<https://leetcode.com/problems/combination-sum-iii/description/>

Find all valid combinations of k numbers that sum up to n such that the following conditions are true:

Only numbers 1 through 9 are used.

Each number is used **at most once**.

Return **a list of all possible valid combinations**. The list must not contain the same combination twice, and the combinations may be returned in any order.

**Example 1:**

**Input:** k = 3, n = 7

**Output:** [[1,2,4]]

**Explanation:**

1 + 2 + 4 = 7

There are no other valid combinations.

**Example 2:**

**Input:** k = 3, n = 9

**Output:** [[1,2,6],[1,3,5],[2,3,4]]

**Explanation:**

1 + 2 + 6 = 9

1 + 3 + 5 = 9

2 + 3 + 4 = 9

There are no other valid combinations.

**Example 3:**

**Input:** k = 4, n = 1

**Output:** []

**Explanation:**

There are no valid combinations.

Using 4 different numbers in the range [1,9], the smallest sum we can get is 1+2+3+4 = 10 and since 10 > 1, there are no valid combination.

**Constraints:**

2 <= k <= 9

1 <= n <= 60

**Attempt 1: 2023-02-18**

**Solution 1: DFS + Backtracking (10 min)**

**Style 1: For loop in helper() method**

class Solution {

    public List<List<Integer>> combinationSum3(int k, int n) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        helper(1, k, n, result, new ArrayList<Integer>());

        return result;

    }

    private void helper(int curNum, int k, int n, List<List<Integer>> result, List<Integer> tmp) {

        if(tmp.size() == k && n == 0) {

            // Because 'tmp' will continue being modified, we have to put a copy of 'tmp' in result instead

            result.add(new ArrayList<>(tmp));

            return;

        }

        // Early return to avoid keep adding number even size > k or n != 0

        if(tmp.size() == k || n == 0) {

            return;

        }

        // Each number is used at most once, use index + 1 model

        // Additional check for i <= n to avoid redundant call in

        // recursion when i already > n but still <= 9.

        // e.g input k = 3, n = 9, when after we have a combination

        // as [1,2,6], the next recursion input n = 6, but in next

        // recursion i will be 7, i > n, no need to do the iteration,

        // same for i = 8 or 9, so let's block calls by adding i <= n

        for(int i = curNum; i <= Math.min(9, n); i++) {

            tmp.add(i);

            helper(i + 1, k, n - i, result, tmp);

            tmp.remove(tmp.size() - 1);

        }

    }

}

Time Complexity: O(C(9,k))-> O(9^k), C(9, k), the combination of picking k elements out of 9

The time complexity of the provided code is exponential.

The algorithm uses a backtracking approach to generate all combinations of size k that sum up

to n using numbers from 1 to 9.

In the worst case, the algorithm explores all possible combinations. The maximum depth of the

recursion tree is k, and at each level, there are up to 9 choices for the next number (limited

by the numbers from 1 to 9). Therefore, the time complexity is roughly O(9^k).

In practice, the actual time complexity may be lower due to the pruning mechanism in the

backtracking process. The algorithm avoids unnecessary recursive calls when the size of the

current combination exceeds k or when n becomes 0. However, in the worst case, the time

complexity remains exponential.

Space Complexity: O(k)

The space complexity of the provided code is O(k) due to the recursive call stack.

In each recursive call, the algorithm adds an element to the tmp list, and this process continues

until a combination of size k is reached. At that point, the algorithm adds a copy of the tmp list

to the result list, and then the elements are removed one by one (backtracked) until the base case

is reached.

The maximum depth of the recursive call stack is k, as the algorithm explores all combinations of

size k. At each level of the recursion, the tmp list contains at most k elements. Therefore, the

space complexity is O(k).

It's important to note that this analysis considers the space used by the call stack and does not

include the space required for the output result, which is separate and depends on the number of

valid combinations generated.

**OR we just move the condition statement out of for loop**

class Solution {

    public List<List<Integer>> combinationSum3(int k, int n) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        helper(1, k, n, result, new ArrayList<Integer>());

        return result;

    }

    private void helper(int curNum, int k, int n, List<List<Integer>> result, List<Integer> tmp) {

        if(tmp.size() == k && n == 0) {

            // Because 'tmp' will continue being modified, we have to put a copy of 'tmp' in result instead

            result.add(new ArrayList<>(tmp));

            return;

        }

        // Early return to avoid keep adding number even size > k or n != 0

        // or current number larger than remain target

        if(tmp.size() == k || n == 0 || curNum > n) {

            return;

        }

        for(int i = curNum; i <= 9; i++) {

            tmp.add(i);

            helper(i + 1, k, n - i, result, tmp);

            tmp.remove(tmp.size() - 1);

        }

    }

}

Time Complexity: O(C(9,k))-> O(9^k)

Space Complexity: O(k)

**Refer to**

<https://leetcode.com/problems/combination-sum-iii/solutions/60614/simple-and-clean-java-code-backtracking/>

public List<List<Integer>> combinationSum3(int k, int n) {

List<List<Integer>> ans = new ArrayList<>();

combination(ans, new ArrayList<Integer>(), k, 1, n);

return ans;

}

private static void combination(List<List<Integer>> ans, List<Integer> comb, int k, int start, int n) {

if (comb.size() > k) {

return;

}

if (comb.size() == k && n == 0) {

List<Integer> li = new ArrayList<Integer>(comb);

ans.add(li);

return;

}

for (int i = start; i <= n && i<=9; i++) {

comb.add(i);

combination(ans, comb, k, i+1, n-i);

comb.remove(comb.size() - 1);

}

}

**Style 2: 0/1 knapsack in helper() method (Pick or Not pick)**

class Solution {

    public List<List<Integer>> combinationSum3(int k, int n) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        helper(1, k, n, result, new ArrayList<Integer>());

        return result;

    }

    private void helper(int curNum, int k, int n, List<List<Integer>> result, List<Integer> tmp) {

        if(tmp.size() == k && n == 0) {

            // Because 'tmp' will continue being modified, we have to put a copy of 'tmp' in result instead

            result.add(new ArrayList<>(tmp));

            return;

        }

        // Early return to avoid keep adding number even size > k or n != 0

        // Also add 'curNum > 9' condition as that's the constrain on original

        // for loop style

        if(tmp.size() == k || n == 0 || curNum > 9) {

            return;

        }

        // Pick current number

        tmp.add(curNum);

        helper(curNum + 1, k, n - curNum, result, tmp);

        tmp.remove(tmp.size() - 1);

        // Not pick current number

        helper(curNum + 1, k, n, result, tmp);

    }

}

Time Complexity: O(C(9,k))-> O(9^k), C(9, k), the combination of picking k elements out of 9

The time complexity of this function can be analyzed by looking at the recursive dfs calls.

Each call to dfs potentially makes two further calls, corresponding to including the current

number i or excluding it. We can view this as a binary tree of decisions, where at each step,

we decide whether to include a number in our current combination (t.append(i)) or not (dfs(i + 1, s)).

Each number between 1 and 9 is considered exactly once in the context of a particular combination,

and there are 9 choices at the first call, 8 at the second, and so forth. However, the depth of our

recursion is limited by k (the size of each combination) and the fact that we do not consider

subsequent numbers once we reach a sum greater than n or a combination length equal to k.

Therefore, the time complexity is O(9! / (9 - k)!) in the worst case (when n is large enough to

allow all combinations of k numbers). Note that since k <= 9, this time complexity remains within

a polynomial bound with respect to 9 (the number of different numbers we can pick).

Space Complexity: O(k)

The space complexity is determined by the space needed for the recursive call stack and the space

used to store the combinations. In the worst case, the recursion can go as deep as k, as we stop

further recursion when the length of the temporary list t reaches k. Therefore the recursion call

stack will contribute O(k).

The space for storing all the combinations also needs to be considered. We have a list of lists to

store the valid combinations, and, at most, each combination contains k elements. In the worst case,

the number of combinations stored will also be bounded by the total number of combinations of k numbers

out of 9, which is O(9! / (k!(9 - k)!)) (this is the binomial coefficient representing the number of

ways to choose k distinct integers from a set of 9).

However, because these combinations are part of the output, we often do not count this as extra space

in terms of space complexity analysis (since the space is required to represent the output). Thus the

space complexity is usually considered to be O(k), which accounts only for the recursion depth and not

the output storage.

Combining both the recursion call stack and the output storage, and if we were to include the output

as part of the space complexity, our total space complexity would be O(9! / (k!(9 - k)!) + k).

The complexities provided assume that k and n are fixed and do not grow with the input size since they

are parameters and not part of the input. Therefore, we should clarify that this complexity analysis

is specific to this problem and not generally applicable to variations where k or n might not be bounded

by small constants.

**Refer to**

<https://leetcode.com/problems/combination-sum-iii/solutions/3538062/dfs-recursion/>

The approach is to run DFS starting with value 1. We can either,

include current value and run DFS with reduced target and next value (since repetition is not allowed), OR

skip current value and run DFS with with same target from next value.

class Solution {

public:

void dfs(int curr, int target, int count, vector<vector<int>> &res, vector<int> &v)

{

if (count==0 && target==0){res.push\_back(v);return;}

if (count<0 || target<0 || curr>=10) return;

v.push\_back(curr);

dfs(curr+1, target-curr, count-1, res, v);

v.pop\_back();

dfs(curr+1, target, count, res, v);

}

vector<vector<int>> combinationSum3(int k, int n) {

vector<vector<int>> res;

vector<int>v;

dfs(1,n,k,res,v);

return res;

}

};

**Refer to**

<https://algo.monster/liteproblems/216>

**Problem Description**

The problem requires us to find all unique combinations of k different numbers, which sum up to a target number n, using only numbers from 1 to 9. Each number can be used at most once in any combination. The resulting combinations are to be returned in a list, where the order of combinations does not matter, and no combination is repeated.

For example, if k=3 and n=7, we have to find all unique combinations of three distinct numbers between 1 and 9 that add up to 7. One such valid combination could be [1, 2, 4] because 1+2+4=7.

**Intuition**

To find the solution, we can utilize a Depth First Search (DFS) algorithm. DFS will help us explore all possible combinations of numbers recursively while adhering to the constraints. Here's the intuition behind using DFS for this problem:

We begin by considering numbers from 1 to 9 and use each of them as a starting point of a combination.

To build a combination, we add a number to the current combination (t in the given code) and recursively call the DFS function to add the next number.

While adding the new number, we have three conditions to check:

We must not exceed the target sum n.

We should not use more than k numbers.

We cannot use numbers greater than 9.

If the sum of numbers in the current combination equals n and we have used exactly k numbers, then we found a valid combination which we add to the answer list (ans).

After exploring a number's inclusion, we backtrack by removing the number from the current combination and exploring the possibility of not including that number.

Through this process of including and excluding each number, and backtracking after exploring each possibility, we ensure that all valid combinations are found.

Each combination is built up incrementally from the smaller numbers towards the larger ones to avoid repeated combinations and maintain uniqueness.

The beauty of DFS in this situation is that it inherently avoids duplicates and handles the "each number used at most once" constraint by the recursive nature of its implementation. DFS explores each branch fully (one specific number added vs. not added) before backtracking, which helps cover all potential unique combinations without repetition.

**Solution Approach**

The provided solution uses a **Depth First Search (DFS)** algorithm to explore all possible unique combinations of numbers that add up to n using at most k numbers. Here is a step-by-step breakdown of the approach, referring to specific parts of the implementation:

**DFS Function:** The function dfs(i: int, s: int) is a recursive function where i represents the current number that we are considering adding to the combination, and s is the remaining sum that we need to achieve. The solution initializes the function by calling dfs(1, n).

**Base Conditions:**

The first base condition checks whether the remaining sum s is 0. If s is 0 and we have exactly k numbers in our temporary combination t, then we have found a valid combination. We then append a a copy of t (using t[:] to create a copy) to our answers list ans.

The second base condition checks whether the current number i is greater than 9 or i is greater than the remaining sum s or we have already selected k numbers. In any of these cases, the function returns without making further recursive calls since no valid combination can be completed under these conditions.

**Choosing the Current Number:**

We add the current number i to our temporary combination t. This is the "exploring the possibility of including the number" part of the DFS.

**Recursive Call for Next Number:**

After including the current number, we make a recursive call to dfs(i + 1, s - i). This call will explore combinations where the current number i is part of the solution. By passing i + 1, we are ensuring that the same number is not used more than once. By passing s - i, we are reducing the sum by the value of the number we've included.

**Backtracking:**

After the recursive call, we backtrack by removing the last number that we added — t.pop(). This is where we explore the possibility of excluding the current number.

**Recursive Call without the Current Number:**

Another recursive call dfs(i + 1, s) is then made. This call will explore combinations that exclude the current number i.

**Initialization:**

The function maintains a list ans to collect all valid combinations found by the DFS.

A list t is used to build a temporary combination that is being explored.

**Return Result:**

Once the initial call to dfs(1, n) has completed, all possible combinations have been explored, and the ans list contains all valid combinations as required by the problem. This list is returned as the final result.

Through the combination of recursive DFS, building combinations incrementally, making sure that each number is used at most once, and backtracking after exploring each possibility, the solution efficiently finds all the valid combinations that sum up to n. The use of a temporary list t for tracking the current combination and the answer list ans are examples of data structures used in this problem to store intermediate and final results, respectively.

**Example Walkthrough**

Let's consider a small example to illustrate the solution approach with k=2 and n=5. We need to find all unique pairs of numbers from 1 to 9 that sum up to 5.

**Initialization:** The ans list is initialized, which will store all valid combinations. The temporary combination list t is also initialized and the function dfs(1, 5) is called to begin the search for combinations that sum up to 5.

**First Call to DFS:** dfs(1, 5) represents the state where we are considering whether to include the number 1 in our combination and the remaining sum to reach our target n is 5. At this state, two branches of recursive calls will occur: one where we include the number 1 and another where we don't.

**Including the Number '1':** We include 1 to our temporary list t which is now [1], and the remaining sum s becomes 4. We call dfs(2, 4) to explore next numbers.

**Recursive Call dfs(2, 4):** Now we are evaluating whether to include the number 2. We choose to include 2, so t updates to [1, 2] and call dfs(3, 2) because now the remaining sum s is 2.

**Base Condition Met:** The recursive call dfs(3, 2) will recursively lead to a point where the remaining sum s is 0, and we have exactly 2 numbers in t. When t = [1, 4], s becomes 0. We add [1, 4] to ans, which is one valid combination.

**Backtracking:** After adding [1, 4] to ans, we backtrack by removing 4 from t. Now we have t = [1] again, and we call dfs(3, 5) since we still need to explore combinations starting with 1 without including 4.

**Not Including the Number '1':** We also need to consider combinations that do not include 1. After the first branch is fully explored with 1 in t, we remove 1 to explore the other branch. We call dfs(2, 5) to move on to the next number without including 1.

**Exploring Further Combinations:** The process of including and excluding numbers continues, now starting from 2 and exploring combinations [2, 3], [3, 2], ..., etc., until all valid pairs are found. This results in finding another valid pair [2, 3].

**Return Result:** Once all numbers from 1 to 9 have been considered through recursive calls and backtracking, the search is complete. The ans list which stored all valid combinations ([[1, 4], [2, 3]]) is returned.

At the end of execution, the returned value is [[1, 4], [2, 3]], which includes all unique combinations of 2 different numbers that sum up to 5.

This example walkthrough demonstrates the step-by-step approach taken by the DFS algorithm to explore all possibilities of including or excluding each number, checking for base conditions, backtracking as necessary, and ultimately collecting all the valid combinations into the answer list.

**Java Solution**

class Solution {

    // List to store the final combinations

    private List<List<Integer>> combinations = new ArrayList<>();

    // Temporary list for the current combination

    private List<Integer> currentCombination = new ArrayList<>();

    // The number of numbers to use in each combination

    private int combinationLength;

    // The public method that initiates the combination search

    public List<List<Integer>> combinationSum3(int k, int n) {

        this.combinationLength = k;

        searchCombinations(1, n);

        return combinations;

    }

    // Helper method to perform depth-first search for combinations

    private void searchCombinations(int start, int remainingSum) {

        // If remaining sum is zero and the current combination's size is k

        if (remainingSum == 0) {

            if (currentCombination.size() == combinationLength) {

                // Found a valid combination, add a copy to the result list

                combinations.add(new ArrayList<>(currentCombination));

            }

            return; // Backtrack

        }

        // If the current number exceeds 9, the remaining sum, or if we have enough numbers in the current combination

        if (start > 9 || start > remainingSum || currentCombination.size() >= combinationLength) {

            return; // Cannot find a valid combination from here, backtrack

        }

        // Include 'start' in the current combination

        currentCombination.add(start);

        // Continue to the next number with the updated remaining sum

        searchCombinations(start + 1, remainingSum - start);

        // Exclude 'start' from the current combination (backtrack)

        currentCombination.remove(currentCombination.size() - 1);

        // Continue to the next number without including 'start'

        searchCombinations(start + 1, remainingSum);

    }

}

**Time and Space Complexity**

The given Python code defines a method combinationSum3 that finds all possible combinations of k different numbers that add up to a number n, where each number is from 1 to 9 (inclusive) and each combination is unique.

**Time Complexity**

The time complexity of this function can be analyzed by looking at the recursive dfs calls. Each call to dfs potentially makes two further calls, corresponding to including the current number i or excluding it. We can view this as a binary tree of decisions, where at each step, we decide whether to include a number in our current combination (t.append(i)) or not (dfs(i + 1, s)).

Each number between 1 and 9 is considered exactly once in the context of a particular combination, and there are 9 choices at the first call, 8 at the second, and so forth. However, the depth of our recursion is limited by k (the size of each combination) and the fact that we do not consider subsequent numbers once we reach a sum greater than n or a combination length equal to k.

Therefore, the time complexity is O(9! / (9 - k)!) in the worst case (when n is large enough to allow all combinations of k numbers). Note that since k <= 9, this time complexity remains within a polynomial bound with respect to 9 (the number of different numbers we can pick).

**Space Complexity**

The space complexity is determined by the space needed for the recursive call stack and the space used to store the combinations. In the worst case, the recursion can go as deep as k, as we stop further recursion when the length of the temporary list t reaches k. Therefore the recursion call stack will contribute O(k).

The space for storing all the combinations also needs to be considered. We have a list of lists to store the valid combinations, and, at most, each combination contains k elements. In the worst case, the number of combinations stored will also be bounded by the total number of combinations of k numbers out of 9, which is O(9! / (k!(9 - k)!)) (this is the binomial coefficient representing the number of ways to choose k distinct integers from a set of 9).

However, because these combinations are part of the output, we often do not count this as extra space in terms of space complexity analysis (since the space is required to represent the output). Thus the space complexity is usually considered to be O(k), which accounts only for the recursion depth and not the output storage.

Combining both the recursion call stack and the output storage, and if we were to include the output as part of the space complexity, our total space complexity would be O(9! / (k!(9 - k)!) + k).

The complexities provided assume that k and n are fixed and do not grow with the input size since they are parameters and not part of the input. Therefore, we should clarify that this complexity analysis is specific to this problem and not generally applicable to variations where k or n might not be bounded by small constants.

**Solution 2: Dynamic Progamming (180 min)**

**Besides traditional Backtracking solution below is a special kind of Dynamic Programming, normally its hard to build pure DFS / DP based on Backtracking.**

class Solution {

    public List<List<Integer>> combinationSum3(int k, int n) {

        List<List<List<Integer>>> dp = new ArrayList<>(n + 1);

        for (int i = 0; i <= n; i++) {

            dp.add(new ArrayList<>());

        }

        dp.get(0).add(new ArrayList<>());

        for (int i = 1; i <= 9; i++) {

            for (int j = n; j >= i; j--) {

                for (List<Integer> v : dp.get(j - i)) {

                    List<Integer> newCombination = new ArrayList<>(v);

                    newCombination.add(i);

                    dp.get(j).add(newCombination);

                }

            }

        }

        List<List<Integer>> res = new ArrayList<>();

        for (List<Integer> t : dp.get(n)) {

            if (t.size() == k) {

                res.add(t);

            }

        }

        return res;

    }

}

**Refer to**

<https://leetcode.com/problems/combination-sum-iii/solutions/2025671/2-solution-using-backtracking-dp-c/>

**Similar to 0-1 knapsack o(n) optimizattion**

class Solution {

public:

vector<vector<int>> combinationSum3(int k, int n) {

vector<vector<vector<int>>> dp(n+1);

dp[0] = {{}};

for(int i=1;i<=9;i++)

{

for(int j = n ; j >=i ; j--)

{

for(auto v : dp[j - i])

{

v.push\_back(i);

dp[j].push\_back(v);

}

}

}

vector<vector<int>>res;

for(auto t: dp[n])

{

if(t.size()==k)res.push\_back(t);

}

return res;

}

};

Divide and Conquer is a way to replace the backtracking ? Similar to L903's Divide and Conquer style (<https://leetcode.com/problems/valid-permutations-for-di-sequence/solutions/168612/top-down-with-memo-bottom-up-dp-n-3-dp-n-2-dp-o-n-space/>) ?

<https://leetcode.com/problems/combination-sum-iii/solutions/60614/Simple-and-clean-Java-code-backtracking/comments/1142235>

**How about if we change condition "Each number is used at most once" to "No limitation for each number" ?**

**Refer to Deepseek**

To modify the solution to allow each number to be used multiple times (no limitation), we need to adjust the backtracking approach to permit reuse of the same number. Here’s the revised solution and explanation:

**Modified Solution (Numbers Can Be Reused)**

class Solution {

public List<List<Integer>> combinationSum3(int k, int n) {

List<List<Integer>> result = new ArrayList<>();

backtrack(1, k, n, new ArrayList<>(), result);

return result;

}

private void backtrack(int start, int k, int remaining, List<Integer> path, List<List<Integer>> result) {

if (path.size() == k && remaining == 0) {

result.add(new ArrayList<>(path));

return;

}

if (path.size() >= k || remaining <= 0) {

return;

}

for (int i = start; i <= 9; i++) {

path.add(i);

backtrack(i, k, remaining - i, path, result); // Allow reuse: pass `i` instead of `i + 1`

path.remove(path.size() - 1);

}

}

}

**Key Changes:**

Reuse of Numbers:

In the recursive call backtrack(i, ...), we pass i (not i + 1), allowing the current number i to be reused in subsequent steps.

Avoiding Duplicates:

By starting the loop at start and incrementing i sequentially, combinations are generated in \*\*non-decreasing order\*\*, ensuring no duplicates (e.g., [1, 2, 3] is valid, but [2, 1, 3] is not generated).

Termination Conditions:

Valid Combination: If path.size() == k and remaining == 0, add to the result.

Early Termination: If path.size() >= k or remaining <= 0, backtrack.

**Example Walkthrough:**

Input: k = 3, n = 6

Valid Combinations:

[1, 1, 4], [1, 2, 3], [2, 2, 2]

Execution:

Start with i = 1:

Add 1, recurse with i = 1, remaining = 5.

Add 1 again, recurse with i = 1, remaining = 4.

Add 1 again, recurse with i = 1, remaining = 3 → Invalid (sum ≠ 6).

Backtrack, add 4 → [1, 1, 4] (valid).

Continue with i = 2:

Add 2, recurse with i = 2, remaining = 4.

Add 2 again, recurse with i = 2, remaining = 2.

Add 2 again → [2, 2, 2] (valid).

Continue with i = 3:

Add 3, recurse with i = 3, remaining = 3.

Add 3 again → [3, 3] (size < 3, invalid).

**Why This Works:**

Reuse Logic: By passing i instead of i + 1, the same number can be reused.

Ordered Combinations: The loop starts at start, ensuring combinations are generated in ascending order, avoiding duplicates.

**Comparison with Original Solution:**

|  |  |  |
| --- | --- | --- |
| Feature | Original (No Reuse) | Modified (Reuse Allowed) |
| **Recursive Call** | backtrack(i + 1, ...) | backtrack(i, ...) |
| **Combinations** | Unique numbers only (e.g., [1, 2, 3]). | Allows duplicates (e.g., [1, 1, 4]). |
| **Time Complexity** | (O(9! / (k!(9 - k)!))) | (O(9^k)) (exponential, but optimized). |

**Conclusion:**

The modified solution efficiently handles the "no limitation" condition by allowing number reuse while maintaining non-decreasing order to avoid duplicates. The key change is passing i instead of i + 1 in the recursive call.

**Refer to**

[DFS Backtracking and Dynamic Programming](note://WEBb2267f049a7493bf71622942b3d5a64d)

[L39.Combination Sum (Ref.L40,L322)](note://79A77D14319840B9A134C272EA618600)

[L40.Combination Sum II (Ref.L39,L90,L377)](note://7895D4AD64EC43CA903579F3C8F803AE)

[L322.Coin Change (Ref.L39,L46,L377)](note://WEB99a663d0191c2bc0c88aec7cf5f3d9fd)

[L377.P16.4.Combination Sum IV(Backpack VI) (Ref.L39,L40,L322)](note://8000F8241A0B4F3AA39E5CA16B6A7BB8)

[L518.Coin Change II (Ref.L377)](note://WEB9160d308503a5b8c326712331320ec56)