LeetCode 39 40 216 377

https://leetcode.com/problems/combination-sum/description/
https://leetcode.com/problems/combination-sum-ii/description/
https://leetcode.com/problems/combination-sum-iii/description/
https://leetcode.com/problems/combination-sum-iv/description/

Yifeng Zeng

Description

39. Combination Sum

40. Combination Sum II

216. Combination Sum III

377. Combination Sum IV

Idea Report

For LC 39, we are looking for any combination that has the sum of the target. So this becomes a problem that is very similar like subset or permutation problem. So we can do a search. For example we have [2,3,6,7] and target is 7. We can add 2 in the result and looking for 7 - 2 = 5 recursively in the [2,3,6,7] input array. And when we find out that if the target decreases to 0, we found one of the result. This becomes a very standard DFS search problem.

Code 1a

```
class Solution {
    // 1a Combination Sum, for loop
    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> res = new ArrayList<>();
        helper(candidates, target, res, new ArrayList<>(), 0);
        return res;
```

```
    }

    private void helper(int[] candidates, int target, List<List<Integer>> res,
                        List<Integer> path, int index) {
        if (target <= 0 || index >= candidates.length) {
            if (target == 0) {
                res.add(new ArrayList<>(path));
            }
            return;
        }

        for (int i = index; i < candidates.length; i++) {
            path.add(candidates[i]);
            helper(candidates, target - candidates[i], res, path, i);
            path.remove(path.size() - 1);
        }
    }
}
```

Looking at the input [2,3,6,7], target 7. We can choose add one element or not add one element. If we add one element 2, then we are looking for target 7 - 2 = 5 in sub input [2,3,6,7]. If we don't add element 2, then we are looing for target 7 in sub input [3,6,7]. Until we are looking for target 0, then the previous elements we have already added are the results that we want.

Code 1b

```
class Solution {
  // 1b Combination Sum, choose/not choose
  public List<List<Integer>> combinationSum(int[] candidates, int target) {
      List<List<Integer>> res = new ArrayList<>();
      search(target, 0, new ArrayList<>(), res, candidates);
      return res;
  }

  private void search(int target, int startIndex, List<Integer> path,
                      List<List<Integer>> res, int[] candidates) {
      if (target < 0) {
          return;
      }

      if (target == 0) {
          res.add(path);
          return;
      }

      if (startIndex >= candidates.length) {
          return;
```

```
        }

        List<Integer> firstPath = new ArrayList<>(path);
        firstPath.add(candidates[startIndex]);
        search(target - candidates[startIndex],
               startIndex, firstPath, res, candidates);
        search(target, startIndex + 1, path, res, candidates);
    }
}
```

For LC 40, each element should only be used once. Then we do the recursion, we need to start to search from the next index, not the current index. And also, we need to remove the duplex by check is the current element we want to add is the same value with the previous value, but the previous element is not in the result. For example, i we search target 8 in [1,2a,2b,3,4]. If we have a result [1,2a, 4], then we can't use [1, 2b, 4] because that is a duplication. So during the search, when we want to add [2b], we have to check if [2a] is in the result [1, 2a] to add [2b], if it is not we can't add it, otherwise will have both [1,2a] and [1,2b] and that would cause duplication.

Code 2a

```
class Solution {
    // 2a Combination Sum II, for loop
    public List<List<Integer>> combinationSum2(int[] candidates, int target) {
        List<List<Integer>> res = new ArrayList<>();
        Arrays.sort(candidates);
        helper(candidates, target, res, new ArrayList<>(), 0);
        return res;
    }

    private void helper(int[] candidates, int target, List<List<Integer>> res,
                        List<Integer> path, int index) {
        if (target <= 0 || index >= candidates.length) {
            if (target == 0) {
                res.add(new ArrayList<>(path));
            }
            return;
        }

        for (int i = index; i < candidates.length; i++) {
            if (i > index && candidates[i] == candidates[i-1]) {
                continue;
            }
            if (target - candidates[i] < 0) {
                break;
            }
            path.add(candidates[i]);
```

```
            helper(candidates, target - candidates[i], res, path, i + 1);
            path.remove(path.size() - 1);
        }
    }
}
```

Based on choose/not choose method from Code 1b, if we choose to add current element, the next search index is startIndex + 1. If we do not choose to add current element, the next search index is the next element that has a different value of the current element. Based on example [1,1,2,5,6,7,10], target = 8, we have the original (target, pos) pair (8,0). If we choose to add first 1, then the next recursion is (7,1). If we do not choose to add first 1, then we can't the second 1 because we do not add any 1 at all, so the next recursion is (8,2) (This is why I answered (8,2) in the class).

Code 2b

```
class Solution {
  // 2b Combination Sum II, choose/not choose
  public List<List<Integer>> combinationSum2(int[] candidates, int target) {
        Arrays.sort(candidates);
        List<List<Integer>> res = new ArrayList<>();
        search(target, 0, new ArrayList<>(), res, candidates);
        return res;
    }

    private void search(int target, int startIndex, List<Integer> path,
                        List<List<Integer>> res, int[] candidates) {
        if (target < 0) {
            return;
        }
        if (target == 0) {
            res.add(new ArrayList<>(path));
            return;
        }
        if (startIndex >= candidates.length) {
            return;
        }

        List<Integer> firstPath = new ArrayList<>(path);
        firstPath.add(candidates[startIndex]);
        search(target - candidates[startIndex],
               startIndex + 1, firstPath, res, candidates);
        while (startIndex + 1 < candidates.length
               && candidates[startIndex] == candidates[startIndex + 1]) {
            startIndex++;
        }
```

```
            search(target, startIndex + 1, path, res, candidates);
        }
    }
```

For LC 216, similarly to code 2a, we have a for loop to add a number x from 1 to 9 into the result, and recursively find the (target - x) until the target == 0. Because we will have to choose k and only k numbers, so if the result's (path's) length is k and target is 0, we add it to the final result. If there are more then k numbers in the path, or the target is less than 0, we won't find any valid answer, then we just return the recursion.

Code 3a

```
class Solution {
    // 3a Combination Sum III, for loop
    public List<List<Integer>> combinationSum3(int k, int target) {
        List<List<Integer>> res = new ArrayList<>();
        helper(target, res, new ArrayList<>(), k, 1);
        return res;
    }

    private void helper(int target, List<List<Integer>> res,
                        List<Integer> path, int k, int value) {
        if (target == 0 && path.size() == k) {
            res.add(new ArrayList<>(path));
            return;
        }
        if (target < 0 || path.size() > k) {
            return;
        }

        for (int v = value; v <= 9; v++) {
            if (target - v < 0) {
                break;
            }
            path.add(v);
            helper(target - v, res, path, k, v + 1);
            path.remove(path.size() - 1);
        }
    }
}
```

For LC 216, similarly to code 2b, for current value we have 2 choices. One is to add it, the then next recursion level is to search target - value, the next value shoulde be (value + 1). The other choise is not add it, then next recursion level is still to search target, and the next value to add

should also be (value + 1). The exit condition is the same as metioned in 3a, plus if the current value is larger than 9, we return, cause we only choose from 1 to 9.

Code 3b

```
class Solution {
    // 3b Combination Sum III, choose/not choose
    public List<List<Integer>> combinationSum3(int k, int target) {
        List<List<Integer>> res = new ArrayList<>();
        search(target, 1, k, new ArrayList<>(), res);
        return res;
    }

    private void search(int target, int value, int k,
                        List<Integer> path, List<List<Integer>> res) {
        if (target == 0 && path.size() == k) {
            res.add(new ArrayList<>(path));
            return;
        }
        if (target < 0 || path.size() > k || value > 9) {
            return;
        }

        // choose current value
        path.add(value);
        search(target - value, value + 1, k, path, res);
        path.remove(path.size() - 1);
        // not choose current value
        search(target, value + 1, k, path, res);
    }
}
```

For LC 377, the primitive idea is to list all the possible combinations very similar to the permutation but same value can be chose many times. This approach list all the possible combinations and got time limit exceeded.

Code 4a

```
class Solution {
    // Time Limit Exceeded
    // 4a Combination Sum IV, for loop
    public int combinationSum4(int[] candidates, int target) {
        List<List<Integer>> res = new ArrayList<>();
        helper(candidates, target, res, new ArrayList<>());
        return res.size();
```

```
    }

    private void helper(int[] candidates, int target, List<List<Integer>> res,
                        List<Integer> path) {
        if (target <= 0) {
            if (target == 0) {
                res.add(new ArrayList<>(path));
            }
            return;
        }

        for (int i = 0; i < candidates.length; i++) {
            if (target - candidates[i] < 0) {
                break;
            }
            path.add(candidates[i]);
            helper(candidates, target - candidates[i], res, path);
            path.remove(path.size() - 1);
        }
    }
}
```

We are actually just returning how many different combinations are there but not care about the actual combinations. So we can just do combinationSum 1 and find the number of combinations of each result in combinationSum 1, this still got time limit exceeded.

Code 4b

```
class Solution {
    // Time Limit Exceeded
    // 4b Combination Sum IV, choose/not choose
    public int combinationSum4(int[] candidates, int target) {
        int[] res = new int[1];
        search(target, 0, new ArrayList<>(), res, candidates);
        return res[0];
    }

    private void search(int target, int startIndex, List<Integer> path,
                        int[] res, int[] candidates) {
        if (target < 0) {
          return;
        }
        if (target == 0) {
            res[0] += findN(path);
            return;
        }
        if (startIndex >= candidates.length) {
```

```
            return;
        }

        List<Integer> firstPath = new ArrayList<>(path);
        firstPath.add(candidates[startIndex]);
        search(target - candidates[startIndex], startIndex, firstPath, res,
               candidates);
        search(target, startIndex + 1, path, res, candidates);
    }

    private int findN(List<Integer> path) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int i : path) {
            map.put(i, map.getOrDefault(i, 0) + 1);
        }

        int n = path.size();
        int res = 1;
        for (int key : map.keySet()) {
            int value = map.get(key);
            res *= C(n, value);
            n = n - value;
            if (n == 0) {
                break;
            }
        }

        return res;
    }

    private int C(int n, int r) {
        // C(n, r) = n! / r! / (n-r)!
        long res = 1;
        for (long x = r + 1; x <= n; x++) {
            res *= x;
        }
        for (long x = n - r; x > 1; x--) {
            res /= x;
        }
        return (int) res;
    }
}
```

Define comb[n] as the number of combinations to get sum of n. So taking the example of[1,2,3], target = 4, we are looking for comb[4]. We consider the last step that we get a sum of 4, we have 3 different situations:

- The last number we add is [1], so that is comb[3] of different answers, each add 1. So there

are comb[3] different answers add 1 to get comb[4].
- Similarly, we have comb[4-2] = comb[2] of different answers, each add 2 to get sum of 4.
- Similarly, we have comb[4-3] = comb[1].

So

- comb[4] = comb[1] + comb[2] + comb[3]; (for loop of the candidates before 4)
- comb[3] = comb[1] + comb[2]; (for loop of the candidates before 3)
- comb[2] = comb[1]; (for loop of the candidates before 2)

The base case is that comb[0] should be 1, because for any value (say 6) in the candidates, it will be always have one sum which is 6 itself, so comb[6] = comb[0] (then add the number 6).

Code DP

```
class Solution {
    // 4b Combination Sum IV, DP, AC
    public int combinationSum4(int[] candidates, int target) {
        int[] comb = new int[target + 1];
        comb[0] = 1;
        Arrays.sort(candidates);
        for (int i = 1; i <= target; i++) {
            for (int j = 0; j < candidates.length; j++) {
                if (candidates[j] >= comb.length) {
                    break;
                }
                if (i - candidates[j] < 0) {
                    continue;
                }
                comb[i] += comb[i - candidates[j]];
            }
        }
        System.out.println(Arrays.toString(comb));
        return comb[target];
    }
}
```

Summary

- Use for loop or choose/not choose an element to divid it into sub problems to do the DFS.
- Use the following code to early stop the search to speed up:

    - `if (target - candidates[i] < 0) {break;}`
- DP would consider the last step and from last to begnning, and consider the first base case.

Combination Sum IV Follow up

The primitive DP idea is to draw the table

	0 amount	1	2	3	4
0 coin	1	0	0	0	0
1	1	1	1	1	1
2	1	1	2	3	5
3	1	1	2	4	7

We define int[][] f, f[i][j] means there are f[i][j] possible combinations that add up to the amount j using the first i coins in the candidates. The initialization is let all f[][0] = 1 because we always have 1 way to make the amount of 0. For each f[i][j], we need to check f[i][j - coins[0]], f[i][j - coins[1]], f[i][j - coins[2]], f[i][j - coins[3]], …, f[i][j - coins[i]] and sum them together. Because for f[i][j], we have f[i][j - coins[i]] ways to make f[i][j] (based on add coins[i] amount to j - coins[i] amount).

```
class Solution {
    public int combinationSum4(int[] candidates, int target) {
        int[][] f = new int[candidates.length + 1][target + 1];
        for (int i = 1; i <= candidates.length; i++) {
            f[i][0] = 1;
            for (int j = 1; j <= target; j++) {
                for (int k = 0; k < i; k++) {
                    int coin = candidates[k];
                    if (j - coin >= 0) {
                        f[i][j] += f[i][j - coin];
                    }
                }
            }
        }
        for (int[] row : f) {
            System.out.println(Arrays.toString(row));
        }
        return f[candidates.length][target];
    }
```

```
}
```

Because we don't really use the information from previous rows, so we can juse use a 1-D array.

```
class Solution {
    // DP optimize space, AC
    public int combinationSum4(int[] candidates, int target) {
        int[] f = new int[target + 1];
        f[0] = 1;
            for (int j = 1; j <= target; j++) {
                for (int k = 0; k < candidates.length; k++) {
                    int coin = candidates[k];
                    if (j - coin >= 0) {
                        f[j] += f[j - coin];
                    }
                }
            }

        System.out.println(Arrays.toString(f));

        return f[target];
    }
}
```

After clearing up

```
class Solution {
    // DP optimization, AC
    public int combinationSum4(int[] candidates, int target) {
        int[] f = new int[target + 1];
        f[0] = 1;
        Arrays.sort(candidates);
        for (int j = 1; j <= target; j++) {
            for (int coin : candidates) {
                if (j - coin < 0) {
                    break;
                }
                f[j] += f[j - coin];
            }
        }
        return f[target];
    }
}
```