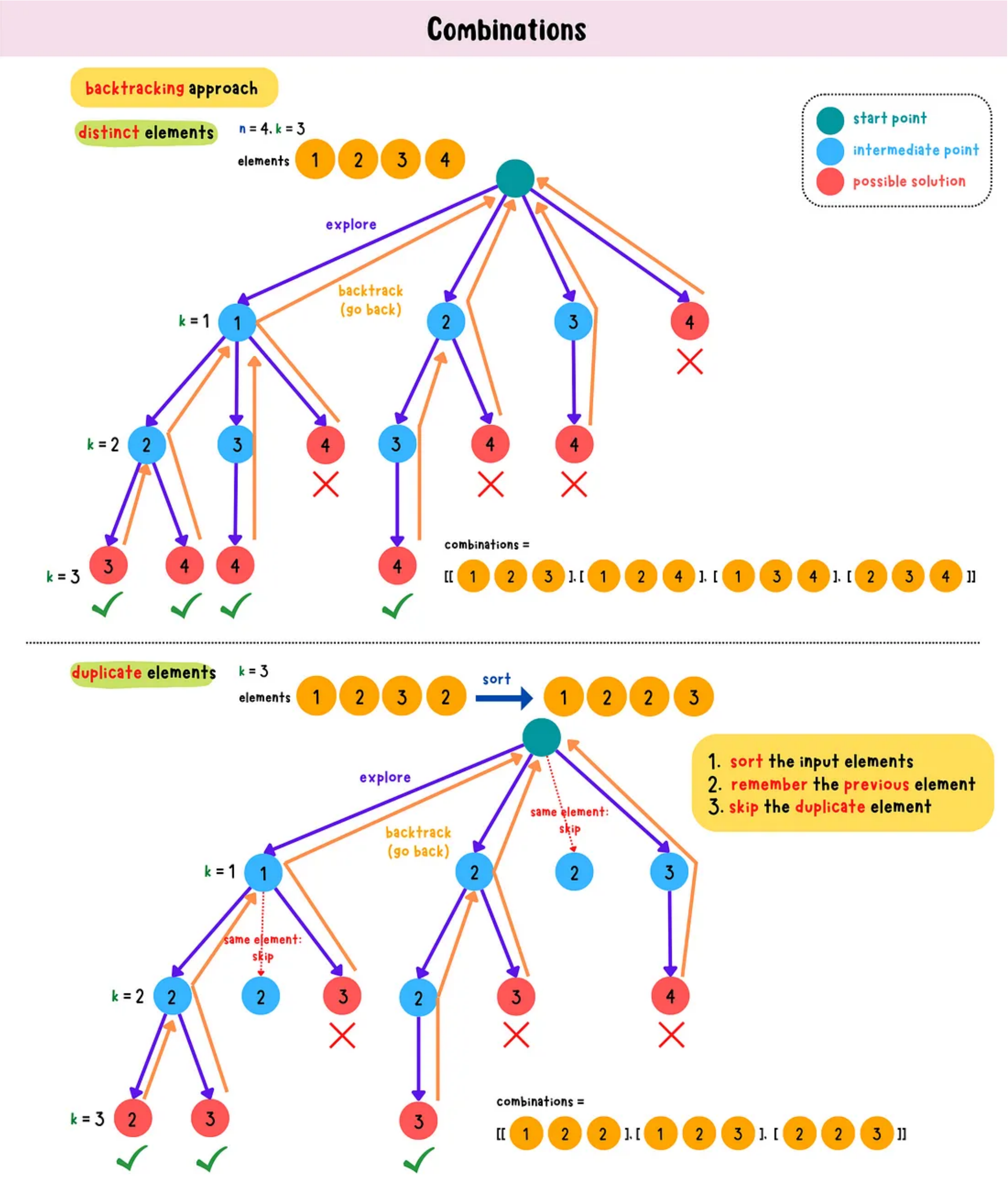
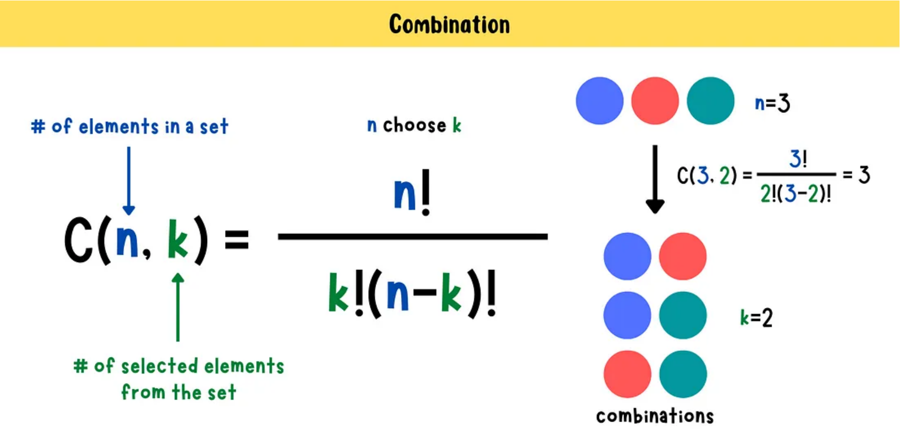
Combinations and Combination Sum With Backtracking

> This article uses **backtracking** approach to find all possible combinations and combinations that sum to target.**Sort** the **input elements** if they contain **duplicate** and **remember the previous element** we explored. If the current element is equal to the previous one, **skip the duplicate** and reach to the different element to try out other possible combinations.



Combination

A selection from a set. The **order does not matter** for a combination.



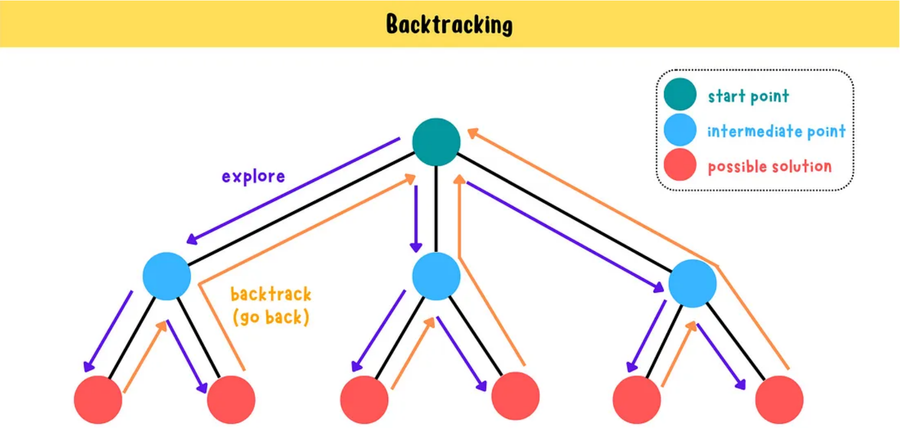
The recursion relation for **combinations** $C(n, k)$ is:

$$C(n,k) = \{ 1 \qquad \text{if } k = 0 \text{ or } k = n,\ C(n-1,k-1) + C(n-1,k) \qquad \text{if } 0 < k < n.$$

- If $k = 0$ or $k = n$, there is exactly one way to choose k elements from n (either all elements or none).
- Otherwise, the number of ways to choose k elements from n is the sum of:
    - $C(n-1, k-1)$: Choosing the current element and $k-1$ from the remaining $n-1$.
    - $C(n-1, k)$: Skipping the current element and choosing all k from the remaining $n-1$.

Backtracking

A **brute force approach** uses **depth-first search(DFS)** to **explore all possible solutions**. If the current solution is not satisfy the constraints, then **eliminate** that and **go back(backtrack)** and check for other solutions.



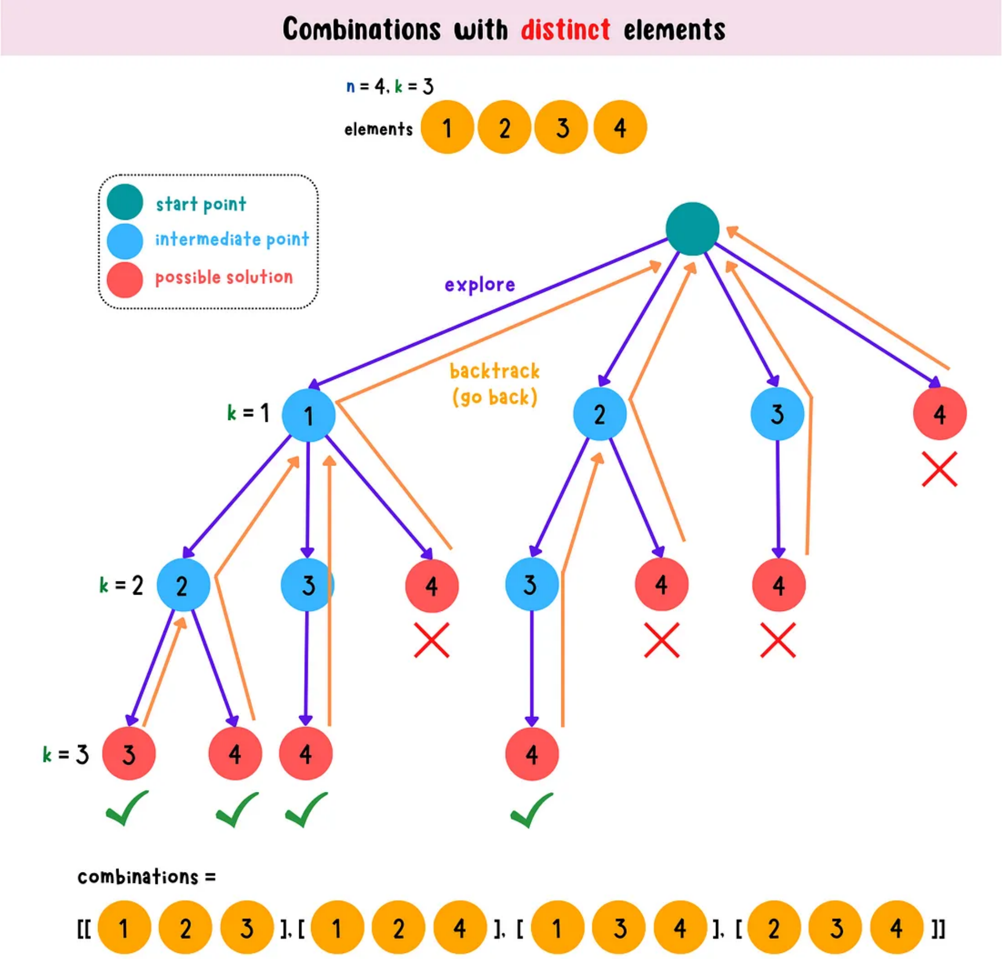
backtracking

Combinations with Distinct Elements

> LeetCode 77: Combinations

Implement **backtracking** technique to find all possible combinations that contains k elements. If the size of a combination equals to k, save the combination. Go back to previous level and keep trying out other possible combinations.

Graphical Explanation



combinations with distinct elements process

Code Implementation

Complexity

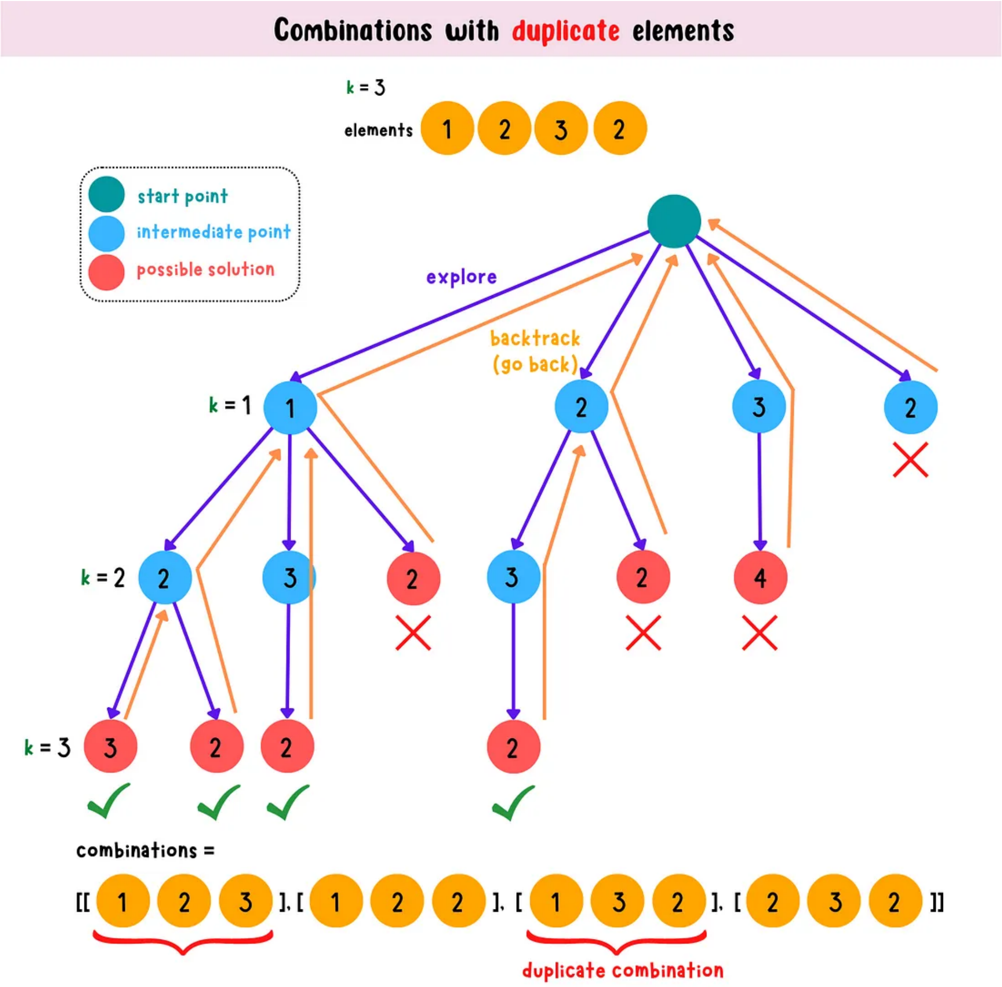
Time: $O(kn*^k)$ ***upper bound*** *(n: branch of the tree, k: the height of the tree) Space: $O(k*(n\ choose k))$, n choose k combinations with a size of k **n**: the total number of elements in the given array **k**: the size of a combination

Python

```python
class Solution:
    def combine(self, n, k):
        combinations = []

        def backtrack(start, currentCombination):
            if len(currentCombination) == k:
                # copy and append
                print("new combination", currentCombination)
                combinations.append(currentCombination[::])
                return

            for i in range(start, n+1):
                currentCombination.append(i)
                backtrack(i+1, currentCombination)
                # clean
                currentCombination.pop()

        backtrack(1, [])
        return combinations


if __name__ == "__main__":
    solution = Solution()
    n, k = 4, 3
    print("combinations", solution.combine(n, k))


"""
output:
('new combination', [1, 2, 3])
('new combination', [1, 2, 4])
('new combination', [1, 3, 4])
('new combination', [2, 3, 4])
('combinations', [[1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]])
"""
```

Combinations with Duplicate Elements

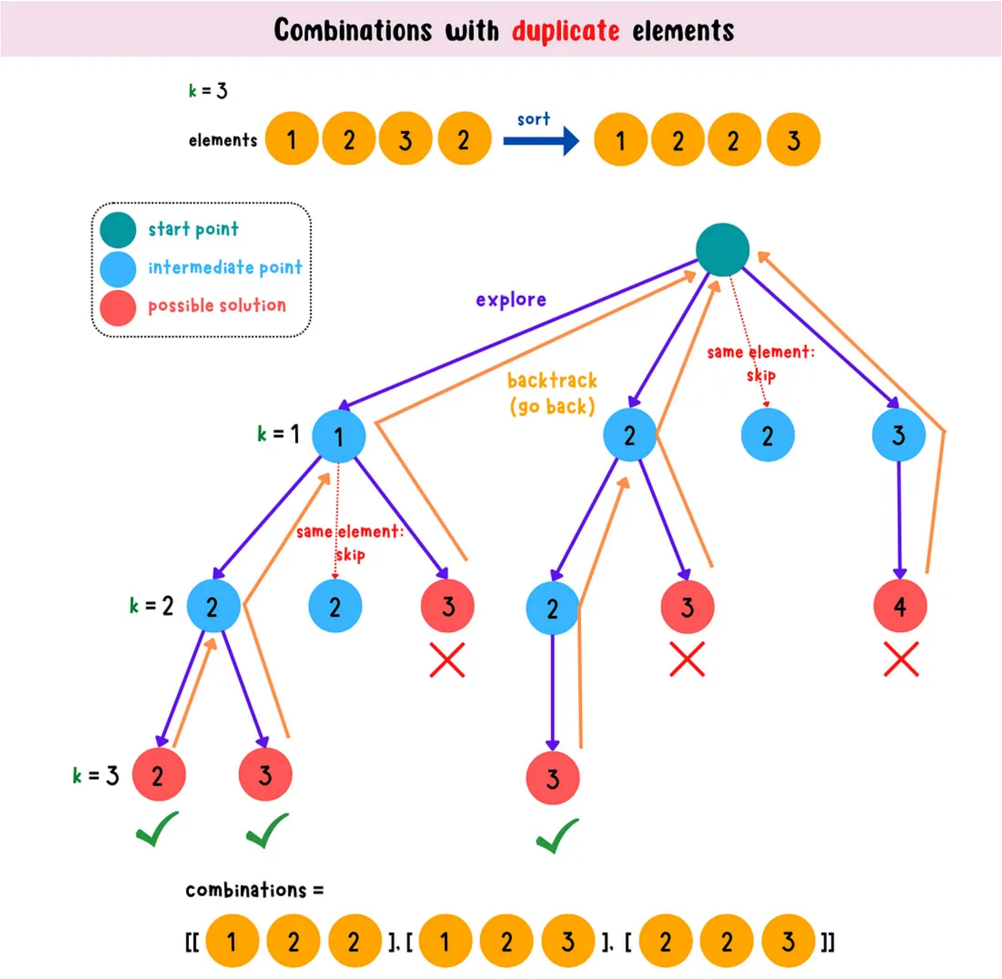
If we follow the same approach as the distinct elements, the combinations are not unique.



not working approach

Sort the input array to allow duplicate numbers to be next to each other and then implement **backtracking** approach. **Remember previous element** we had explored. If the current element equals to the previous one, we **skip this duplicate** and reach to a different element to try out other possible combinations.

Graphical Explanation



combinations with duplicate elements process

Code Implementation

Complexity

Time: $O(kn^k)$ ***upper bound*** *(n: branch of the tree, k: the height of the tree) Space: $O(k*(n\ choose\ k))$, n choose k combinations with a size of k n: the total number of elements in the given array k: the size of a combination

Python

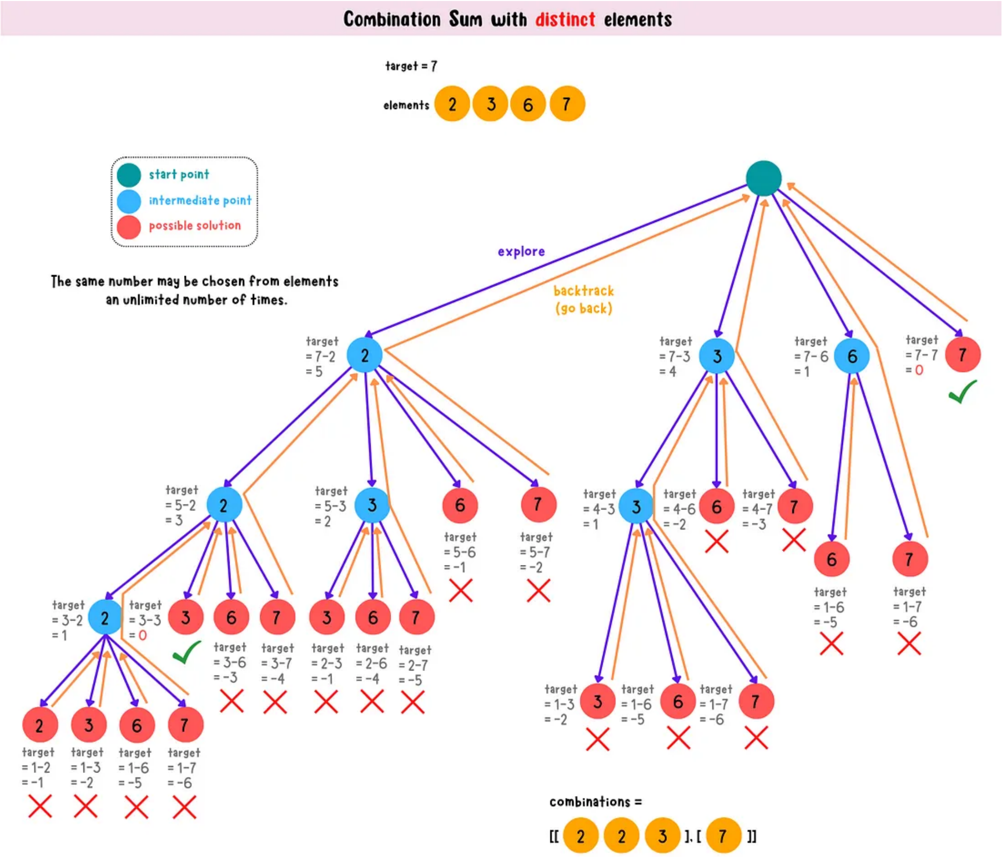
```
class Solution:
    def combine(self, elements, k):
        elements.sort()
        combinations = []

        def backtrack(pos, currentCombination):
            if len(currentCombination) == k:
                # copy and append
                print("new combination", currentCombination)
                combinations.append(currentCombination[::])
                return

            prev = None
            for i in range(pos, len(elements)):
                currentElement = elements[i]
                if prev == currentElement:
                    print("skip", currentElement)
                    continue
                currentCombination.append(currentElement)
                backtrack(i+1, currentCombination)
                # clean
                currentCombination.pop()
                prev = currentElement

        backtrack(0, [])
        return combinations


if __name__ == "__main__":
    solution = Solution()
    elements = [1, 2, 3, 2]
    k = 3
    print("combinations", solution.combine(elements, k))


"""
output:
('new combination', [1, 2, 2])
('new combination', [1, 2, 3])
('skip', 2)
('new combination', [2, 2, 3])
('skip', 2)
('combinations', [[1, 2, 2], [1, 2, 3], [2, 2, 3]])
"""
```

Combination Sum with Distinct Elements

> LeetCode 39: Combination Sum The **same** number may be chosen from `candidates` an **unlimited number of times**.

Implement **backtracking** technique to find all possible combinations that sum to target. If the sum of elements in a combination equals to target, save the combination. Go back to previous level and keep exploring other possible combinations.

Graphical Explanation



combination sum with distinct elements process

Code Implementation

Complexity

Time: $O(2^t)$ Space: $O(2^t)$ t: target value

Python

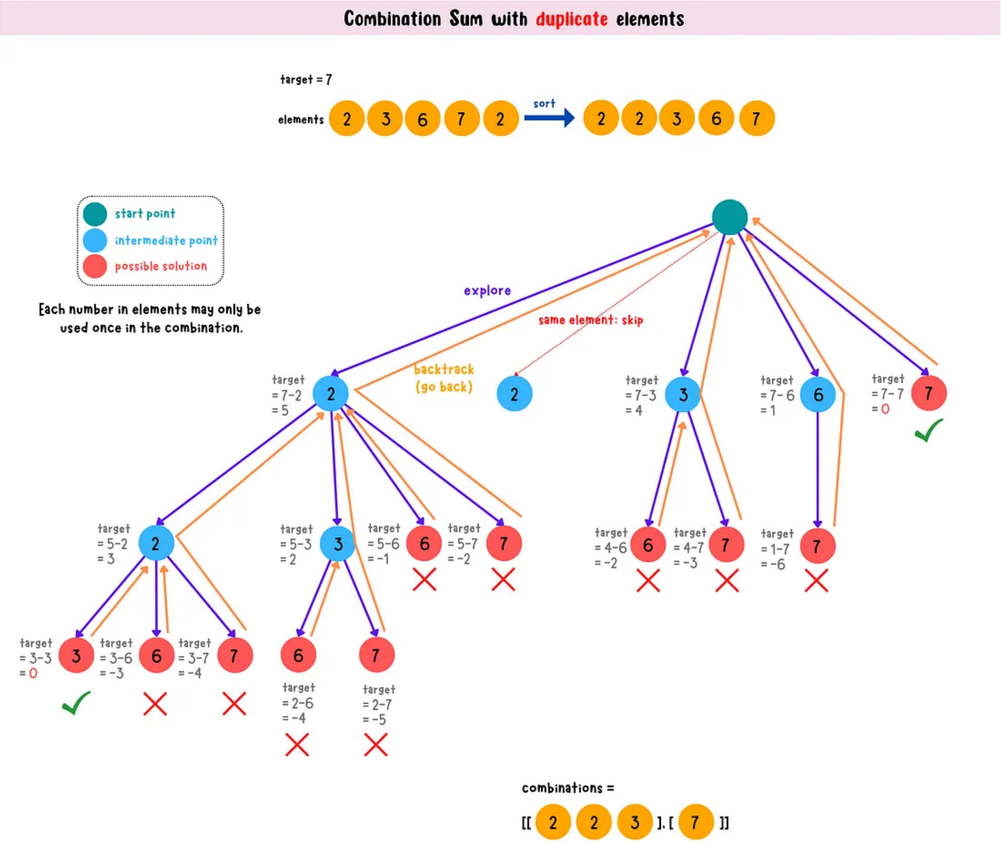
```
class Solution(object):
    def combinationSum(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        result = []

        def backtrack(pos, target, currentCombination):
            if target == 0:
                print("new combination", currentCombination)
                result.append(currentCombination[::])
                return

            if pos == len(candidates) or target < 0:
                return

            for i in range(pos, len(candidates)):
                currentElement = candidates[i]
                currentCombination.append(currentElement)
                backtrack(i, target-currentElement, currentCombination)
                currentCombination.pop()

        backtrack(0, target, [])

        return result

if __name__ == "__main__":
    solution = Solution()
    candidates, target = [2, 3, 6, 7], 7
    print("combinations", solution.combinationSum(candidates, target))


"""
output:
('new combination', [2, 2, 3])
('new combination', [7])
('combinations', [[2, 2, 3], [7]])
"""
```

Combination Sum with Duplicate Elements

> LeetCode 40: Combination Sum II Each number in `candidates` may only be used **once** in the combination.

Sort the input array to allow duplicate numbers to be next to each other and then implement **backtracking** approach. **Remember previous element** we had explored. If the current element equals to the previous one, we **skip this duplicate** and reach to a different element to try out other possible combinations that sum to target.

Graphical Explanation



combination sum with duplicate elements process

Code Implementation

Complexity

Time: $O(2^n)$ Space: $O(2^n)$ n: the size of the input array

Python

```
class Solution(object):
    def combinationSum2(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        candidates.sort()
        result = []

        def backtrack(pos, currentTarget, currentCombination):
            if currentTarget == 0:
                print("new combination", currentCombination)
                result.append(currentCombination[::])
                return

            if pos == len(candidates) or currentTarget < 0:
                return

            prev = None
            for i in range(pos, len(candidates)):
                currentNum = candidates[i]
                if currentNum == prev:
                    print("skip", currentNum)
                    continue

                currentCombination.append(currentNum)
                backtrack(i+1, currentTarget - currentNum, currentCombination)
                currentCombination.pop()
                prev = currentNum

        backtrack(0, target, [])

        return result

if __name__ == "__main__":
    solution = Solution()
    candidates, target = [2, 3, 6, 7, 2], 7
    print("combinations", solution.combinationSum2(candidates, target))

"""
output:
('new combination', [2, 2, 3])
('skip', 2)
('new combination', [7])
('combinations', [[2, 2, 3], [7]])
"""
```

You can access the source code here.

LeetCode Problems:

- 77. Combinations
- 39. Combination Sum
- 40. Combination Sum II