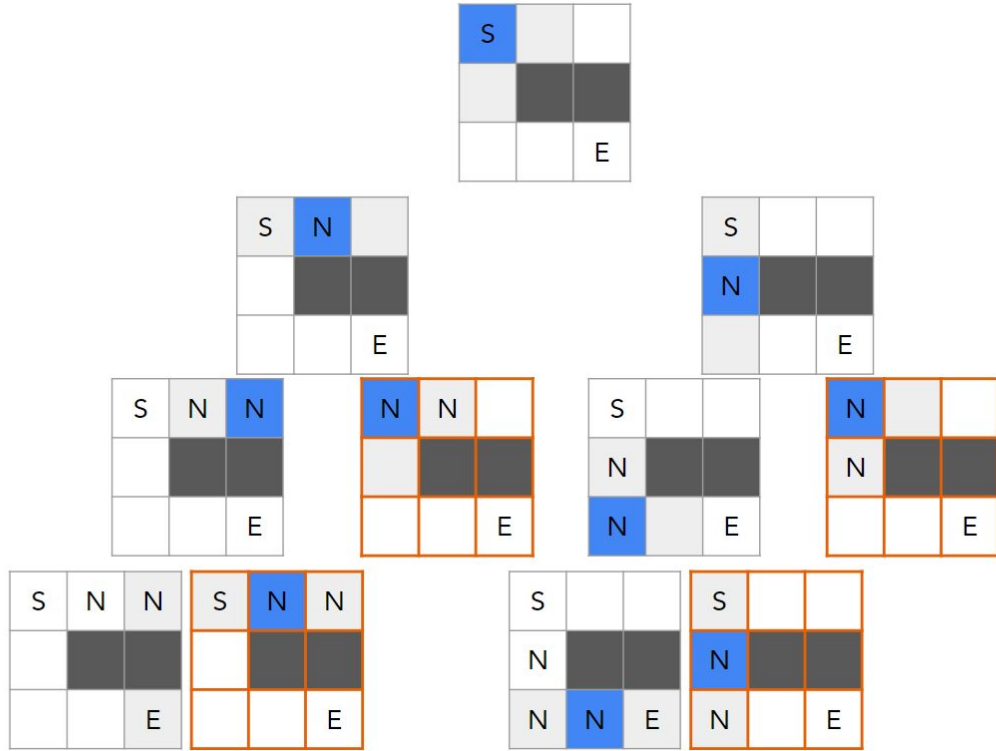


Recursion II



Lecture Flow

- 1) Pre-requisites
- 2) Review of previous lecture
- 3) Backtracking
- 4) Time and space complexity
- 5) Things to Pay Attention (common pitfalls)
- 6) Divide and Conquer
- 7) Practice questions
- 8) Resources
- 9) Quote of the day

Pre-requisites

- Recursion I
- Time and Space Complexity Analysis

Revision

Instances where you have seen recursion useful from last lecture and practice?

- Unidentified number of levels
 - [Decode String](#)
- Input is expressed using recursive rules explicitly or implicitly
 - [Fibonacci](#)
 - [Pascal's Triangles](#)
 - [Find Kth Bit in Nth Binary String](#)
- Complete Search Problems
 - [Predict the Winner](#)
- Tree Structure Traversal/Tree related problems
 - [Lowest common ancestor of a binary tree](#)
 - [Validate BST](#)

Let's visualize fibonacci numbers

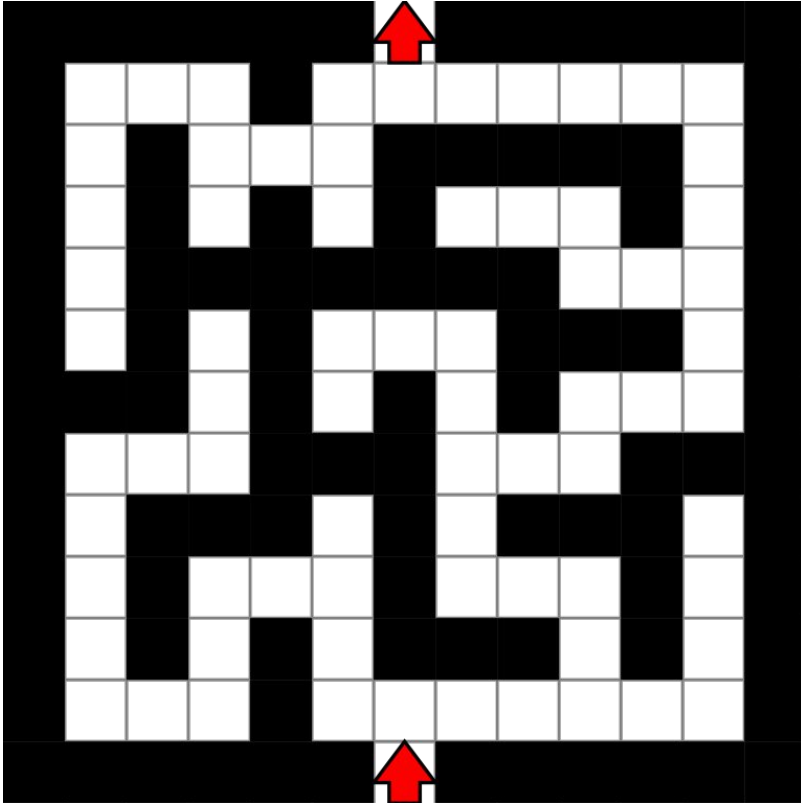
[Link](#)

Is that all recursion can do?

Today, we will add more areas
where recursion shines

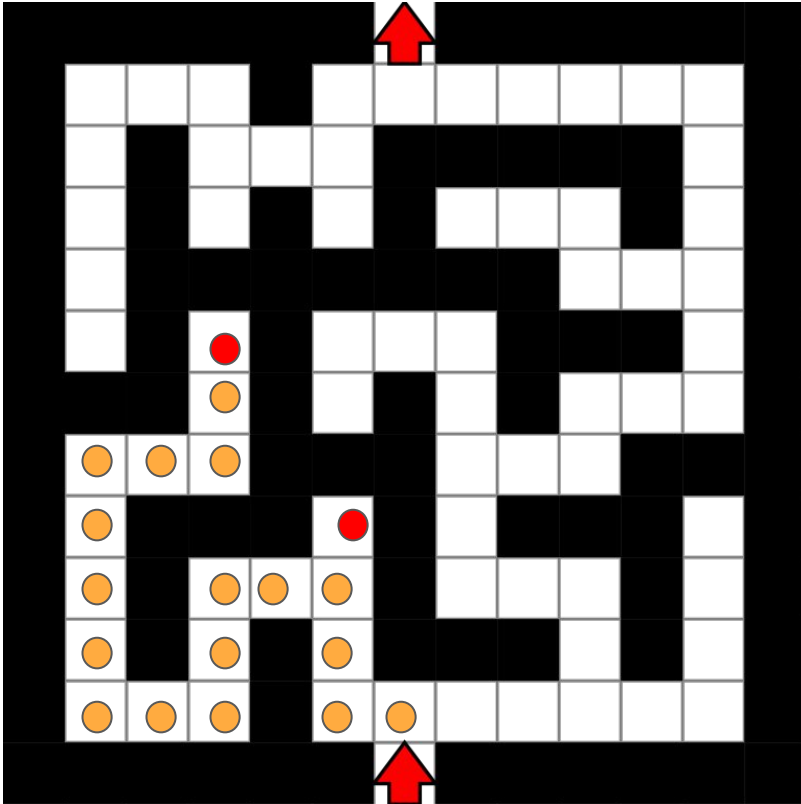
Backtracking

How would you find your way out?



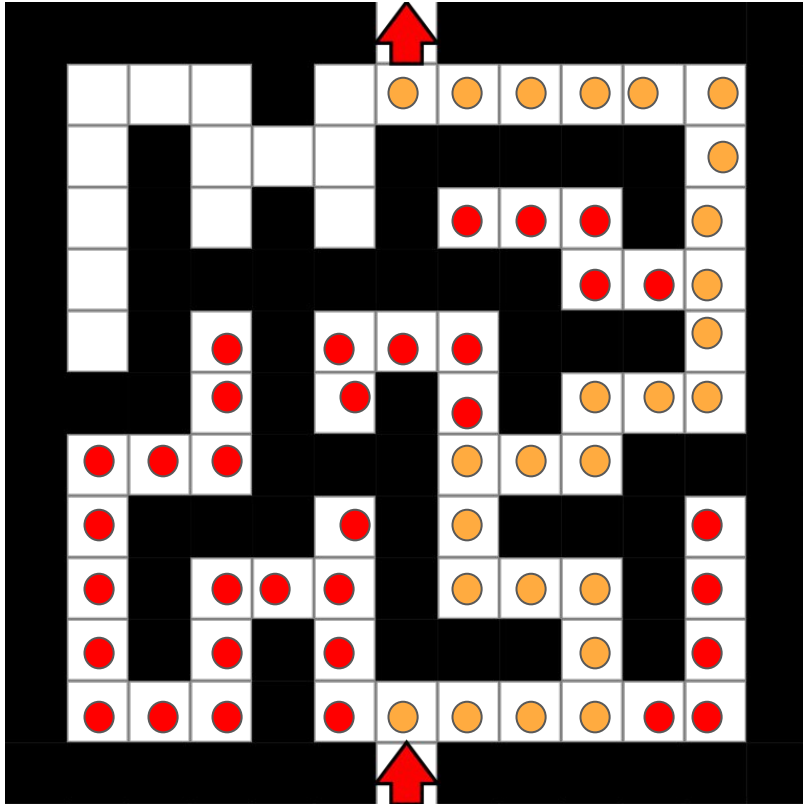
- Let's **put peanuts** on our path
- When you reach a **dead end**, go **back** to where you came from and
- **Explore** a different path
- Red peanut to mark we have been here
- Orange peanut currently in exploring path

How would you find your way out?



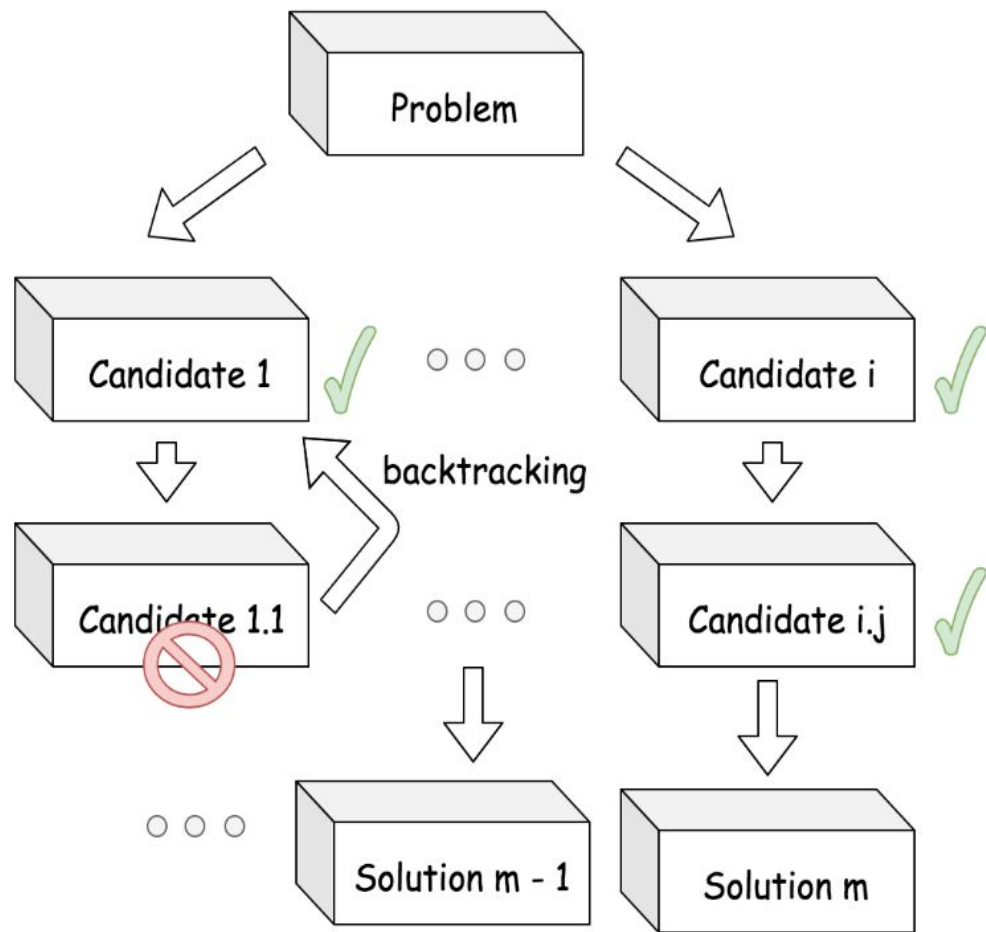
- Let's **put peanuts** on our path
- When you reach a **dead end**, go **back** to where you came from and
- **Explore** a different path
- Red peanut to mark we have been here
- Orange peanut currently in exploring path

How would you find your way out?



- Let's **put peanuts** on our path
- When you reach a **dead end**, go **back** to where you came from and
- **Explore** a different path

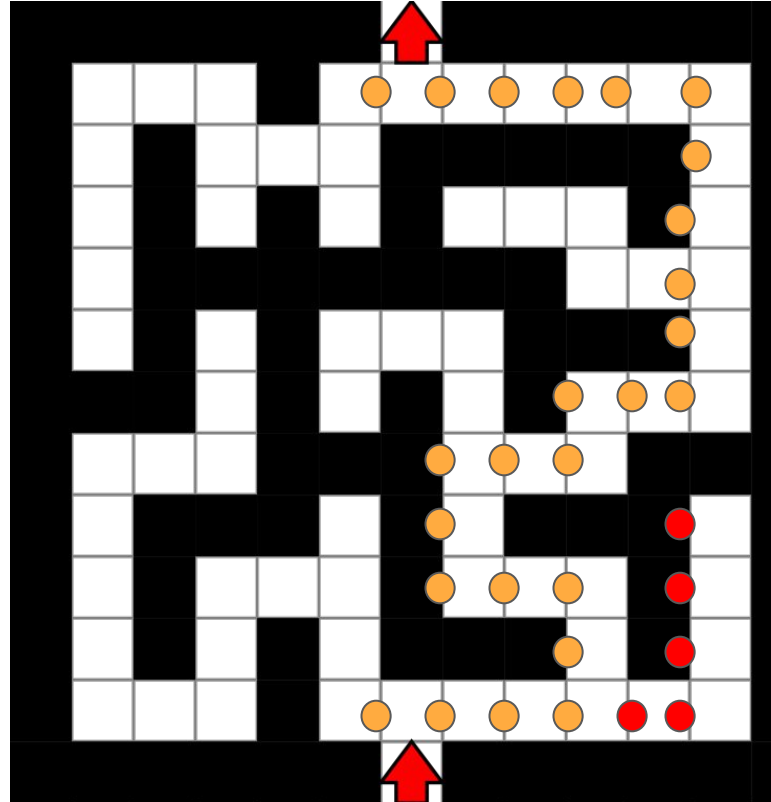
- Red peanut to mark we have been here
- Orange peanut currently in exploring path



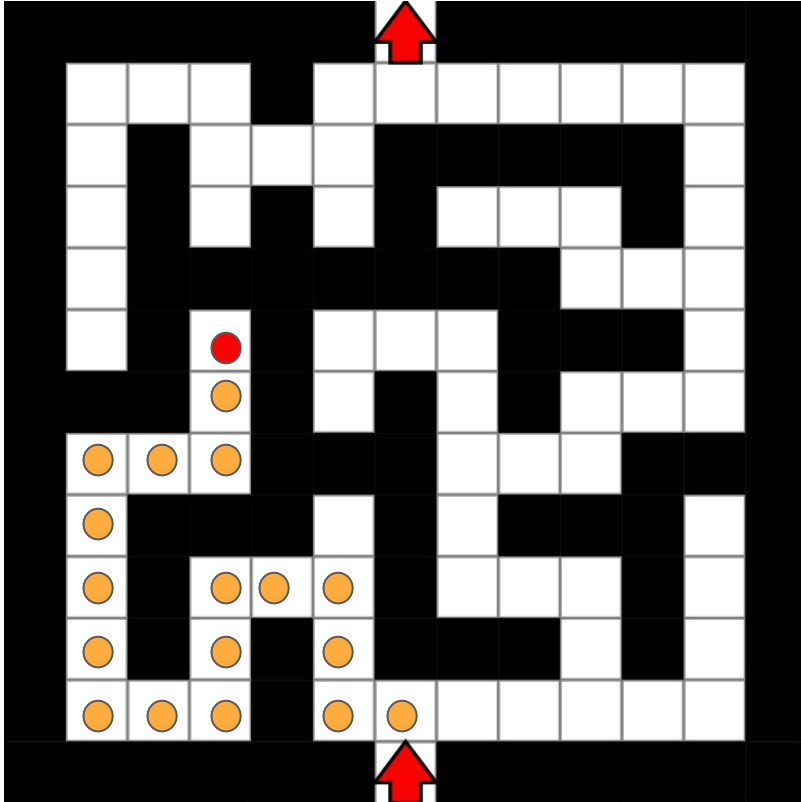
For our maze our candidates at each cell where

- Go forward
- Take right Turn
- Take left Turn
- Go backward?

How would we simulate this with code?



Template



```
def backtrack(candidate):
    if candidate is a solution:
        # process or output a candidate
        return

    # handle basecases

    # iterate all possible candidates.
    for next_candidate in list_of_candidates:
        if next_candidate is valid:
            # try this partial candidate solution
            place(next_candidate)

            # given the candidate, explore further.
            backtrack(next_candidate)

        # backtrack
        remove(next_candidate)
```

What are the next candidates for this state?

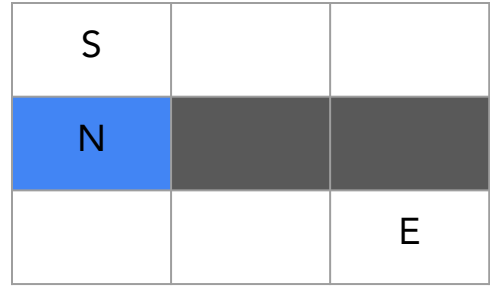
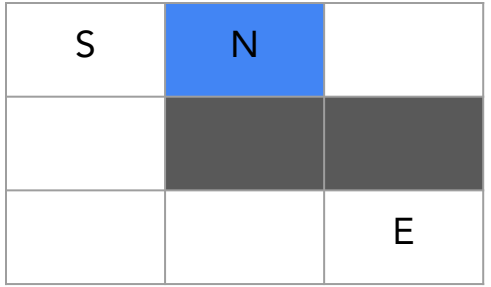
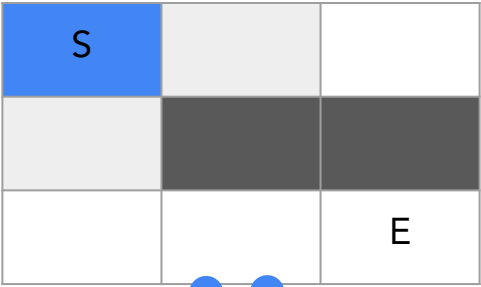
S		
		E

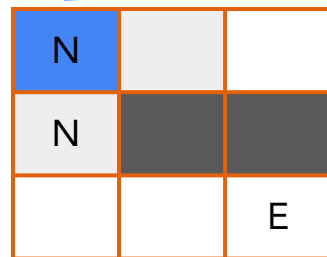
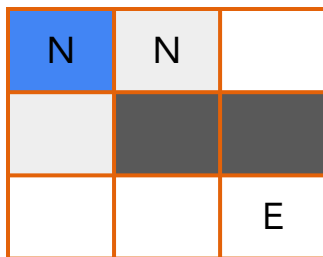
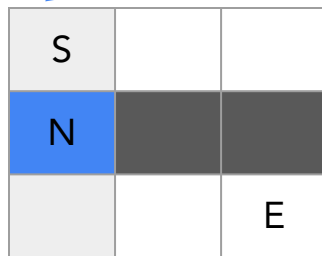
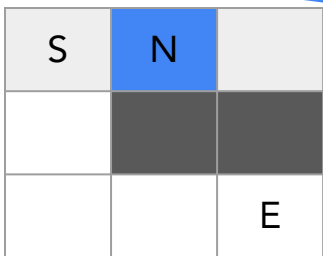
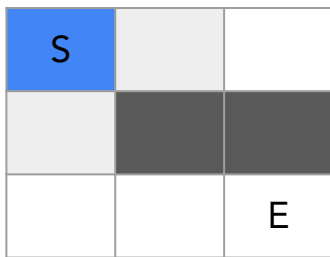
Legend

S - Start

N - Next

E - End





S		
		E

S	N	
		E

S		
N		
		E

S	N	N
		E

N	N	
		E

S		
N		
N		E

N		
N		
		E

S	N	N
		E

S	N	N
		E

S		
N		
N	N	E

S		
N		
N		E

S		
		E

S	N	
		E

S		
N		
		E

S	N	N
		E

N	N	
		E

S		
N		
N		E

N		
N		
		E

S	N	N
		E

S	N	N
		E

S		
N		
N	N	E

S		
N		
N		E

S		
N		
N	N	E

S		
N		
N	N	E

Reflection

- How do we describe one candidate?
 - i, j, grid ; where (i, j) is our current position
- Why was grid one of the states?
- Why can't the next state be our caller's state?
- What is the base case?

Backtracking is a method to solve problems by trying out different options and exploring **all possible paths** until a solution is found.

If it reaches a dead end, it goes back to the last decision point and tries another option until all possibilities have been exhausted.

Types of backtracking

- Decision Problem
 - In this, we search for a feasible solution.
- Optimization Problem
 - In this, we search for the best solution.
- Enumeration Problem
 - In this, we find all feasible solutions. (Permutation and Combination)

Use the backtracking template to solve the
following problem

Combinations

Medium



5.8K



178



Companies

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

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

Example 1:

Input: $n = 4, k = 2$

Output: `[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]`

Explanation: There are $4 \text{ choose } 2 = 6$ total combinations.

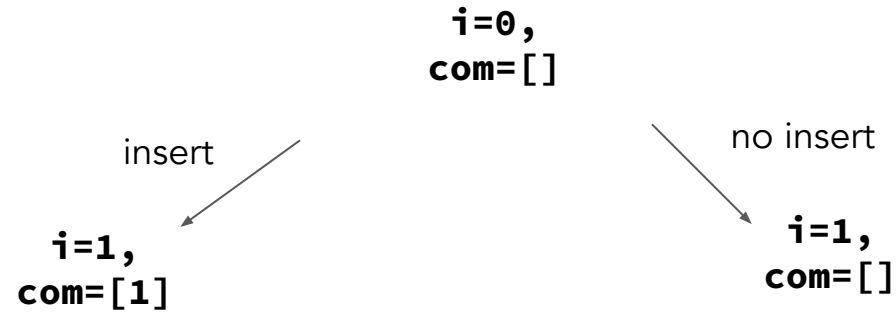
Note that combinations are unordered, i.e., `[1,2]` and `[2,1]` are considered to be the same combination.

nums = [1,2,3] | k = 2 <> [[1,2],[1,3],[2,3]]

i=0,
com=[]

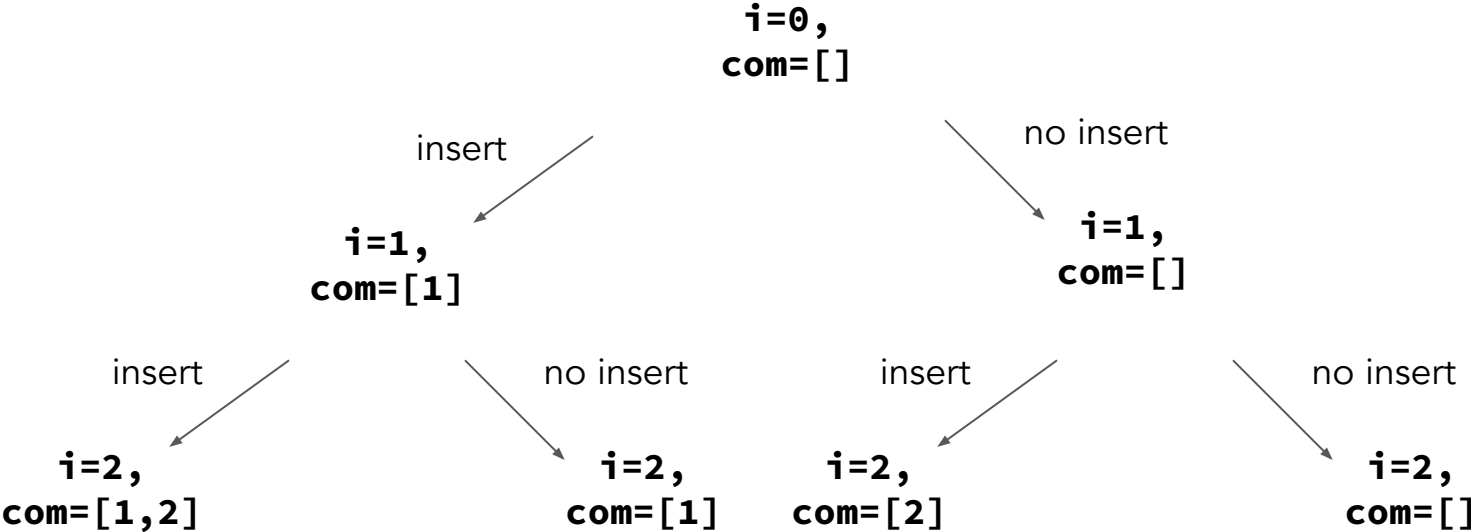
What are the
different calls at this
level?

nums = [1,2,3] | k = 2 <> [[1,2],[1,3],[2,3]]



What are the
different calls at this
level?

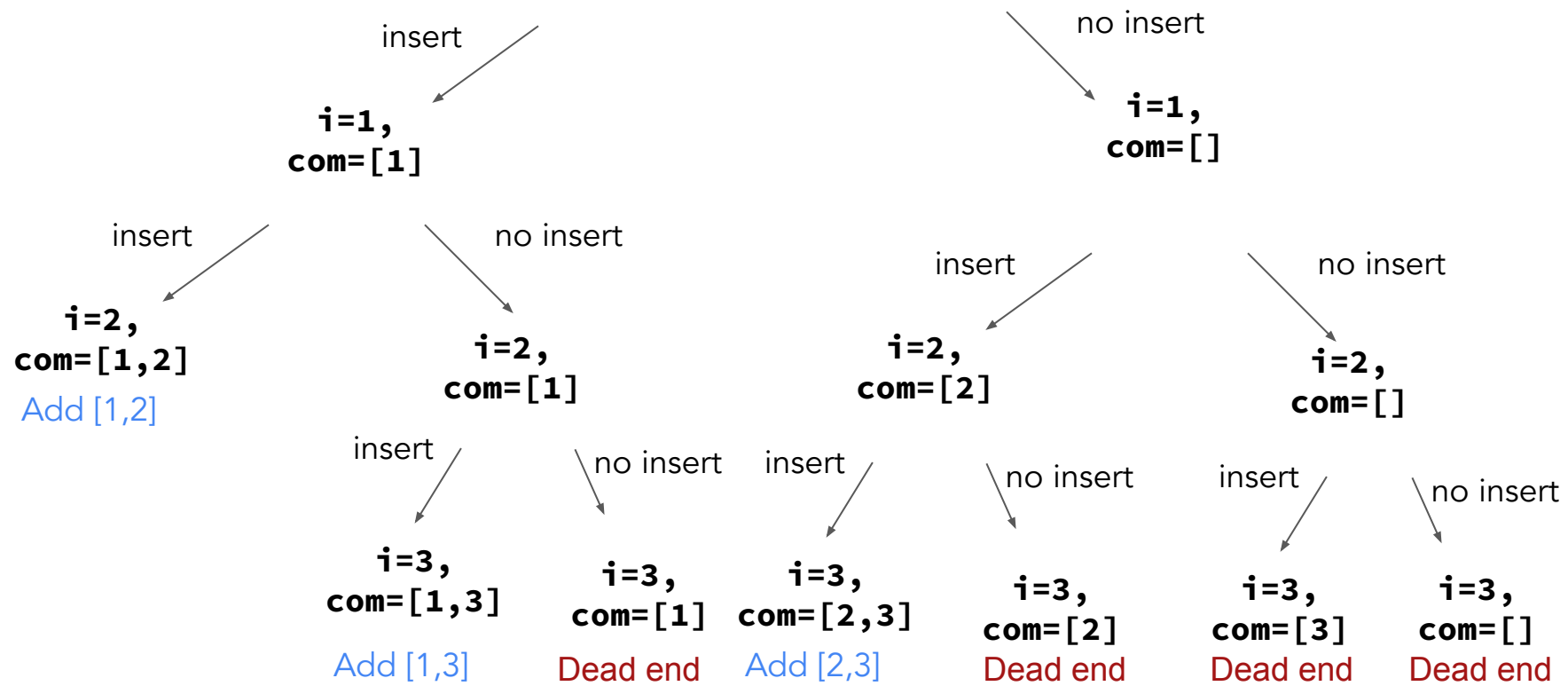
nums = [1,2,3] | k = 2 <> [[1,2],[1,3],[2,3]]



What are the
different calls at this
level?

nums = [1,2,3] | k = 2 <> [[1,2],[1,3],[2,3]]

**i=0,
com=[]**



Time Complexity

- We have two branches
- We have a depth of size n
- When ever we find an answer we add a list of size K to our answer.

Time for the recursion = branches \wedge depth

Time for the list addition = count of combinations $\ast K = [n! / (K!) \ast (n-K)!] \ast k$

$$O(2^n + \{n! / [(K!) \ast (n-K)!]\} \ast K)$$

Space Complexity

- We have the call stack
- **Note:** The array is pass by reference so it won't count as function space cost

Space Complexity = Depth * Space of function and arguments

$$O(n*s)$$

What type of backtracking was
the previous problem?

Splitting a String into Descending Consecutive values

You are given a string `s` that consists of only digits.

Check if we can split `s` into **two or more non-empty substrings** such that the **numerical values** of the substrings are in **descending order** and the **difference** between numerical values of every two **adjacent substrings** is equal to `1`.

- For example, the string `s = "0090089"` can be split into `["0090", "089"]` with numerical values `[90, 89]`. The values are in descending order and adjacent values differ by `1`, so this way is valid.
- Another example, the string `s = "001"` can be split into `["0", "01"]`, `["00", "1"]`, or `["0", "0", "1"]`. However all the ways are invalid because they have numerical values `[0, 1]`, `[0, 1]`, and `[0, 0, 1]` respectively, all of which are not in descending order.

Return `true` if it is possible to split `s` as described above, or `false` otherwise.

A **substring** is a contiguous sequence of characters in a string.

Example 2:

Input: `s = "050043"`

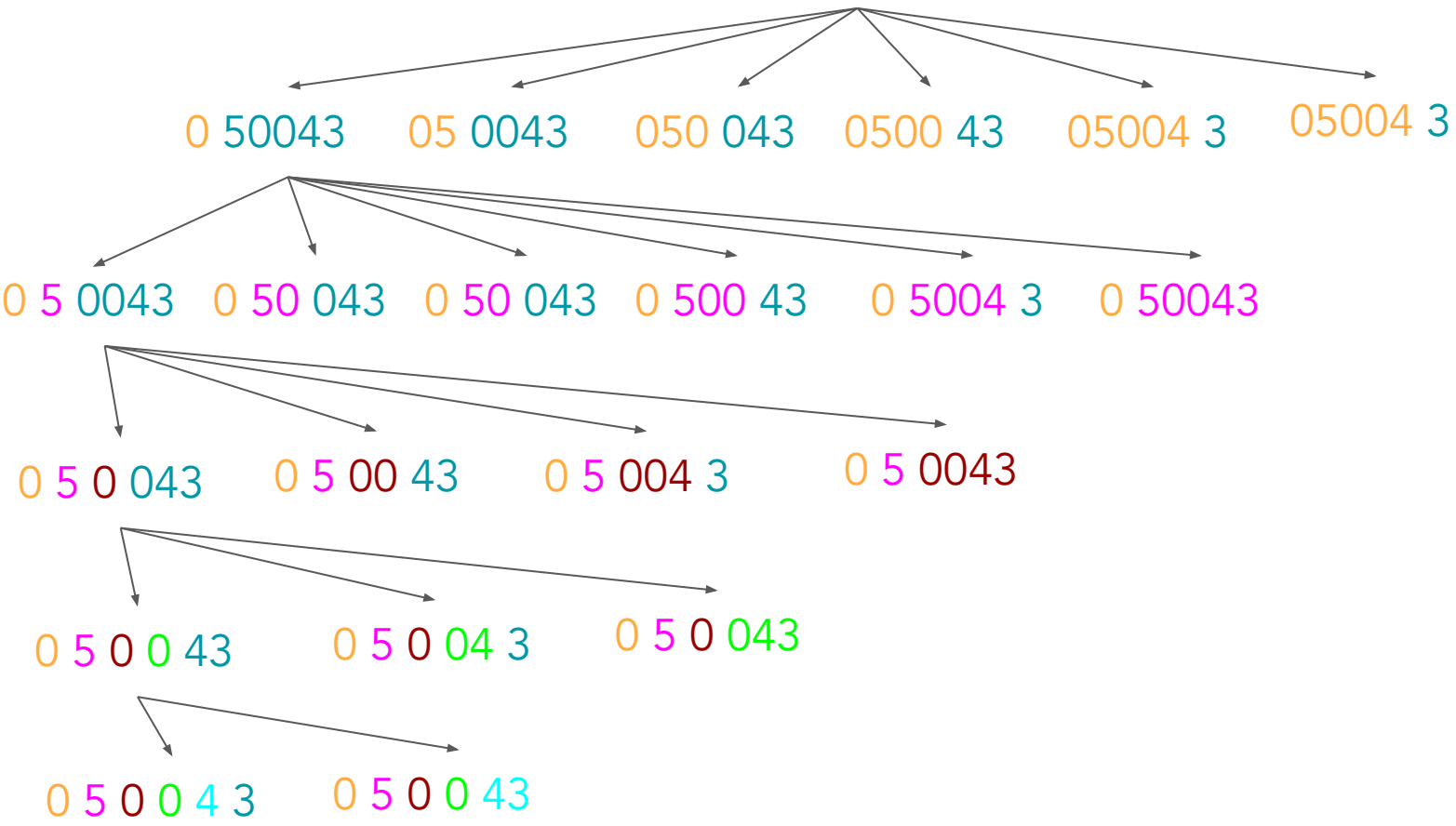
Output: `true`

Explanation: `s` can be split into `["05", "004", "3"]` with numerical values `[5, 4, 3]`.

The values are in descending order with adjacent values differing by `1`.

num = "050043"

050043



Time Complexity

- We have a depth of n
- At each step we have a choice of n

$$\text{Time Complexity} = O(n * (2^n))$$

Space Complexity

- Size of the call stack
- The array we are keeping:

Note: The array is global.

$$\text{Space Complexity} = O(n + n) = O(n)$$

What type of backtracking was
the previous problem?

Decision Problem: we were trying to find at least one solution

How would we approach an
optimization problem using
backtracking?

How would we approach an optimization problem using backtracking?

- Do the same thing we did for enumeration
- From all the solution, we pick the optimal solution

Fair Distribution of Cookies

Medium



👍 790

👎 40



Companies

You are given an integer array `cookies`, where `cookies[i]` denotes the number of cookies in the i^{th} bag. You are also given an integer `k` that denotes the number of children to distribute **all** the bags of cookies to. All the cookies in the same bag must go to the same child and cannot be split up.

The **unfairness** of a distribution is defined as the **maximum total** cookies obtained by a single child in the distribution.

Return *the **minimum** unfairness of all distributions.*

Can you draw what the possible branching would look like before you go to implementation?

cookie= [1,2,3] | bucket = []

i=0,
bucket=[0,0]

What are the
different calls at this
level?

cookie= [1,2,3] | bucket = []

i=0,
bucket=[0,0]

put at
0
↙
i=1,
bucket=[1,0]

↘ put at 1
i=1,
bucket=[0,1]

What are the
different calls at this
level?

cookie= [1,2,3] | bucket = []

**i=0,
bucket=[0,0]**

put at
0

**i=1,
bucket=[1,0]**

put at 1

**i=1,
bucket=[0,1]**

put at
0

**i=2,
bucket=[3,0]**

put at 1

**i=2,
bucket=[1,2]**

Put at 0

**i=2,
bucket=[2,1]**

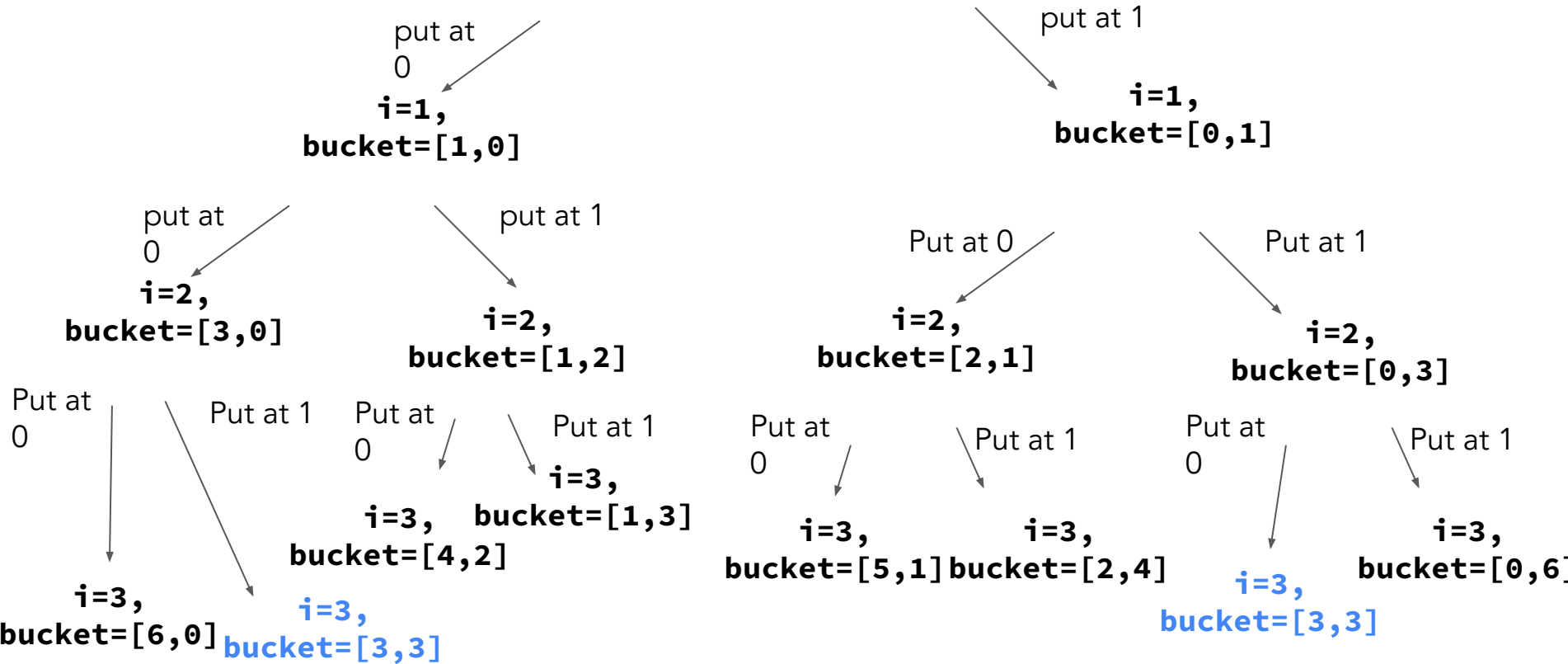
Put at 1

**i=2,
bucket=[0,3]**

What are the
different calls at this
level?

cookie= [1,2,3] | bucket = []

i=0,
bucket=[0,0]



Time Complexity

- Depth of the recursion tree
- Number of branches

$$\begin{aligned}\text{Time complexity} &= \text{Branches}^{(\text{Depth})} \\ &= O(K^n)\end{aligned}$$

Space Complexity

- Size of call stack
- The bucket list

Note: The list is passed by reference

$$\text{Space Complexity} = O(n + k) = O(n)$$

Some of the paths will eventually lead to a dead end or it is not optimal anymore

Shouldn't we just stop searching that path?

Pruning the execution tree

- What if we added more bases cases?
- Cases to prune
 - When **path is not optimal anymore** (check previous problem for instance)
 - Path **eventually leads to dead end**
- Stating facts before writing might help in discovering pruning cases

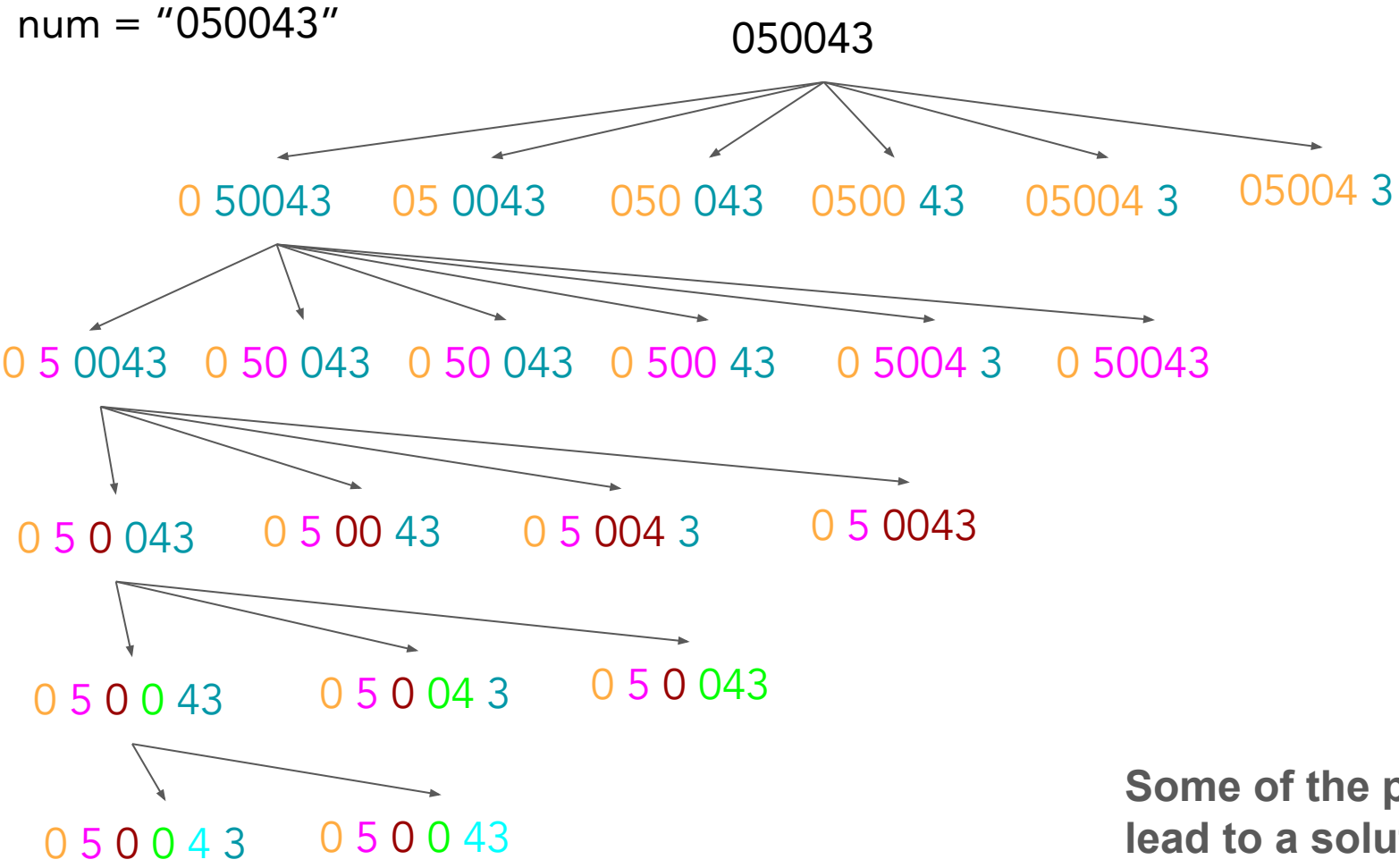
Optimize the previous solutions using pruning

Combinations

Optimize the previous solutions using pruning

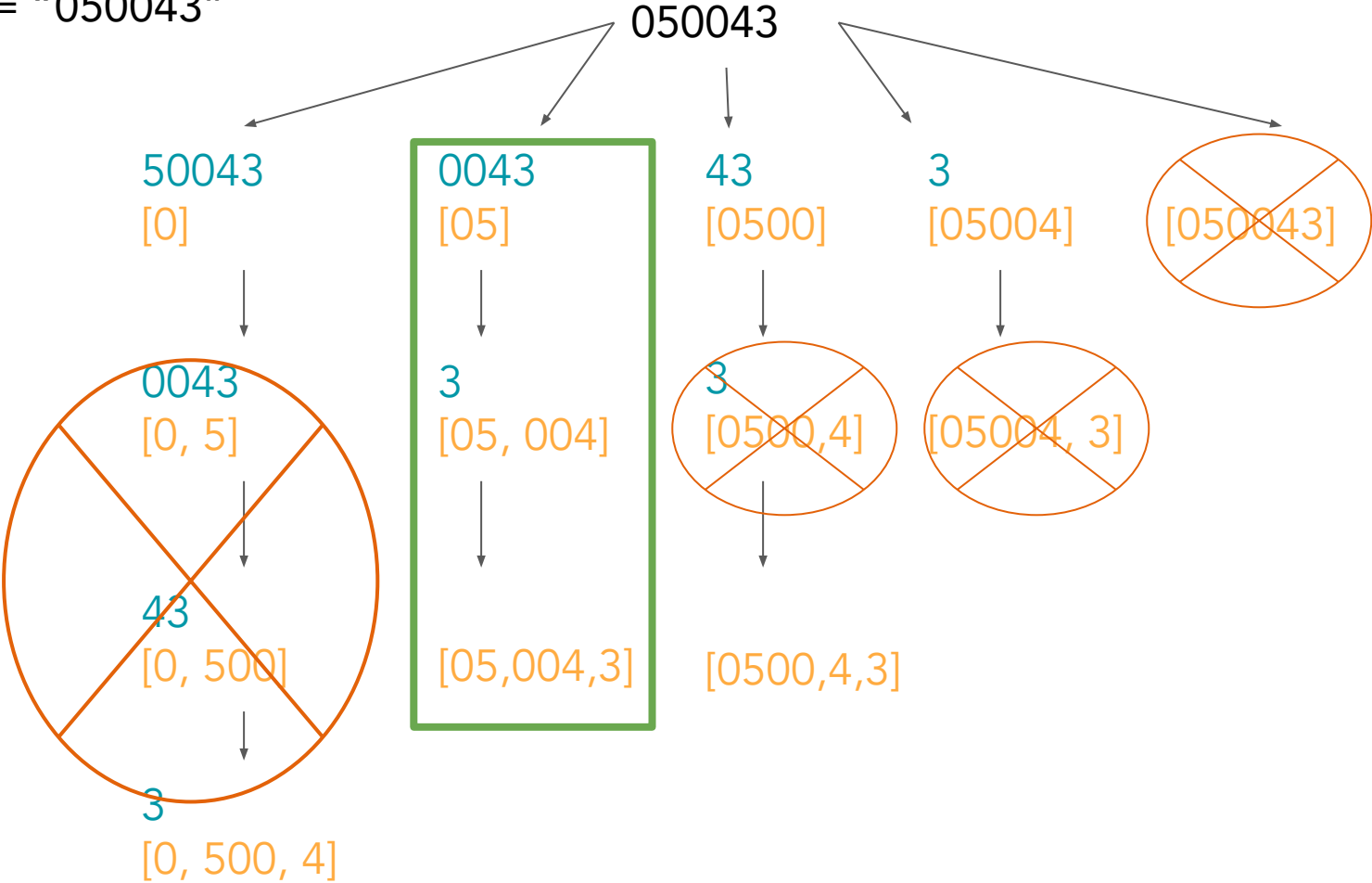
Splitting a String into Descending Consecutive values

num = "050043"



Some of the paths never lead to a solution

num = "050043"



Optimize the previous solutions using pruning

Fair Distribution of Cookies

Pair Programming

Subsets

Is that all recursion can do for us?

Divide and Conquer

Let's start with a problem

Convert Sorted Array to Binary
Search Tree

Divide and Conquer is a problem-solving technique where a large problem is broken down into smaller sub-problems that are easier to solve independently

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

Combine the solutions to the subproblems into the solution for the original problem.

If you have noticed, Tree problems are usually divide and conquer problems

Can you mention one problem from your previous lecture?

Pair Programming


Longest Nice

Substring

Things to pay attention

Incorrectly updating state variables

```
def backtrack(s, partial, remaining):  
    if is_solution(partial):  
        return partial  
  
    for c in remaining:  
        # Error: modifying iterable while iterating  
        remaining.remove(c)  
        backtrack(s, partial+c, remaining)  
        remaining.append(c)
```



Cannot edit dictionary
while iterating

Shallow copying a list

```
def permutation():  
    if len(path) == len(nums):  
        answer.append(path) →  
        return  
  
    for num in nums:  
        if num in path:  
            continue  
        path.append(num)  
        permutation()  
        path.pop()
```

Since the path is passed by reference the final permutation list is going to be a list of the same path

Practice Questions

Backtracking

[Combinations](#)

[Permutation](#)

[Subsets](#)

[Subsets II](#)

[Combination Sum](#)

[Splitting a string into descending consecutive values](#)

[Additive Numbers](#)

[Sudoku Solver](#)

[N-Queens](#)

[N-Queens II](#)

Divide and Conquer

[Majority Element](#)

[Maximum Binary Tree](#)

[Sort List](#)

[Balance a binary search tree](#)