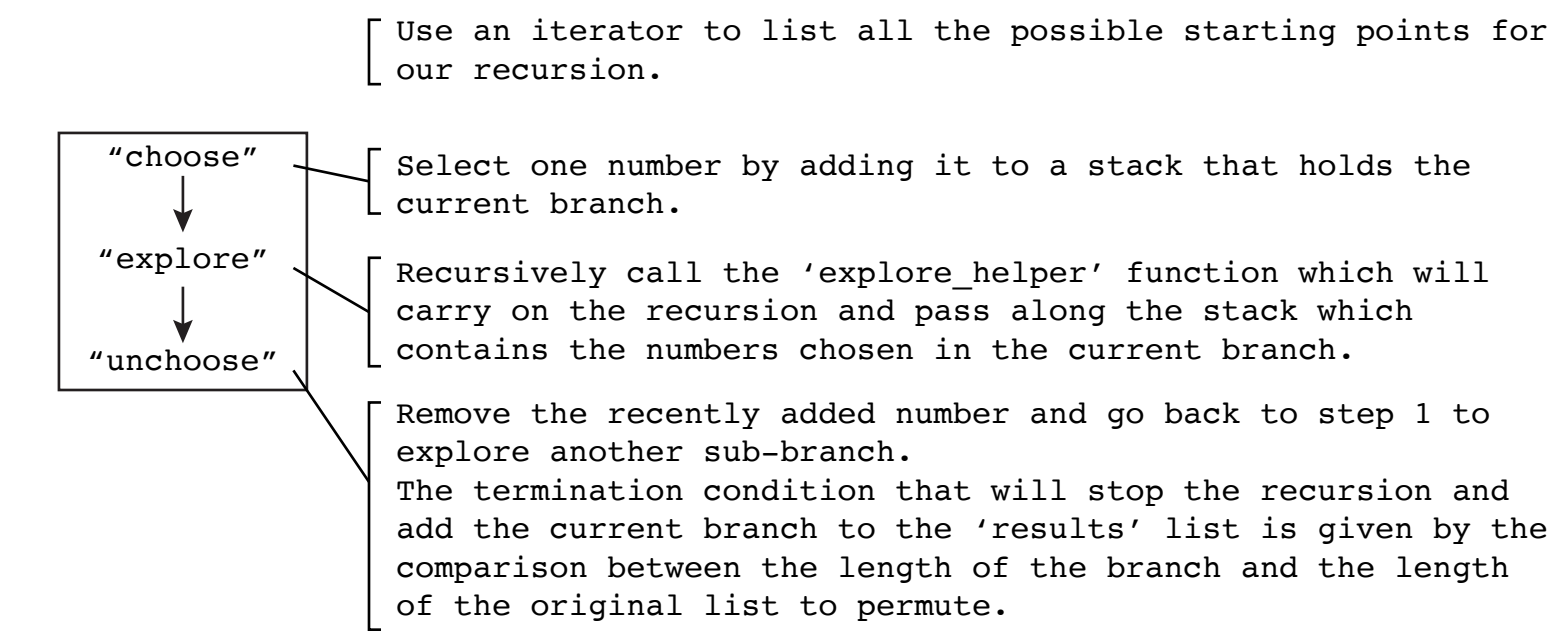


Every backtracking problem can be solved by the following strategy:



Subset (LC 78)

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.

```
lst = ["a", "b", "c"]
```

```
answer = [
    ['a', 'b', 'c'],
    ['a', 'b'],
    ['a', 'c'],
    ['a'],
    ['b', 'c'],
    ['b'],
    ['c'],
    []]
```

Subset II (LC 90)

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.

```
lst = ["a", "b", "c", "c"]
```

```
answer = [
    ['a', 'b', 'c', 'c'],
    ['a', 'b', 'c'],
    ['a', 'b'],
    ['a', 'c', 'c'],
    ['a', 'c'],
    ['a'],
    ['b', 'c', 'c'],
    ['b', 'c'],
    ['b'],
    ['c', 'c'],
    ['c'],
    []]
```

```
class SolutionSubset:
    def subsets(self, nums: List[any]) -> List[List[any]]:
        answer = []
        subset = []

        def backtrack(i):
            if i >= len(nums):
                answer.append(subset.copy())
                return

            subset.append(nums[i])
            backtrack(i + 1)

            subset.pop()
            backtrack(i + 1)

        backtrack(0)

        return answer
```

```
class SolutionSubsetII:
    def subsetWithDup(self, nums: List[any]) -> List[List[any]]:
        answer = []
        subset = []
        nums.sort()

        def backtrack(i):
            if i == len(nums):
                answer.append(subset.copy())
                return

            subset.append(nums[i])
            backtrack(i + 1)
            subset.pop()

            while (i + 1 < len(nums) and
                  nums[i] == nums[i + 1]):
                i += 1

            backtrack(i + 1)

        backtrack(0)

        return answer
```

sort the nums first

add to the subset, try the backtrack by incrementing i; remove from subset, try the backtrack by incrementing i

keep incrementing i if the nums[i] is a duplicate of the previous.

then try the backtrack by incrementing i;

this allows getting all duplicated values. without it, returns a set without duplicates (incorrect).

```
answer = [['a', 'b', 'c'],
          ['a', 'b'],
          ['a', 'c'],
          ['a'],
          ['b', 'c'],
          ['b'],
          ['c'],
          []]
```

BACKTRACKING

Combination Sum (LC 39)

Given a collection of candidate numbers (candidates) and a target number (target), return a list of all unique combinations of candidates where the chosen numbers sum to target.
The same number may be chosen from candidates an unlimited number of times.
The solution set must not contain duplicate combinations.

```
candidates = [2, 3, 5]
target = 8
```

```
answer = [
    [2, 2, 2, 2],
    [2, 3, 3],
    [3, 5]]
```

```
class SolutionCombinationSum:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        answer = []
        permutation = []

        def backtrack(i):
            if sum(permutation) == target:
                answer.append(permutation.copy())
                return

            if i >= len(candidates) or sum(permutation) > target:
                return

            permutation.append(candidates[i])
            backtrack(i)

            permutation.pop()
            backtrack(i + 1)

        backtrack(0)

        return answer
```

check if the sum() of permutation == target also check that it doesn't hit edge cases

add to the permutation, try the backtrack with i; remove last item from permutation, try the backtrack by incrementing i

same number may be chosen from candidates an unlimited number of times

the for-loop inside of the backtrack() function ensures that all indices that are left in candidates is tested, and also allows skipping duplicates.

Inside the for-loop, the algorithm iterates through the candidates list, starting from the i index. The loop allows the algorithm to consider different candidates for the next element in the combination.

Combination Sum II (LC 40)

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target.
Number in candidates may only be used once.

```
candidates = [10,1,2,7,6,1,5]
target = 8
```

```
answer = [
    [1,1,6],
    [1,2,5],
    [1,7],
    [2,6]]
```

```
class SolutionCombinationSumII:
    def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:
        answer = []
        permutation = []
        candidates.sort()

        def backtrack(i):
            if sum(permutation) == target:
                answer.append(permutation.copy())
                return

            if sum(permutation) >= target:
                return

            prev = -1
            for idx in range(i, len(candidates)):
                if candidates[idx] == prev:
                    continue
                permutation.append(candidates[idx])
                backtrack(idx + 1)
                permutation.pop()
                prev = candidates[idx]

        backtrack(0)

        return answer
```

sort the candidates first

for loop; idx starting from i -> end of length of candidates

this SKIPS the idx if there is a duplicate, runs continue

add to the permutation, try the backtrack with i + 1;

candidates.sort() and this code allows skipping duplicate values

Number in candidates may only be used once

Permutations (LC 46)

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

```
lst = [1,2,3]
```

```
answer = [
    [1,2,3],
    [1,3,2],
    [2,1,3],
    [2,3,1],
    [3,1,2],
    [3,2,1]]
```

```
class SolutionPermutations:
    def permute(self, nums: List[any]) -> List[List[any]]:
        answer = []
        permutation = []

        def backtrack():
            if len(permutation) == len(nums):
                answer.append(permutation.copy())
                return

            for num in nums:
                if num not in permutation:
                    permutation.append(num)
                    backtrack()
                    permutation.pop()

        backtrack()

        return answer
```

for loop; goes through each element in nums

append()

backtrack()

pop()

The for-loop iterates through each element (num) in the nums list. For each element, it checks if that element is not already in the permutation list. This check ensures that the same element is not added multiple times to the same permutation

Palindrome Partitioning (LC 131)

Given a string s, partition s such that every substring of the partition is a palindrome.
Return all possible palindrome partitioning of s.

```
s = "aab"
```

```
s2 = "abababab"
```

```
answer = [
    ["a","a","b"],
    ["aa","b"]]

answer2 = [
    ['a', 'b', 'a', 'b', 'r', 'a', 'b', 'a'],
    ['a', 'b', 'a', 'b', 'r', 'a', 'b', 'a'],
    ['a', 'bab', 'r', 'a', 'b', 'a'],
    ['a', 'bab', 'r', 'aba'],
    ['aba', 'b', 'r', 'a', 'b', 'a'],
    ['aba', 'b', 'r', 'aba']]
```

```
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        res, part = [], []

        def backtrack(i):
            if i >= len(s):
                res.append(part.copy())
                return

            for j in range(i, len(s)):
                if self.isPalindrome(s, i, j):
                    part.append(s[i : j + 1])
                    backtrack(j + 1)
                    part.pop()

            backtrack(0)
            return res

        def isPalindrome(self, s, l, r):
            while l < r:
                if s[l] != s[r]:
                    return False
                l, r = l + 1, r - 1
            return True
```

takes a snapshot of part, append to res when i >= len(s)

Inside the for-loop, the variable j ranges from i to the end of the string s. This loop iterates over all possible substrings starting from the current position i.

0 i -----> len(s)

a b a b r a b a

↑↑↑↑↑↑↑↑

```
['a']
['a', 'b']
['a', 'b', 'a']
['a', 'b', 'a', 'b']
['a', 'b', 'a', 'b', 'r']
['a', 'b', 'a', 'b', 'r', 'a']
['a', 'b', 'a', 'b', 'r', 'a', 'b']
['a', 'b', 'a', 'b', 'r', 'a', 'b', 'a']
res.append(part.copy())
```

The for-loop acts to iterate over i -> len(s).
The backtrack() function is iteratively called as i moves up, in the form of j + 1 as the new parameter.
As a result, the backtrack() function first generates a single-letter list.

Then pop() the last item

```
['a', 'b', 'a', 'b', 'r', 'a', 'b', 'a'] part.pop()
['a', 'b', 'a', 'b', 'r', 'a', 'b'] part.pop()
['a', 'b', 'a', 'b', 'r', 'a'] part.pop()
['a', 'b', 'a', 'b', 'r'] part.pop()
i=5; j=6 checks "ab" for palindrome
i=5; j=7 checks "aba" for palindrome
['a', 'b', 'a', 'b', 'r', 'aba']
res.append(part.copy())
```