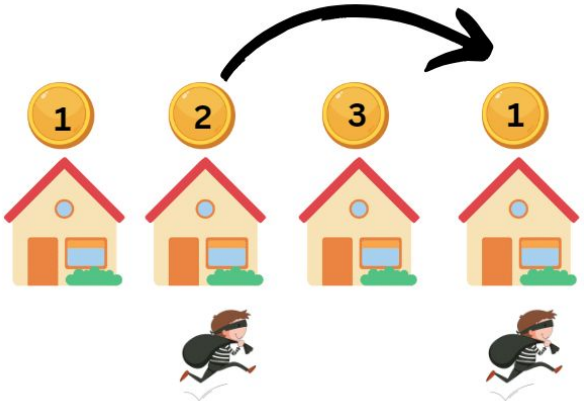
Newton School of Technology

Input : [1,2,3,1]

Output : 4

Explanation: there are two possible scenarios which gives outcome 3 and 4 and max(3,4) is 4

1 + 3 = 4



2 + 1 = 3

Example

Newton School of Technology

Input : [20,4,6,30]

Output : 50

Building Intuition :

Newton School of Technology

- If there was only one house, how much would you rob?
- If there were two houses, what would be your choice?
- If you stand in front of a house, what are your choices?-
- If you rob this house, what happens to the next house?
- If you skip this house, how does that change your future decisions?
- Can you break this problem down into **smaller subproblems**?
- If you knew the answer for smaller inputs (fewer houses), could you use that to build a solution for the larger input?

Approach

Newton School of Technology

1. Base Case

The base case is the stopping condition for the recursion:

- If the current house index is **out of bounds** (i.e i >= len(nums))
  - This means there are no more houses to rob.
  - **Return 0**, as there is no money to add beyond this point.

2. Recursive Call

17

At each step, you make two recursive calls:

1. **Rob the current house (iii)**:
   - Add the money in the current house (nums[i]\text{nums}[i]nums[i]) to the result of robbing houses starting from index i+2 (since you cannot rob the adjacent house).
   - Recursive call: **robFrom(i+2)**

Approach

Newton School of Technology

2 Skip the current house (iii):

i. Move directly to the next house **(i+1)** without adding the money from the current house.
ii. Recursive call: **robFrom(i+1)**

3. Return Type

For each house iii, you return the maximum of the two choices:

18

1. Rob the current house **(nums[i] + robFrom(i+2))**
2. Skip the current house **(robFrom (i+1)).**

The return value ensures you always maximize the total amount robbed.

Example

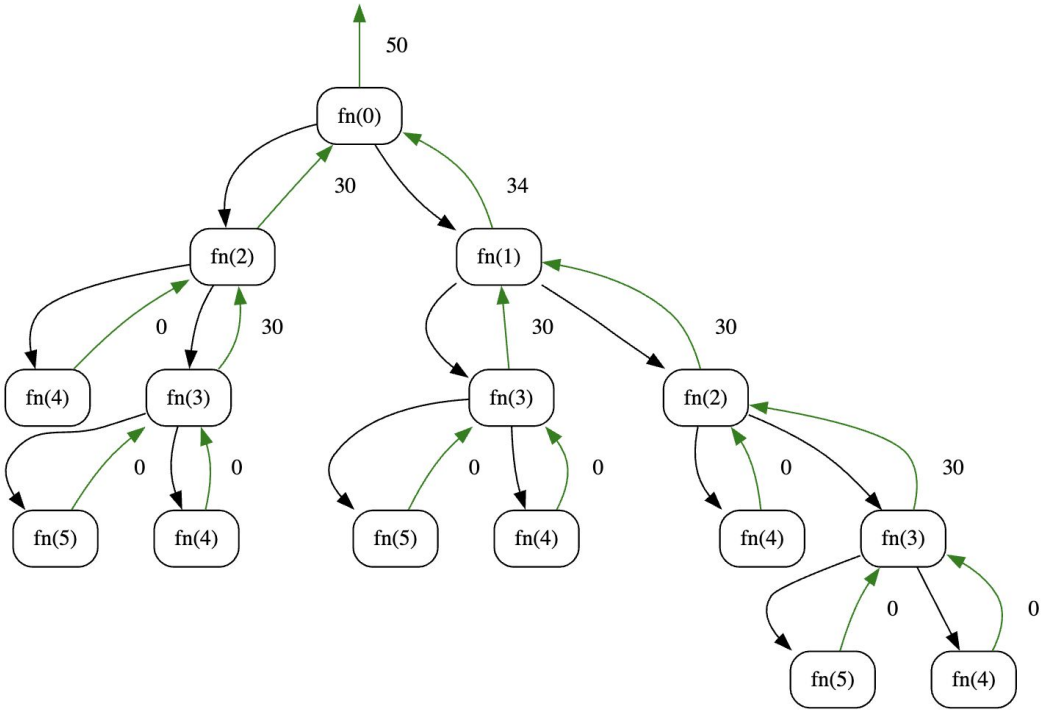
Newton School of Technology

Input : [20,4,6,30]

Output : 50

Recursive Tree

Newton School of Technology



Code

Newton School of Technology

```python
def robFrom(nums, i):
    # Base Case: No more houses left to rob
    if i >= len(nums):
        return 0

    # Recursive Calls
    rob_this = nums[i] + robFrom(nums, i + 2)  # Rob the current house
    skip_this = robFrom(nums, i + 1)           # Skip the current house

    # Return Type: Maximum of robbing or skipping
    return max(rob_this, skip_this)

def rob(nums):
    return robFrom(nums, 0)  # Start robbing from the first house
```

Code Analysis

Newton School of Technology

Time Complexity : O(2^N)

Space Complexity : O(N)

Newton School of Technology

Frog Jump

Frog Jump-1

Newton School of Technology

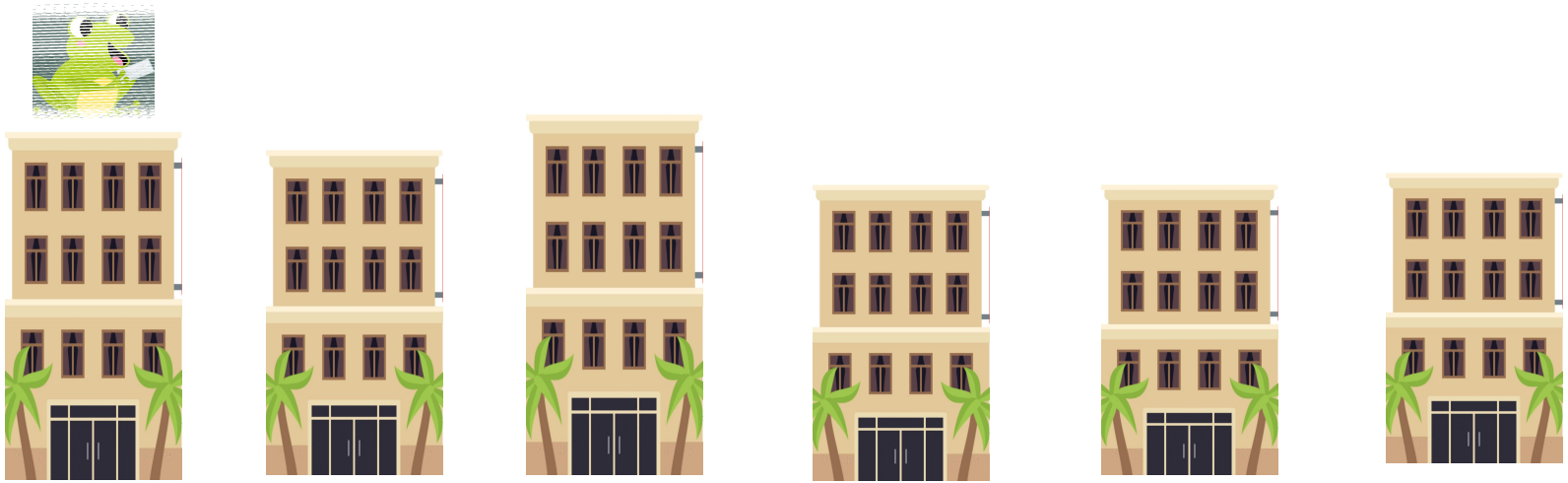
There are N buildings, numbered 1,2,3,…,N. For each i (1 ≤ i ≤ N), the height of building **i** is hi.
There is a frog who is initially on Building 1's roof, He wants to reach Building N.
The frog can jump from one building to another but there are some conditions to his jumps.

- If the frog is currently on Building **i**, he can jump on either building i+1 or i+2.
- The energy used by the frog for each jump is the difference in heights of the initial and final buildings. meaning | hi - hj |, where **i** is the initial building and **j** is the final building.

Find the minimum possible energy used by the frog to reach building N.

Frog Jump-1

Newton School of Technology



Frog Jump – 1

Newton School of Technology

Different options for the frog can be.

[10, 20, 30, 10]

Building Intuition :

Newton School of Technology

- If you are standing on step i, what are your choices?
- **What factors influence your decision on whether to jump 1 step or 2 steps?**
- Can you always take big jumps? What if a big jump makes you lose more energy than a small jump?
- Can you break this problem down into **smaller subproblems**?

27

Approach

Newton School of Technology

1. Base Case

-

28

Approach

Newton School of Technology

29

Recursive Tree

Newton School of Technology

30

Let’s Code:

Newton School of Technology

Code

Newton School of Technology

```python
def frogJump(heights):
    return f(0,heights)

def f(ind, heights):

    if ind == len(heights)-1:
        return 0
    onejump = float('inf')
    twojump = float('inf')
    if ind+1 < len(heights):
        onejump = abs(heights[ind+1] - heights[ind]) + f(ind+1, heights)
    if ind+2 < len(heights):
        twojump = abs(heights[ind+2] - heights[ind]) + f(ind+2, heights)

    return min(onejump,twojump)

```

Code Analysis

Newton School of Technology

Time Complexity : O(2^N)

Space Complexity : O(N)

Common pitfalls in recursion

Newton School of Technology

Summary

Newton School of Technology

In today's lecture we have studied about **recursive tree** which is used for visualisation of recursive call with branch showing each step

The problems which we have covered in this class has **two recursive call and only one parameter is changing**

In **Climbing stairs** problem we have two calls one for next step and one for next to next step and changing parameter was the **step of stairs**

In **House robber** problem we have two calls one for next house and one for next to next house and changing parameter was **index of list** or we can say **house number**

Thanks for Attending!