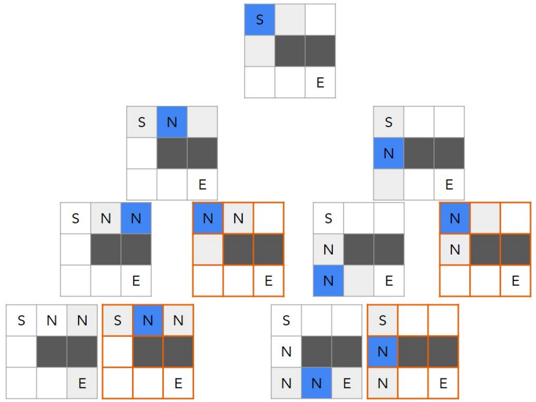
Fundamentals of Recursion II



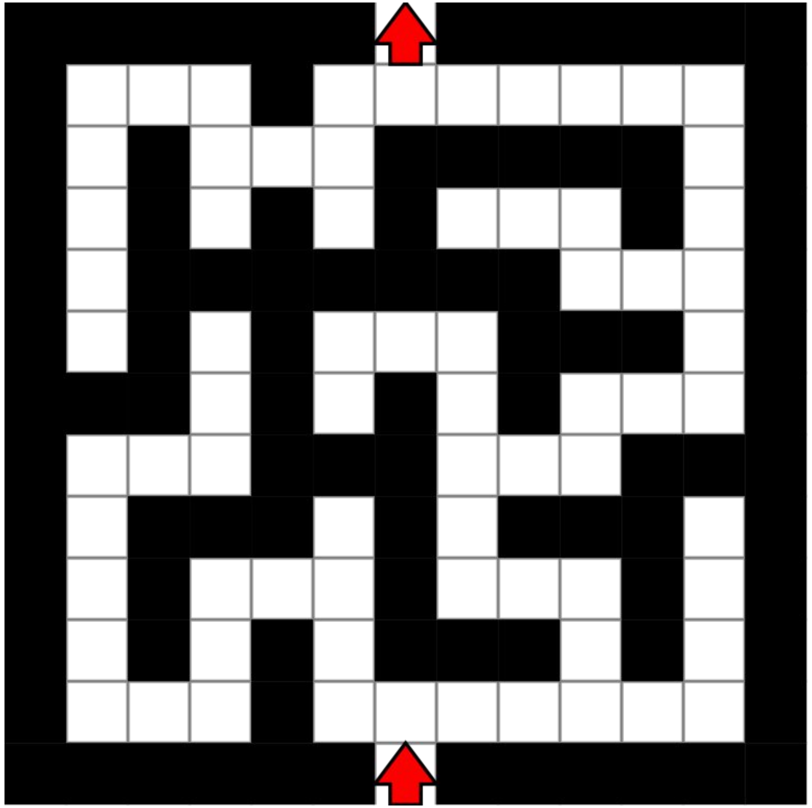
What is on today's Lecture?

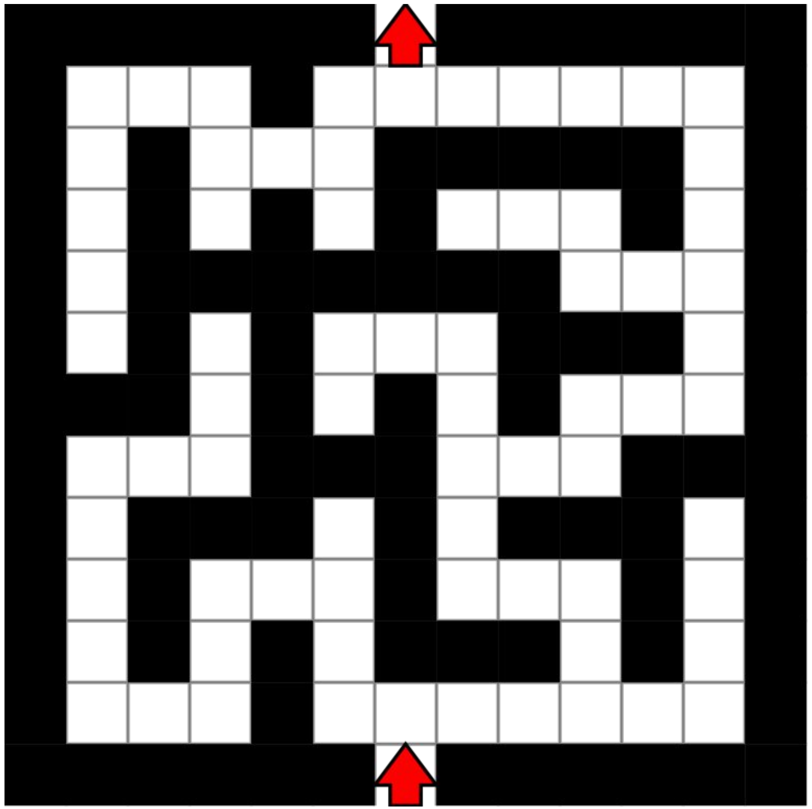
- We have seen the basics of recursion before.
- We have also used recursion for tree traversals.
- But the world of recursion is wider than you might have guessed.

- In this lecture, we will cover two main classes of algorithms that are strongly related to recursion.
    - ❏ Backtracking
    - ❏ Divide and conquer

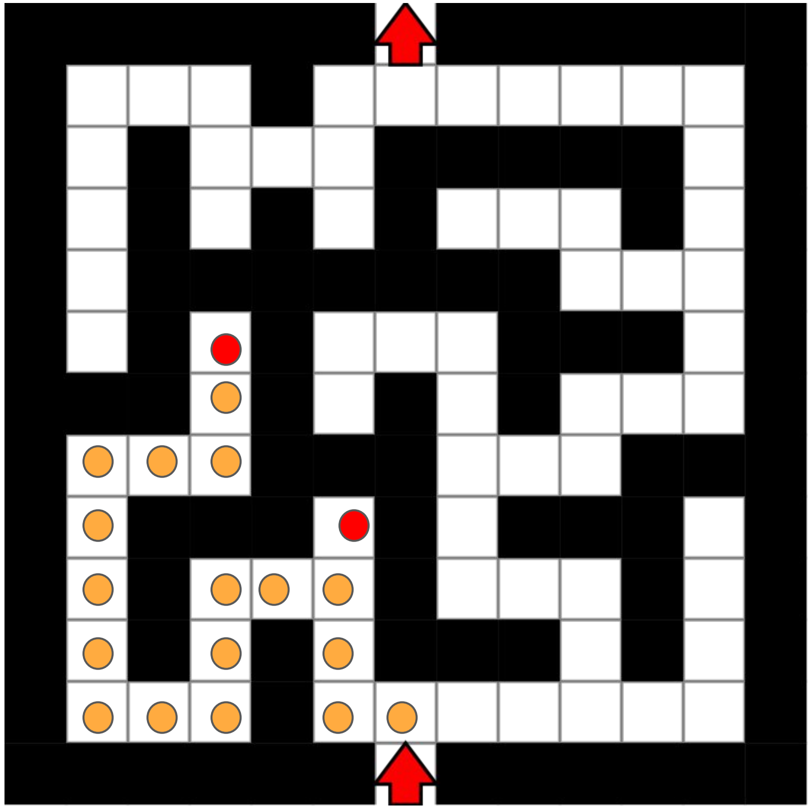
Backtracking

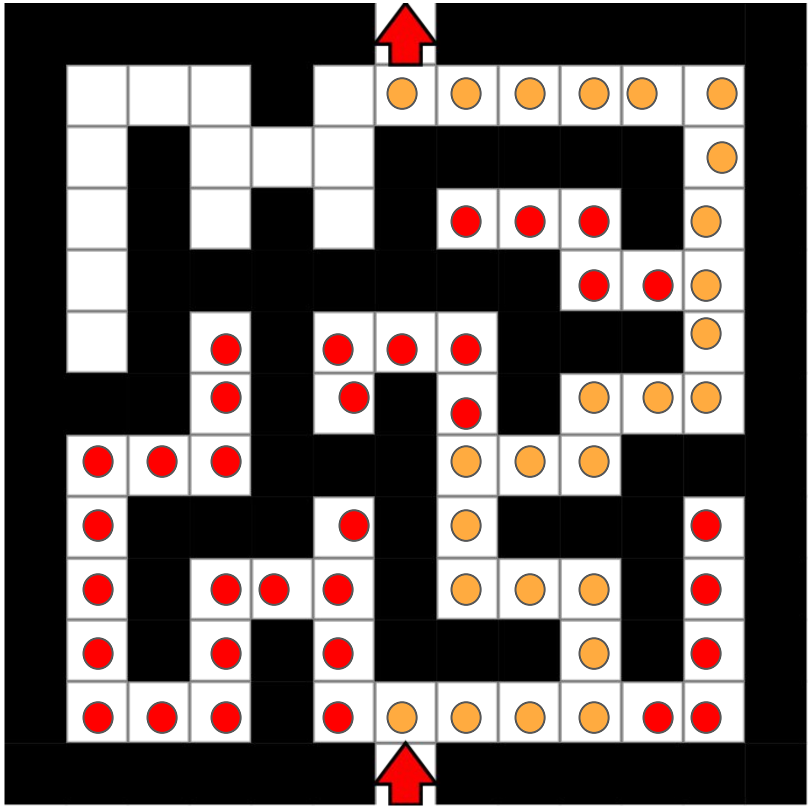
How would a robot find its way out??



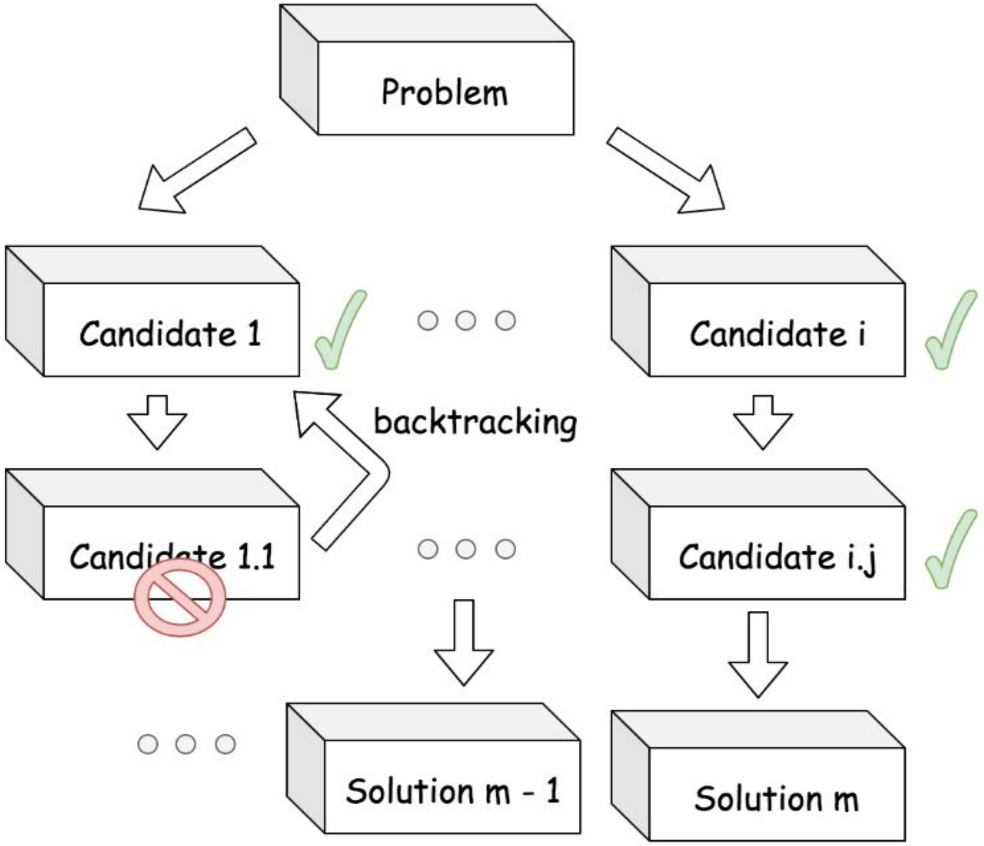


1. Try every available direction.
2. When you reach a dead end, go back to where you came from and
3. Explore a different path





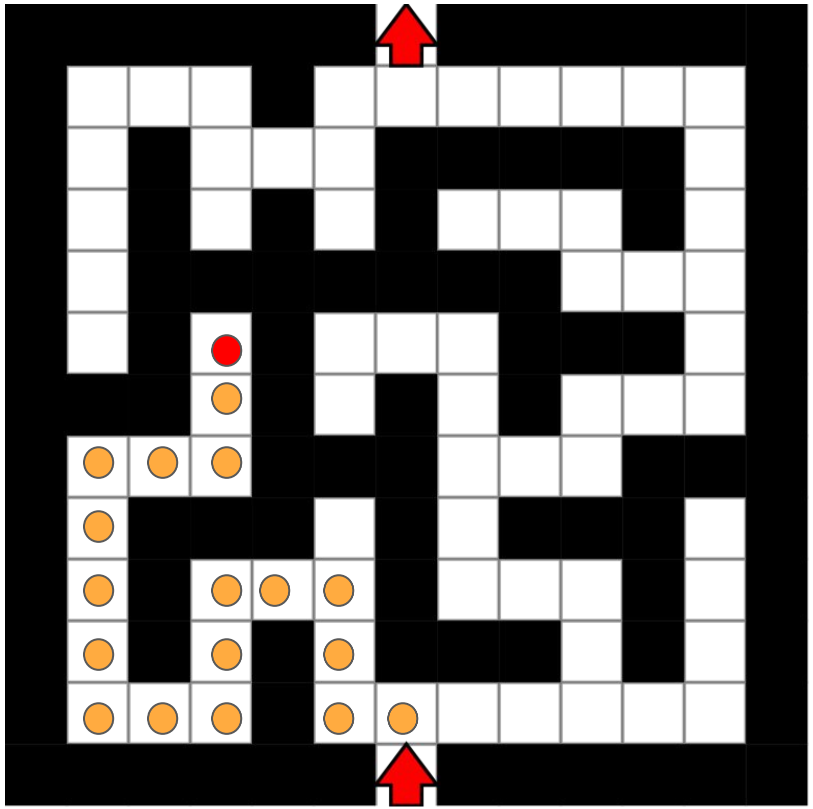
Generally speaking,



What were the candidates to the maze problem discussed earlier?

1. Take right Turn
2. Take left Turn
3. Go Up

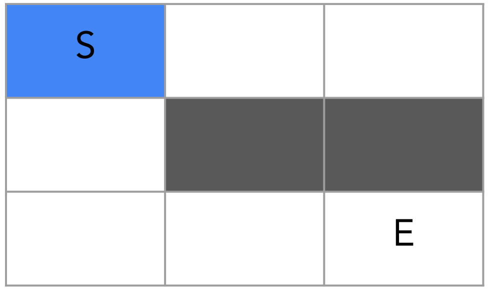
Now, let's discuss the code structure!



```
def find_path(candidate):
    # base case
    if  candidate is a solution:
        # process or output a candidate
        return
    # iterate all possible candidates.
    for next_candidate in list_of_candidates:
        if  next_candidate is valid:
            # try this partial candidate solution
             # for example: push it to a path list
            place(next_candidate) # mark
            # given the candidate, explore further.
            find_path(next_candidate)
            # backtrack
             # for example: pop it from path list
            remove(next_candidate) # could be unmark
```

Let's take a closer look at the solution.

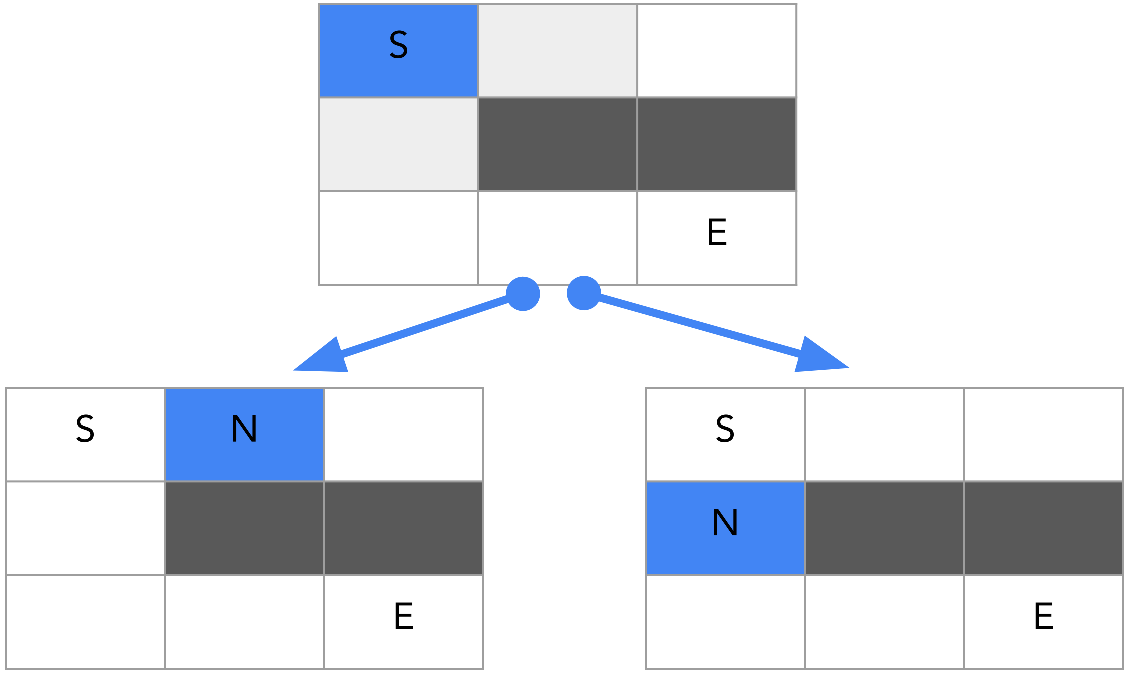
The robot wants to go from S to E.

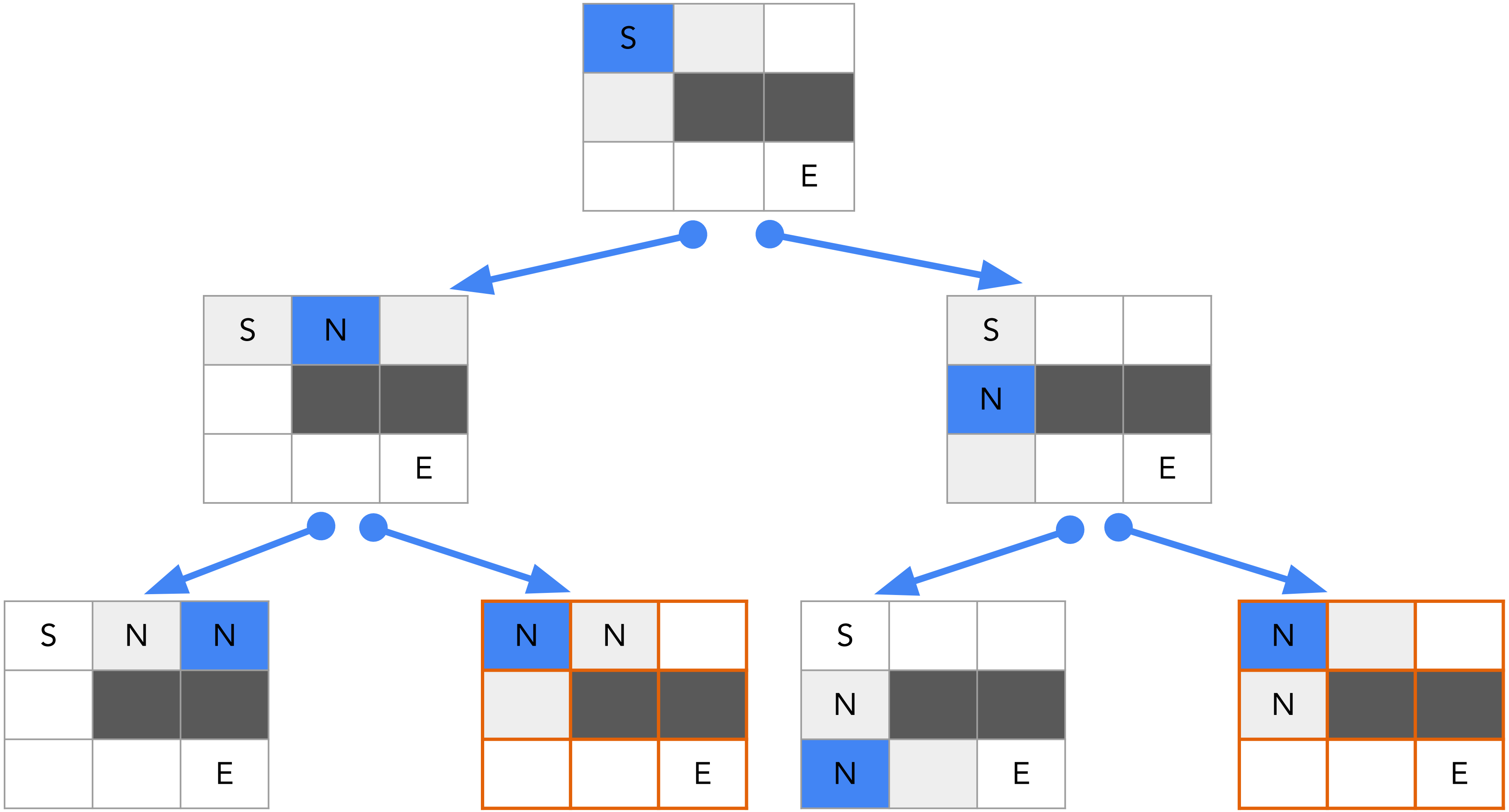


Legend

S - Start
N - Next
E - End

What are the next candidates?


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

- What are the states involved in the problem described earlier?
    - What changes as the robot moves from one spot to another?

- The answer is (i, j).
- If we think of the maze as a grid, we can use the row and column indices as the state.

What is the base case for the recursion?

- Two base cases for the recursion are: having no available next candidates and reaching the final destination.

In Summary

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.

Exercise Time

Let's try to understand the following problem and then we will solve it together.

Binary Tree Paths

I intentionally left space to draw on using tabs.

Implementation

```python
class Solution:
    def find_paths(self, node, path):
        # leaf node, a base case
        path.append(node.val)

        if node.left == None and node.right == None:

            str_path = [str(val) for val in path]
            result = "->".join(str_path)

            self.all_paths.append(result)

        if node.left != None:
            self.find_paths(node.left, path)
        if node.right != None:
            self.find_paths(node.right, path)

        path.pop()

    def binaryTreePaths(self, root: Optional[TreeNode]) -> List[str]:
        self.all_paths = []
        path = []
        self.find_paths(root, path)

        return self.all_paths
```

Time Complexity

- As with any recursive algorithm, one part of our time complexity is branches raised to the power of depth.
- In addition to that, we are also concatenating strings and performing other linear time operations.
- If the tree was a perfect tree.
- Branch ⇒ 2

  Depth ⇒ Log(N)

  2^Log(N) ⇒ N

- In total O(N^2)

space complexity

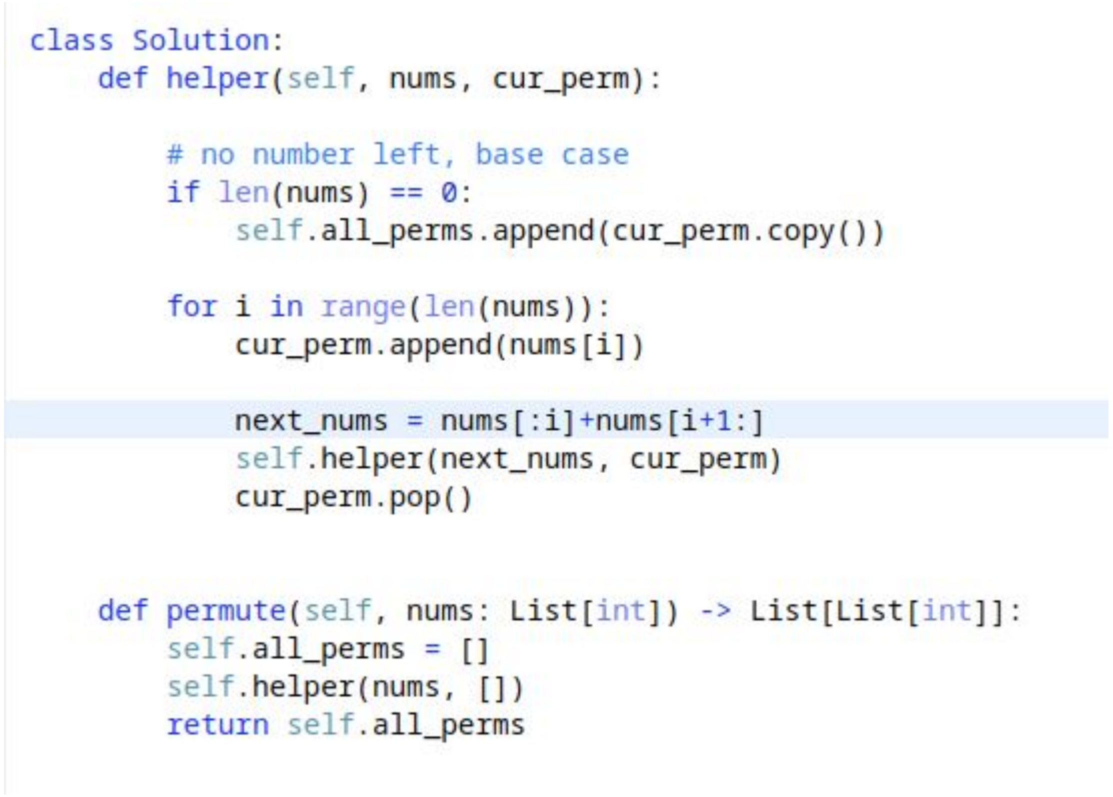
- Like with any recursive algorithm, there is hidden space used in the form of a recursion stack. This stack can be at most the depth of the recursion.
- O(d) and d is log(N) for full tree and N in skewed tree.

Here's another problem. Let's try to understand and then we will solve it together

Permutations

I intentionally left space to draw on using tabs.

Implementation



```
class Solution:
    def helper(self, nums, cur_perm):

        # no number left, base case
        if len(nums) == 0:
            self.all_perms.append(cur_perm.copy())

        for i in range(len(nums)):
            cur_perm.append(nums[i])

            next_nums = nums[:i]+nums[i+1:]
            self.helper(next_nums, cur_perm)
            cur_perm.pop()


    def permute(self, nums: List[int]) -> List[List[int]]:
        self.all_perms = []
        self.helper(nums, [])
        return self.all_perms
```

Time Complexity

- We will perform a rough analysis because the exact analysis is complex.
- Similar to other recursive algorithms, the time complexity for this recursion can be represented as branches raised to the power of depth.
- Considering that branches are typically less than N (approximately), and the depth is also N, the time complexity can be roughly approximated as O(N^N).
- The space complexity of this algorithm is dominated by the recursion stack if we don not include the output. Since the depth of recursion is N, the space complexity can be expressed as O(N).

Divide and Conquer

Divide and conquer is a problem-solving technique where a large problem is broken down into smaller subproblems that are easier to solve independently and then combine them.

Here are the steps

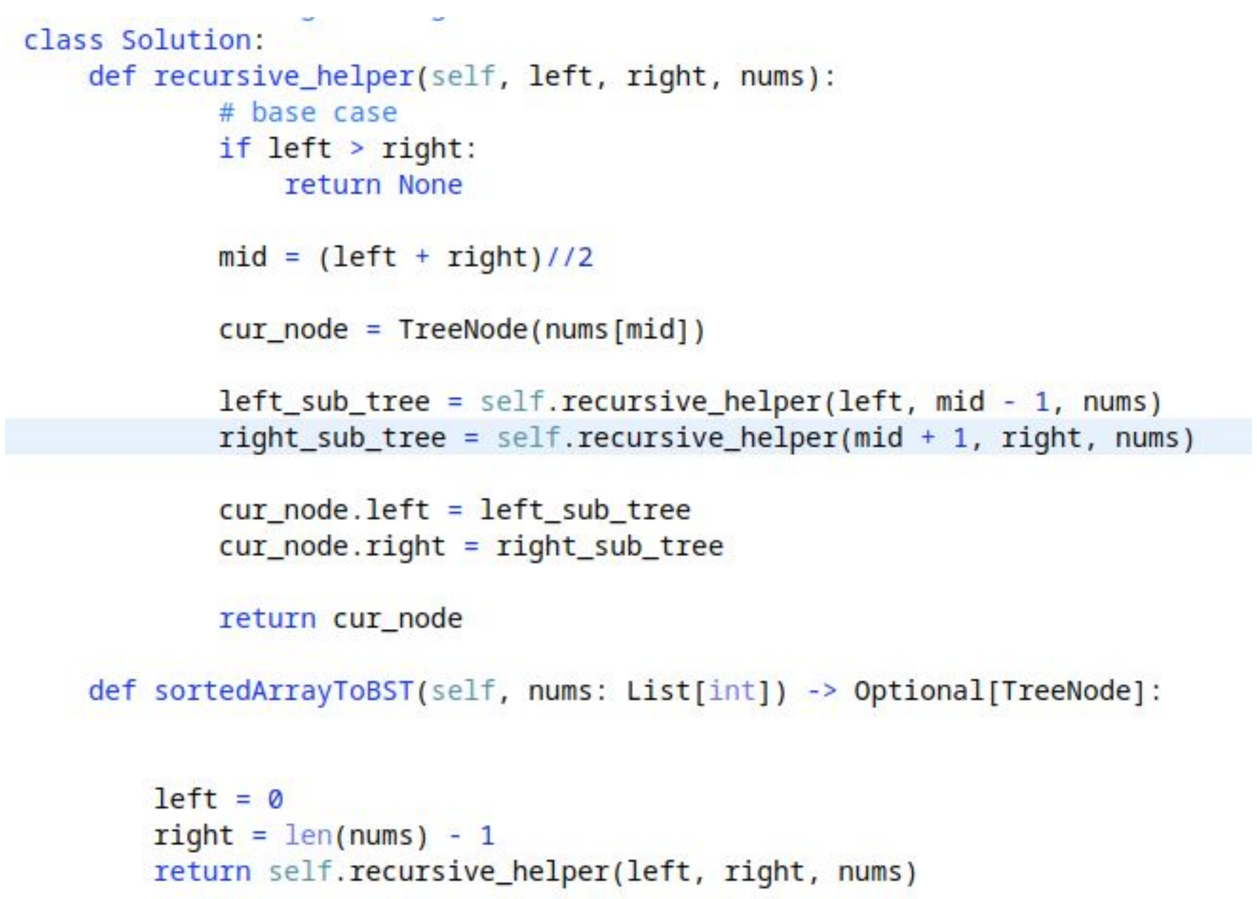
1. Divide the problem into a number of subproblems that are smaller instances of the same problem.
2. Conquer the subproblems by solving them recursively.
3. Combine the solutions to the subproblems into the solution for the original problem.

To truly understand the technique, first try to understand and solve the following problem. Then, we will work on it together.

Convert Sorted Array to Binary Search Tree

I intentionally left space to draw on using tabs.

implementation



```
class Solution:
    def recursive_helper(self, left, right, nums):
            # base case
            if left > right:
                return None

            mid = (left + right)//2

            cur_node = TreeNode(nums[mid])

            left_sub_tree = self.recursive_helper(left, mid - 1, nums)
            right_sub_tree = self.recursive_helper(mid + 1, right, nums)

            cur_node.left = left_sub_tree
            cur_node.right = right_sub_tree

            return cur_node

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


        left = 0
        right = len(nums) - 1
        return self.recursive_helper(left, right, nums)
```

Time and space complexity

- We can think of the time complexity of this algorithm as O(N) since we are accessing each number in 'nums' only once
- And for space complexity its the depth.
- This means, O(log(N)) for a full binary tree and O(N) for a skewed binary tree, where N is the number of nodes in the tree.

Here is another Divide and conquer problem

Validate BST

Things to pay attention

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

Backtracking

Permutation

Combinations

Subsets

Subsets II

Combination Sum

Divide and Conquer

Convert Sorted Array to Binary Search Tree

Balance a binary search tree

Maximum Binary Tree

Validate BST

Sort List

Good can almost always be better

Emre Varol