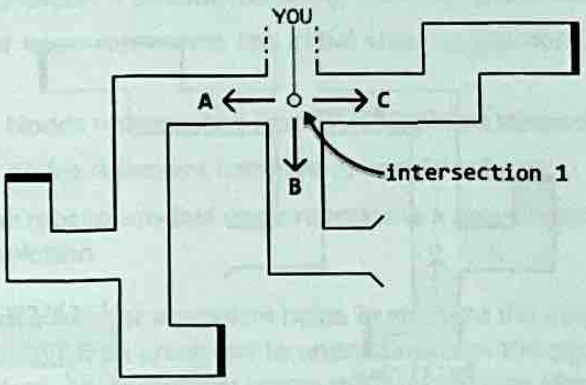
14

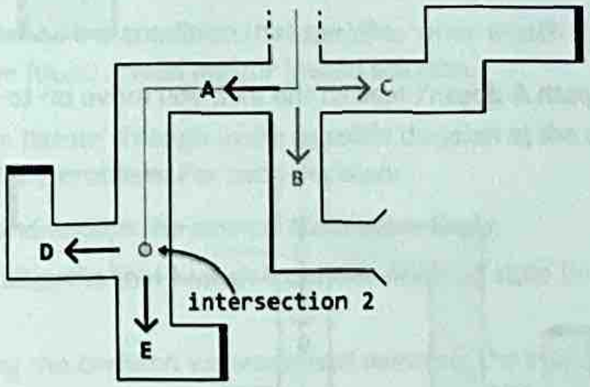
Backtracking

Introduction to Backtracking

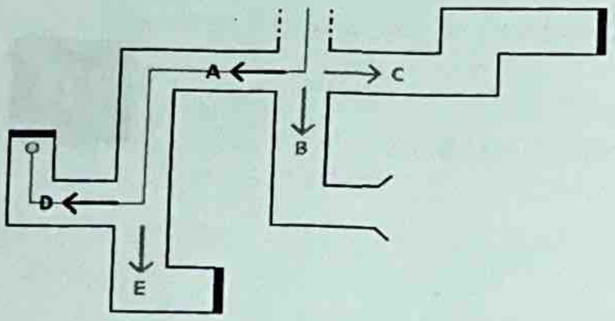
Imagine you're stuck at an intersection point in a maze, and you know one of the three routes ahead leads to the exit:

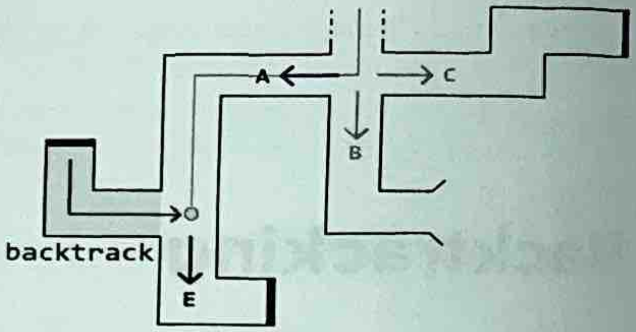


However, you're not sure which route to take. To find the exit, you decide to try each option one by one, starting with option A. As you walk through passage A, you encounter a new intersection containing two more routes, D and E:

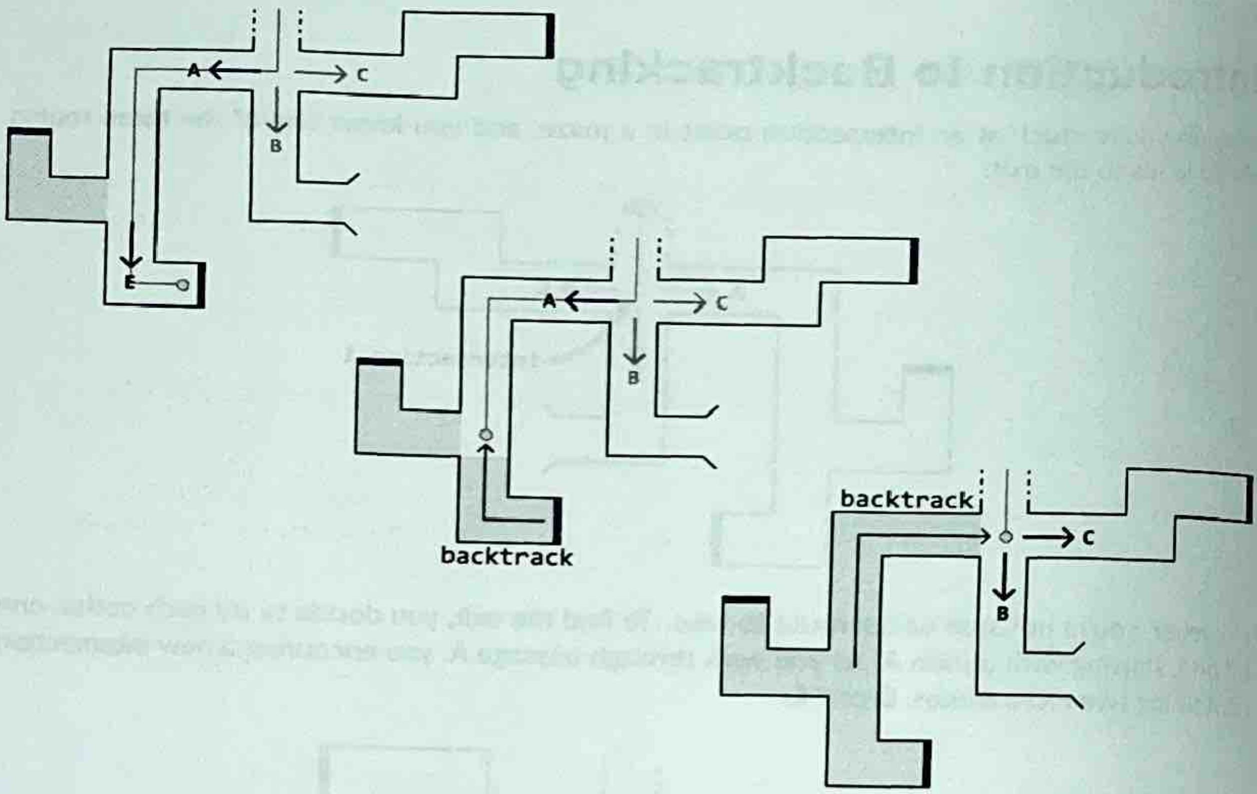


You try option D, but it leads to a dead end. So, you backtrack to the second intersection point:

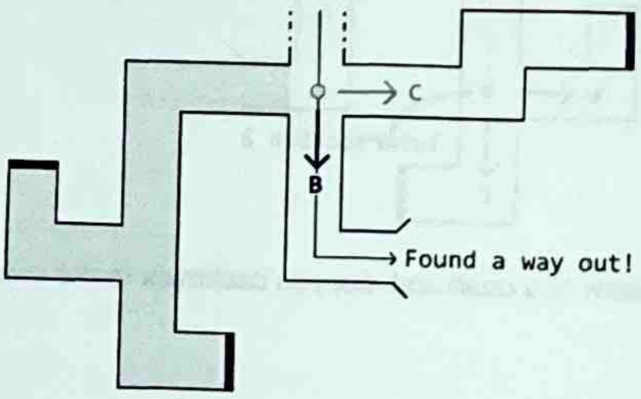




Next, you try option E, but it also leads to a dead end, so you backtrack to the second junction point. After concluding that neither path D or E works, you backtrack again to the first intersection point:



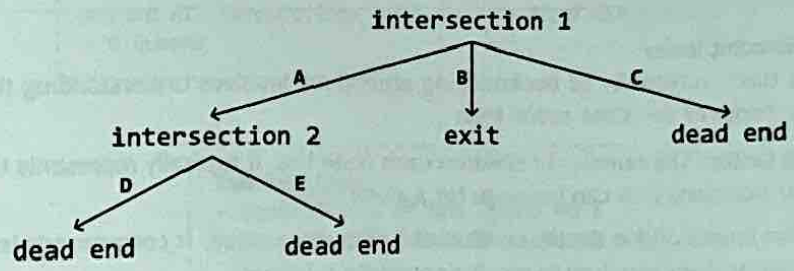
Having determined that path A doesn't lead to the exit, you move on to path B and discover that it leads to the exit:



This brute force process of testing all possible paths and backtracking upon failure is called 'back-tracking.'

State space tree
In backtracking, the state space tree, also known as the decision tree, is a conceptual tree constructed by considering every possible decision that can be made at each point in a process.

For example, here's how we would represent the state space tree for the maze scenario:



Here's a simplified explanation of a state space tree:

- Edges: Each edge represents a possible decision, move, or action.
- Root node: The root node represents the initial state or position before any decisions are made.
- Intermediate nodes: Nodes representing partially completed states or intermediate positions.
- Leaf nodes: The leaf nodes represent complete or invalid solutions.
- Path: A path from the root to any leaf node represents a sequence of decisions that lead to a complete or invalid solution.

Drawing out the state space tree for a problem helps to visualize the entire solution space, and all possible decisions. In addition, it's a great way to understand how the algorithm works. By figuring out how to traverse this tree, we essentially create the backtracking algorithm.

Backtracking algorithm
Traversing the state space tree is typically done using recursive DFS. Let's discuss how it's implemented at a high level.

Termination condition: Define the condition that specifies when a path should end. This condition should define when we've found a valid and/or invalid solution.

Iterate through decisions: Iterate through every possible decision at the current node, which contains the current state of the problem. For each decision:

1. Make that decision and update the current state accordingly.
2. Recursively explore all paths that branch from this updated state by calling the DFS function on this state.
3. Backtrack by undoing the decision we made and reverting the state.

Below is a crude template for backtracking:

```
def dfs(state):
    # Termination condition.
    if meets_termination_condition(state):
        process_solution(state)
        return
    # Explore each possible decision that can be made at the current
    # state.
    for decision in possible_decisions(state):
        make_decision(state, decision)
        dfs(state)
        undo_decision(state, decision)  # Backtrack.
```

Analyzing time complexity

Analyzing the time complexity of backtracking algorithms involves understanding the branching factor and the depth of the state space tree:

- Branching factor: The number of children each node has. It typically represents the maximum number of decisions that can be made for a given state.
- Depth: The length of the deepest path in the state space tree. It corresponds to the number of decisions or steps required to reach a complete solution.

The time complexity is often estimated as $O(b^d)$, where b denotes the branching factor and d denotes the depth. This is because in the worst case, every node at each level of the tree needs to be explored during a typical backtracking algorithm.

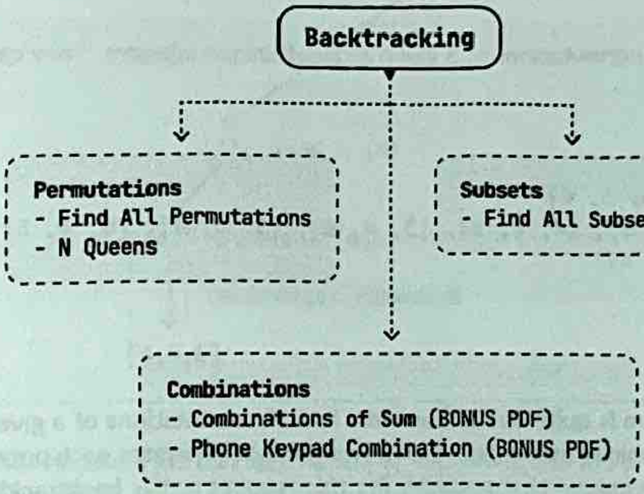
When to use backtracking

Backtracking is useful when we need to explore all possible solutions to a problem. For example, if we need to find all possible ways to arrange items, or generate all possible subsets, permutations, or combinations, backtracking can help to identify every possible solution.

Real-world Example

AI algorithms for games: backtracking is used in AI algorithms for games like chess and Go to explore possible moves and strategies. The programs examine each potential move, simulate the game's progression, and evaluate the outcome. If a move leads to an unfavorable position, the program will backtrack to the previous move and try alternative options, systematically exploring the game tree until it finds the optimal strategy.

Chapter Outline



Find All Permutations

Return all possible permutations of a given array of unique integers. They can be returned in any order.

Example:

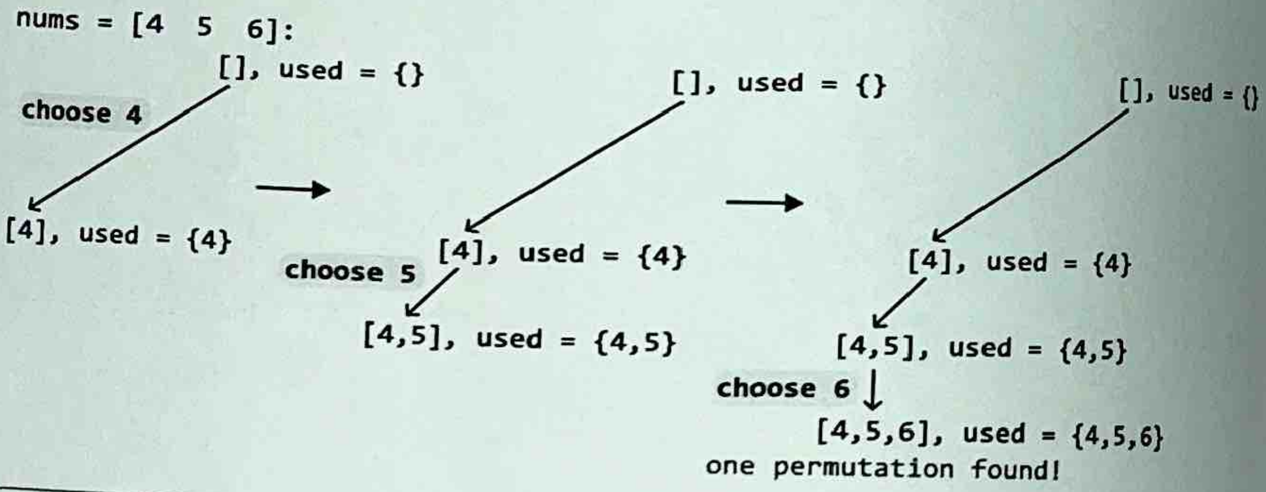
```
Input: nums = [4, 5, 6]
Output: [[4, 5, 6], [4, 6, 5], [5, 4, 6], [5, 6, 4], [6, 4, 5],
         [6, 5, 4]]
```

Intuition

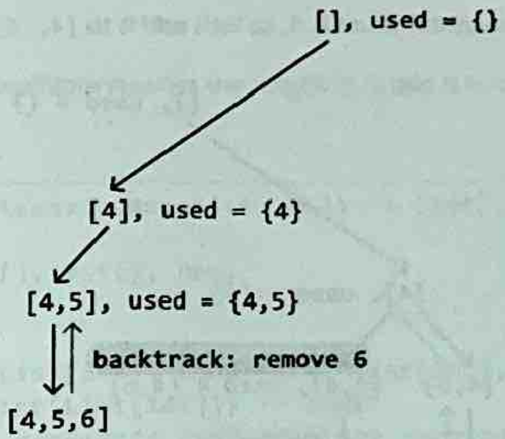
Our task in this problem is quite straightforward: find all permutations of a given array. The key word here is "all". To achieve this, we need an algorithm that generates each possible permutation one at a time. The technique that naturally fits this requirement is **backtracking**. As with any backtracking solution, it's useful to first visualize the state space tree.

State space tree

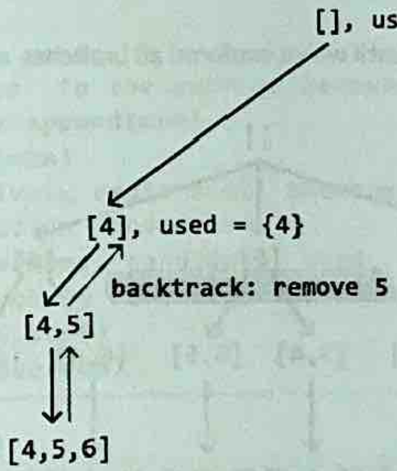
Let's figure out how to build just one permutation. Consider the array [4, 5, 6]. We can start by picking one number from this array for the first position of this permutation. For the second position, let's pick a different number. We can keep adding numbers like this until all the numbers from the array are used. To avoid reusing numbers, let's also keep track of the used numbers using a hash set.



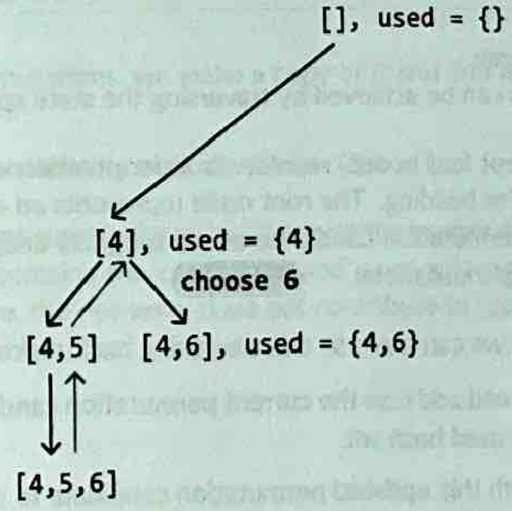
Now that we've found one permutation, let's *backtrack* to find others. Start by removing the most recently added number, 6, bringing us back to [4, 5]:



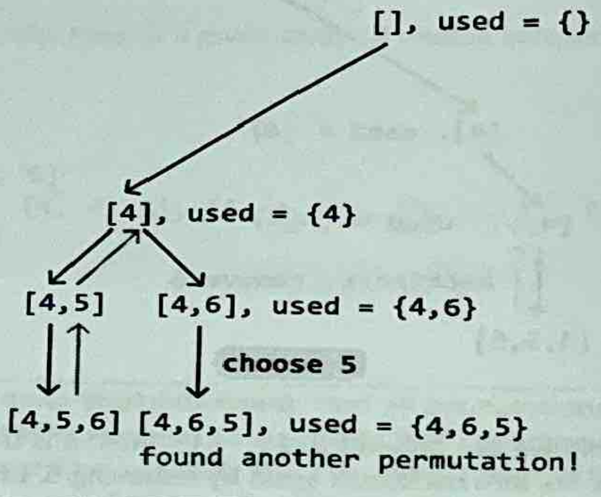
Are there any other numbers we can append to [4, 5]? Well, 6 is the only option at this point, which we already explored. So, let's backtrack again by removing 5, bringing us back to [4]:



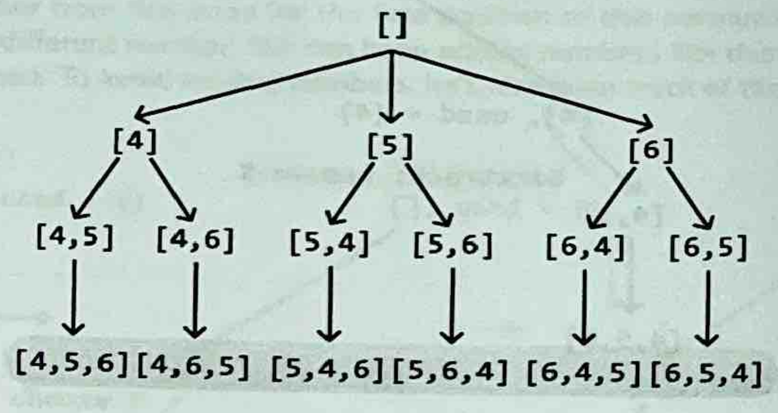
Are there any numbers other than 5 we can add to [4] at this point? Yes, we can use 6, so let's add it and continue searching:



The only number we can use at this point is 5, so let's add it to `[4, 6]`, giving us another permutation:



Following this backtracking process until we've explored all branches allows us to generate all permutations:



Every time we reach a permutation (i.e., when the permutation we're building reaches a size of n, where n denotes the length of the input array), add it to our output.

Traversing the state space tree

Generating all permutations can be achieved by traversing the state space tree.

Each node in this tree, except leaf nodes, represents a permutation candidate: a partially completed permutation that we're building. The root node represents an empty permutation, and an element is added to each permutation candidate as we progress deeper into the tree. The leaf nodes represent completed permutations.

Starting from the root node, we can traverse this tree using backtracking:

1. Pick an unused number and add it to the current permutation candidate. Mark this number as used by adding it to the used hash set.
2. Make a recursive call with this updated permutation candidate to explore its branches.
3. Backtrack: remove the last number we added to the current candidate array, and the used

hash set.

Whenever a permutation candidate reaches the length of n, add it to our output.

Implementation

```
def find_all_permutations(nums: List[int]) -> List[List[int]]:
    res = []
    backtrack(nums, [], set(), res)
    return res

def backtrack(nums: List[int], candidate: List[int], used: Set[int],
              res: List[List[int]]) -> None:
    # If the current candidate is a complete permutation, add it to the
    # result.
    if len(candidate) == len(nums):
        res.append(candidate[:])
        return
    for num in nums:
        if num not in used:
            # Add 'num' to the current permutation and mark it as used.
            candidate.append(num)
            used.add(num)
            # Recursively explore all branches using the updated
            # permutation candidate.
            backtrack(nums, candidate, used, res)
            # Backtrack by reversing the changes made.
            candidate.pop()
            used.remove(num)
```

Complexity Analysis

Time complexity: The time complexity of `find_all_permutations` is $O(n \cdot n!)$. Here's why:

- Starting from the root, we recursively explore n candidates.
- For each of these n candidates, we explore $n - 1$ more candidates, then $n - 2$ more candidates, etc, until we have explored all permutations. This results in a total of $n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 1 = n!$ permutations.
- For each of the $n!$ permutations, we make a copy of it and add it to the output, which takes $O(n)$ time.

This results in a total time complexity of $O(n!) \cdot O(n) = O(n \cdot n!)$.

Space complexity: The space complexity is $O(n)$ because the maximum depth of the recursion tree is n. The algorithm also maintains the `candidate` and `used` data structures, both of which also contribute $O(n)$ space. Note, the `res` array does not contribute to space complexity.

Find All Subsets

Return all possible subsets of a given set of unique integers. Each subset can be ordered in any way, and the subsets can be returned in any order.

Example:

```
Input: nums = [4, 5, 6]
Output: [[], [4], [4, 5], [4, 5, 6], [4, 6], [5], [5, 6], [6]]
```

Intuition

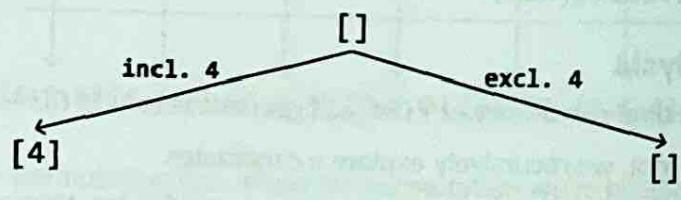
The key intuition for solving this problem lies in understanding that each subset is formed by making a specific decision for every number in the input array: **to include the number, or exclude it.** For example, from the array `[4, 5, 6]`, the subset `[4, 6]` is created by including 4, excluding 5, and including 6.

Let's have a look at what the state space tree looks like when making this decision for every element.

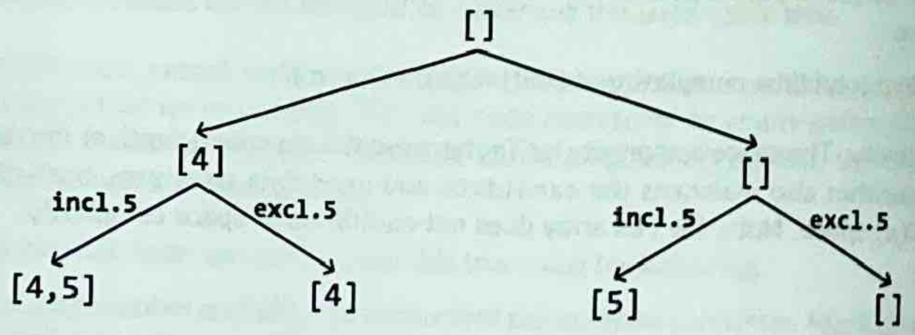
State space tree

Consider the input array `[4, 5, 6]`. Let's start with the root node of the tree, which is an empty subset:

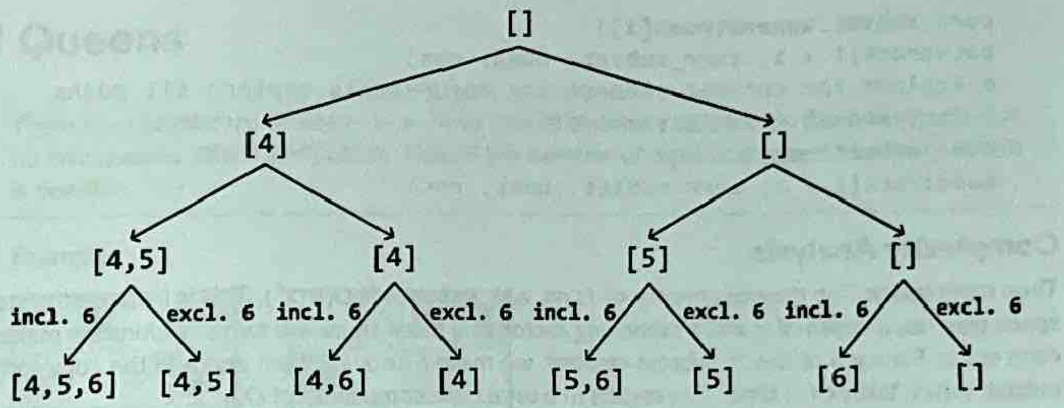
To figure out how we branch out from here, let's consider our decision of whether to include or exclude an element. Let's make this decision with the first element of the input array, 4:



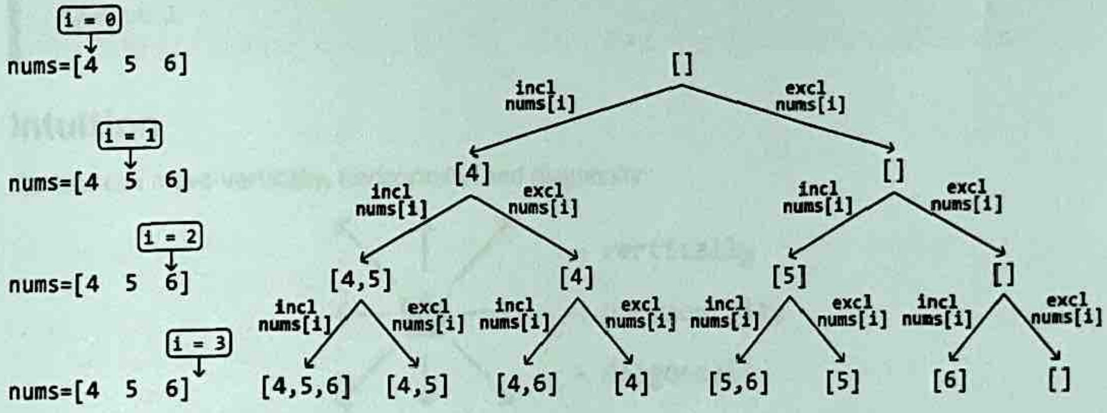
For each of these subsets, we repeat the process, branching out again based on the same choice for the second element: include or exclude it:



Finally, for the third element, we continue branching out for each existing subset based on whether we include or exclude this element:



One important thing missing from this state space tree is a way to tell which element of the input array we're making a decision on at each node of the tree. We can use an index, `i`, for this:



As shown, the final level of the tree (i.e., when `i == n`, where n denotes the length of the input array) contains all the subsets of the input array. We can add each of these subsets to our output. To get to these subsets, we need to traverse the tree, and **backtracking** is great for this.

Implementation

```python
def find_all_subsets(nums: List[int]) -> List[List[int]]:
    res = []
    backtrack(0, [], nums, res)
    return res

def backtrack(i: int, curr_subset: List[int], nums: List[int],
              res: List[List[int]]) -> None:
    # Base case: if all elements have been considered, add the
    # current subset to the output.
    if i == len(nums):
        res.append(curr_subset[:])
        return
    # Include the current element and recursively explore all paths
    # that branch from this subset.
    curr_subset.append(nums[i])
    backtrack(i + 1, curr_subset, nums, res)
    # Exclude the current element and recursively explore all paths
    # that branch from this subset.
    curr_subset.pop()
    backtrack(i + 1, curr_subset, nums, res)
```

Complexity Analysis

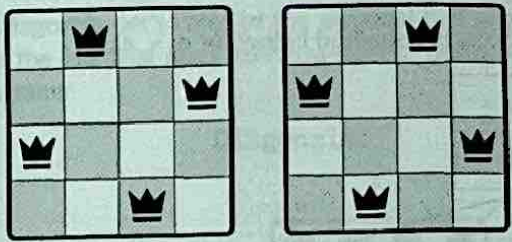
Time complexity: The time complexity of `find_all_subsets` is $O(n \cdot 2^n)$. This is because the state space tree has a depth of n and a branching factor of 2 since there are two decisions we make at each state. For each of the 2^n subsets created, we make a copy of them and add the copy to the output, which takes $O(n)$ time. This results in a total time complexity of $O(n \cdot 2^n)$.

Space complexity: The space complexity is $O(n)$ because the maximum depth of the recursion tree is n. The algorithm also maintains the `curr_subset` data structure, which also contributes $O(n)$ space. Note, the res array does not contribute to space complexity.

N Queens

There is a chessboard of size n x n. Your goal is to place n queens on the board such that no two queens attack each other. Return the number of distinct configurations where this is possible.

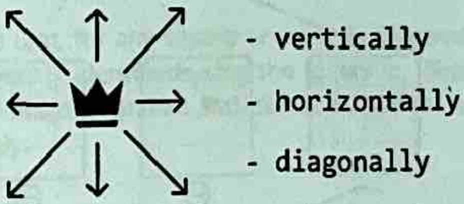
Example:



Input: n = 4
Output: 2

Intuition

Queens can move vertically, horizontally, and diagonally:



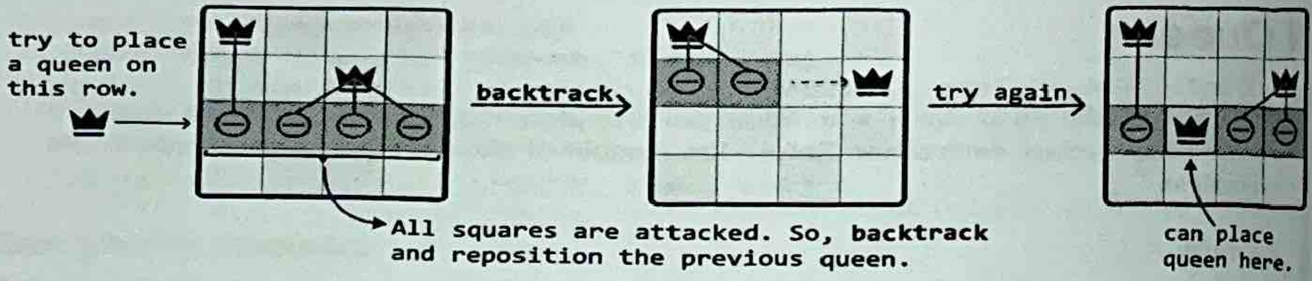
So, it's only possible to place a queen on a square of the board when:

- No other queen occupies the same row of that square.
- No other queen occupies the same column of that square.
- No other queen occupies either diagonal of that square.

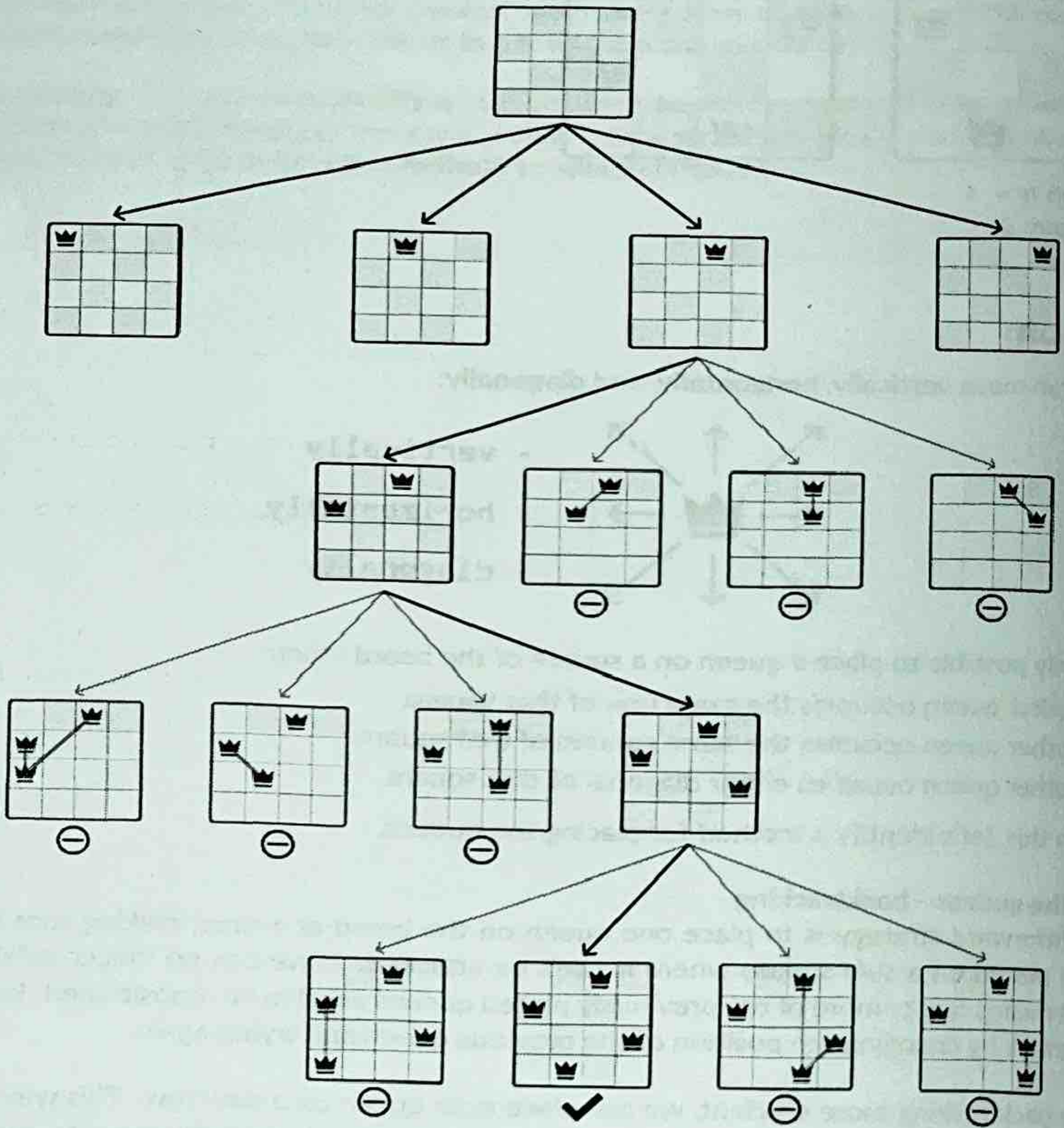
Based on this, let's identify a method for placing the queens.

Placing the queens - backtracking
A straightforward strategy is to place one queen on the board at a time, making sure each new queen is placed on a safe square where it can't be attacked. If we can no longer safely place a queen, it means one or more of the previously placed queens need to be repositioned. In this case, we backtrack by changing the position of the previous queen and trying again.

To make backtracking more efficient, we can place each queen on a new row. This way, we don't have to worry about conflicts between queens on the same row, and only need to check for an opposing queen on the same column and along the diagonals of the square where the new queen is placed. If a queen cannot be placed anywhere on this new row, we backtrack, reposition the previous row's queen, and then try again:


try to place a queen on this row.
backtrack
try again
All squares are attacked. So, backtrack and reposition the previous queen.
can place queen here.

A partial state space tree for this backtracking process is visualized below for n = 4:



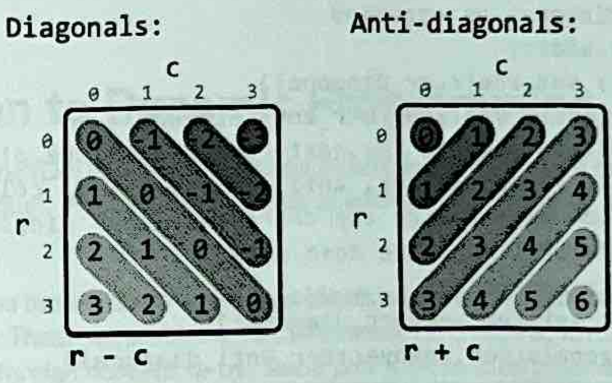
We're still left with some questions. In particular, how can we tell if a square is being attacked, and how exactly do we 'place' or 'remove' a queen?

Detecting opposing queens

One challenge in this problem is determining if a square is attacked by another queen. We could do a linear search across the row, column, and diagonals every time we want to place a new queen, but this is quite inefficient. A key observation is that we don't necessarily need to know the exact positions of all the queens. We only need to know if there exists a queen in any given square's row, column, or diagonals. We can use **hash sets** to efficiently check for this.

Note that we don't need a hash set for rows because we always place each queen on a different row. For columns, whenever we place a new queen on a square `(r, c)`, we can add that square's column id (`c`) to a column hash set.

What about diagonals? How can we determine which diagonal we're on? Since there are two types of diagonals, let's refer to the diagonal that goes from top-left to bottom-right as the **'diagonal'**, and the one that goes from top-right to bottom-left as the **'anti-diagonal'**. Consider the following diagrams:



The key observation here is that, for any square `(r, c)`, its diagonal can be identified using the id `r - c`, and its anti-diagonal is identified using the id `r + c`. Similarly to how we keep track of column ids, we can use a diagonal and an anti-diagonal hash set to keep track of diagonal and anti-diagonal ids, respectively.

Placing and removing a queen

Now that we have a way to identify opposing queens, we know the action of 'placing' a queen means adding its column, diagonal, and anti-diagonal ids to their respective hash sets. Inversely, to remove a queen, we just remove those exact ids from the hash sets.

Implementation

Note, this implementation uses a global variable as it leads to a more readable solution. However, it's important to confirm with your interviewer whether global variables are acceptable.

```python
res = 0

def n_queens(n: int) -> int:
    dfs(0, set(), set(), set(), n)
    return res

def dfs(r: int, diagonals_set: Set[int], anti_diagonals_set: Set[int],
        cols_set: Set[int], n: int) -> None:
    global res
    # Termination condition: If we have reached the end of the rows,
    # we've placed all 'n' queens.
    if r == n:
        res += 1
        return
    for c in range(n):
        curr_diagonal = r - c
        curr_anti_diagonal = r + c
        # If there are queens on the current column, diagonal or
        # anti-diagonal, skip this square.
        if (c in cols_set or curr_diagonal in diagonals_set or
            curr_anti_diagonal in anti_diagonals_set):
            continue
        # Place the queen by marking the current column, diagonal, and
        # anti-diagonal as occupied.
        cols_set.add(c)
        diagonals_set.add(curr_diagonal)
        anti_diagonals_set.add(curr_anti_diagonal)
        # Recursively move to the next row to continue placing queens.
        dfs(r + 1, diagonals_set, anti_diagonals_set, cols_set, n)
        # Backtrack by removing the current column, diagonal, and
        # anti-diagonal from the hash sets.
        cols_set.remove(c)
        diagonals_set.remove(curr_diagonal)
        anti_diagonals_set.remove(curr_anti_diagonal)
```

Complexity Analysis

Time complexity: The time complexity of n_queens is $O(n!)$. Here's why:

- For the first queen, there are n choices for its position.
- For the second queen, there are $n - a$ choices for its position, where a denotes the number of squares on the second row attacked by the first queen.
- The third queen has $n - b$ choices, where b denotes the number of squares on the third row attacked by the previous two queens, and $b < a$.
- This process continues for subsequent queens, resulting in a total of $n \cdot (n - a) \cdot (n - b) \cdot ... \cdot 1$ choices. Even though this doesn't exactly equate to $n!$ $(n \cdot (n - 1) \cdot (n - 2) \cdot ... \cdot 1)$, this trend approximately results in a factorial growth of the search space.

Space complexity: The space complexity is $O(n)$ because the maximum depth of the recursion tree is n. The hash sets also contribute to this space complexity because they each store up to n values.