Recursion

Instructor: David Hulak

{ik} INTERVIEW KICKSTART

Agenda

Block 1
Introduction & foundation recap
Letter Case Permutation
5-minute break

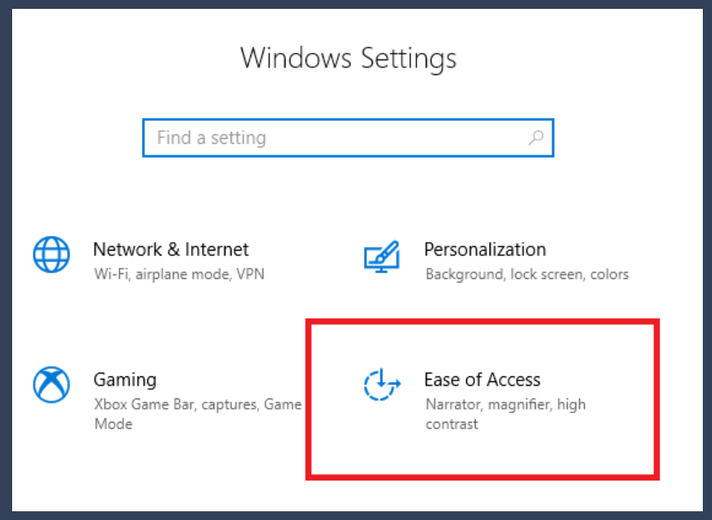
Block 2
LCP optimization
Subsets
25-minute break

Block 3
Permutations
Backtracking
5-minute break

Block 4
Combinations
Other exercises to practices
Q&A

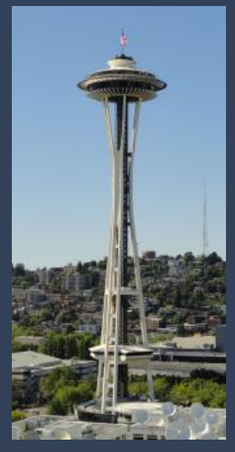
INTERVIEW KICKSTART

Introduction



UNIVERSIDADE FEDERAL DE PERNAMBUCO

VIRTUS IMPAVIDA



{ik} INTERVIEW KICKSTART

Operational notes

- Please use the Q&A feature for **questions**
  - Please **don't** use it for answers or comments
  - Alternatively: raise hand

- Answer most voted questions

- I'll stay after class to answer the ones that remain

{ik} INTERVIEW KICKSTART

Takeaways from preclass videos

- Why recursion is hard, and how thinking like a lazy manager is the best way to proceed.
- Examples of simple recursive mathematical functions and how they get executed on the call stack
- How you come up with recursive code: Recursive implementations naturally follow from decrease-and-conquer / divide-and-conquer algorithm design strategies. Divide the overall problem into one or more subproblems ***of the same form***.
- How you think about recursive code: Make a “Recursive leap of faith”. Avoid thinking about the entire execution. Think only about the base case and how (if your subordinates do their job correctly), your code will correctly assemble the solution to the overall problem.
- Basic combinatorics (counting the number of permutations and combinations)

“Mathematics:
Some interviewers ask basic discrete math questions. This is more prevalent at Google than at other companies because we are surrounded by counting problems, probability problems, and other Discrete Math 101 situations. Spend some time before the interview refreshing your memory on (or teaching yourself) the essentials of combinatorics and probability. You should be familiar with n-choose-k problems and their ilk - the more the better.”

https://careers.google.com/how-we-hire/interview/#interviews-for-software-engineering-and-technical-roles

Today’s live class

Recursion is a bit of a misnomer for today’s class.
- You have seen recursion in sorting: insertion sort, merge sort and quicksort.
- You have seen recursion in searching: e.g, binary search
- Binary trees are an example of a recursive data structure. It is easy to write recursive algorithms on recursive data structures.
- You will see examples of recursion in Graphs and Dynamic Programming.

*So **recursion pervades the whole curriculum.***

- What is distinctive about today’s class is the “class of problems” we are going to look at: **Combinatorial Enumeration** problems.
- Writing recursive code for these is more tricky.

Exhaustive enumeration

Requires a systematic enumeration / generation of all possible combinatorial objects (permutations or combinations).

Leads to a ***combinatorial explosion*** - exponential time complexity. Hence, *impractical for even moderate input sizes*.

Still, for many problems, we cannot do better than this.

General template

You have to fill in a series of blanks, and there is an exponential number of ways to do it.

_ _ _ _ _ _ _ _ _ _ . . . $N \cdot (N-1) \cdot (N-2) \cdots 1$

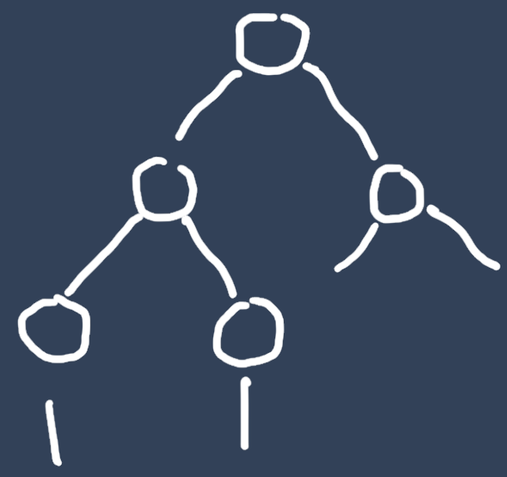
This series of blanks denotes a combinatorial object (some permutation or combination).

You proceed left to right, and as a lazy manager, only take responsibility for filling in the first blank. Delegate the rest of the work to subordinates.

Write up the general strategy used by any arbitrary worker in the hierarchy as a recursive function.

Lazy manager approach

- Each parent is a lazy manager
- Recursive code written for any arbitrary worker



General template

Note the three cases:

1. The boss at the top (root) of the hierarchy does not receive any partial solution (or slate) from above. The boss executes the general strategy on the entire problem definition and starts from a blank slate.
2. The workers in any of the internal nodes above receive a subproblem to solve, and also the partial solution constructed by the bosses above them. They fill in the leftmost blank of the subproblem, and delegate the remaining work to their subordinates (by means of a recursive call).
3. The leaf-level-workers receive the entire solution from above (and a subproblem of size 0), and their job is to print / append / filter / analyse the solutions. Their job description is mentioned in the base case of the recursive function.

General template: code

```
function subordinate(input, subproblem, partial_solution, result):
    if base_case_condition:
        # increment result with partial_solution, which, at this point, will be a fully formed
solution
        return

    # increment partial_solution
    # decrement subproblem
    # recursively call subordinate()

function root_manager(input):
    result = []

    # initialize subproblem
    # initialize the partial_solution
    subordinate()

    return result
```

784. Letter Case Permutation

Medium 1680 114 Add to List Share

Given a string S, we can transform every letter individually to be lowercase or uppercase to create another string.

Return *a list of all possible strings we could create*. You can return the output in **any order**.

Example 1:

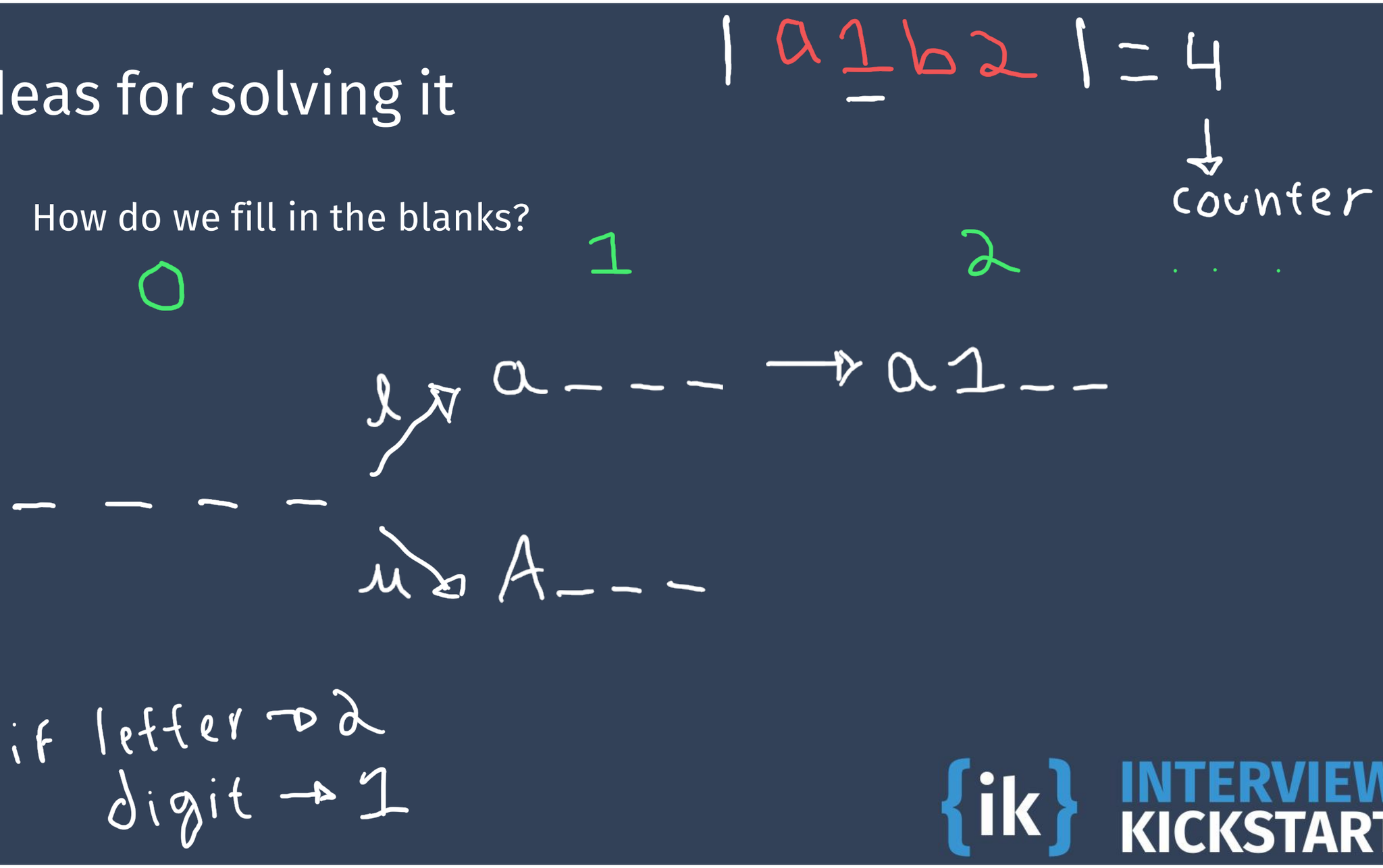
```
Input: S = "a1b2"
Output: ["a1b2","a1B2","A1b2","A1B2"]
```

Constraints:

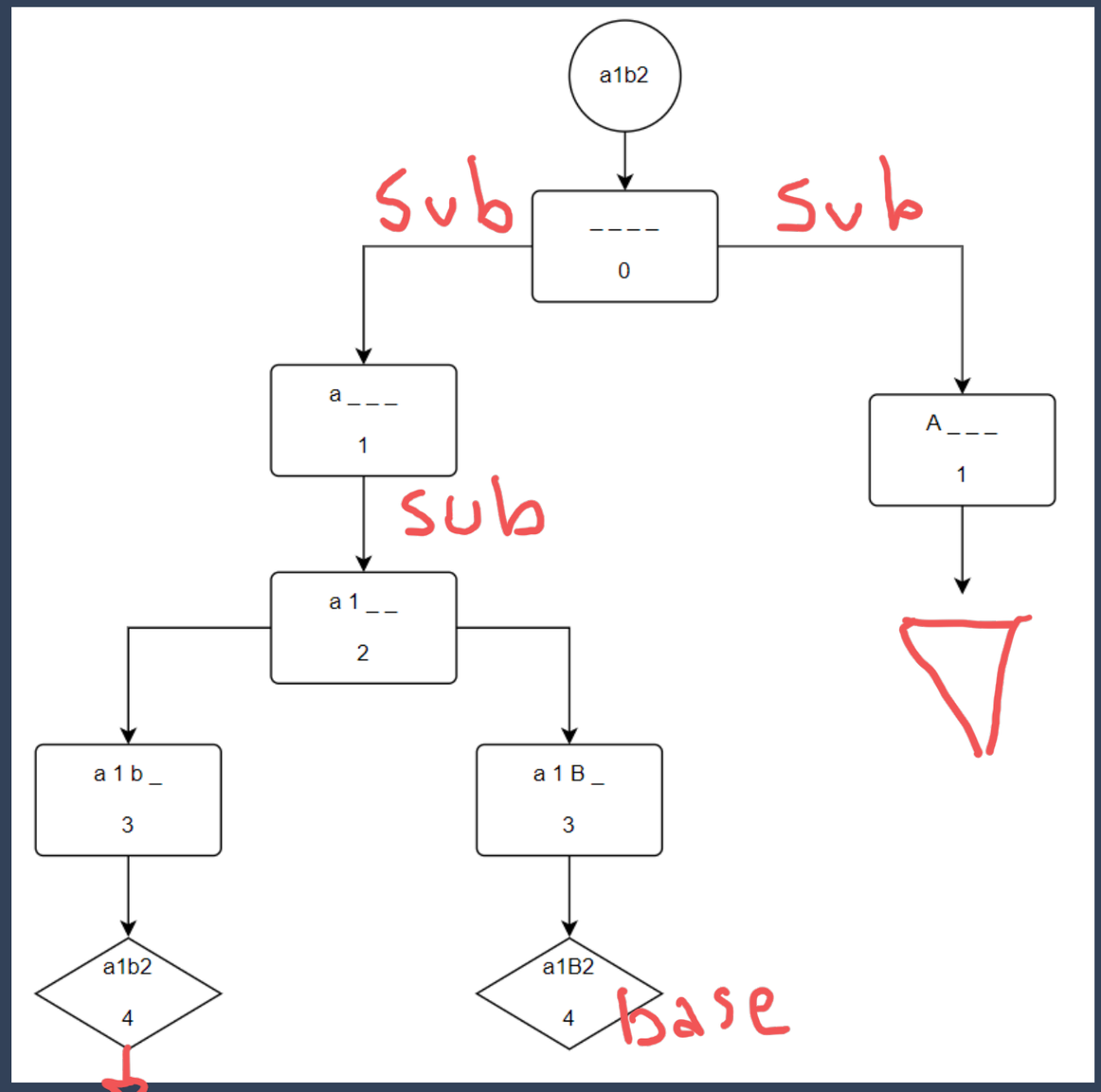
- `S` will be a string with length between `1` and `12`.
- `S` will consist only of letters or digits.

Ideas for solving it

- How do we fill in the blanks?



LCP Tree



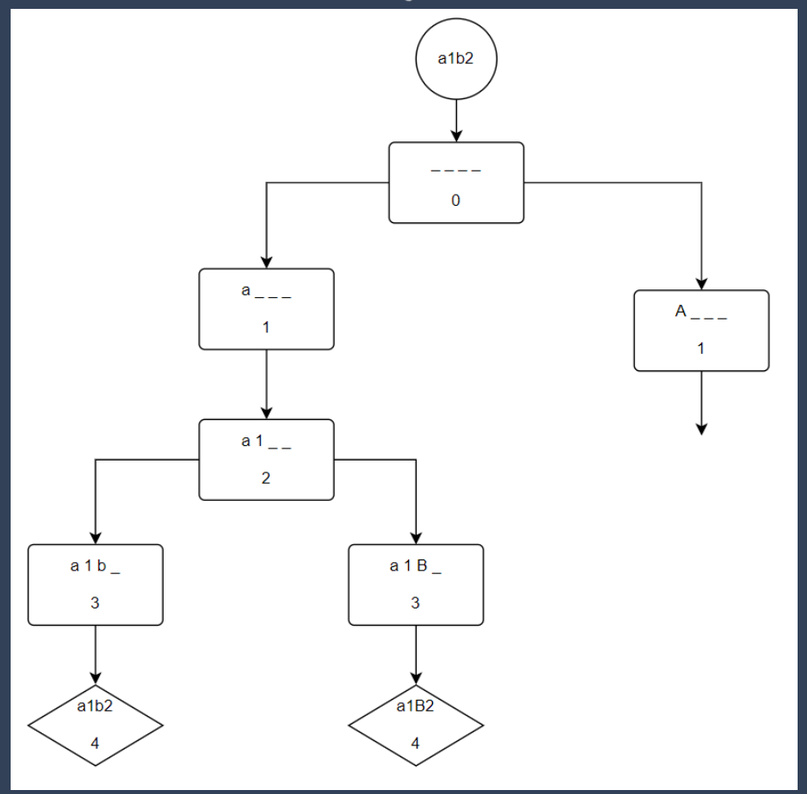
Hints for the code

- #1: Base case: idx_subproblem == len(input)
- #2: Use an index to keep track where we are (subproblem)
- #3: Internal node workers span different calls depending on whether they’re a letter or a number (partial solution)
- #4: Internal node workers will call at most two subordinates
- #5: Root manager initializes the partial solution and subproblem

LCP Coding #1

```
        result.push(partial_solution)
        return

    if input[idx_subproblem].isdigit():
        subordinate(input, idx_subproblem + 1, partial_solution + input[idx_subproblem],
result)
    else:
        subordinate(input, idx_subproblem + 1, partial_solution +
input[idx_subproblem].tolower(), result)
        subordinate(input, idx_subproblem + 1, partial_solution +
input[idx_subproblem].toupper(), result)

function root_manager(input):
    result = []

    subordinate(input, 0, '', result)

    return result
```

Time complexity = $1 \cdot 2^N + N \cdot 2^N = O(N \cdot 2^N)$



Leaf nodes:

$2 \cdot 2 \cdot 2 \cdot 2 \rightarrow 2^N$

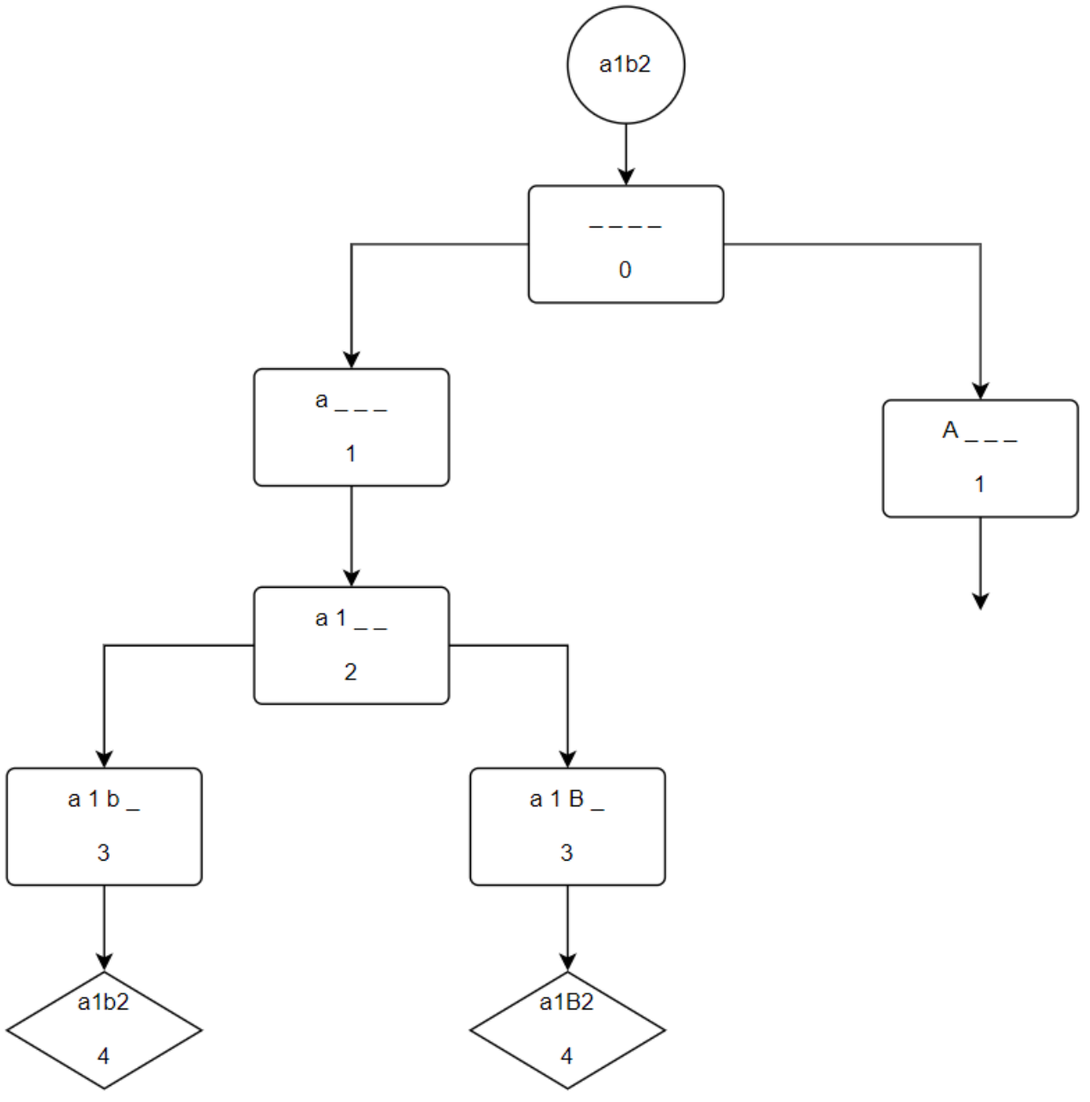
Leaf nodes work: $O(1)$

Internal nodes:

$2^0 + 2^1 + \ldots 2^N = O(2^N)$

Internal nodes work: $O(N)$

{ik} INTERVIEW KICKSTART



Return result space complexity

$O(N \cdot 2^N)$

objects returned

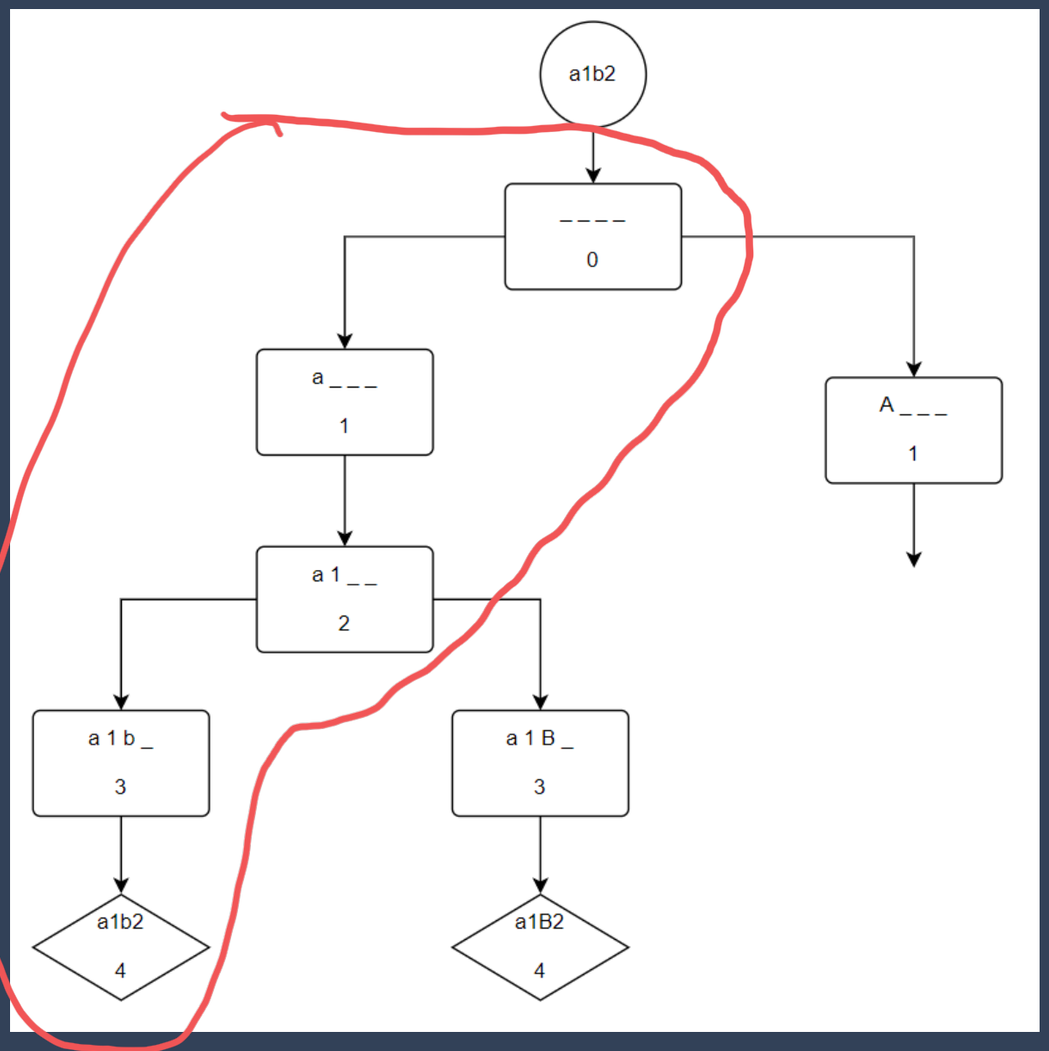
2^N

size objects returned

N

{ik} INTERVIEW KICKSTART

Space complexity (call stack) = $N \cdot N + 1 = O(N^2)$



max # internal nodes on stack:

N

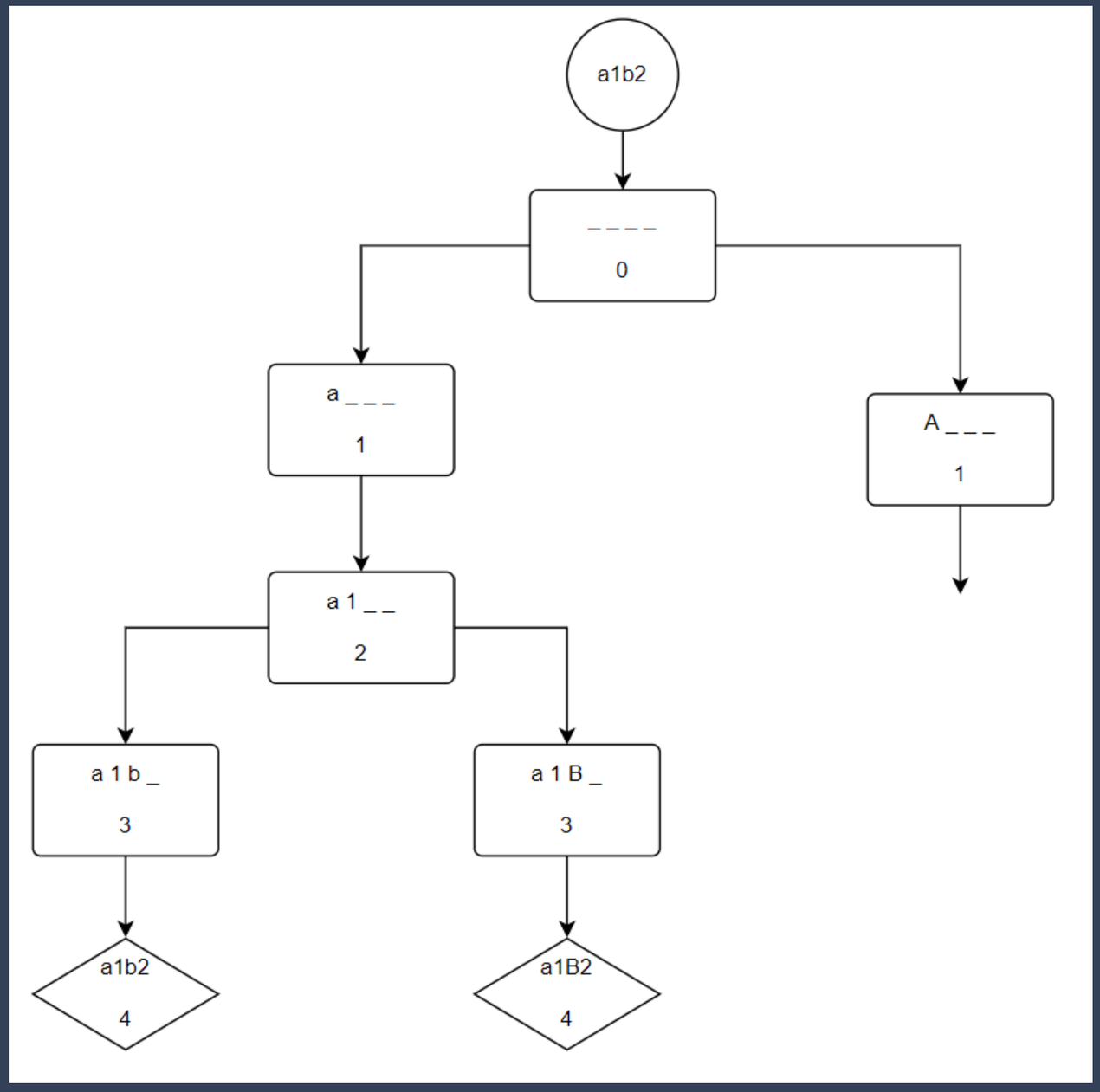
internal nodes memory:

$O(N)$

leaf nodes memory:

$O(1)$

{ik} INTERVIEW KICKSTART



Space complexity

Return result: $n * 2^n$

Call stack: n^2

Result: $O((n * 2^n) + n^2) =$ **$O(n * 2^n)$**

{ik} INTERVIEW KICKSTART

ab111

abcd

O(n) → check

n → #

n (n-1) (n-2) . . . .

n^2

1111

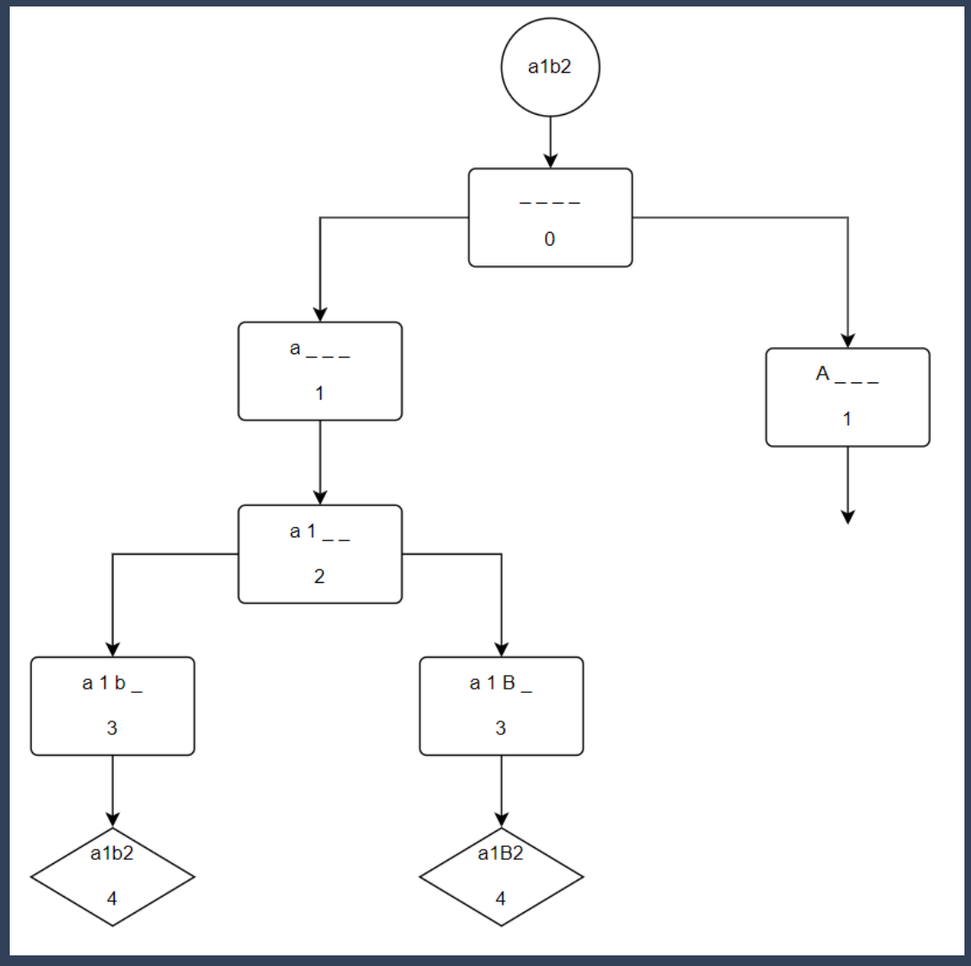
1abc

5-minute break

- We'll be right back ☺

String concatenation

- Immutable languages
  - “+” operator creates new string



Optimizing: inefficiency

- “”
- “a”
- “a1”
- “a1B”
- “a1B2”

$1 + 2 + \dots + N = \frac{N \cdot N}{2} = O(N^2)$

- How many strings are created until we get to a fully formed object?

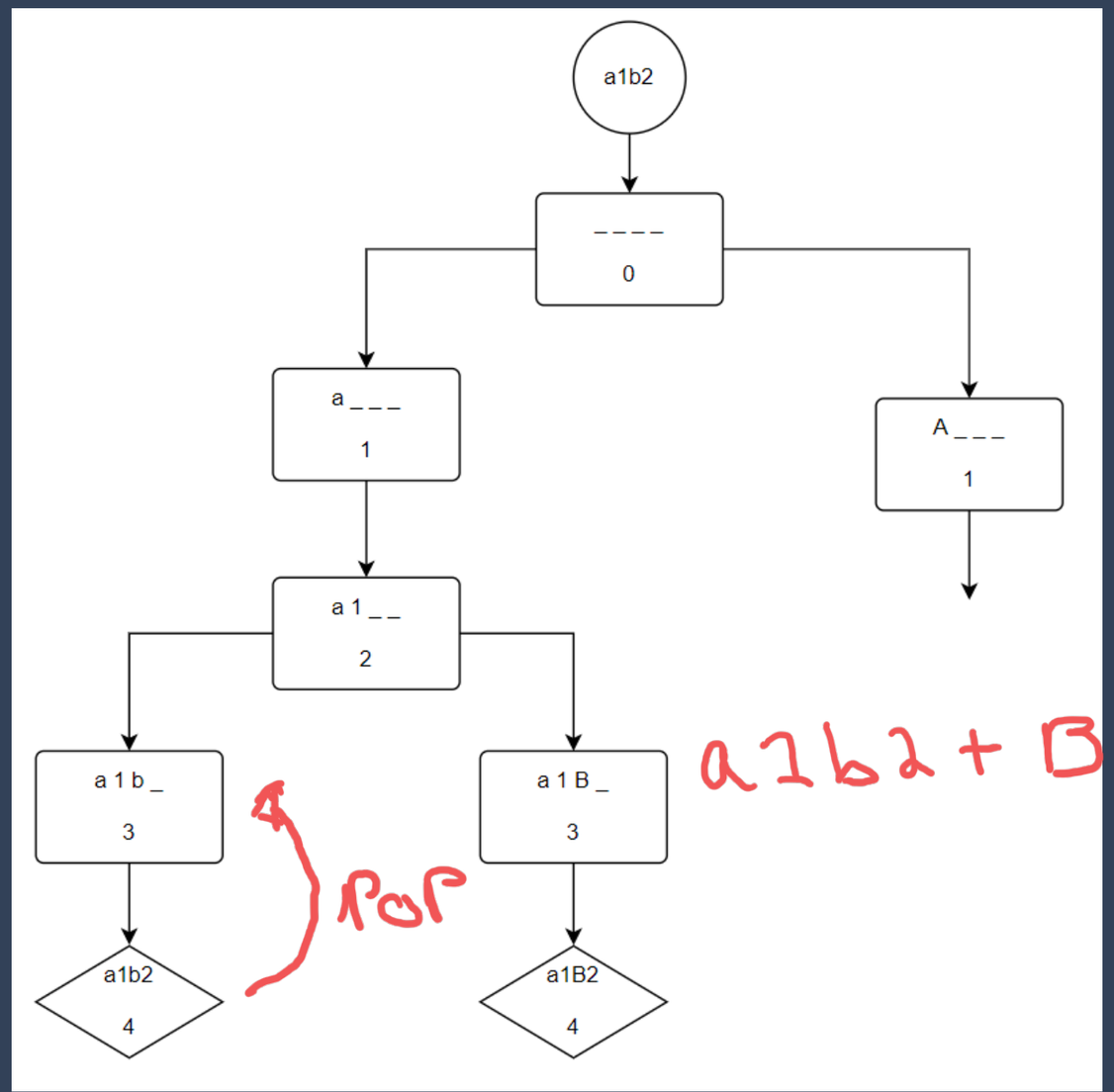
Optimizing: better solution

- Part one: create a buffer
  - []
  - [a] O(1)
  - [a, 1] O(1) > M · 1 = O(M)
  - [a, 1, B]
  - [a, 1, B, 2]

→ M + M = 2M = O(M)

- Part two: create a string
  - “a1B2” O(M)
- How much does each operation cost?

Optimizing: slate maintenance



- What should we do when going back up the tree?

{ik} INTERVIEW KICKSTART

LCP Coding #2

```
    subordinate(input, idx_subproblem + 1, partial_solution, result)
    partial_solution.pop()
else:
    partial_solution.push(input[idx_subproblem].tolower())
    subordinate(input, idx_subproblem + 1, partial_solution, result)
    partial_solution.pop()

    partial_solution.push(input[idx_subproblem].toupper())
    subordinate(input, idx_subproblem + 1, partial_solution, result)
    partial_solution.pop()

function root_manager(input):
    result = []

    subordinate(input, 0, [], result)

    return result
```

Time and space complexity update

- Time complexity
  - # leaf nodes: O(2^n)
  - Leaf nodes work: O(n)
  - Return result (same): O(n * 2^n)

- Memory complexity:
  - Return result (same): O(n * 2^n)
  - Call stack: O(n)

Space complexity with immutable slate = $O(n^2)$

Space complexity with mutable slate = O(n)

Henceforth, we'll prefer mutable parameters

78. Subsets

Medium 5207 109 Add to List Share

Given an integer array `nums` of **unique** elements, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

Example 1:

```
Input: nums = [1,2,3]
Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

Example 2:

```
Input: nums = [0]
Output: [[],[0]]
```

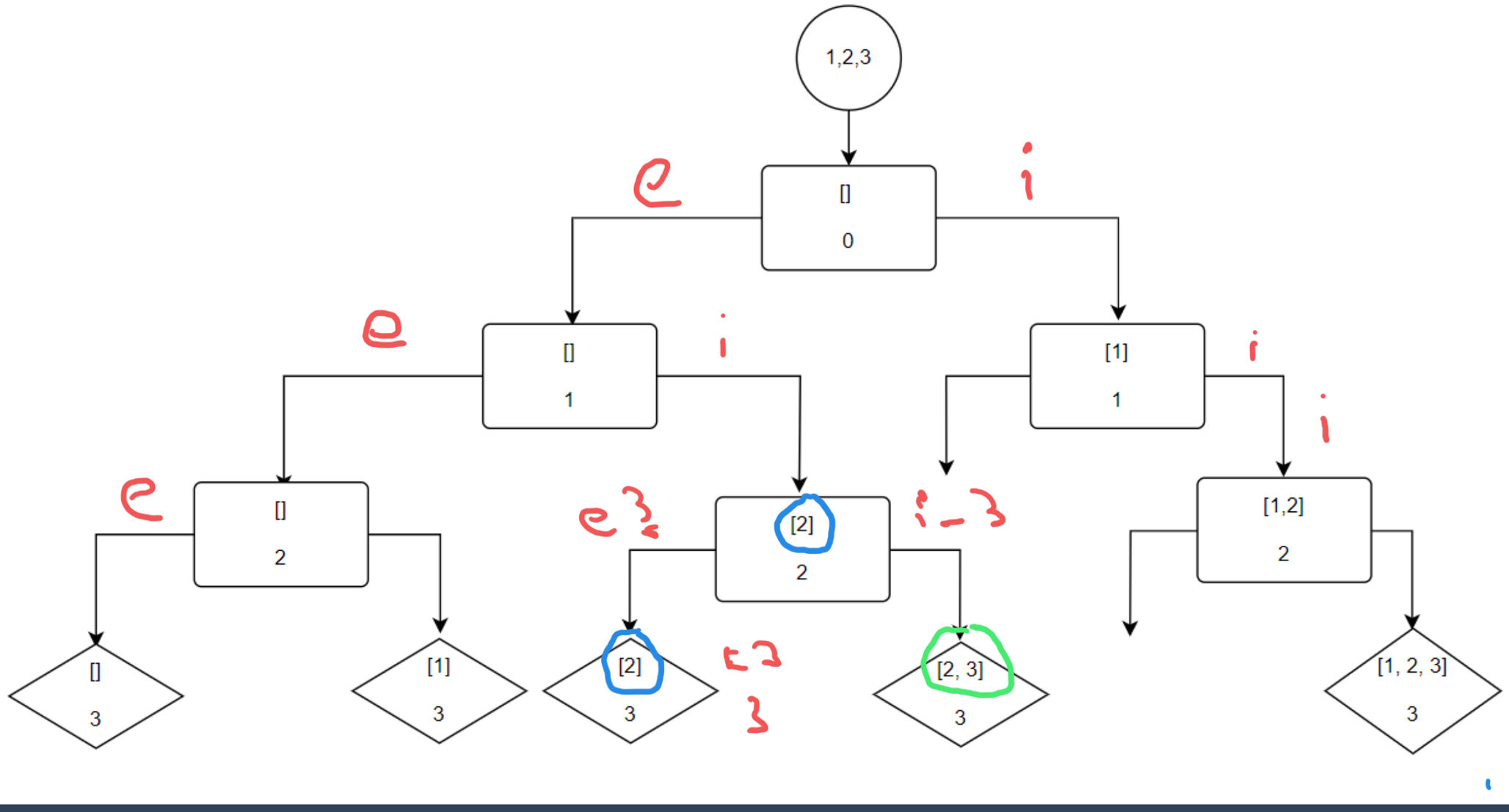
Constraints:

- `1 <= nums.length <= 10`
- `-10 <= nums[i] <= 10`
- All the numbers of `nums` are **unique**.

Subsets: filling in the blanks

[1, 2, 3]

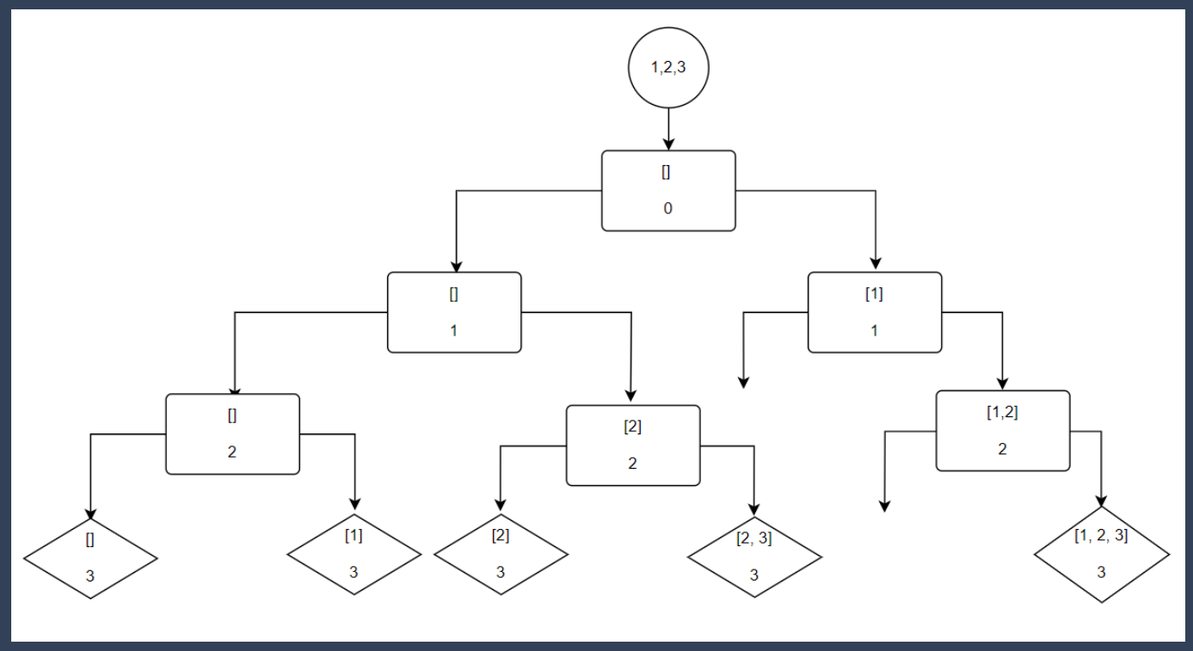
Subsets - Tree



Subsets - Code

```
    if len(input) == idx_subproblem:
        result.push(copy(partial_solution))
        return

    # exclude current element -> don't add it
    subordinate(input, idx_subproblem + 1, partial_solution, result)

    # include current
        partial_solution.push(input[idx_subproblem])
        subordinate(input, idx_subproblem + 1, partial_solution, result)
        partial_solution.pop()

function root_manager(input):
    result = []

    subordinate(input, 0, [], result)

    return result
```

Time complexity = $1 \cdot 2^N + N \cdot 2^N = O(N \cdot 2^N)$



2^0
2^1
2^2

internal nodes:

$2^0 + 2^1 \ldots + 2^{N-1} = O(2^N)$

internal nodes work:

$O(1)$

leaf nodes:

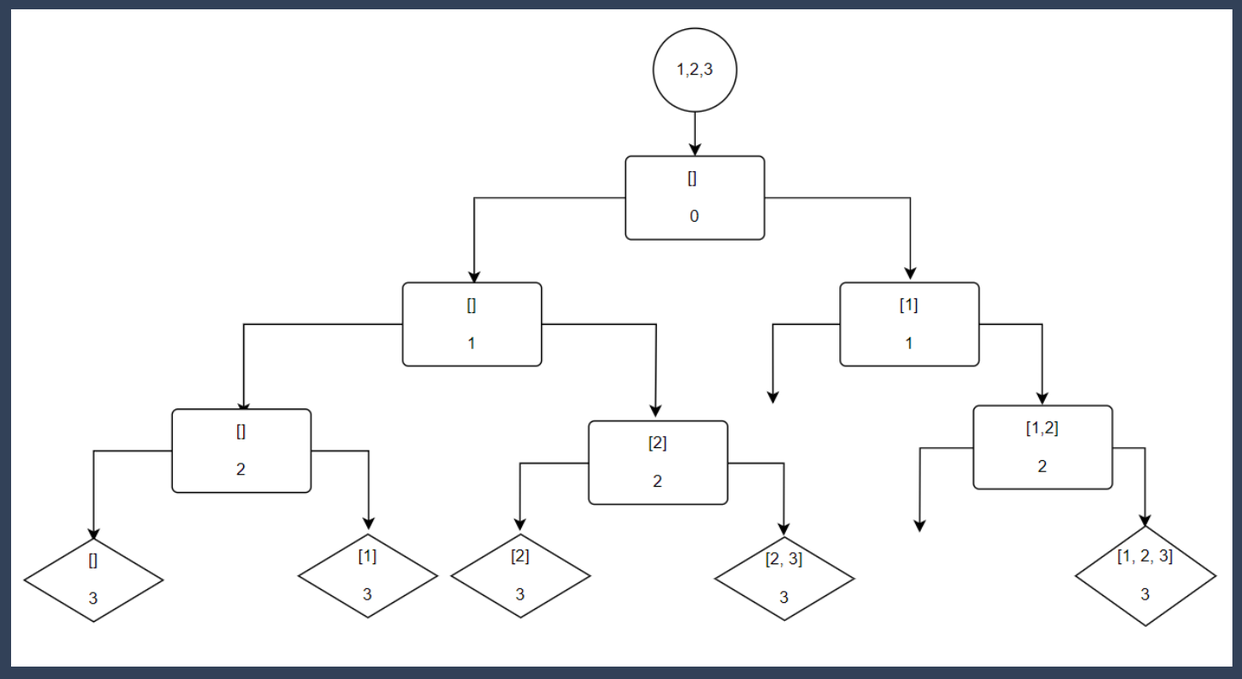
2^N

leaf nodes work:

$O(N)$

{ik} INTERVIEW KICKSTART

Space complexity (return result) = $O(N \cdot 2^N)$



objects returned:

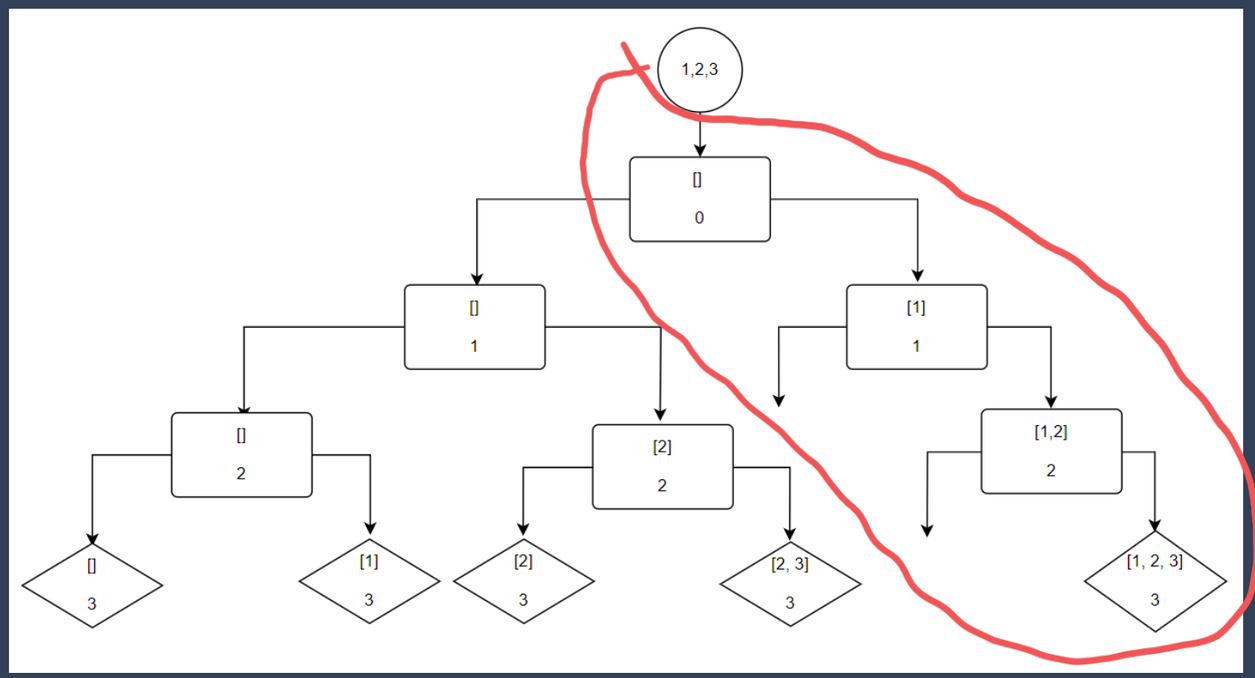
2^N

size objects returned:

N

{ik} INTERVIEW KICKSTART

Space complexity (call stack) = $N + N = 2N = O(N)$



max # internal nodes on stack:

N

internal nodes memory:

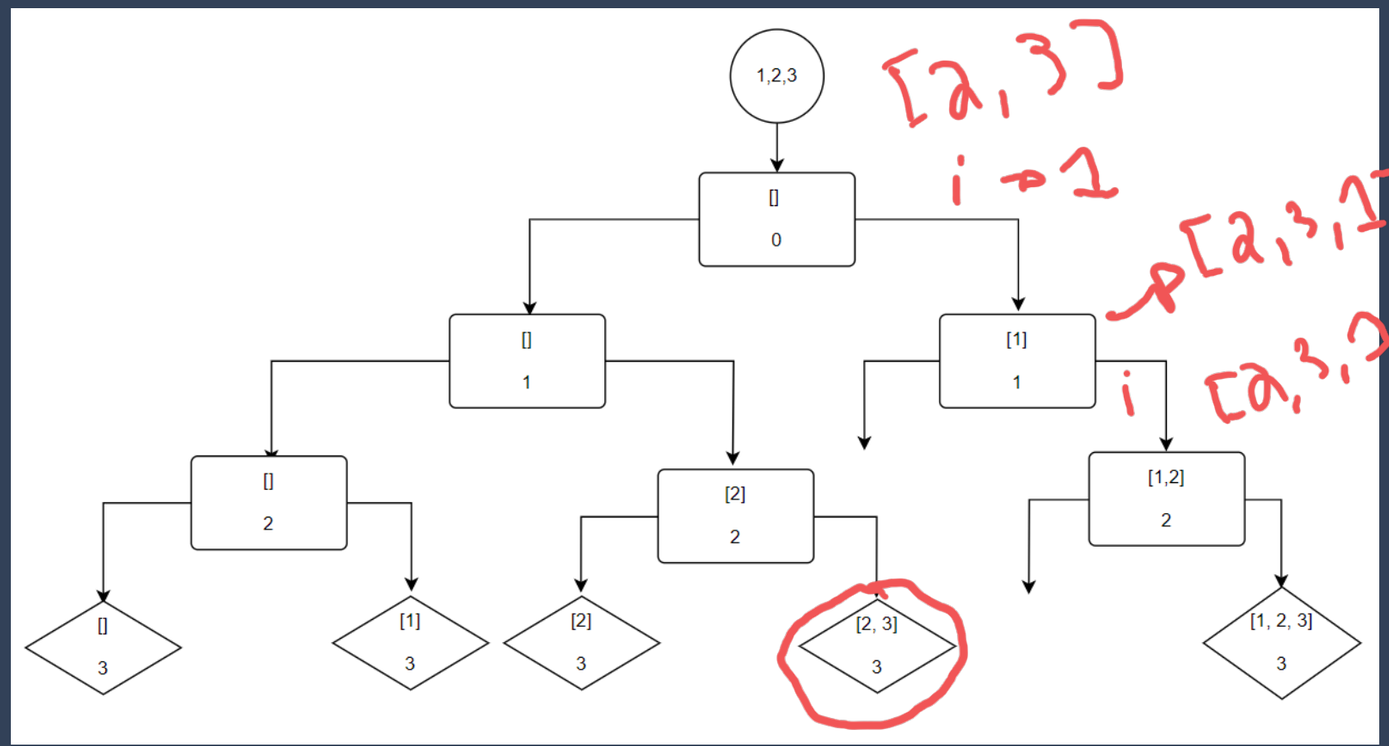
$O(1)$

leaf nodes memory:

$O(N)$

{ik} INTERVIEW KICKSTART

Space complexity



Return result: n * 2^n

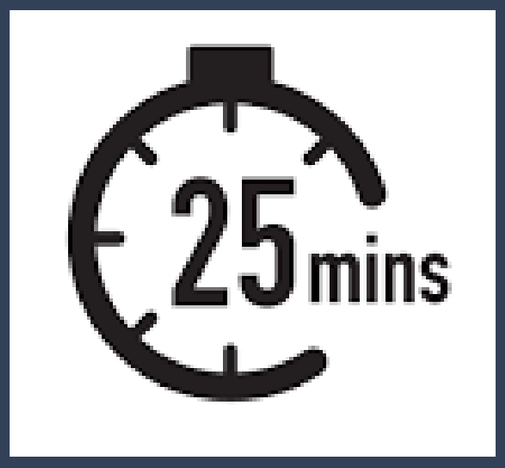
Call stack: n

Result: O((n * 2^n) + n) = **O(n * 2^n)**

{ik} INTERVIEW KICKSTART

25-minute break

- We'll be right back ☺



{ik} INTERVIEW KICKSTART

N → N!

46. Permutations

1,1,2 → 211

Medium 5391 126 Add to List Share

Given an array `nums` of distinct integers, return *all the possible permutations*. You can return the answer in **any order**.

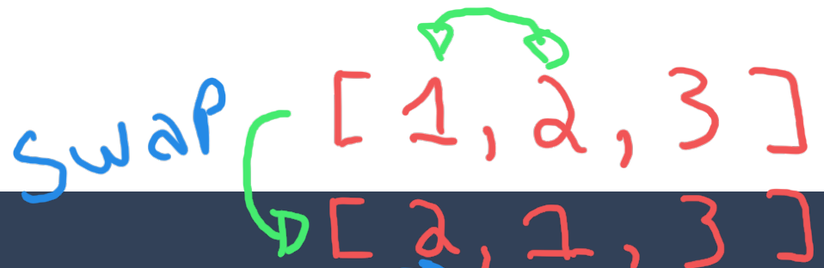
N N-1 ... 1

Example 1:

```
Input: nums = [1,2,3]
Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

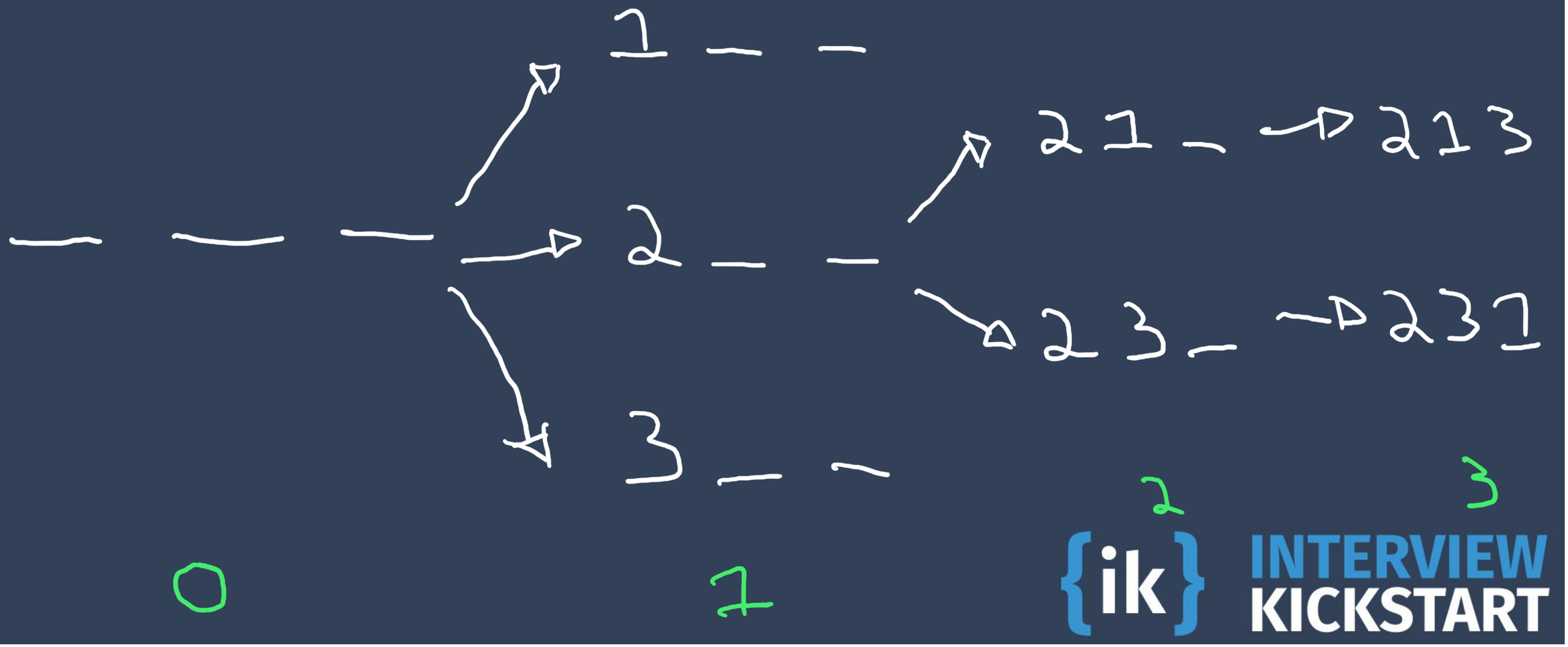
Constraints:

- `1 <= nums.length <= 6`
- `-10 <= nums[i] <= 10`
- All the integers of `nums` are **unique**.



1 A B 2
0 1 2 3

Designing the solution: filling the blanks



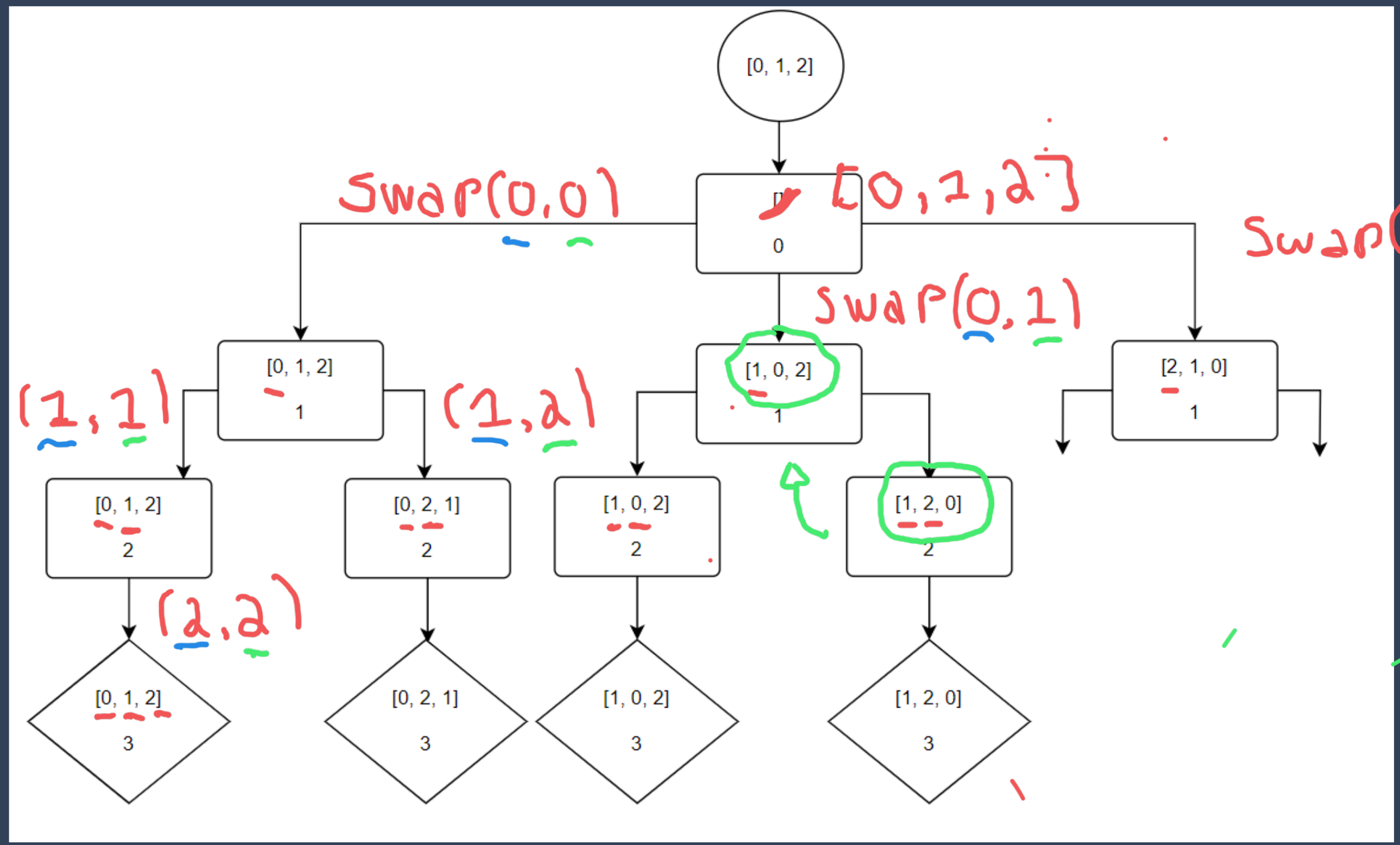
Permutations - Tree

{ik} INTERVIEW KICKSTART

swap(idx_sp, idx_pick)

0 → 0 → 2
1 → 1 → 2
2 → 2 → 2

idx_sp → N-1



swap(0,0) [0,1,2] swap(0,2)

swap(0,1)

(1,1) (1,2)

(2,2)

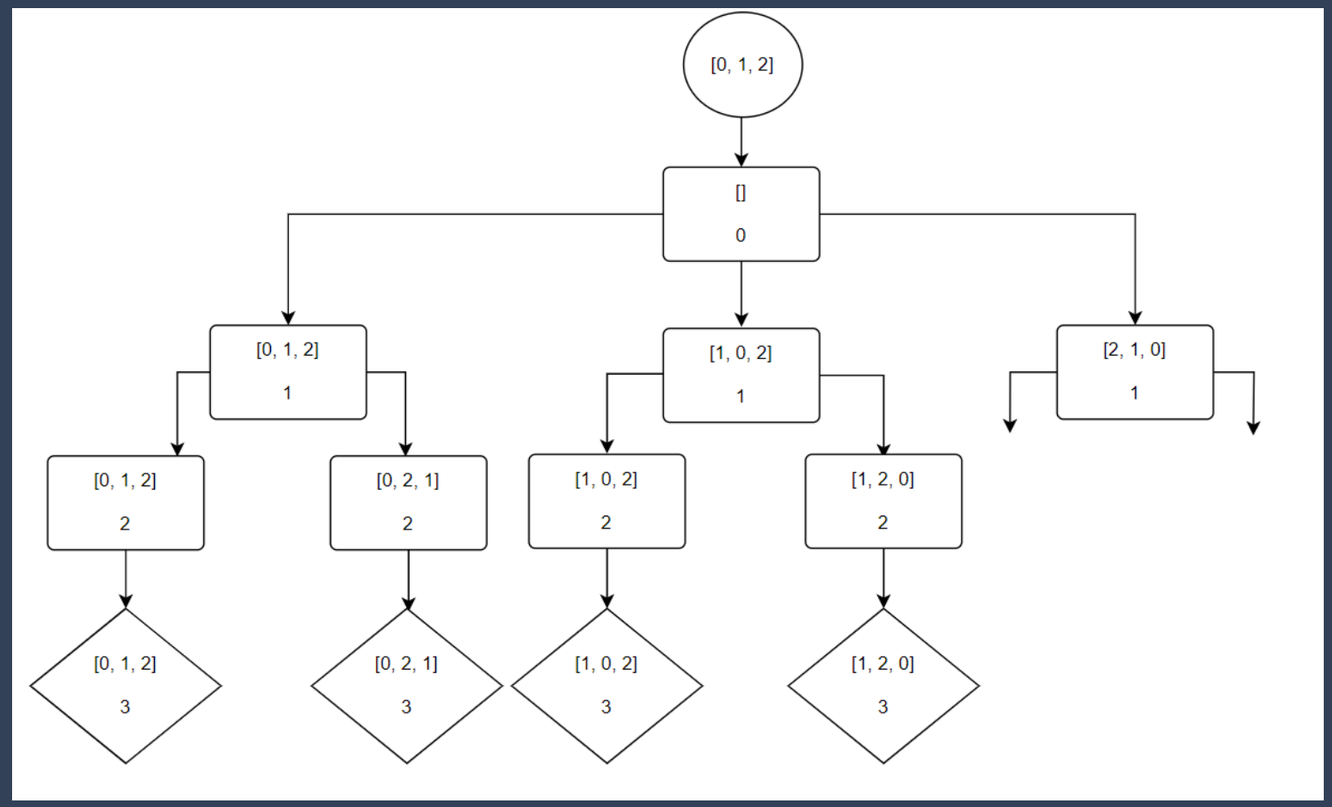
idx_pick ∈ [0, N-1]

Permutations - Code

{ik} INTERVIEW KICKSTART

```
function subordinate(input, idx_subproblem, partial_solution, result):
    if idx_subproblem == len(input):
        result.push(copy(partial_solution))
        return

    for idx_pick <- idx_subproblem to len(input) - 1:
        swap(idx_subproblem, idx_pick)
        subordinate(input, idx_subproblem + 1, partial_solution, result)
        swap(idx_pick, idx_subproblem)

function root_manager(input):
    result = []

    subordinate(input, 0, copy(input), result)

    return result

------
```

Leaf nodes time complexity: $O(N \cdot N!)$



Leaf nodes: $N!$

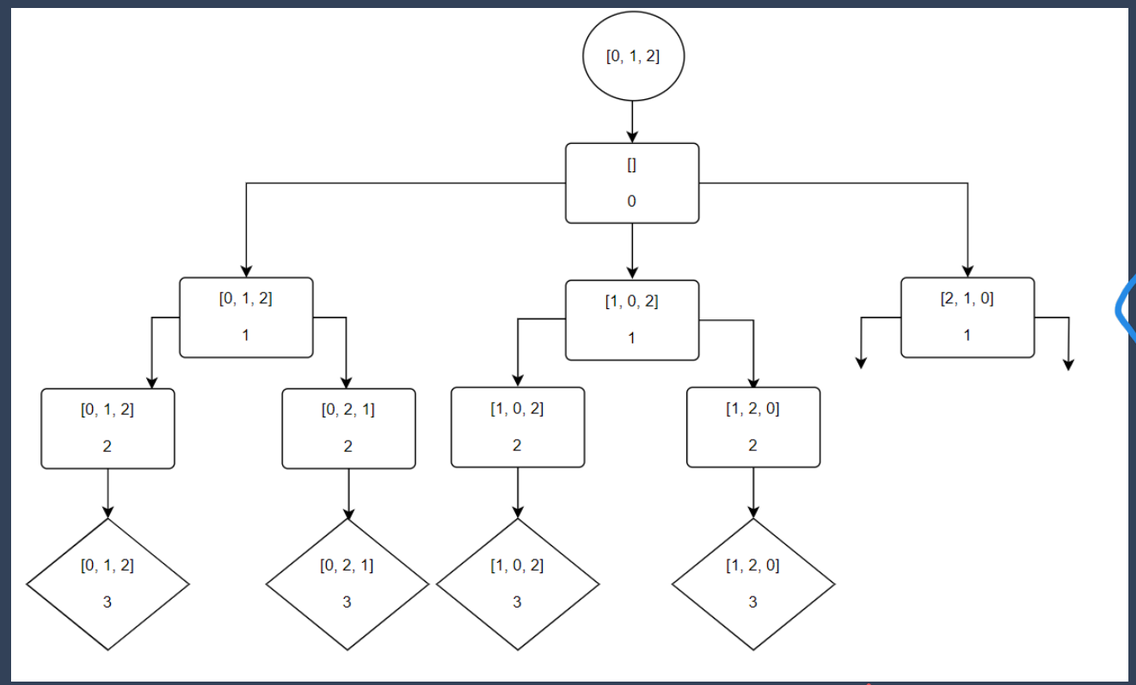
Leaf nodes work:

$O(N)$

{ik} INTERVIEW KICKSTART

3 · 2 · 1 = 3!

Time complexity (internal nodes)



3

3 · 2

3 · 2 · 1

Internal nodes work: O(1)

Internal nodes

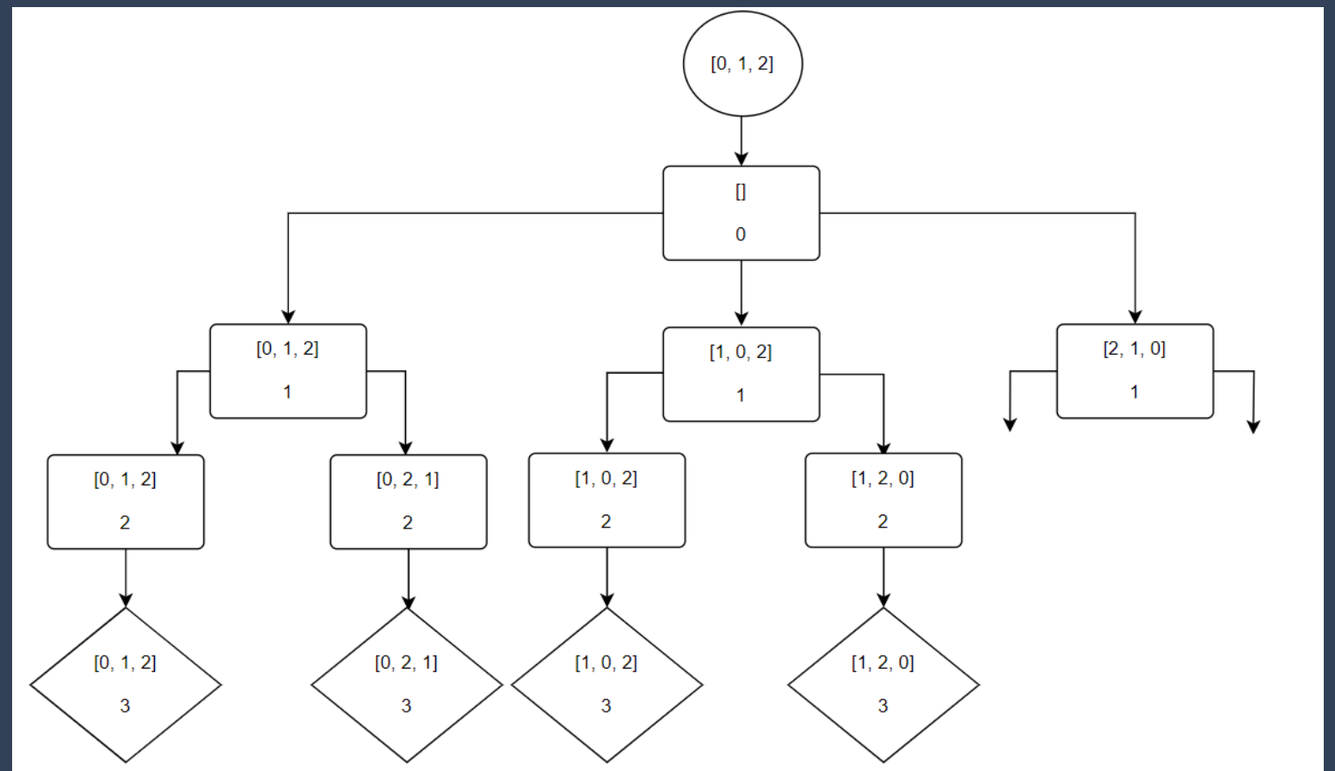
$3 + 3 \cdot 2 + 3 \cdot 2 \cdot 1$

$N + N \cdot (N-1) + N(N-1)(N-2)$

$< N!$ $< N!$ $N!$

N

$O(N \cdot N!)$



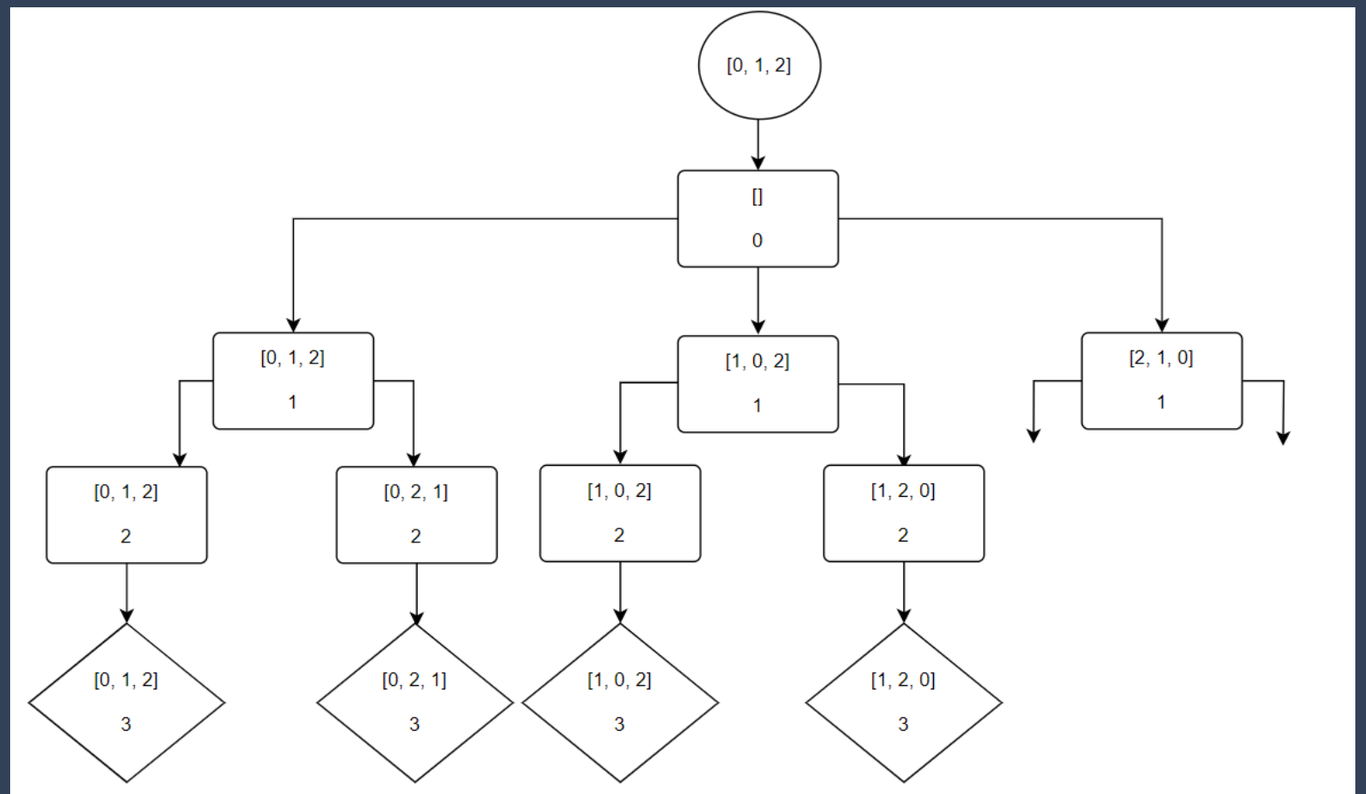
Time complexity

Internal nodes: O(n * n!)

Leaf nodes: O(n * n !)

Result: O(2 * n * n!) = **O(n * n!)**

{ik} INTERVIEW KICKSTART



Return result space?

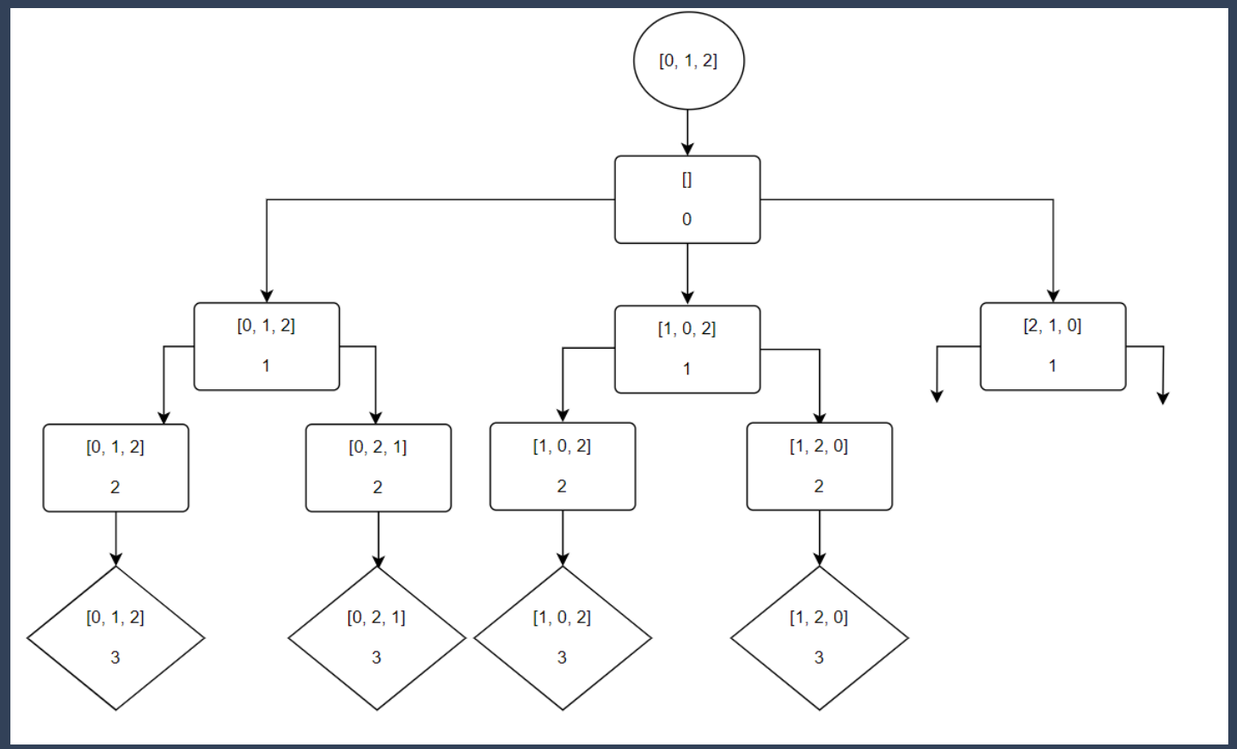
$N!$

N

$O(N \cdot N!)$

{ik} INTERVIEW KICKSTART

Space complexity (call stack) = $N + N \simeq O(N)$



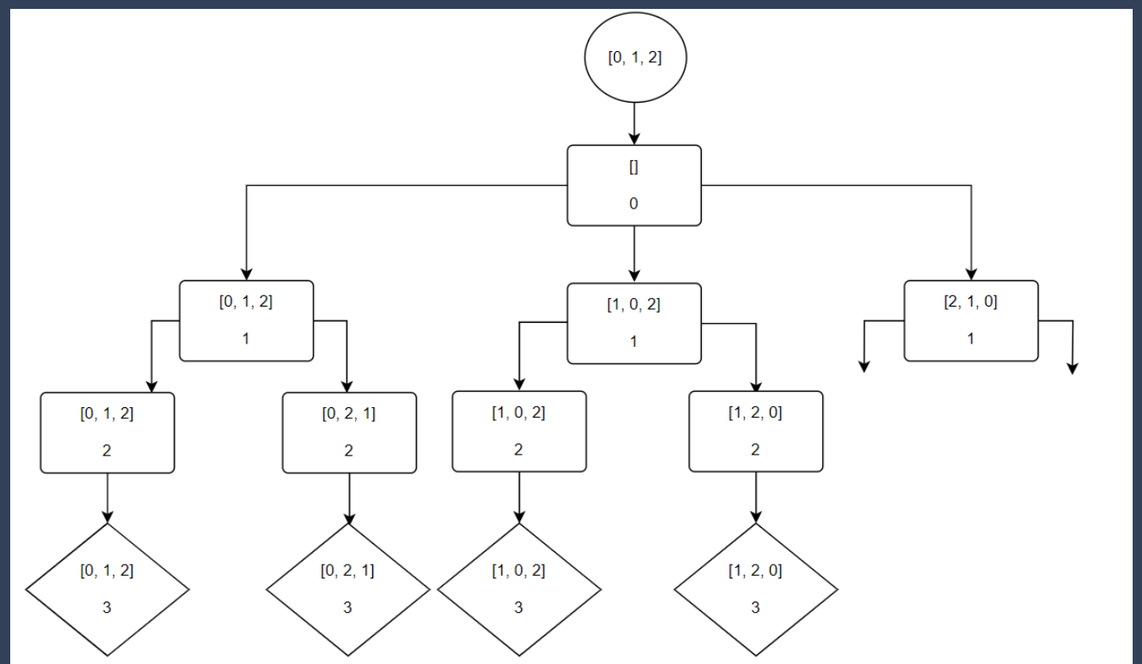
Leaf node memory: $O(N)$

Max # of internal nodes on the stack? N

Space needed for an internal node? 1

{ik} INTERVIEW KICKSTART

Space complexity (call stack) =



Space complexity

Return result: O(n * n!)

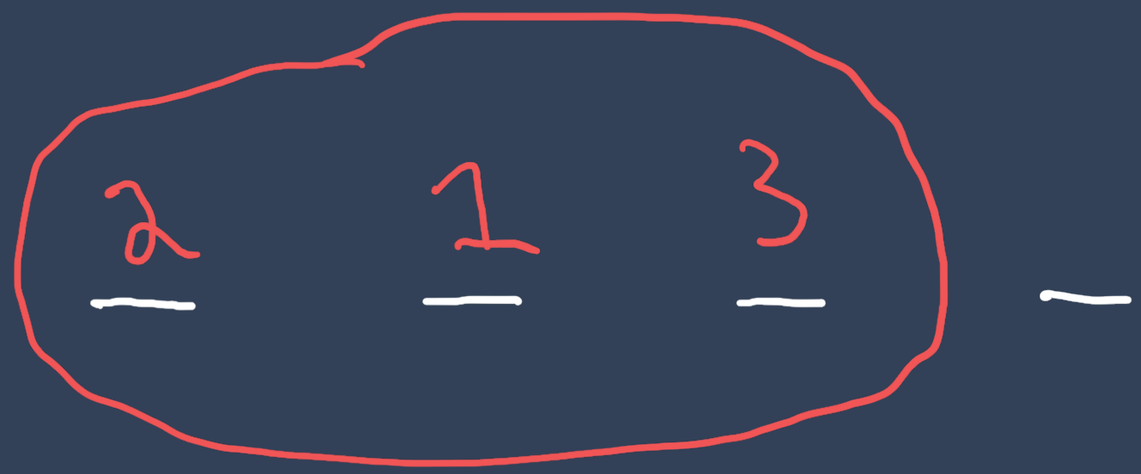
Call stack: O(n)

Result: O(n * n! + n) = **O(n * n!)**

{ik} INTERVIEW KICKSTART

Backtracking

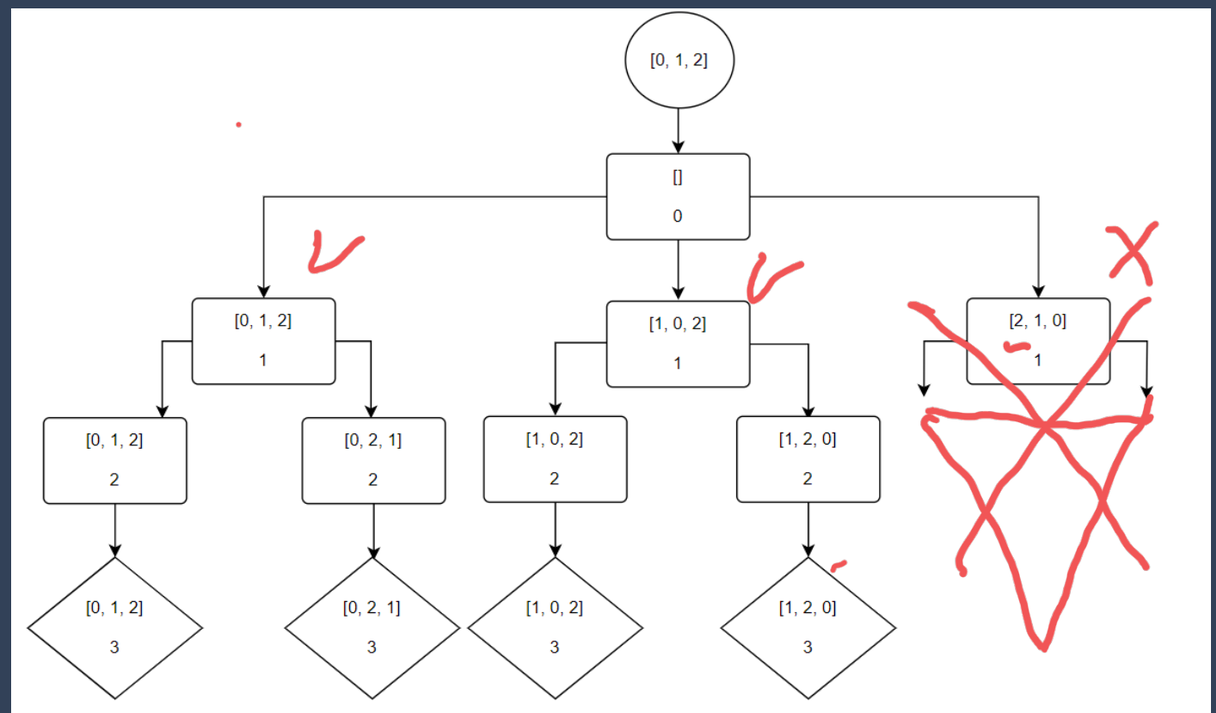
- Sometimes we don’t have to finish filling all the blanks



{ik} INTERVIEW KICKSTART

Backtracking: constraint and pruning

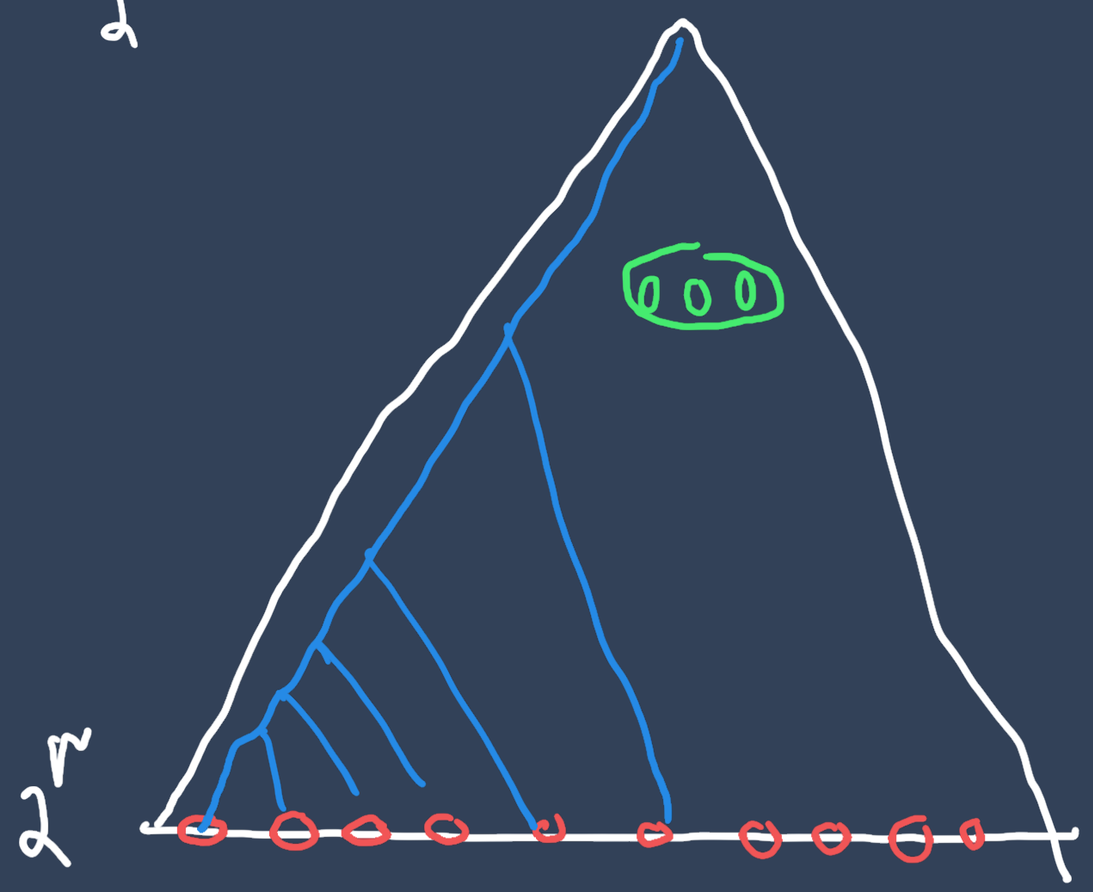
- What if someone asks you to return all possible permutations where 0 does not appear to the right of 2?



{ik} INTERVIEW KICKSTART

Backtracking: approach

- Let’s say you’re given problems with the following characteristics
    - The width of the tree for a particular problem is potentially large
    - Solutions tend to not be close to the root node
    - Would you consider DFS or BFS?



Backtracking

- Solves for a constraint
- Depth-First Search
- Heuristic
- Pruning
    - Worst case scenario: same as brute force
    - In practice, leads to significant gains in running time (sometimes exponential)

Backtracking comparison

Recursion w/o backtracking so far

- Base case
  - Action
    - Go back in the tree
  - When:
    - State limit
- Recursive case
  - Internal nodes
  - Filling in the blanks

- Recursion with backtracking
  - Base case
    - Action
      - Go back in the tree
    - When
      - State limit
      - **Pruning case**
  - Recursive case
    - Internal nodes
    - Filling in the blanks

5-minute break

- We'll be right back ☺

$\binom{N}{K}$

77. Combinations Medium

Given two integers `n` and `k`, return *all possible combinations of* `k` *numbers out of the range* `[1, n]`.

You may return the answer in **any order**.

Example 1:

[1,2,3,4]

```
Input: n = 4, k = 2
Output:
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

Constraints:

- `1 <= n <= 20`
- `1 <= k <= n`

n = 3

1

2

3

4

[ ]

[ ] [1]

[1] [1, 2]

[1, 2] [1, 2, 3] 3

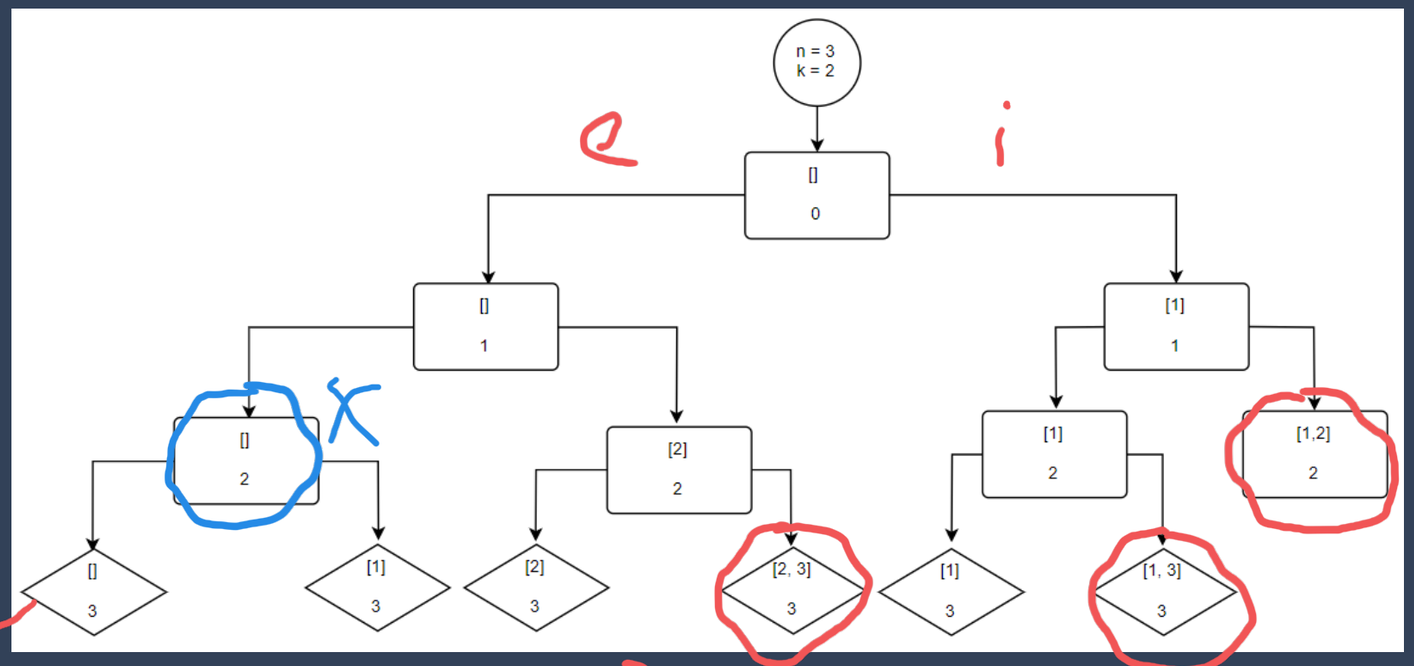
Remembering subsets

- Set = {1,2,3,4}
- Subsets
  - {}
  - {1}, {2}, {3}, {4}
  - {1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4} K=2
  - {1,2,3}, {1,2,4}, {1,3,4}, {2,3,4} K=3
  - {1,2,3,4}

Combinations: different approaches

- Two ways of rephrasing
- Let's say that n = 3 and k = 2
- #1: Find all subsets of size 2 using [1,2,3] as your elements
    - We'll show you that one
- #2: Arrange [1,2,3] in groups of 2 such that the order doesn't matter
    - Your exercise: think about how to adapt the permutations solution

Combinations: constraint



How do we know we've found a solution?

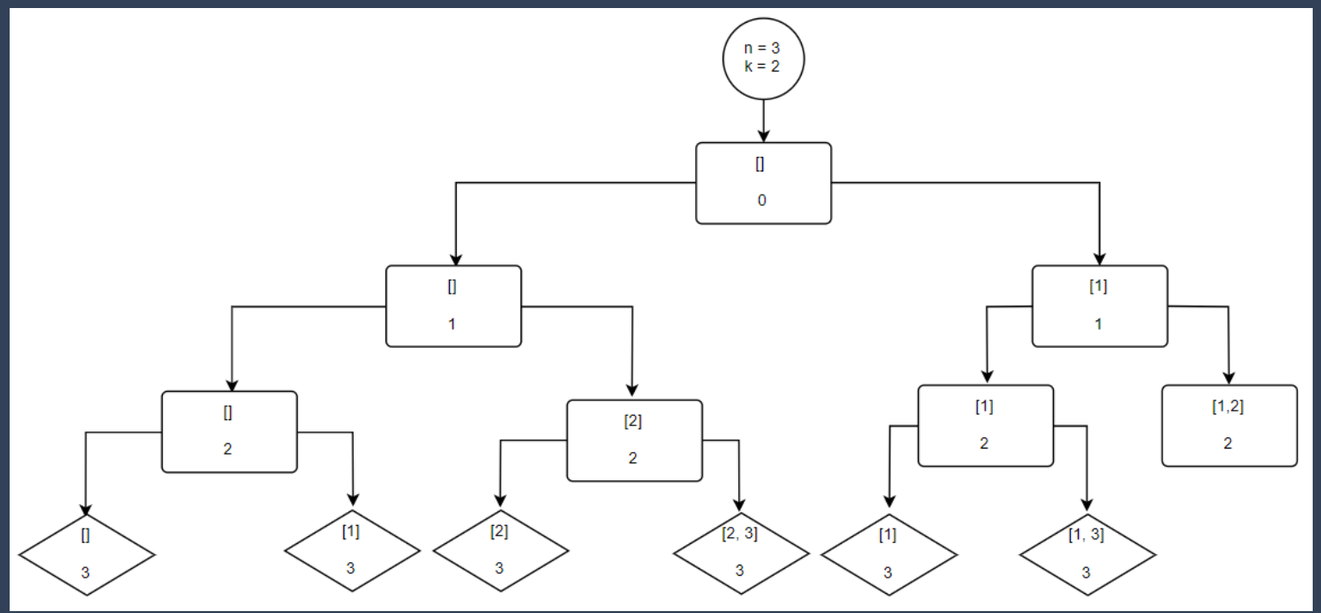
What is the state limit?

Combinations - Code

{ik} INTERVIEW KICKSTART


{ik}
INTERVIEW
KICKSTART

Time complexity = $O(N \cdot 2^N)$



Leaf nodes work: $O(N)$

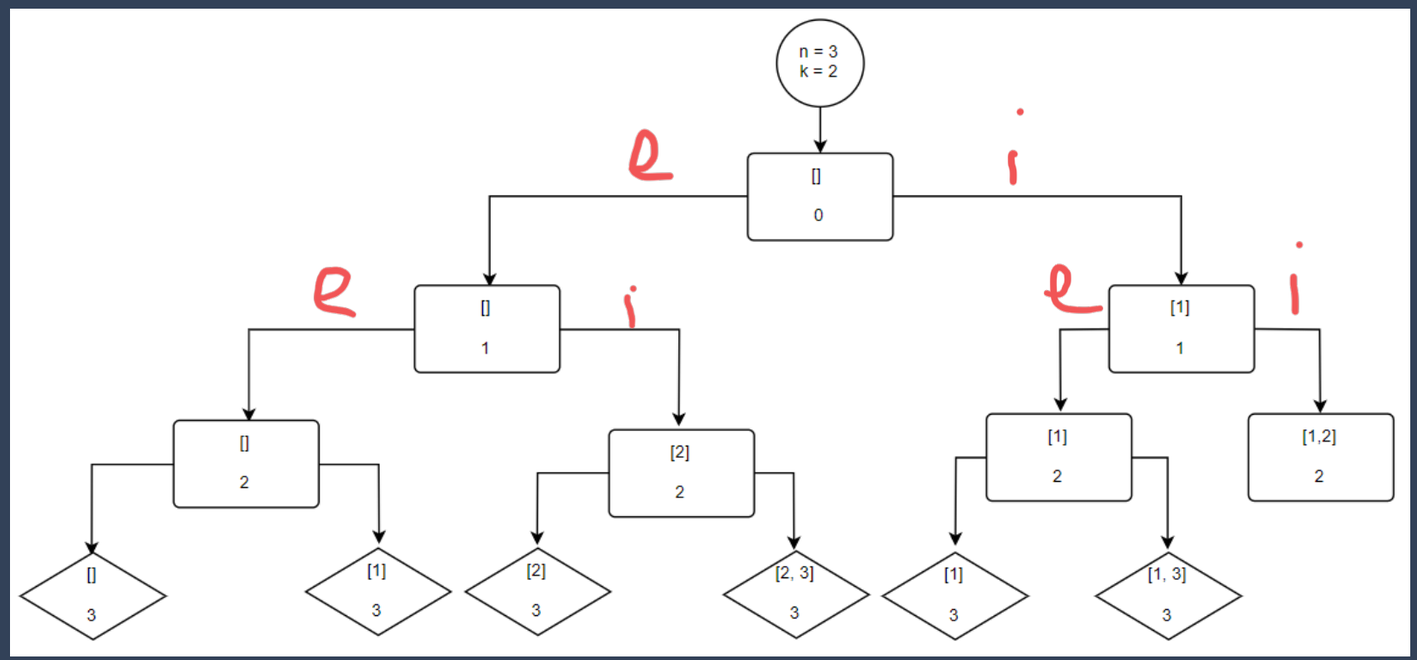
Internal nodes work: $O(1)$

nodes: $O(2^N)$

{ik} INTERVIEW KICKSTART

2^N

$\binom{N}{N} = 1$

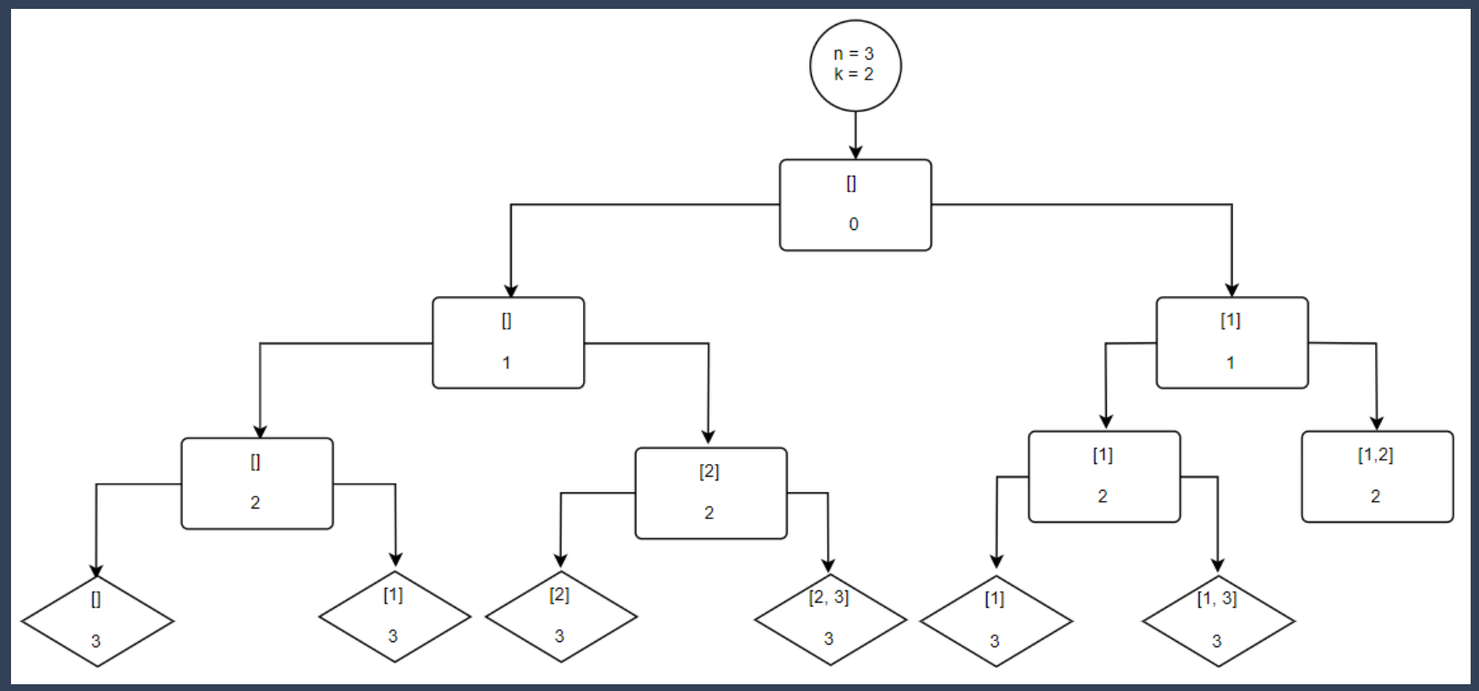


Return result space

$N \cdot \binom{N}{K}$

{ik} INTERVIEW KICKSTART

Space complexity (call stack) = $O(N)$



Leaf nodes space?

$O(N)$

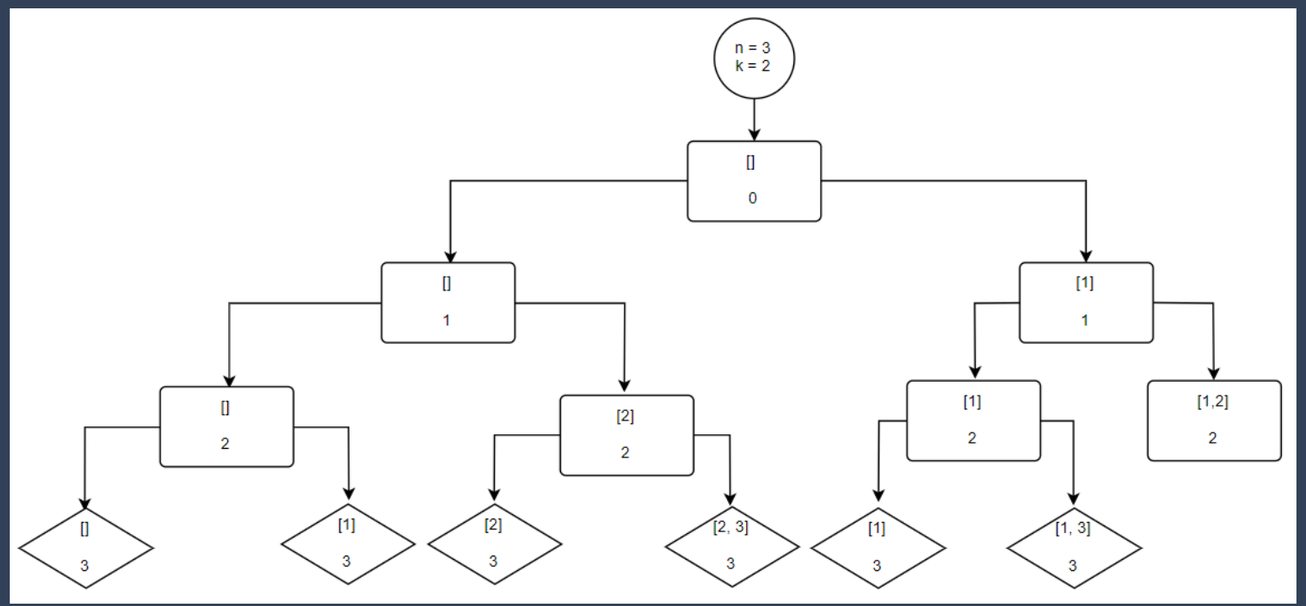
Internal nodes space?

$O(1)$

Max # internal nodes on stack?

$O(N)$

{ik} INTERVIEW KICKSTART



Space complexity

Return result: O(n * ~~2^n~~)

Call stack: O(n)

Result: O((n * ~~2^n~~) + n) = **O(n * ~~2^n~~)**

{ik} INTERVIEW KICKSTART

Other problems to practice

{ik} INTERVIEW KICKSTART

Other problems to practice

- Solutions to other backtracking problems (such as N queens) are covered in the supplementary lecture on UpLevel

- Practice the problems on our platform

Subset Sum: variations

- Given a list of numbers, how many subsets add up to a given sum?
  - Variations: how many subsets are smaller / higher than a given sum
- **Enumeration**: return all subsets
- **Counting**: number of subsets
- **Decision**: is there a subset
- **Optimization**: which subset is the best considering weights (knapsack)

Subset Sum

- Given
  - a list of positive numbers L
  - a positive constant k

- How many subsets of L add up to a number <= k

Other recursive problems for you to practice

22. Generate Parentheses

Medium 3017 188 Favorite Share

Given *n* pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given *n* = 3, a solution set is:

```
[
  "((()))",
  "(()())",
  "(())()",
  "()(())",
  "()()()"
]
```

{ik} INTERVIEW KICKSTART

Other recursive problems for you to practice

51. N-Queens

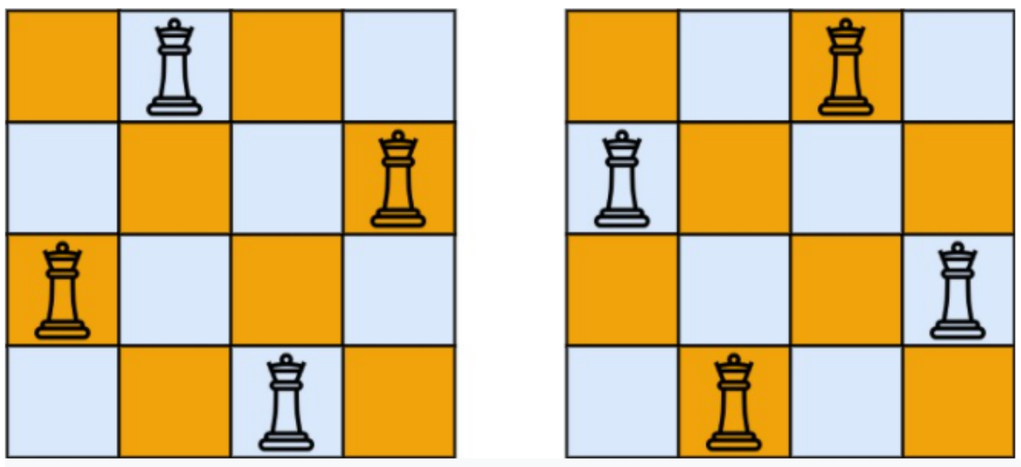
Hard 2628 98 Add to List Share

The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *all distinct solutions to the **n-queens puzzle***.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.

Example 1:



{ik} INTERVIEW KICKSTART

Other recursive problems for you to practice

90. Subsets II

Medium 2242 100 Add to List Share

Given an integer array `nums` that may contain duplicates, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

Example 1:

```
Input: nums = [1,2,2]
Output: [[],[1],[1,2],[1,2,2],[2],[2,2]]
```

Example 2:

```
Input: nums = [0]
Output: [[],[0]]
```

{ik} INTERVIEW KICKSTART

Other recursive problems for you to practice

47. Permutations II

Medium 2723 76 Add to List Share

Given a collection of numbers, `nums`, that might contain duplicates, return *all possible unique permutations* ***in any order***.

Example 1:

```
Input: nums = [1,1,2]
Output:
[[1,1,2],
 [1,2,1],
 [2,1,1]]
```

Constraints:

- `1 <= nums.length <= 8`
- `-10 <= nums[i] <= 10`

Other recursive problems for you to practice

131. Palindrome Partitioning

Medium 1013 39 Favorite Share

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of *s*.

Example:

```
Input: "aab"
Output:
[
  ["aa","b"],
  ["a","a","b"]
]
```

50. Pow(x, n)

Medium 1036 2474 Favorite Share

Implement pow(*x*, *n*), which calculates *x* raised to the power *n* (x^n).

Example 1:

```
Input: 2.00000, 10
Output: 1024.00000
```

Example 2:

```
Input: 2.10000, 3
Output: 9.26100
```

Example 3:

```
Input: 2.00000, -2
Output: 0.25000
Explanation: 2^-2 = 1/2^2 = 1/4 = 0.25
```

Note:

- -100.0 < *x* < 100.0
- *n* is a 32-bit signed integer, within the range $[-2^{31}, 2^{31} - 1]$

Solutions library

- https://drive.google.com/drive/folders/1NFn1kcyncXeHE4kjRasxeLFYjKDt5z--

Questions

Exercises

- We’ll solve some of the following problems if we have extra time

- Otherwise, practice them using our tools

Subset Sum

- Given
  - a list of positive numbers L
  - a positive constant k

- How many subsets of L add up to a number <= k

Subset Sum

- Given
  - a list of positive numbers L
  - a positive constant k

- How many subsets of L add up to a number <= k

Subset Sum: time and memory complexity

- Time complexity
- Memory complexity

Generate parentheses

22. Generate Parentheses

Medium 3017 188 Favorite Share

Given *n* pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given *n* = 3, a solution set is:

```
[
  "((()))",
  "(()())",
  "(())()",
  "()(())",
  "()()()"
]
```

{ik} INTERVIEW KICKSTART

Designing the solution

- How do we fill in the blanks?
- How do we know when a partial solution is well formed?

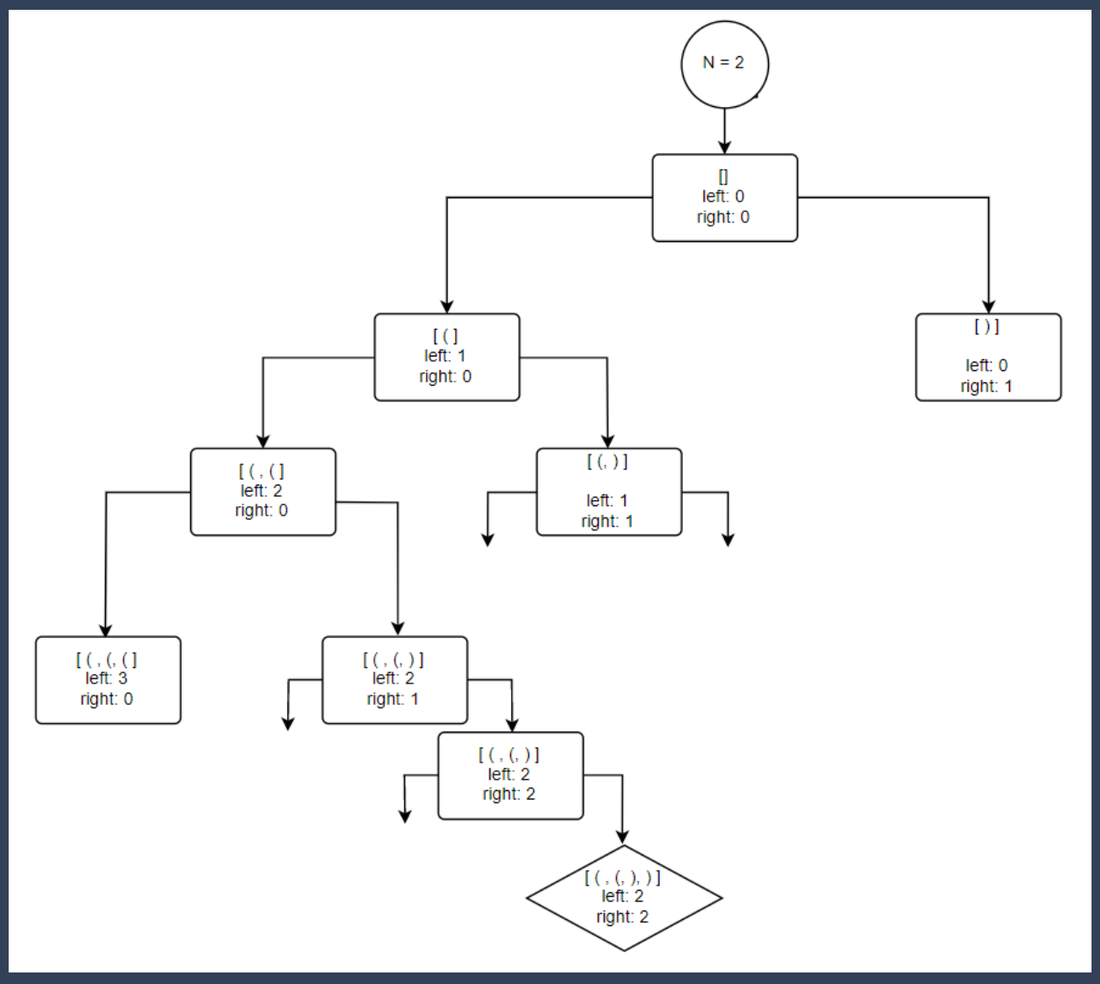
Generate Parentheses - Tree


{ik}
INTERVIEW
KICKSTART

Generate Parentheses - Code


{ik}
INTERVIEW
KICKSTART

Time complexity =

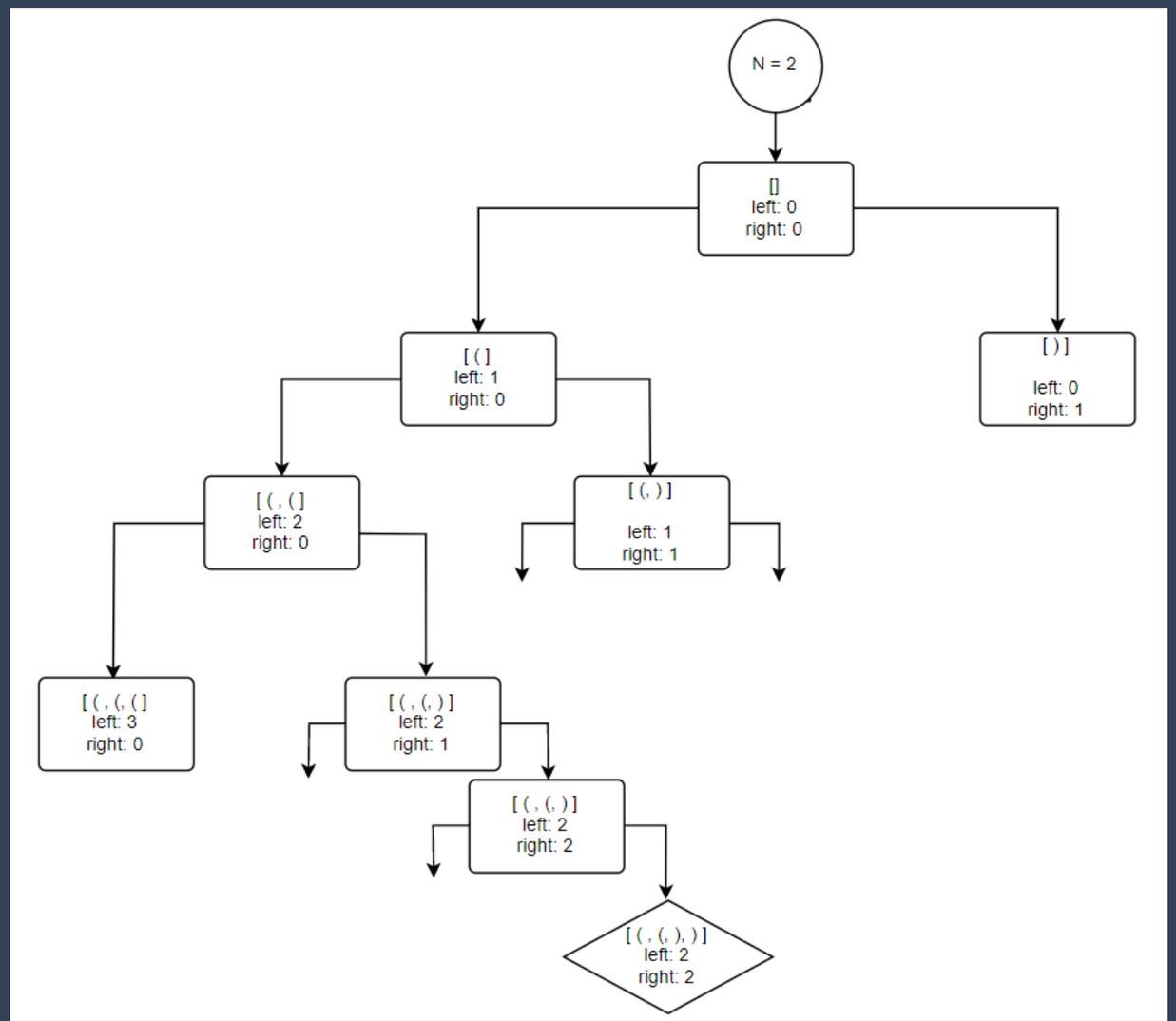


nodes:

nodes work:

Result:

{ik} INTERVIEW KICKSTART



Return result space?

Space for everything else?

{ik} INTERVIEW KICKSTART

Credits: David Hulak and Omkar Deshpande

{ik} INTERVIEW KICKSTART