## Foundations of recursion

Program -> Call Stack

Function -> Activation Record

* *Parameters*
* *Local Data*
* Metadata
  + Code pointers
  + Return addresses

***DRAW RECURSION TREES***

1. Mental model of the code
2. S(n) is proportional to the height of the tree

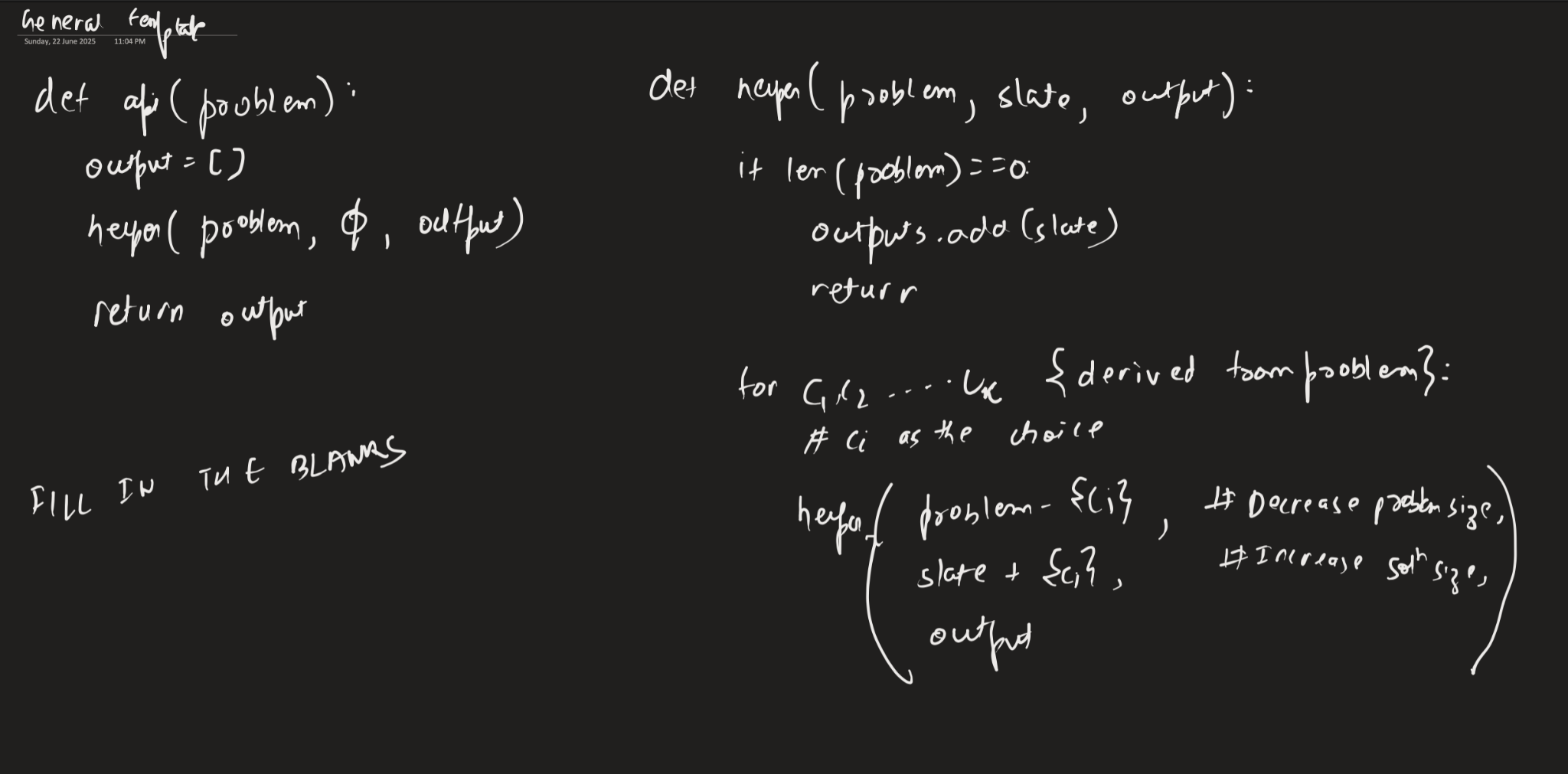
= height \* space per node

1. T(n) is proportional to the no of nodes

= no of nodes \* time per node

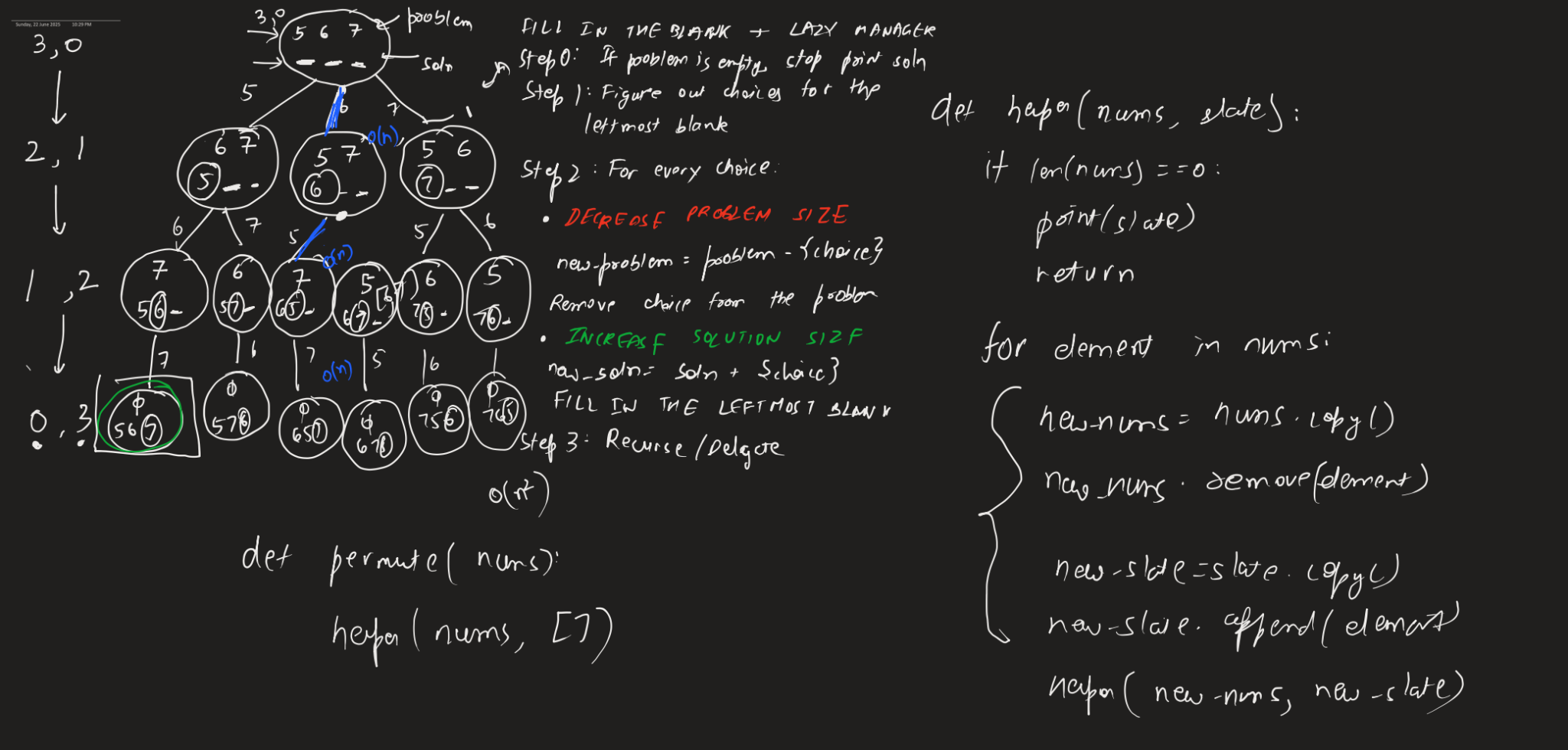
## Exhaustive enumeration

Systematic generation of a large number of outputs (possibly exponential)



## Permutations (46)

<https://leetcode.com/problems/permutations/>



class Solution(object):

def helper(self, nums, slate, output):

# Step 0: Base case for a trivial problem

if len(nums) == 0:

output.append(slate)

return

# Step 1: Figure out a list of choices

for element in nums:

# Step 2: Decrease problem, increase solution

new\_nums = nums[:]

new\_nums.remove(element)

new\_slate = slate[:]

new\_slate.append(element)

# Step 3: Delegate

self.helper(new\_nums, new\_slate, output)

def permute(self, nums):

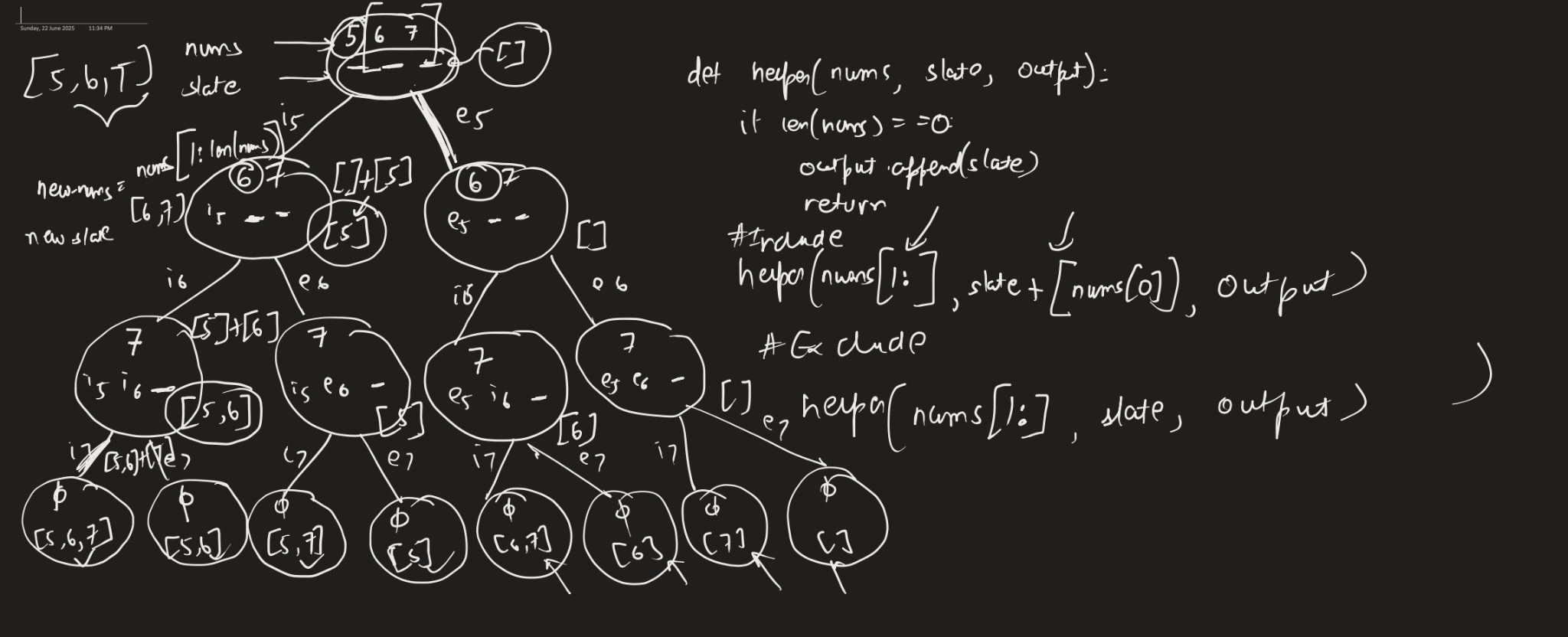
output = []

self.helper(nums, [], output)

return output

## Subsets

<https://leetcode.com/problems/subsets/description/>



class Solution(object):

def helper(self, nums, slate, output):

if len(nums) == 0:

output.append(slate)

return

# Include

self.helper(nums[1:], slate + [nums[0]], output)

# Exclude

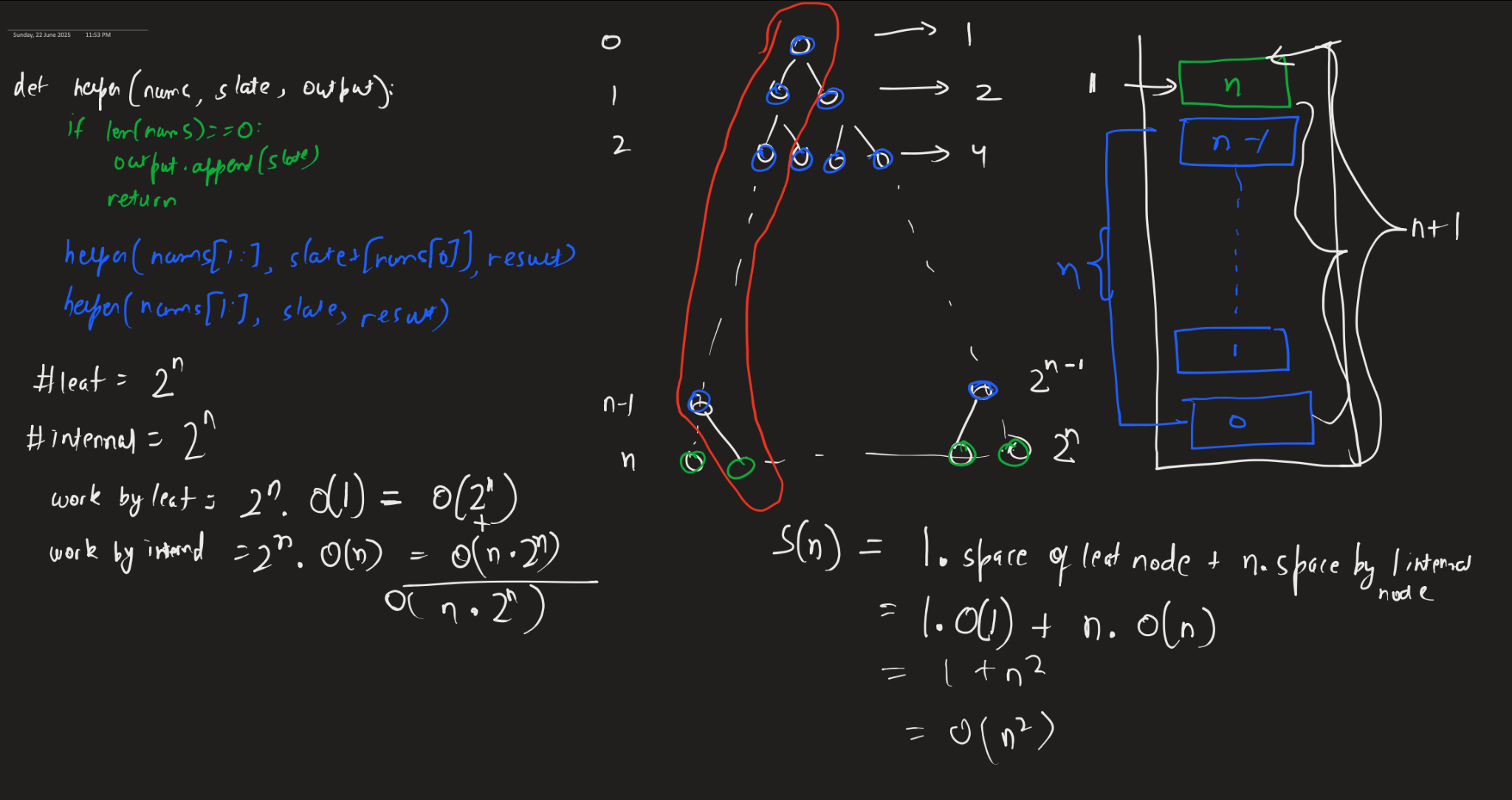
self.helper(nums[1:], slate, output)

def subsets(self, nums):

output = []

self.helper(nums, [], output)

return output



## Optimization

Use index to keep track of the current element

Use a mutable data structure to append and pop for the solution

class Solution(object):

def helper(self, nums, i, slate, output):

if len(nums) == i:

output.append(slate[:])

return

# Include

slate.append(nums[i])

self.helper(nums, i+1, slate, output)

slate.pop()

# Exclude

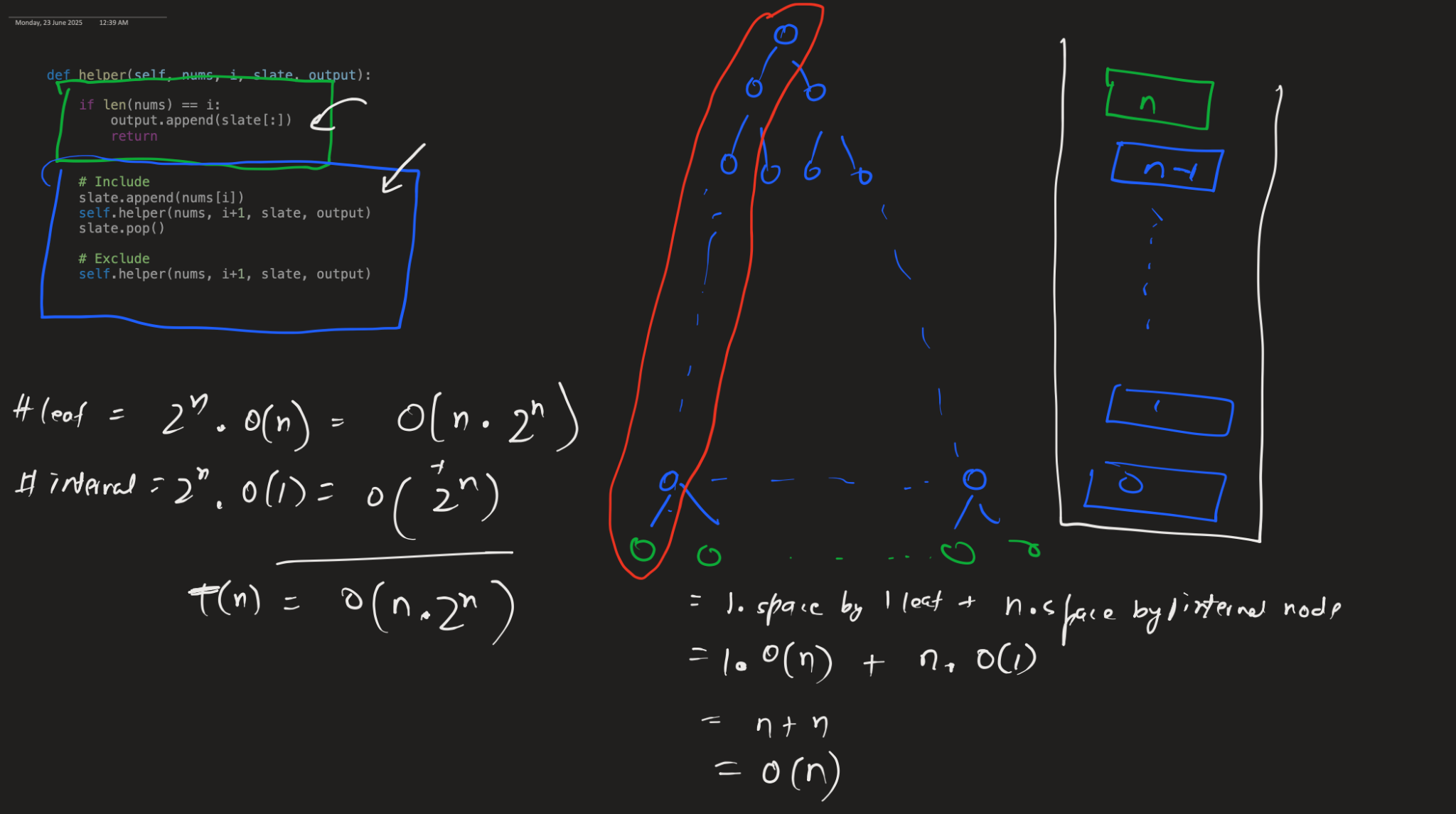
self.helper(nums, i+1, slate, output)

def subsets(self, nums):

output = []

self.helper(nums, 0, [], output)

return output



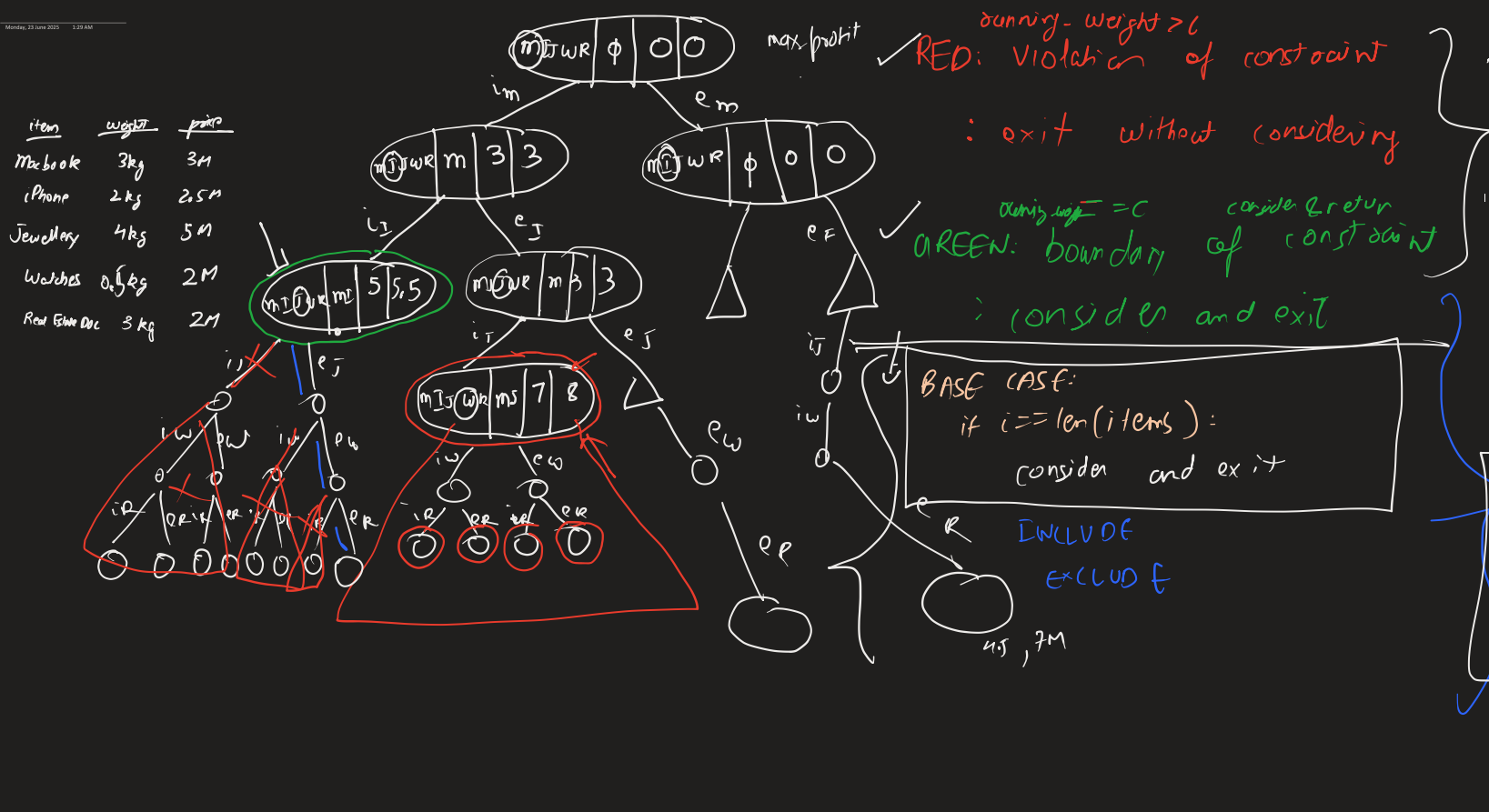
T(n) = O(n.2^n)

**S(n) = O(n)**

## Backtracking

In an exhaustive enumeration problem with constraints, we can decide to exit early if we reach a state where there is no solution that can exist (RED NODE), or a single solution exists (GREEN NODE).

**It results in an improved exponential time complexity.**

****

from dataclasses import dataclass

@dataclass

class Item:

name: str

weight: float

price: float

class Solution:

def helper(self, items, i, c, slate, running\_weight, running\_profit):

if running\_weight > c:

return

if running\_weight == c:

if running\_profit > self.max\_profit:

self.max\_profit = running\_profit

self.max\_slate = slate[:]

return

if i == len(items):

if running\_profit > self.max\_profit:

self.max\_profit = running\_profit

self.max\_slate = slate[:]

return

slate.append(items[i])

self.helper(items, i+1, c, slate, running\_weight + items[i].weight, running\_profit + items[i].price)

slate.pop()

self.helper(items, i+1, c, slate, running\_weight, running\_profit)

def knapsack(self, items, c):

self.max\_profit = 0

self.max\_slate = []

self.helper(items, 0, c, [], 0, 0)

return self.max\_profit, self.max\_slate

s = Solution()

items = [Item("Macbook", 3, 3), Item("iPhone", 2, 2.5), Item("Jewellery", 4, 5), Item("Watch", 1, 2), Item("Real Estate Documents", 3, 2)]

print(s.knapsack(items, 5))

T(n) = O(2^n)

S(n) = O(n)

## Combinations of size k

<https://leetcode.com/problems/combinations/>

class Solution(object):

def helper(self, nums, i, k, slate, output):

if len(slate) + len(nums) - i < k:

return

if len(slate) == k:

output.append(slate[:])

return

if len(nums) == i:

return

# Include

slate.append(nums[i])

self.helper(nums, i+1, k, slate, output)

slate.pop()

# Exclude

self.helper(nums, i+1, k, slate, output)

def combine(self, n, k):

nums = [i for i in range(1, n+1)]

output = []

self.helper(nums, 0, k, [], output)

return output