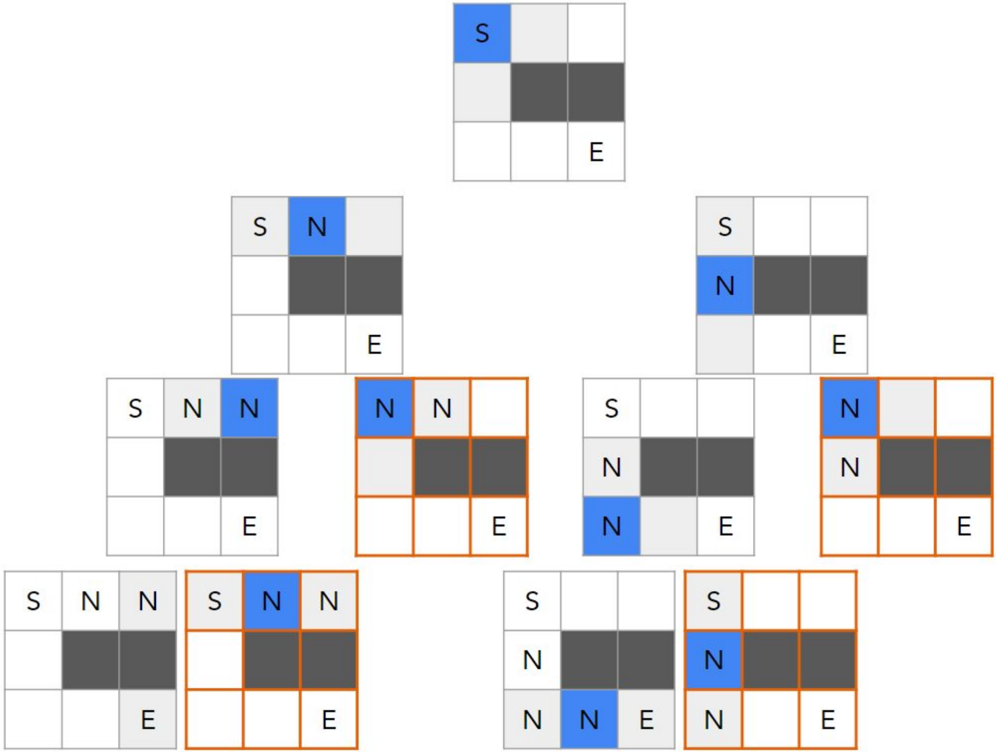
Recursion II



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

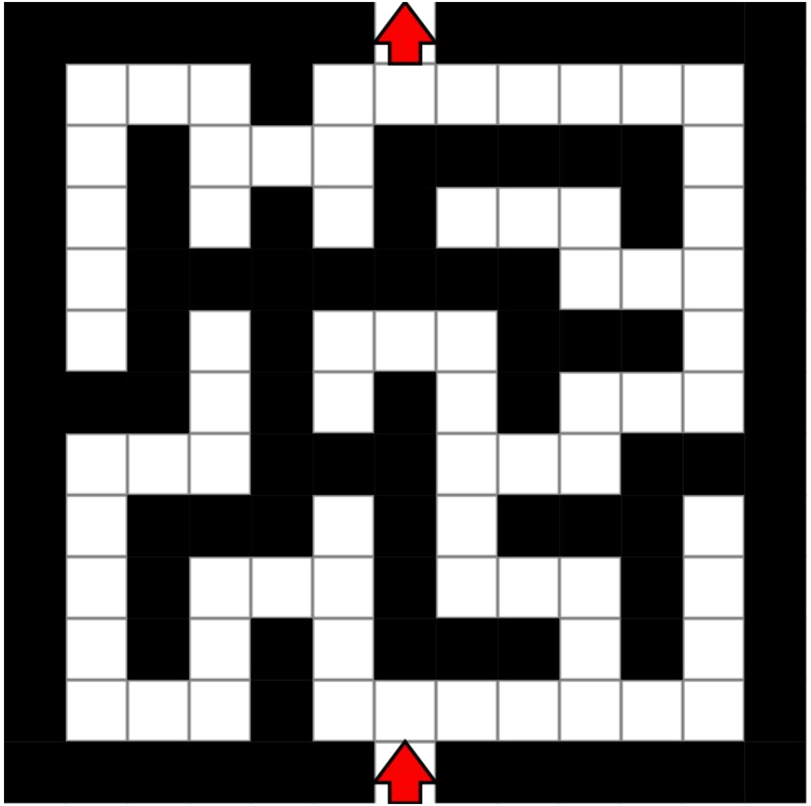
Link

Is that all recursion can do?

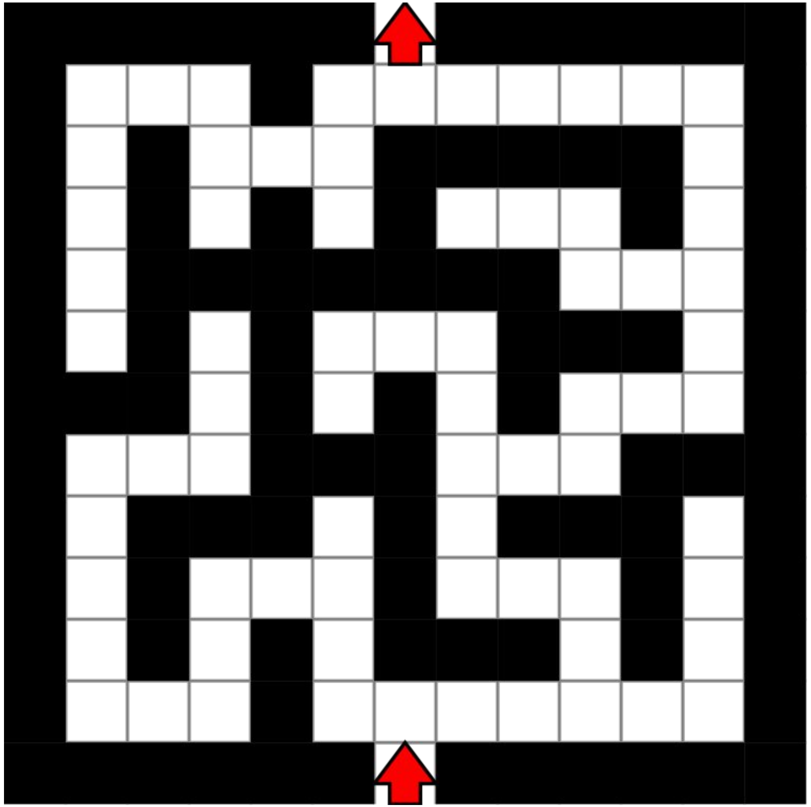
Today, we will add more areas where recursion shines

Backtracking

How would you find your way out?



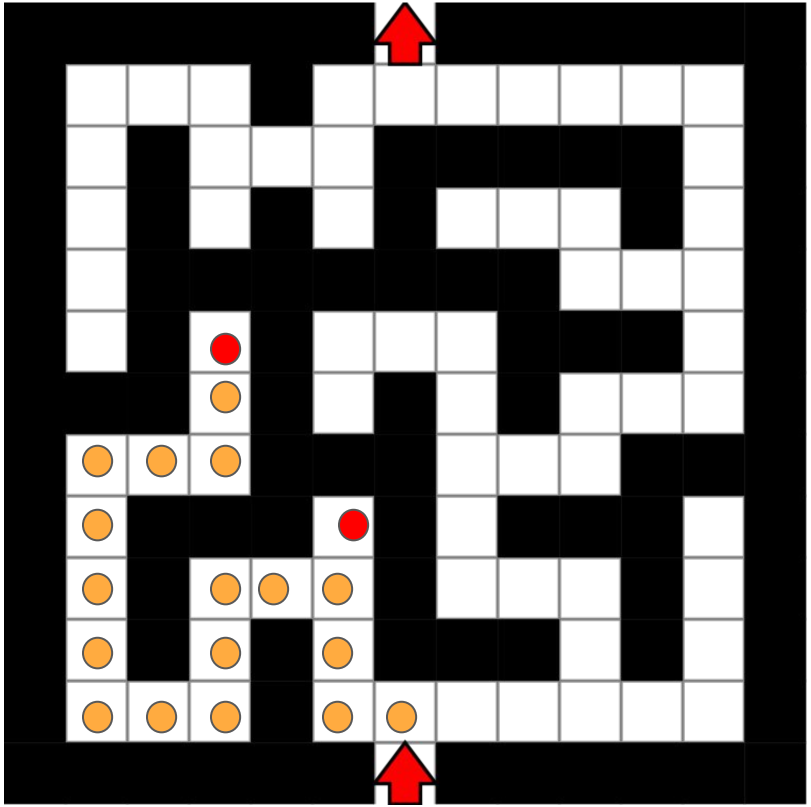
How would you find your way out?



- 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

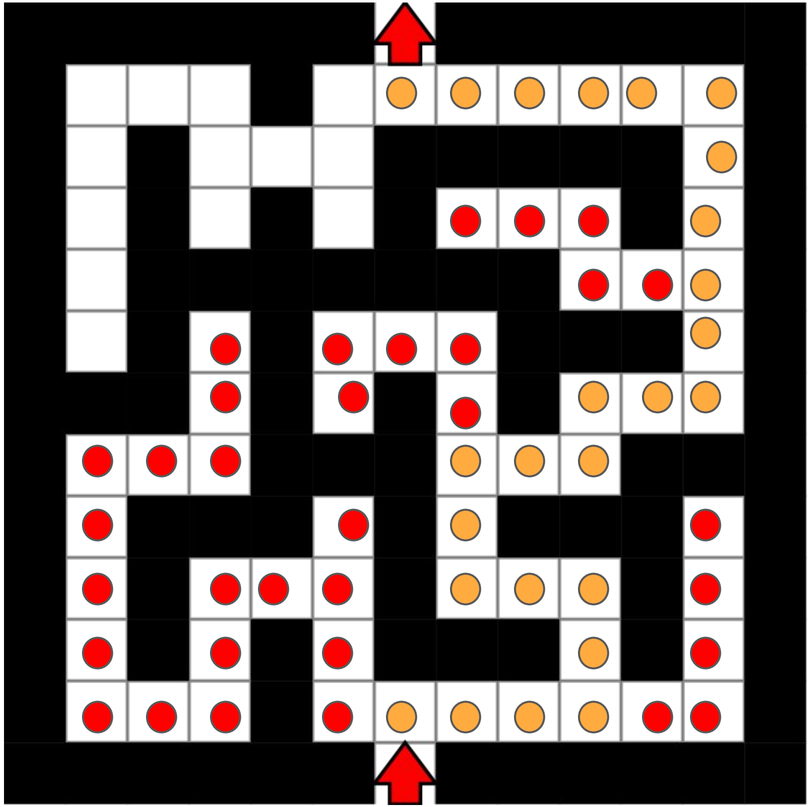
How would you find your way out?



- 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

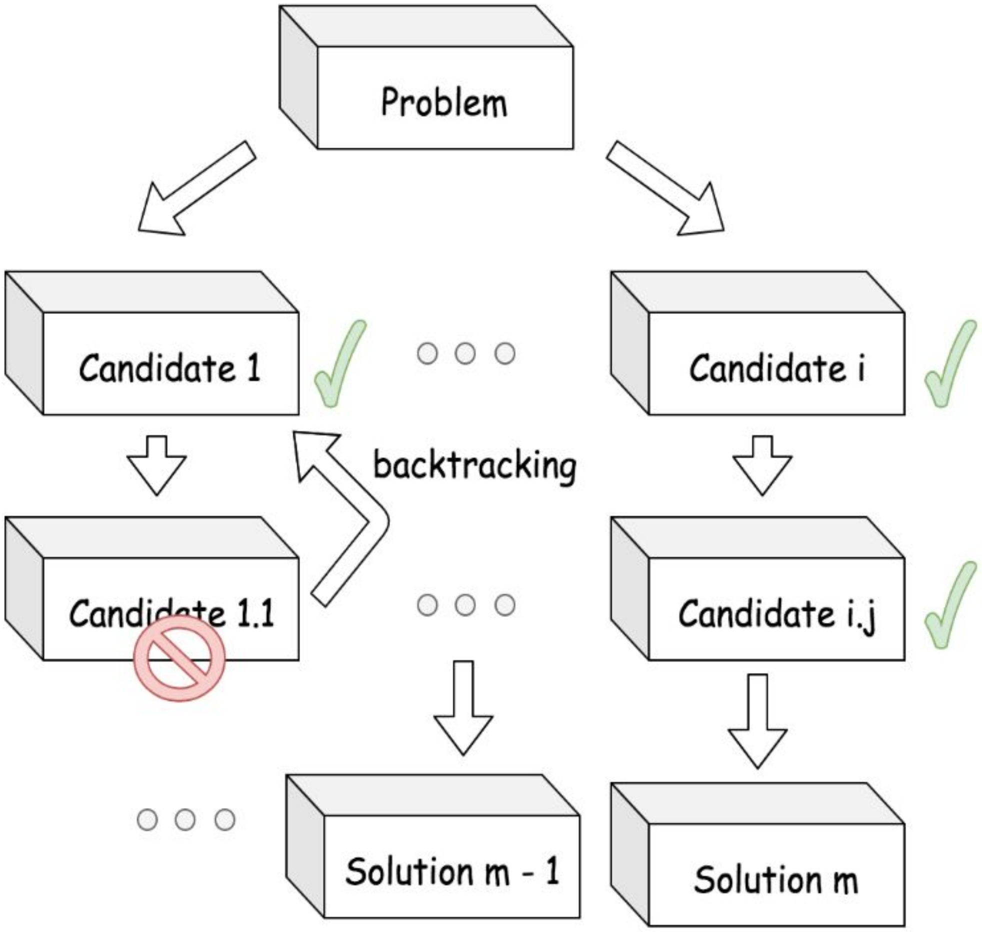
- Red peanut to mark we have been here
- Orange peanut currently in exploring path

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

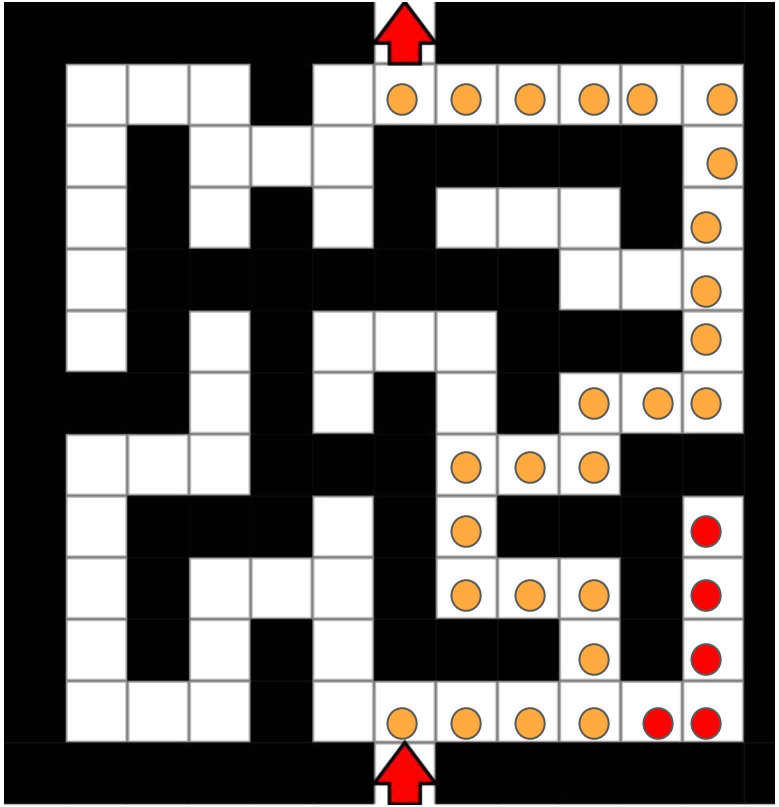
- Red peanut to mark we have been here
- Orange peanut currently in exploring path


Problem
Candidate 1
Candidate i
backtracking
Candidate 1.1
Candidate i.j
Solution m - 1
Solution m

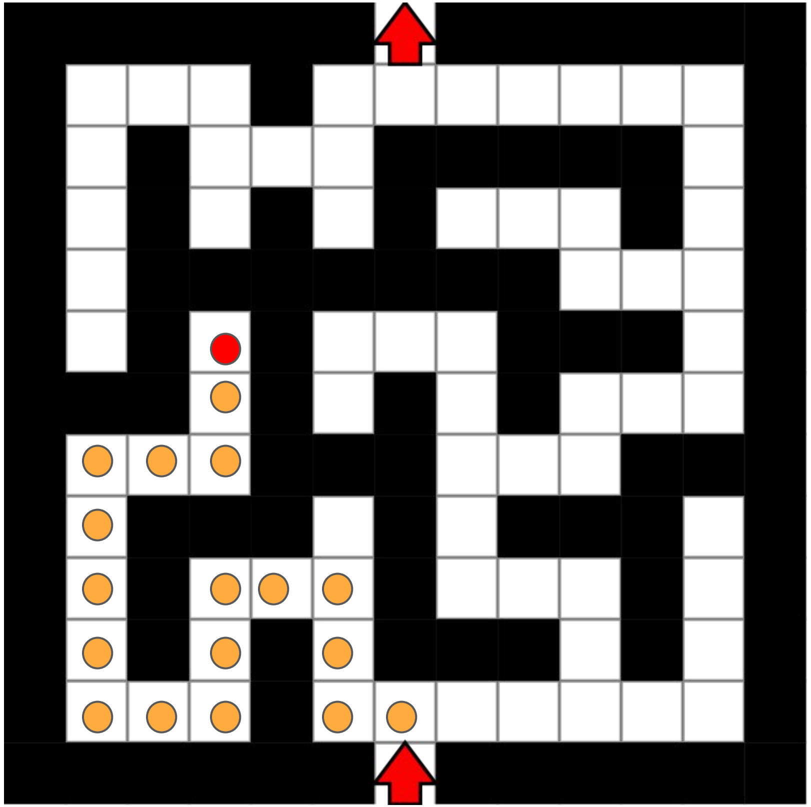
For our maze our candidates at each cell were

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

How would we simulate this with code?



Template

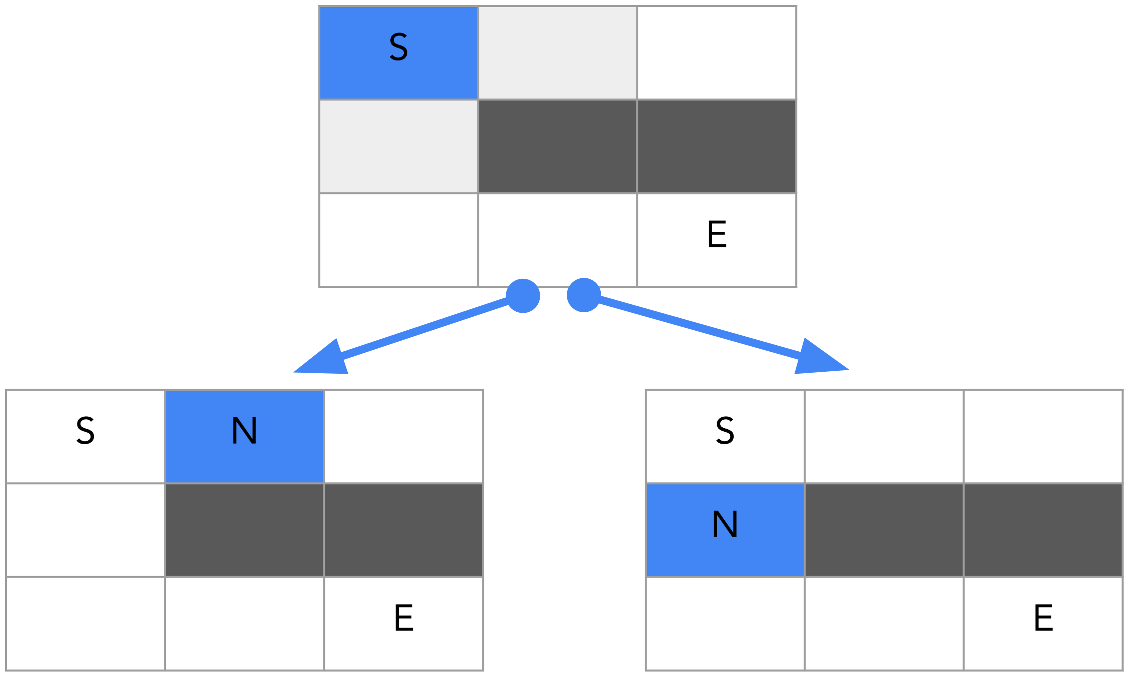


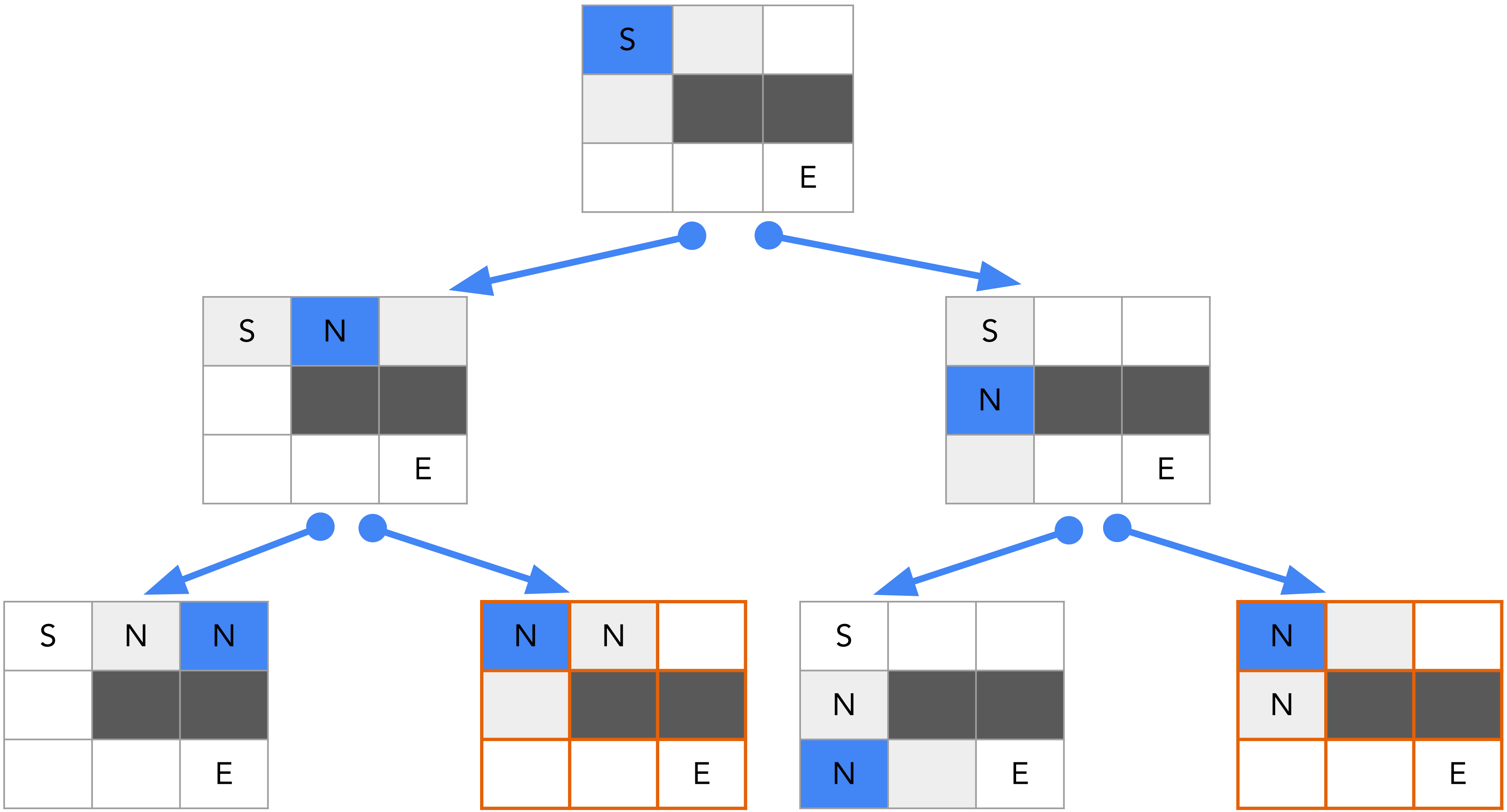
```
def backtrack(candidate):
    if  candidate is a solution:
        # process or output a candidate
        return

      # 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)
```

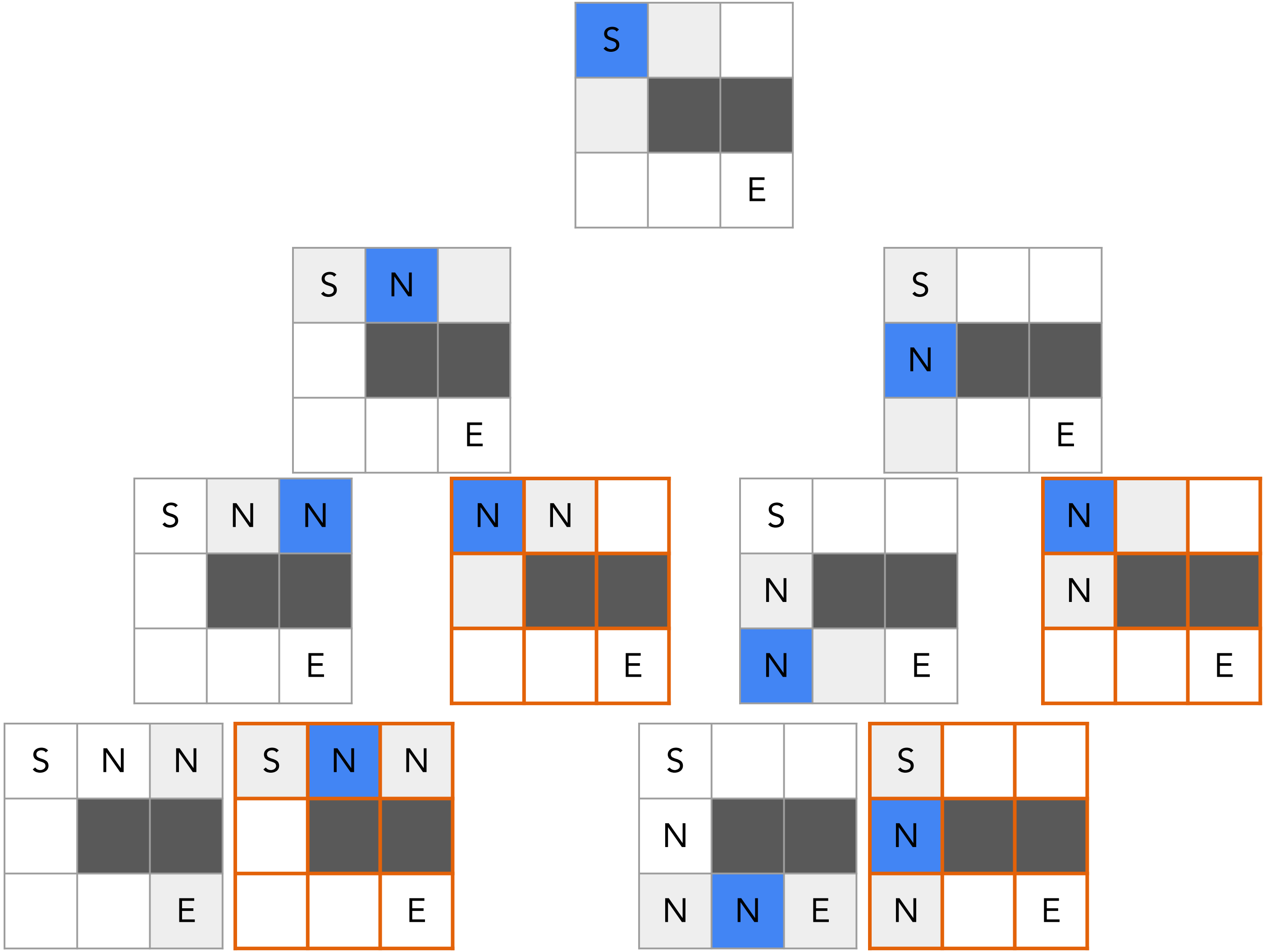
What are the next candidates for this state?

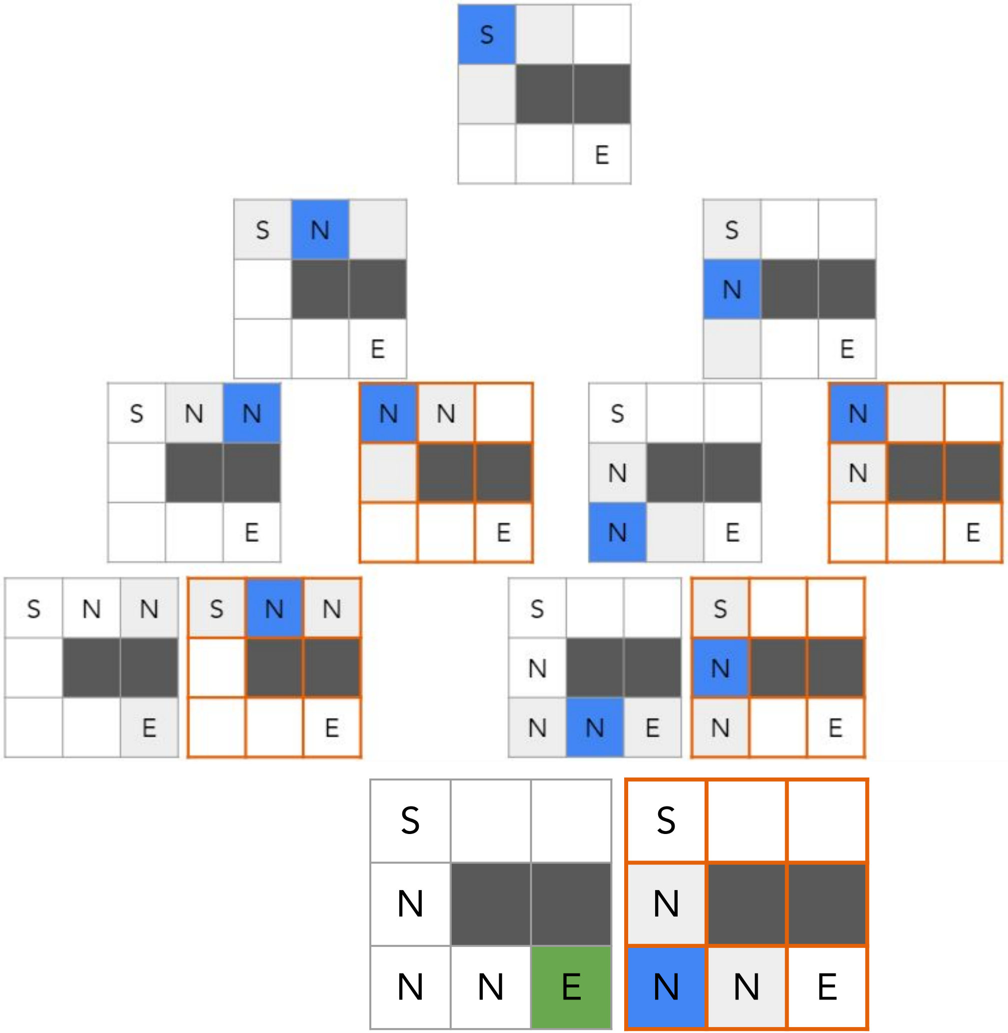
Legend

S - Start
N - Next
E - End


S
E
S
N
E
S
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
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 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 choose 2 = 6 total combinations.
Note that combinations are unordered, i.e., [1,2] and [2,1]
are considered to be the same combination.
```

Implementation

```
def combine(self, n: int, k: int) -> List[List[int]]:
    def backtrack(first_num, path):
        if len(path) == k:
            ans.append(path[:])
            return

        for num in range(first_num, n + 1):
            path.append(num)
            backtrack(num + 1, path)
            path.pop()

    ans = []
    backtrack(1, [])
    return ans
```

Another Implementation

```
def combine(self, n: int, k: int) -> List[List[int]]:
    nums = [num for num in range(1, n+1)]
    combinations = []

    def backtrack(i, combination):
         if len(combination) == k:
            combinations.append(combination[:])# copying
            return
        if i >= n:
             return
        #insert
        combination.append(nums[i])
        backtrack(i+1, combination)
        combination.pop()

         #no insert
        backtrack(i+1, combination)

    backtrack(0, [])
    return combinations
```

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]]

i=0,
com=[]

insert

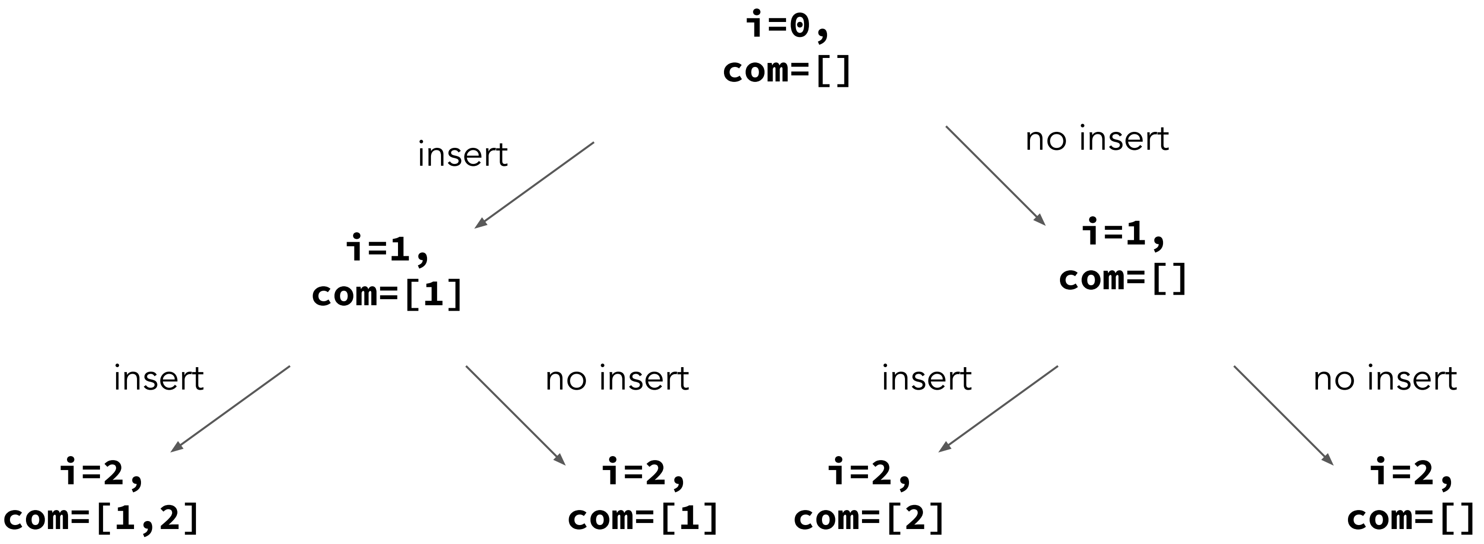
no insert

i=1,
com=[1]

i=1,
com=[]

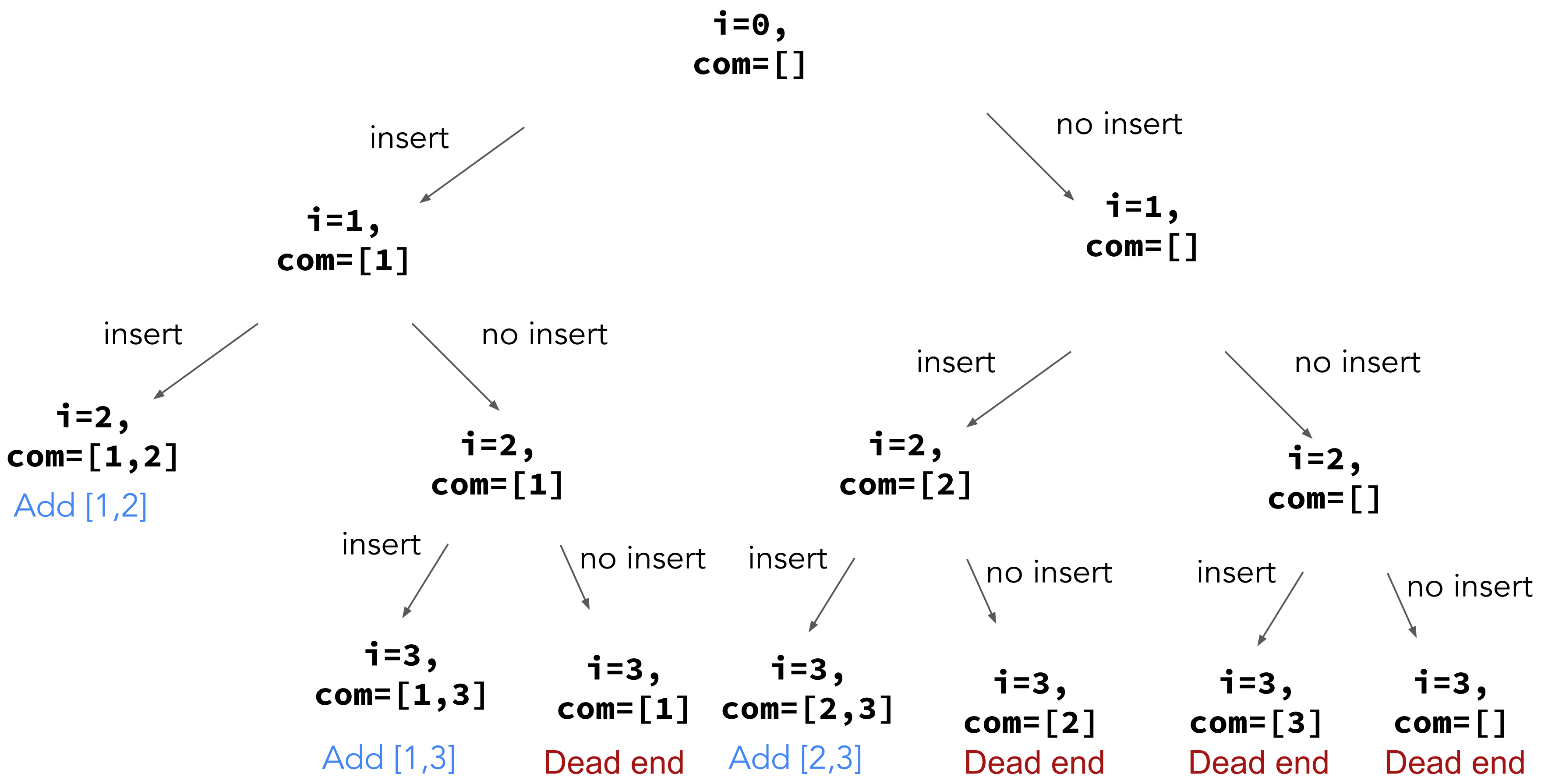
What are the different calls at this level?

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]]



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 ^ depth

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

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

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.

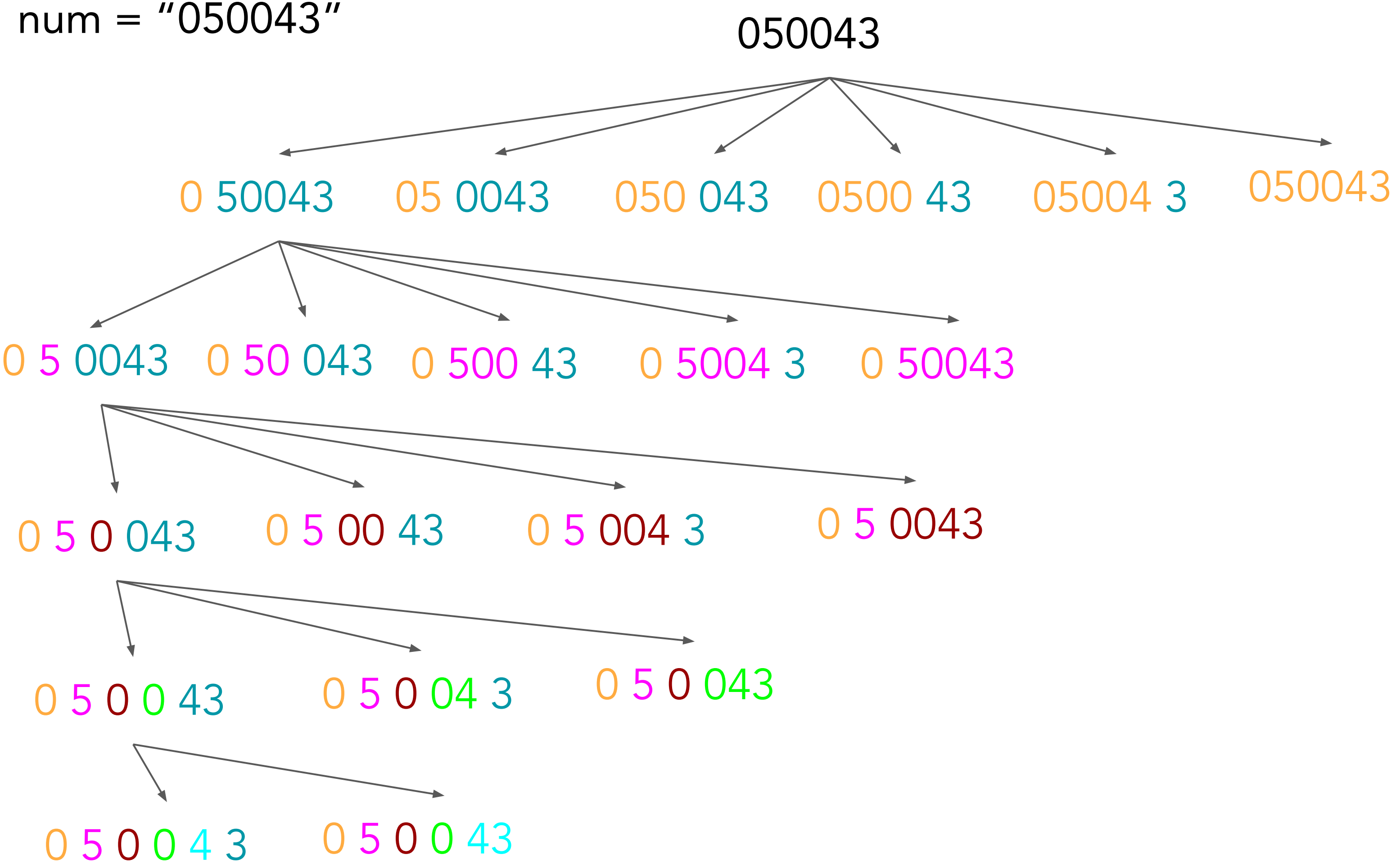
Return `true` *if it is possible to split* `s` *as described above, or* `false` *otherwise.*

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
0 50043
05 0043
050 043
0500 43
05004 3
050043
0 5 0043
0 50 043
0 500 43
0 5004 3
0 50043
0 5 0 043
0 5 00 43
0 5 004 3
0 5 0043
0 5 0 0 43
0 5 0 04 3
0 5 0 043
0 5 0 0 4 3
0 5 0 0 43

Implementation

```
def splitString(self, s: str) -> bool:
    current = []

    def backtrack(idx):
        if idx >= len(s):
            for i in range(1, len(current)):
                if current[i - 1] - current[i] != 1:
                    return False
            return len(current) >= 2

        for i in range(idx, len(s)):
            val = int(s[idx:i+1])
            current.append(val)
            if backtrack(i + 1):
                return True
            current.pop()

        return False
    return backtrack(0)
```

Time Complexity

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

Time Complexity = O (n * (2 ^ n))

Space Complexity

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

Note: The array is global.

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

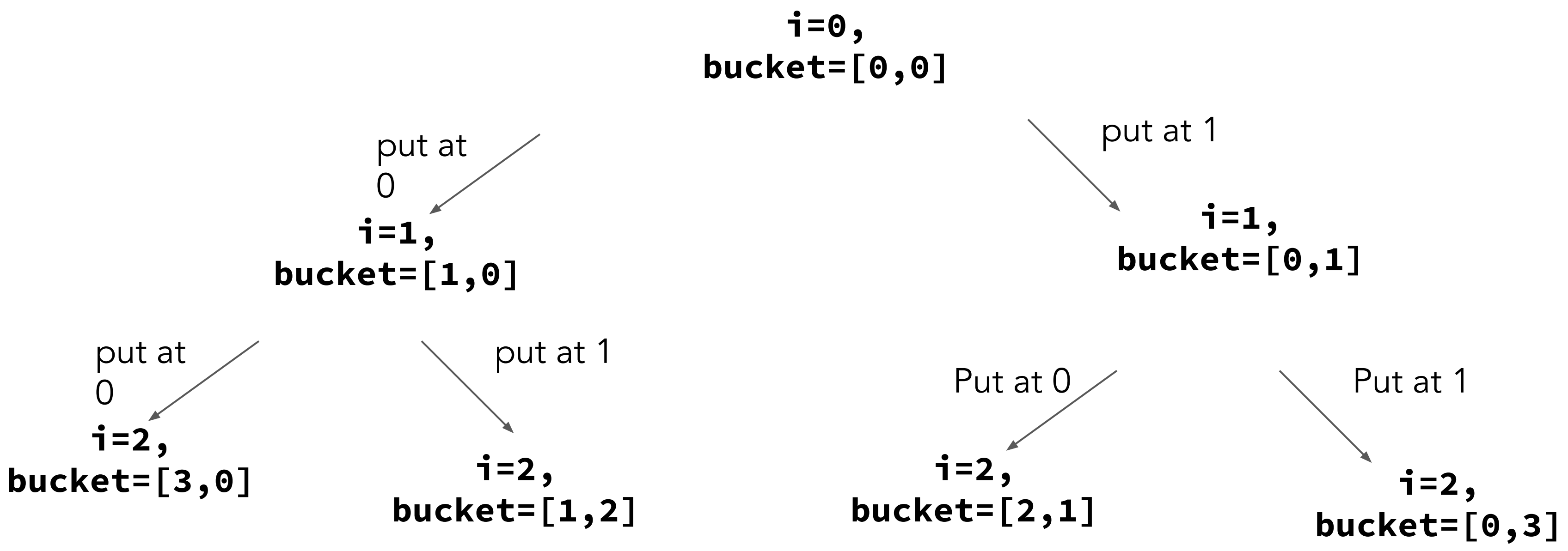
i=1,
bucket=[1,0]

put at 1

i=1,
bucket=[0,1]

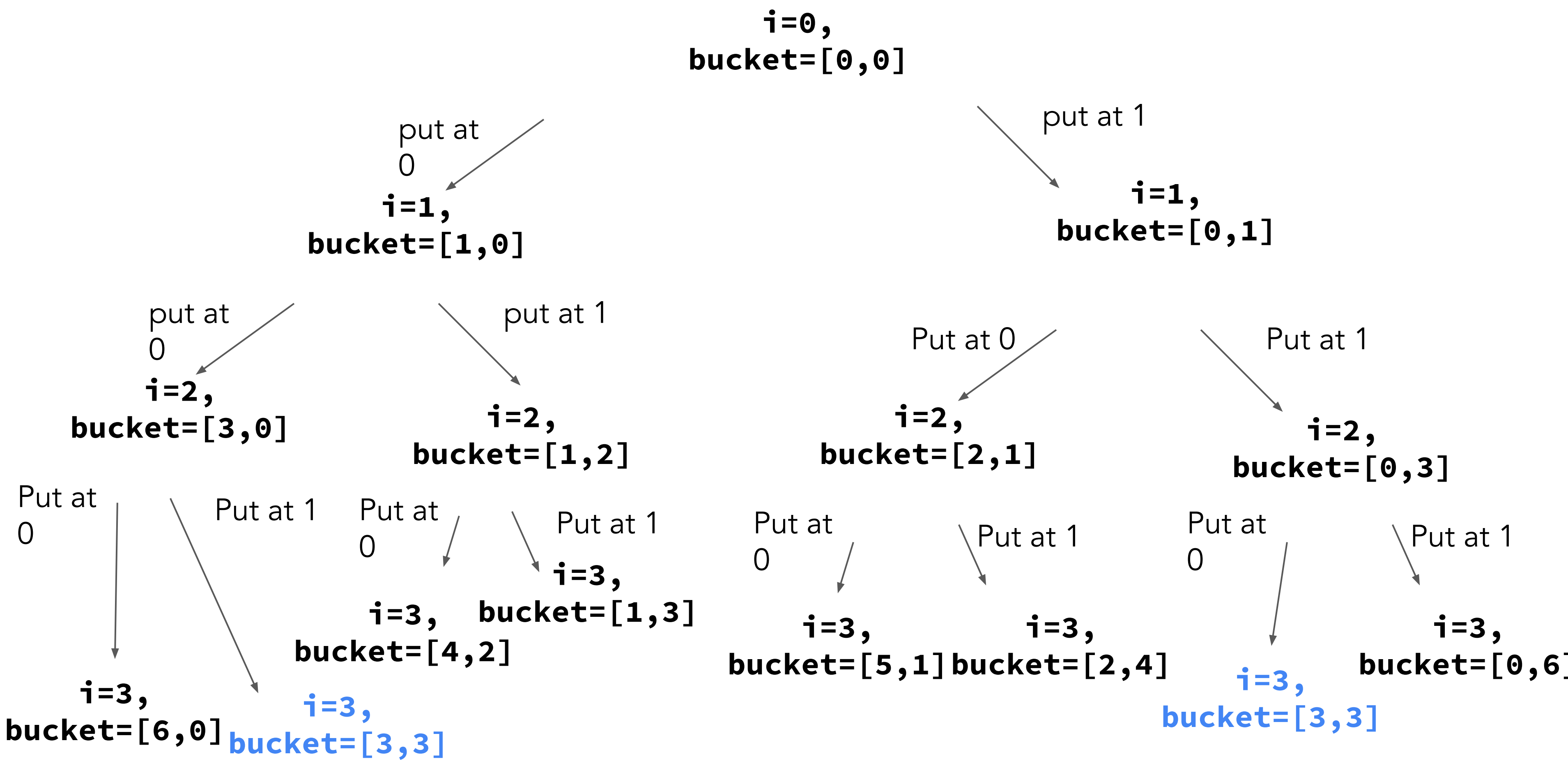
What are the different calls at this level?

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



What are the different calls at this level?

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



Implementation

```python
def distributeCookies(self, cookies: List[int], k: int) -> int:
    bucket = [0] * k
    self.minUnfairness = float('inf')

    def backtrack(i, bucket):
        if i >= len(cookies):
            self.minUnfairness = min(self.minUnfairness, max(bucket))
            return

        for j in range(k):
            bucket[j] += cookies[i]
            backtrack(i+1, bucket)
            bucket[j] -= cookies[i]

    backtrack(0, bucket)
    return self.minUnfairness
```

Time Complexity

- Depth of the recursion tree
- Number of branches

Time complexity = Branches$^{(\text{Depth})}$

$= O(K^n)$

Space Complexity

- Size of call stack
- The bucket list

Note: The list is passed by reference

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

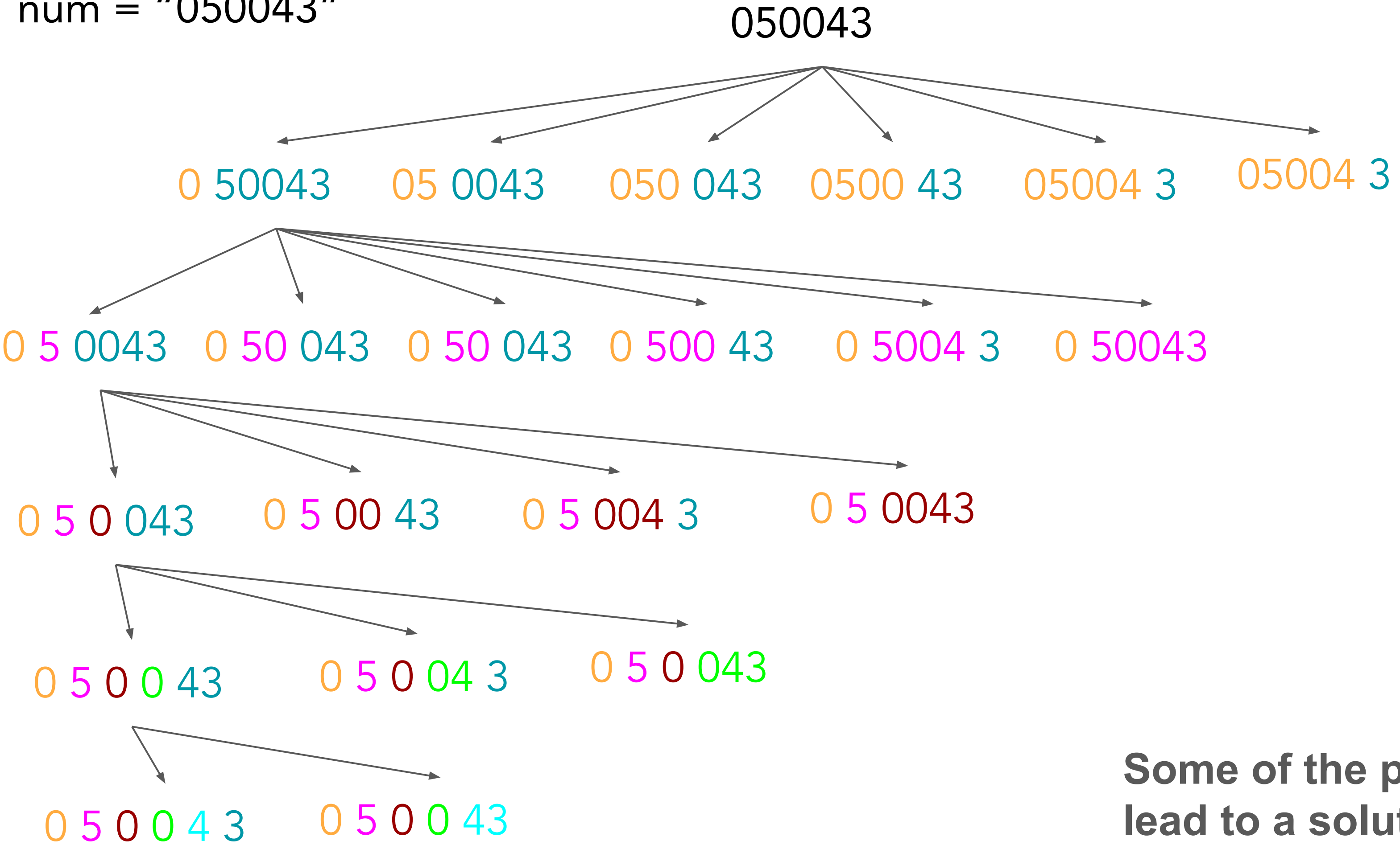
Combinations

Implementation

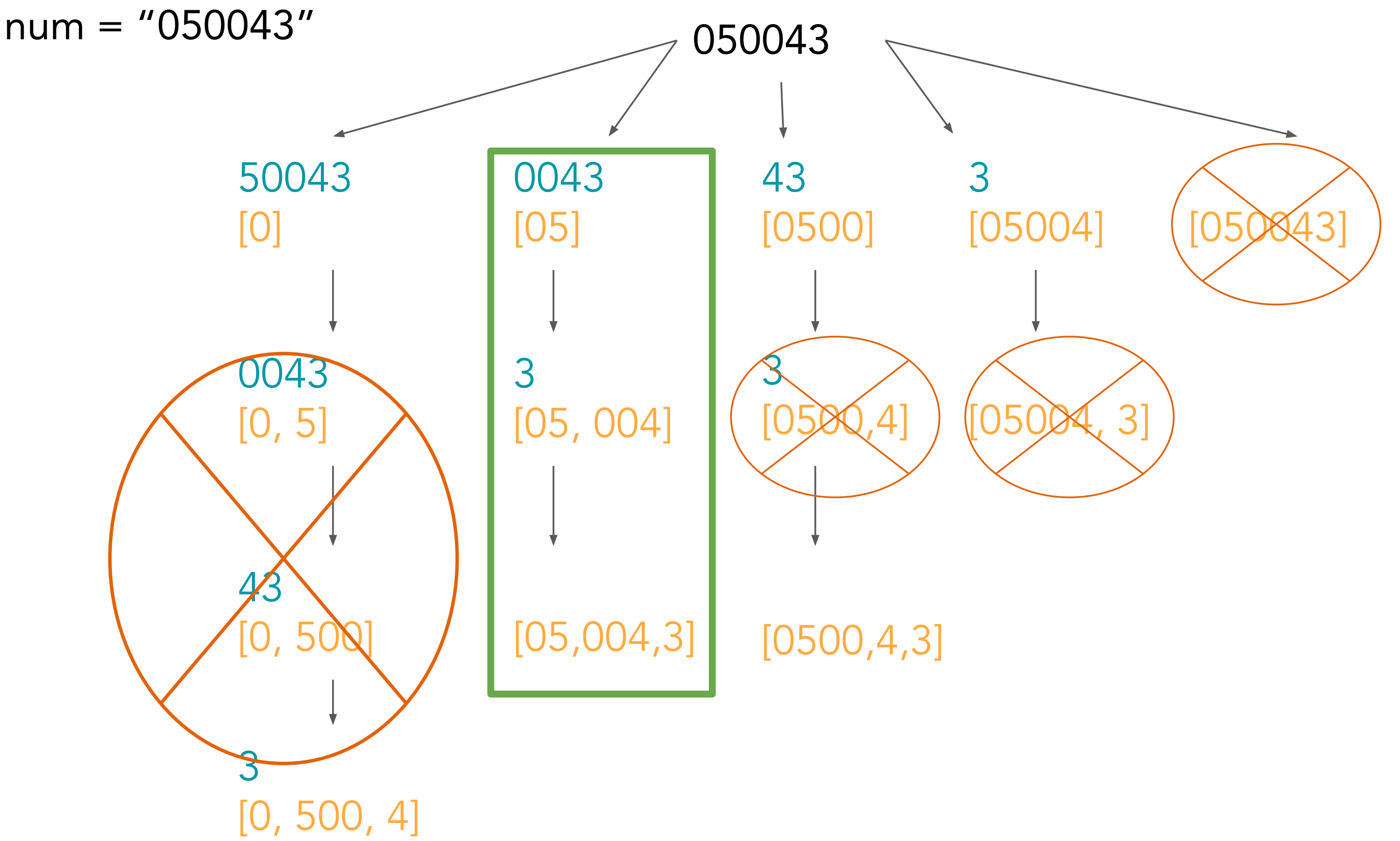
```python
def combine(self, n: int, k: int) -> List[List[int]]:
    def backtrack(curr, first_num):
        if len(curr) == k:
            ans.append(curr[:])
            return

        need = k - len(curr)
        remain = n - first_num + 1
        available = remain - need

        for num in range(first_num, first_num + available + 1):
            curr.append(num)
            backtrack(curr, num + 1)
            curr.pop()

        return

    ans = []
    backtrack([], 1)
    return ans
```

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"
050043
50043
[0]
0043
[0, 5]
43
[0, 500]
3
[0, 500, 4]
0043
[05]
3
[05, 004]
[05,004,3]
43
[0500]
3
[0500,4]
[0500,4,3]
3
[05004]
[05004, 3]
[050043]

Implementation

```
def splitString(self, s: str) -> bool:
    current = []

    def backtrack(idx):
        if idx >= len(s):
            return len(current) >= 2

        for i in range(idx, len(s)):
            val = int(s[idx:i+1])
            if len(current) == 0 or val == current[-1] - 1:
                current.append(val)
                if backtrack(i + 1):
                    return True
                current.pop()

        return False
    return backtrack(0)
```

Optimize the previous solutions using pruning

[Fair Distribution of Cookies]

Implementation

```python
def distributeCookies(self, cookies: List[int], k: int) -> int:
    bucket = [0]*k
    self.minUnfairness = float('inf')

    def backtrack(i,bucket):
        if i >= len(cookies):
            self.minUnfairness = min(self.minUnfairness, max(bucket))
            return
        if max(bucket) > self.minUnfairness: # pruning case
            return

        for j in range(k):
            bucket[j] += cookies[i]
            backtrack(i+1, bucket)
            bucket[j] -= cookies[i]

    backtrack(0, bucket)
    return self.minUnfairness
```

Pair Programming

[Subsets]()

Implementation

```
def subsets(self, nums: List[int]) -> List[List[int]]:
    def backtrack(i, path, length):
        if len(path) == length:
            ans.append(path[:])
            return

        for j in range(i, N):
            path.append(nums[j])
            backtrack(j + 1, path, length)
            path.pop()

    ans = []
    N = len(nums)
    for l in range(N + 1):
        backtrack(0, [], l)

    return ans
```

Is that all recursion can do for us?

Divide and Conquer

Let’s start with a problem

[Convert Sorted Array to Binary Search Tree]

Implementation

```python
def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:

    def helper(left, right):
        if left > right:
            return None

        mid = (left + right) // 2
        left = helper(left, mid - 1)
        right = helper(mid + 1, right)
        return TreeNode(nums[mid], left, right)

    return helper(0, len(nums) - 1)
```

Divide and Conquer is a problem-solving technique where a large problem is broken down into smaller subproblems 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]

Implementation

```
class Solution:
    def longestNiceSubstring(self, s: str) -> str:
        if len(s) < 2:
            return ""

        s_set = set(s)

        for i,c in enumerate(s):
            if c.swapcase() not in s_set:
                s1 = self.longestNiceSubstring(s[0:i])
                s2 = self.longestNiceSubstring(s[i+1:])

                return s2 if len(s2) > len(s1) else s1

        return s
```

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

Quote of the day

The bad news is time flies, the good news is you're the pilot

Michael Altshuler