https://leetcode.com/problems/combinations/

Given two integers n and k, return *all possible combinations of* k *numbers chosen from the range* [1, n].

You may return the answer in **any order**.

Example 1:

Input: n = 4, k = 2

Output: [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]

Explanation: There are 4 choose 2 = 6 total combinations.

Note that combinations are unordered, i.e., [1,2] and [2,1] are considered to be the same combination.

Example 2:

Input: n = 1, k = 1

Output: [[1]]

Explanation: There is 1 choose 1 = 1 total combination.

Constraints:

- 1 <= n <= 20
- 1 <= k <= n

Attempt 1: 2022-10-17

Solution 1: Backtracking style 1 (10min, initialize with combinations)

```
class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        int[] candidates = new int[n + 1];
        for(int i = 1; i <= n; i++) {
            candidates[i] = i;
        }
        // Since range [1,n], start index not 0 but 1
        helper(candidates, result, new ArrayList<Integer>(), k, 1);
        return result;
    }

```

```
    private void helper(int[] candidates, List<List<Integer>> result, List<Integer> tmp, int k, int index) {
        if(tmp.size() == k) {
            result.add(new ArrayList<Integer>(tmp));
            return;
        }
        for(int i = index; i < candidates.length; i++) {
            tmp.add(candidates[i]);
            helper(candidates, result, tmp, k, i + 1);
            tmp.remove(tmp.size() - 1);
        }
    }
}
```

Time Complexity: O(nlogn + 2^n) ~= O(2^n) where n is size of candidates array
But according to
https://leetcode.com/problems/combinations/discuss/395558/Time-complexity-analysis-of-Backtracking-Java
O(2^n) is not tight, O(n!) is not tight, answer is O(n!/(k-1)!)
The simplest way to analysis is that : every round of for loop it will add one and only one number for sure, so how many numbers means how many round of loops, there are k*C(n,k) numbers in the output
so it is O(k*C(n,k)) which is O(n!/(k-1)!)
O(n!/(k-1)!) is not O(n!) because k is not some constant but a input that has impact on time complexity
Space Complexity: O(length_of_longest_combination)

Solution 2: Backtracking style 2 (10min, initialize without combinations, directly use 'n')

```
class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        // Since range [1,n], start index not 0 but 1
        helper(n, result, new ArrayList<Integer>(), k, 1);
        return result;
    }

```

```
    private void helper(int n, List<List<Integer>> result, List<Integer> tmp, int k, int index) {
        if(tmp.size() == k) {
            result.add(new ArrayList<Integer>(tmp));
            return;
        }
        for(int i = index; i <= n; i++) {
            tmp.add(i);
            helper(n, result, tmp, k, i + 1);
            tmp.remove(tmp.size() - 1);
        }
    }
}
```

For Backtracking style 1 & 2 Tree Structure Analysis

```
Tree Structure Analysis
e.g.
Input: n = 3, k = 3
Output: [[1,2,3]]

                          { }            index=0
                    /      |      \
                  {1}     {2}    {3}     index=1
                 /   \     |
               {1,2} {1,3} {2,3}          index=2
               /
            {1,2,3}                       index=3
```

Refer to

https://medium.com/algorithms-and-leetcode/backtracking-with-leetcode-problems-part-2-705c9cc70e52

The traversal and backtrack process is below

Solution 3: Backtracking style 3 (10min, "Not pick" or "Pick" branch)

```java
class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        // Since range [1,n], start index not 0 but 1
        helper(n, result, new ArrayList<Integer>(), k, 1);
        return result;
    }

    private void helper(int n, List<List<Integer>> result, List<Integer> tmp, int k, int index) {
        if(tmp.size() == k) {
            result.add(new ArrayList<Integer>(tmp));
            return;
        }
        // Based on tree analysis, add return condition when index > n
        if(index > n) {
            return;
        }
        // Not pick
        helper(n, result, tmp, k, index + 1);
        // Pick
        tmp.add(index);
        helper(n, result, tmp, k, index + 1);
        tmp.remove(tmp.size() - 1);
    }
}
```

For Backtracking style 3 Tree Structure Analysis

```
Tree Structure Analysis
e.g.
Input: n = 4, k = 2
Output: [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
                                    {}
                      /                          \
```

```
                    {}                          {1}         index=1
                /       \                   /       \
               {}       {2}              {1}       {1,2}   index=2
             /    \     /   \            /   \
           {}     {3}  {2}  {2,3}      {1}  {1,3}          index=3
          /  \    /  \   /  \           /   \
        {}   {4} {3}{3,4} {2}{2,4}     {1} {1,4}           index=4
-----------------------------------------------------------------
In leaf nodes, tmp={},{4},{3},{2},{1} should discard
When index > 4 means even 'tmp' size not equal to 2 have to discard, add return condition index > n
```

Time complexity analysis of Backtracking Java

https://leetcode.com/problems/combinations/discuss/395558/Time-complexity-analysis-of-Backtracking-Java

Here is the code, classic DFS backtracking.

```java
public List<List<Integer>> combine(int n, int k) {
        List<Integer> curr=new ArrayList<Integer>();
        List<List<Integer>> ans =new ArrayList<List<Integer>>();
        dfs(n,k,1,curr,ans);
        return ans;
    }

    private void dfs(int n, int k,int next,List<Integer> curr, List<List<Integer>> ans){
        if(curr.size()==k){
            ans.add(new ArrayList(curr));
            return;
        }
        for(int i=next;i<=n;i++){
            curr.add(i);
            dfs(n,k,i+1,curr,ans);
            curr.remove(curr.size()-1);
        }
    }
```

There are a lot of different answers for time complexity. Finally I think I got it right and clear, please let me know if you find I got anything wrong.

First of all, O(2^n) is not tight, O(n!) is not tight. My answer is O(n!/(k-1)!)

The simplest way to analysis is that : every round of for loop it will add one and only one number for sure, so how many numbers means how many round of loops, there are k*C(n, k) numbers in the output

so it is O(k*C(n, k)) which is O(n!/(k-1)!)

O(n!/(k-1)!) is not O(n!) because k is not some constant but a input that has impact on time complexity

Also, I can prove it another way, more academic way:

Prove: T(n) = n!/(k-1)!

from the DFS we can simply see that

T(n) = C1 + n [(T(n-1) + C2]

= nT(n-1) + C2*n + C1

when you see T(n) = nT(n-1)..., it usually means n!

because

T(0) = 1

T(n) = nT(n-1)

= n*[(n-1)*T(n-2)]

= (n)*(n-1)*(n-2)*(n-3).....T(0)

= n!

But, here comes the core part of this analysis.

```
if(curr.size()==k){
            ans.add(new ArrayList(curr));
            return;
        }
```

because of this part of code, which is the bounding condition of backtracking, **the recursive call will only reach kth level of call,** so

T(n) = nT(n-1)

= n*[(n-1)*T(n-2)]

= (n)*(n-1)*(n-2)*(n-3).....(n-k)*T(n-k-1)when recursive reach the kth level and try to do T(n-k-1)

curr.size()==k is true, because every level curr add one numberT(n-k-1) will simply

ans.add(new ArrayList(curr)) and returns which means T(n-k-1) = 1

so

T(n) = (n)*(n-1)*(n-2)*(n-3).....(n-k) * 1

= n!/(k-1)!

= k * n!/k!

= k * C(n,k)

compare with n!/(k-1)! , both 2^n and n! are not tight.

Video explain time complexity for combinations

Combinations - Leetcode 77 - Python **[time: 4:30]**

https://www.youtube.com/watch?v=q0s6m7AiM7o

根据Backtracking模式下的DFS没法直接转变成DP的情况，我们采用Divide and Conquer技术下的DFS来做DP演化

Refer to

DFS Backtracking and Dynamic Programming

Solution 4: Divide and Conquer (360 min)

参考这里基于这个公式 **C (n, k) = C (n - 1, k - 1) + C (n - 1, k)** 所用的思想。

从 n 个数字选 k 个，我们把所有结果分为两种，包含第 n 个数和不包含第 n 个数。这样的话，就可以把问题转换成

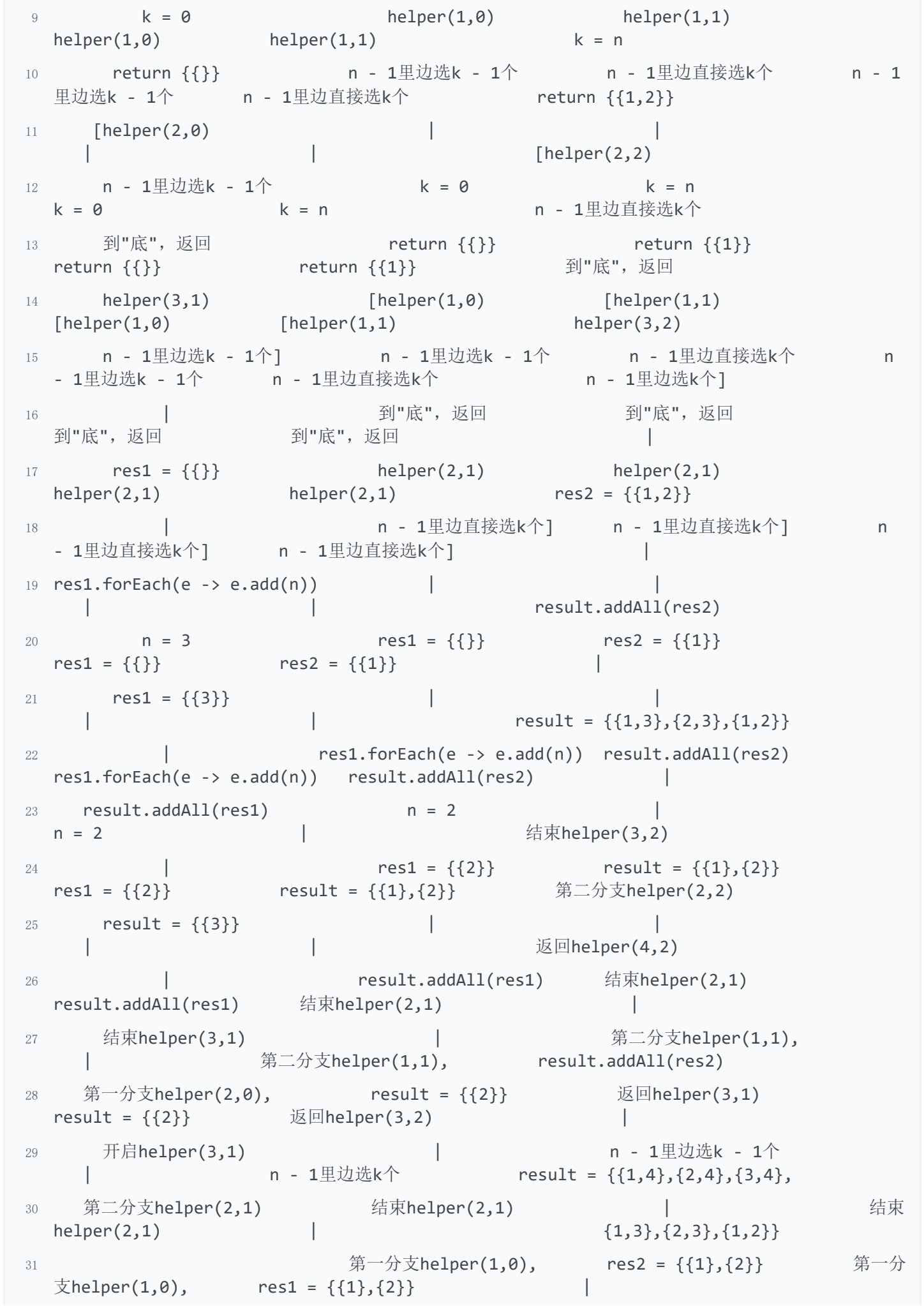
- 从 n - 1 里边选 k - 1 个，然后每个结果加上 n
- 从 n - 1 个里边直接选 k 个。

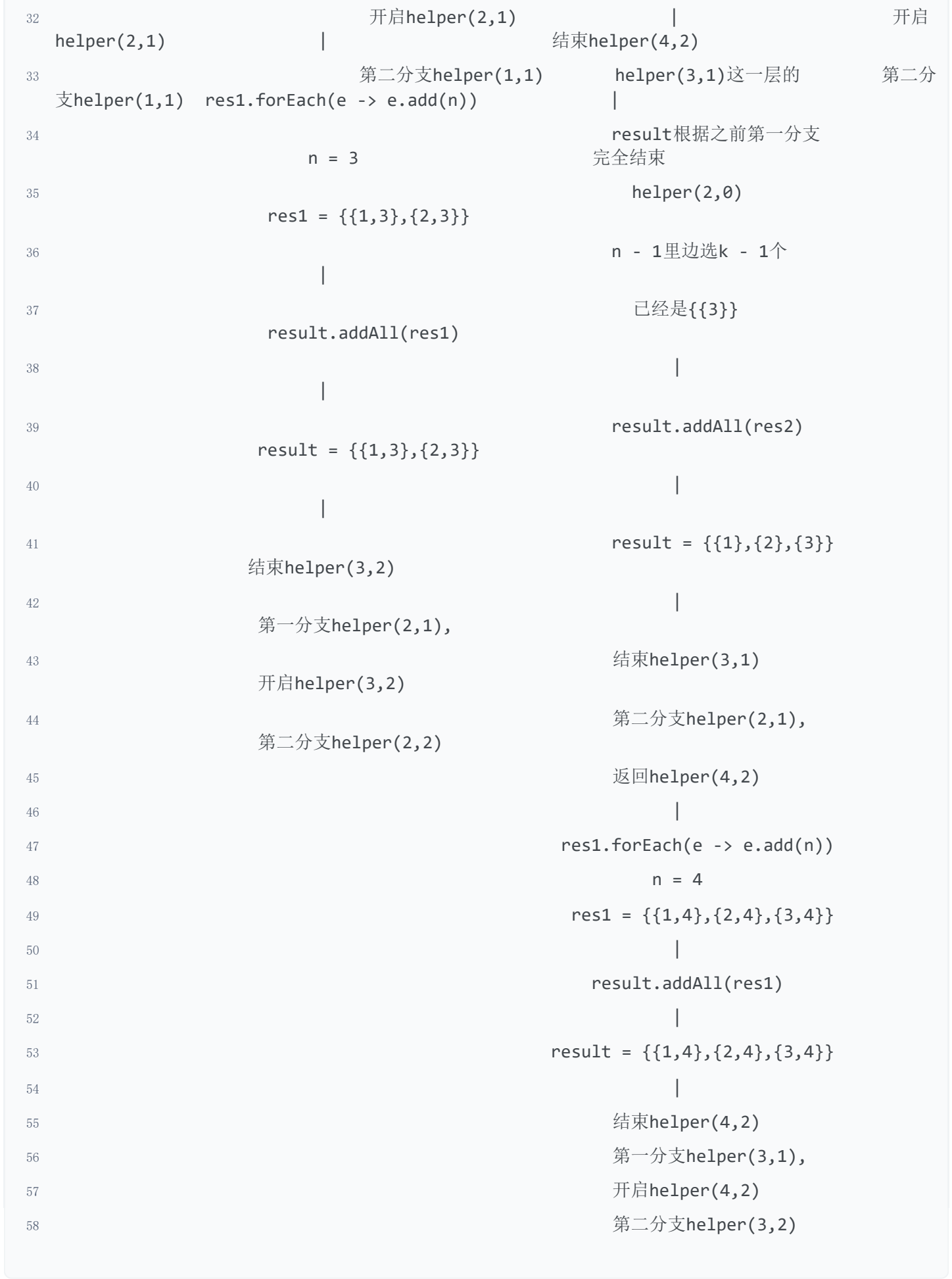
```
class Solution {
    public List<List<Integer>> combine(int n, int k) {
        return helper(n, k);
    }

    // 基于C(n, k) = C(n - 1, k - 1) + C(n - 1, k)构建两种分支
    private List<List<Integer>> helper(int n, int k) {
        // Base condition 1: helper(n - 1, k - 1)的底
        if(k == 0) {
            return new ArrayList<>(Arrays.asList(new ArrayList<Integer>()));
        }
        // Base condition 2: helper(n - 1, k)的底
        if(k == n) {
```

```
            List<Integer> row = new ArrayList<>();
            for (int i = 1; i <= k; ++i) {
                row.add(i);
            }
            return new ArrayList<>(Arrays.asList(row));
        }
        // Divide and Conquer
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        // 选择1: n - 1里边选k - 1个
        List<List<Integer>> res1 = helper(n - 1, k - 1);
        // 每个结果加上n
        res1.forEach(e -> e.add(n));
        // 把n - 1个选k - 1个的结果加入
        result.addAll(res1);
        // 选择2: n - 1里边直接选k个
        List<List<Integer>> res2 = helper(n - 1, k);
        // 把n - 1个选k个的结果也加入
        result.addAll(res2);
        return result;
    }
}
```

需要展开来看全部流程


k = 0 helper(1,0) helper(1,1) helper(1,0) helper(1,1) k = n
return {{}} n - 1里边选k - 1个 n - 1里边直接选k个 n - 1里边选k - 1个 n - 1里边直接选k个 return {{1,2}}
[helper(2,0) [helper(2,2)
n - 1里边选k - 1个 k = 0 k = n k = 0 k = n n - 1里边直接选k个
到"底"，返回 return {{}} return {{1}} return {{}} return {{1}} 到"底"，返回
helper(3,1) [helper(1,0) [helper(1,1) [helper(1,0) [helper(1,1) helper(3,2)
n - 1里边选k - 1个] n - 1里边选k - 1个 n - 1里边直接选k个 n - 1里边选k - 1个 n - 1里边直接选k个 n - 1里边选k个]
到"底"，返回 到"底"，返回 到"底"，返回 到"底"，返回
res1 = {{}} helper(2,1) helper(2,1) helper(2,1) helper(2,1) res2 = {{1,2}}
n - 1里边直接选k个] n - 1里边直接选k个] n - 1里边直接选k个] n - 1里边直接选k个]
res1.forEach(e -> e.add(n)) result.addAll(res2)
n = 3 res1 = {{}} res2 = {{1}} res1 = {{}} res2 = {{1}}
res1 = {{3}} result = {{1,3},{2,3},{1,2}}
res1.forEach(e -> e.add(n)) result.addAll(res2) res1.forEach(e -> e.add(n)) result.addAll(res2)
result.addAll(res1) n = 2 n = 2 结束helper(3,2)
res1 = {{2}} result = {{1},{2}} res1 = {{2}} result = {{1},{2}} 第二分支helper(2,2)
result = {{3}} 返回helper(4,2)
result.addAll(res1) 结束helper(2,1) result.addAll(res1) 结束helper(2,1)
结束helper(3,1) 第二分支helper(1,1), 第二分支helper(1,1), result.addAll(res2)
第一分支helper(2,0), result = {{2}} 返回helper(3,1) result = {{2}} 返回helper(3,2)
开启helper(3,1) n - 1里边选k - 1个 n - 1里边选k个 result = {{1,4},{2,4},{3,4},
第二分支helper(2,1) 结束helper(2,1) 结束helper(2,1) {1,3},{2,3},{1,2}}
第一分支helper(1,0), res2 = {{1},{2}} 第一分支helper(1,0), res1 = {{1},{2}}



Solution 5: Divide and Conquer with Memoization (60 min)

参考这里基于这个公式 **C (n, k) = C (n - 1, k - 1) + C (n - 1, k)** 所用的思想，这个思想之前刷题也用过，但忘记是哪道了。

从 n 个数字选 k 个，我们把所有结果分为两种，包含第 n 个数和不包含第 n 个数。这样的话，就可以把问题转换成

- 从 n - 1 里边选 k - 1 个，然后每个结果加上 n
- 从 n - 1 个里边直接选 k 个。

关于非基础类型的memo元素如何使用deep copy来避免元素状态改变

Note: Be careful with the deep copy utilization when memo stores object, e.g memo[i][j] is List<List<Integer>> (when memo stores primitive no need deep copy, e.g memo[i][j] is int)

```
class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>>[][] memo = new ArrayList[n + 1][k + 1];
        return helper(n, k, memo);
    }

    // 基于C(n, k) = C(n - 1, k - 1) + C(n - 1, k)构建两种分支
    private List<List<Integer>> helper(int n, int k, List<List<Integer>>[][] memo) {
        // Base condition 1: helper(n - 1, k - 1)的底
        if(k == 0) {
            return new ArrayList<>(Arrays.asList(new ArrayList<Integer>()));
        }
        // Base condition 2: helper(n - 1, k)的底
        if(k == n) {
            List<Integer> row = new ArrayList<>();
            for (int i = 1; i <= k; ++i) {
                row.add(i);
            }
            return new ArrayList<>(Arrays.asList(row));
        }
        // For memo must use deep copy since type of memo[n][k] is
        // "List<Integer>" as object, the object stored into memo[n][k]
        // should not change during further recursion, so original object's
        // reference should not return back, must do deep copy to return
        // a new reference for further recursion
        if(memo[n][k] != null) {
            List<List<Integer>> copy = new ArrayList<>();
            for(List<Integer> sublist : memo[n][k]) {
```

```
                copy.add(new ArrayList<>(sublist));
            }
            return copy;
        }
        // Divide and Conquer
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        // 选择1: n - 1里边选k - 1个
        List<List<Integer>> res1 = helper(n - 1, k - 1, memo);
        // 每个结果加上n
        res1.forEach(e -> e.add(n));
        // 把n - 1个选k - 1个的结果加入
        result.addAll(res1);
        // 选择2: n - 1里边直接选k个
        List<List<Integer>> res2 = helper(n - 1, k, memo);
        // 把n - 1个选k个的结果也加入
        result.addAll(res2);
        // For memo must use deep copy again to store current 'result'
        // status into memo[n][k], a deep copy will create a new reference
        // of current 'result' object, hence when 'result' object update
        // during further recursion with original reference, that won't
        // impact object stored at memo[n][k]
        memo[n][k] = new ArrayList<>();
        for(List<Integer> sublist : result) {
            memo[n][k].add(new ArrayList<>(sublist));
        }
        return result;
    }
}
```

Solution 6: 2D DP (60 min)

```
class Solution {
    /**
        e.g n = 4, k = 2, 1 <= k <= n

          Start with
          k   0    1    2    -> j
```

```
        n
        0    {{}}  X    X
        1    {{}}{{1}}  X
        2    {{}}    {{1,2}}
        3    {{}}
        4    {{}}       ?
     -> i
          End with
          k   0             1            2    -> j
        n
        0    {{}}           X            X
        1    {{}}         {{1}}          X
        2    {{}}       {{1},{2}}     {{1,2}}
        3    {{}}     {{1},{2},{3}}   {{1,2},{1,3},{2,3}}
        4    {{}}   {{1},{2},{3},{4}} {{1,2},{1,3},{2,3},{1,4},{2,4},{3,4}} -> dp[4][2]
     -> i
     */
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>>[][] dp = new ArrayList[n + 1][k + 1];
        for(int i = 0; i <= n; i++) {
            dp[i][0] = new ArrayList<>();
            dp[i][0].add(new ArrayList<Integer>());
        }
        for(int i = 1; i <= n; i++) {
            for(int j = 1; j <= i && j <= k; j++) {
                dp[i][j] = new ArrayList<>();
                // 判断是否可以从i - 1里边选j个
                if(i - 1 >= j) {
                    dp[i][j].addAll(dp[i - 1][j]);
                }
                // 不再额外需要判断是否可以从i - 1里边选j - 1个，因为循环条件里面包含j <= i
                for(List<Integer> list : dp[i - 1][j - 1]) {
                    List<Integer> tmp = new ArrayList<>(list);
                    // 每个结果加上n(这里替换为i)
                    tmp.add(i);
                    dp[i][j].add(tmp);
                }
            }
        }
        return dp[n][k];
```

```
    }
}
```

Step by step derive from recursion to DP

```
Initial status
e.g n = 4, k = 2, 1 <= k <= n
      k   0    1    2    -> j
    n
    0    {{}}  X    X
    1    {{}}{{1}}  X
    2    {{}}    {{1,2}}
    3    {{}}
    4    {{}}       ?
 -> i
Note: 'X' means not avaialble since to match condition 1 <= k <= n
dp[n][k] = ?

dp[0][0] must be {{}} even it satisfy both k = 0 and k = n, but if we see two base conditions
in Divide and Conquer recursion, we will see for k = 0, dp[0][0] will be {{}}, for k = n, dp[0][0]
will also be {{}} since loop of i to fill the row start from i = 1 till i <= k(=0), which means
no element will be added

// Base condition 1: helper(n - 1, k - 1)的底
if(k == 0) {
    return new ArrayList<>(Arrays.asList(new ArrayList<Integer>()));
}
// Base condition 2: helper(n - 1, k)的底
if(k == n) {
    List<Integer> row = new ArrayList<>();
    for(int i = 1; i <= k; ++i) {
       row.add(i);
    }
    return new ArrayList<>(Arrays.asList(row));
}
```

```
=>
for(int i = 0; i <= n; i++) {
    dp[i][0] = new ArrayList<Integer>();
    dp[i][0].add(new ArrayList<Integer>());
}

And for how to create the formula, it also corresponding to the recursion relationship
in Divide and Conquer recursion, basically we have below 2 recursion calls:
(1) 选择1: n - 1里边选k - 1个
    List<List<Integer>> res1 = helper(n - 1, k - 1)
    每个结果加上n
    res1.forEach(e -> e.add(n));
(2) 选择2: n - 1里边直接选k个
    List<List<Integer>> res2 = helper(n - 1, k)
=>
when create formula we just replace 'n' to 'i' and 'k' to 'j'
dp[i][j] = (dp[i - 1][j - 1] and plus element {i}) + (dp[i - 1][j])

Start with
      k   0    1    2    -> j
    n
    0    {{}}  X    X
    1    {{}}{{1}}  X
    2    {{}}    {{1,2}}
    3    {{}}
    4    {{}}       ?
 -> i

End with
      k   0             1            2    -> j
    n
    0    {{}}           X            X
    1    {{}}         {{1}}          X
    2    {{}}       {{1},{2}}     {{1,2}}
    3    {{}}     {{1},{2},{3}}   {{1,2},{1,3},{2,3}}
    4    {{}}   {{1},{2},{3},{4}} {{1,2},{1,3},{2,3},{1,4},{2,4},{3,4}} -> dp[4][2]
 -> i

for(int i = 1; i <= n; i++) {
    for(int j = 1; j <= i && j <= k; j++) {
```

```
        // 判断是否可以从i - 1里边选j个
        if(i - 1 >= j) {
            dp[i][j].addAll(dp[i - 1][j])
        }
        // 不再额外需要判断是否可以从i - 1里边选j - 1个，因为循环条件里面包含j <= i
        for(List<Integer> list : dp[i - 1][j - 1]) {
            List<Integer> tmp = new ArrayList<>(list);
            // 每个结果加上n(这里替换为i)
            tmp.add(i);
            dp[i][j].add(tmp);
        }
    }
}
```

Solution 7: 2 rows array DP (10 min, refer to L72.Edit Distance)

```
class Solution {
    public List<List<Integer>> combine(int n, int k) {
        //List<List<Integer>>[][] dp = new ArrayList[n + 1][k + 1];
        List<List<Integer>>[] dpPrev = new ArrayList[k + 1];
        List<List<Integer>>[] dp = new ArrayList[k + 1];
        //for(int i = 0; i <= n; i++) {
        //    dp[i][0] = new ArrayList<>();
        //    dp[i][0].add(new ArrayList<Integer>());
        //}
        // 2 rows array方法的初始化必须从dpPrev开始，而非dp，dp的初始化最好
        // 在外层row维度循环中进行，以应对每一层初始化数值不一定相同的情况，参见L72
        dpPrev[0] = new ArrayList<>();
        dpPrev[0].add(new ArrayList<Integer>());
        for(int i = 1; i <= n; i++) {
            dp[0] = new ArrayList<>();
            dp[0].add(new ArrayList<Integer>());
            for(int j = 1; j <= i && j <= k; j++) {
                //dp[i][j] = new ArrayList<>();
                dp[j] = new ArrayList<>();
                // 判断是否可以从i - 1里边选j个
                if(i - 1 >= j) {
```

```
                    //dp[i][j].addAll(dp[i - 1][j]);
                    dp[j].addAll(dpPrev[j]);
                }
                // 不再额外需要判断是否可以从i - 1里边选j - 1个，因为循环条件里面包含j <= i
                //for(List<Integer> list : dp[i - 1][j - 1]) {
                //    List<Integer> tmp = new ArrayList<>(list);
                //    // 每个结果加上n(这里替换为i)
                //    tmp.add(i);
                //    dp[i][j].add(tmp);
                //}
                for(List<Integer> list : dpPrev[j - 1]) {
                    List<Integer> tmp = new ArrayList<>(list);
                    // 每个结果加上n(这里替换为i)
                    tmp.add(i);
                    dp[j].add(tmp);
                }
            }
            dpPrev = dp.clone();
        }
        //return dp[n][k];
        return dpPrev[k];
    }
}
```

Solution 8: 1 row array DP (60 min, refer to L72.Edit Distance)

进一步优化为1 row (不是真正的1 row方案，内层循环不需要反转，因为只是用2个变量替代了2 rows中的1 row)

```
class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>>[] dp = new ArrayList[k + 1];
        // i 从 1 到 n
        dp[0] = new ArrayList<>();
        dp[0].add(new ArrayList<Integer>());
        for (int i = 1; i <= n; i++) {
            // j 从 1 到 i 或者 k
            List<List<Integer>> temp = new ArrayList<>(dp[0]);
```

```
            for (int j = 1; j <= i && j <= k; j++) {
                List<List<Integer>> last = temp;
                if(dp[j]!=null){
                    temp = new ArrayList<>(dp[j]);
                }
                // 判断是否可以从 i - 1 里边选 j 个
                if (i <= j) {
                    dp[j] = new ArrayList<>();
                }
                // 把 i - 1 里边选 j - 1 个的每个结果加上 i
                for (List<Integer> list : last) {
                    List<Integer> tmpList = new ArrayList<>(list);
                    tmpList.add(i);
                    dp[j].add(tmpList);
                }
            }
        }
        return dp[k];
    }
}
```

L77. Combinations 和 L46. Permutations & L47. Permutations II应用回溯法最大的区别就是L77真的需要在每一层递归的传递的时候都需要向前（从左向右视为前）移动坐标一位，因为是求Combinations，不能折返回已经用过的坐标的数（包括当前坐标），所有的坐标和坐标所代表的数值只能最多使用一次或者不用，但L46，L47不一样，因为是Permutations，所以可以折返回已经用过的所有坐标再次选取，所以每一层递归的时候传递进去的坐标都必须从0开始，重新从头开始选取数值，除了当前选取集合中已经包含的坐标之外的所有坐标都可以再次选取（例如，从{1,2,3,4}中选2个，在当前递归过程中在到达"底"之前已经形成的集合已经包含了{2,3}，除了数值4可选之外，还可以回头选择坐标为0的数值1，但是坐标位于1,2的数值2,3都不能再选了，因为已经在当前集合中出现过了）

Refer to

https://leetcode.wang/leetCode-77-Combinations.html

解法一 回溯法

这种选数字很经典的回溯法问题了，先选一个数字，然后进入递归继续选，满足条件后加到结果中，然后回溯到上一步，继续递归。直接看代码吧，很好理解。

```java
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>> ans = new ArrayList<>();
    getAns(1, n, k, new ArrayList<Integer>(), ans);
    return ans;
}
private void getAns(int start, int n, int k, ArrayList<Integer>
temp,List<List<Integer>> ans) {
    //如果 temp 里的数字够了 k 个，就把它加入到结果中
    if(temp.size() == k){
        ans.add(new ArrayList<Integer>(temp));
        return;
    }
    //从 start 到 n
    for (int i = start; i <= n; i++) {
        //将当前数字加入 temp
        temp.add(i);
        //进入递归
        getAns(i+1, n, k, temp, ans);
        //将当前数字删除，进入下次 for 循环
        temp.remove(temp.size() - 1);
    }
}
```

一个 for 循环，添加，递归，删除，很经典的回溯框架了。在这里发现了一个优化方法。for 循环里 i 从 start 到 n，其实没必要到 n。比如，n = 5，k = 4，temp.size( ) == 1，此时代表我们还需要（4 - 1 = 3）个数字，如果 i = 4 的话（这里i 的起点是i = 1，不是普通的数组从i = 0开始循环（基于坐标index的体系），因为本题中i的范围是基于数值本身的[1, n]，在n = 5的设定下，i = 4意味着在{1, 2, 3, 4, 5}中剩下{4, 5}可选），以后最多把 4 和 5 加入到 temp 中，而此时 temp.size() 才等于 1 + 2 = 3，不够 4 个，所以 i 没必要等于 4，i 循环到 3 就足够了。

所以 for 循环的结束条件可以改成， i <= n - (k - temp.size ( )) + 1，k - temp.size ( ) 代表我们还需要的数字个数。因为我们最后取到了 n，所以还要加 1。

```java
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>> ans = new ArrayList<>();
    getAns(1,n, k, new ArrayList<Integer>(), ans);
    return ans;
```

```
}
private void getAns(int start, int n, int k, ArrayList<Integer> temp,
List<List<Integer>> ans) {
    if(temp.size() == k){
        ans.add(new ArrayList<Integer>(temp));
        return;
    }
    for (int i = start; i <= n - (k - temp.size()) + 1; i++) {
        temp.add(i);
        getAns(i+1, n, k, temp, ans);
        temp.remove(temp.size() - 1);
    }
}
```

虽然只改了一句代码，速度却快了很多。

解法二 迭代

参考这里，完全按照解法一回溯的思想改成迭代。我们思考一下，回溯其实有三个过程。

- for 循环结束，也就是 i == n + 1，然后回到上一层的 for 循环
- temp.size() == k，也就是所需要的数字够了，然后把它加入到结果中。
- 每个 for 循环里边，进入递归，添加下一个数字

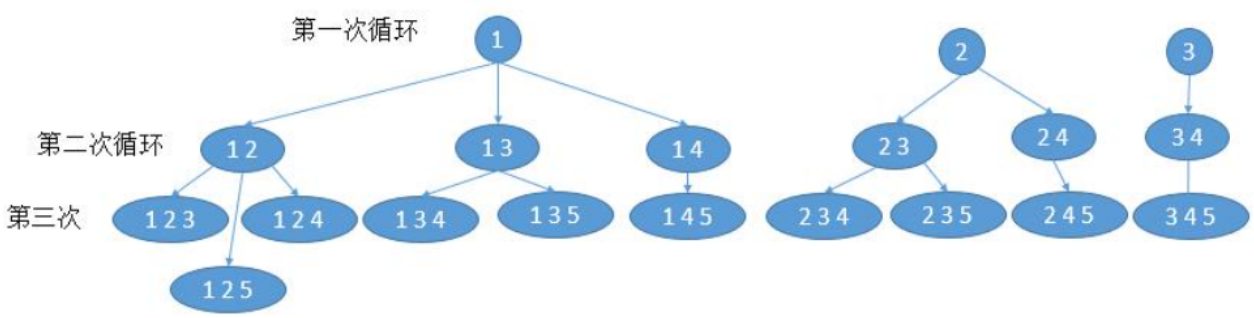
```
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>> ans = new ArrayList<>();
    List<Integer> temp = new ArrayList<>();
    for(int i = 0;i<k;i++){
        temp.add(0);
    }
    int i = 0;
    while (i >= 0) {
        temp.set(i, temp.get(i)+ 1); //当前数字加 1
        //当前数字大于 n，对应回溯法的 i == n + 1，然后回到上一层
        if (temp.get(i) > n) {
            i--;
        // 当前数字个数够了
        } else if (i == k - 1) {
            ans.add(new ArrayList<>(temp));
        //进入更新下一个数字
```

```
        }else {
            i++;
            //把下一个数字置为上一个数字，类似于回溯法中的 start
            temp.set(i, temp.get(i-1));
        }
    }
    return ans;
}
```

解法三 迭代法2

解法二的迭代法是基于回溯的思想，还有一种思想，参考这里。类似于46题的解法一，找 k 个数，我们可以先找出 1 个的所有结果，然后在 1 个的所有结果再添加 1 个数，变成 2 个，然后依次迭代，直到有 k 个数。

比如 n = 5， k = 3



第 1 次循环，我们找出所有 1 个数的可能 [1]，[2]，[3]。4 和 5 不可能，解法一分析过了，因为总共需要 3 个数，4，5 全加上才 2 个数。
第 2 次循环，在每个 list 添加 1 个数， [1] 扩展为 [1 , 2]，[1 , 3]，[1 , 4]。[1 , 5] 不可能，因为 5 后边没有数字了。 [2] 扩展为 [2 , 3]，[2 , 4]。[3] 扩展为 [3 , 4];
第 3 次循环，在每个 list 添加 1 个数， [1，2] 扩展为[1，2，3]， [1，2，4]， [1，2，5]; [1，3] 扩展为 [1，3，4]， [1，3，5]; [1，4] 扩展为 [1，4，5]; [2，3] 扩展为 [2，3，4]， [2，3，5]; [2，4] 扩展为 [2，4，5]; [3，4] 扩展为 [3，4，5];
最后结果就是，[[1，2，3]， [1，2，4]， [1，2，5]，[1，3，4]， [1，3，5]， [1，4，5]， [2，3，4]， [2，3，5]，[2，4，5]， [3，4，5]]。
上边分析很明显了，三个循环，第一层循环是 1 到 k ，代表当前有多少个数。第二层循环就是遍历之前的所有结果。第三次循环就是将当前结果扩展为多个。

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

```
    if (n == 0 || k == 0 || k > n) return Collections.emptyList();
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    //个数为 1 的所有可能
    for (int i = 1; i <= n + 1 - k; i++) res.add(Arrays.asList(i));
    //第一层循环，从 2 到 k
    for (int i = 2; i <= k; i++) {
        List<List<Integer>> tmp = new ArrayList<List<Integer>>();
        //第二层循环，遍历之前所有的结果
        for (List<Integer> list : res) {
            //第三次循环，对每个结果进行扩展
            //从最后一个元素加 1 开始，然后不是到 n ，而是和解法一的优化一样
            //(k - (i - 1)) 代表当前已经有的个数，最后再加 1 是因为取了 n
            for (int m = list.get(list.size() - 1) + 1; m <= n - (k - (i - 1)) + 1; m++) {
                List<Integer> newList = new ArrayList<Integer>(list);
                newList.add(m);
                tmp.add(newList);
            }
        }
        res = tmp;
    }
    return res;
}
```

解法四 递归

参考这里基于这个公式 **C (n, k) = C (n - 1, k - 1) + C (n - 1, k)** 所用的思想，这个思想之前刷题也用过，但忘记是哪道了。

从 n 个数字选 k 个，我们把所有结果分为两种，包含第 n 个数和不包含第 n 个数。这样的话，就可以把问题转换成

- 从 n - 1 里边选 k - 1 个，然后每个结果加上 n
- 从 n - 1 个里边直接选 k 个。

把上边两个的结果合起来就可以了。

```
public List<List<Integer>> combine(int n, int k) {
    if (k == n || k == 0) {
        List<Integer> row = new LinkedList<>();
        for (int i = 1; i <= k; ++i) {
```

```
            row.add(i);
        }
        return new LinkedList<>(Arrays.asList(row));
    }
    // n - 1 里边选 k - 1 个
    List<List<Integer>> result = combine(n - 1, k - 1);
    //每个结果加上 n
    result.forEach(e -> e.add(n));
    //把 n - 1 个选 k 个的结果也加入
    result.addAll(combine(n - 1, k));
    return result;
}
```

解法五 动态规划

参考这里，既然有了解法四的递归，那么一定可以有动态规划。递归就是压栈压栈压栈，然后到了递归出口，开始出栈出栈出栈。而动态规划一个好处就是省略了出栈的过程，我们直接从递归出口往上走。

```
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>>[][] dp = new List[n + 1][k + 1];
    //更新 k = 0 的所有情况
    for (int i = 0; i <= n; i++) {
        dp[i][0] = new ArrayList<>();
        dp[i][0].add(new ArrayList<Integer>());
    }
    // i 从 1 到 n
    for (int i = 1; i <= n; i++) {
        // j 从 1 到 i 或者 k
        for (int j = 1; j <= i && j <= k; j++) {
            dp[i][j] = new ArrayList<>();
            //判断是否可以从 i - 1 里边选 j 个
            if (i > j){
                dp[i][j].addAll(dp[i - 1][j]);
            }
            //把 i - 1 里边选 j - 1 个的每个结果加上 i
            for (List<Integer> list: dp[i - 1][j - 1]) {
                List<Integer> tmpList = new ArrayList<>(list);
```

```
                tmpList.add(i);
                dp[i][j].add(tmpList);
            }
        }
    }
    return dp[n][k];
}
//这里遇到个神奇的问题，提一下，开始的的时候，最里边的 for 循环是这样写的
for (List<Integer> list: dp[i - 1][j - 1]) {
    List<Integer> tmpList = new LinkedList<>(list);
    tmpList.add(i);
    dp[i][j].add(tmpList);
}
//就是 List 用的 Linked，而不是 Array，看起来没什么大问题，在 leetcode 上竟然报了超时。看了下 java 的源码。
//ArrayList
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
//LinkedList
public boolean add(E e) {
    linkLast(e);
    return true;
}
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
```

猜测原因可能是因为 linked 每次 add 的时候，都需要 new 一个节点对象，而我们进行了很多次 add，所以这里造成了时间的耗费，导致了超时。所以刷题的时候还是优先用 ArrayList 吧。

接下来就是动态规划的常规操作了，空间复杂度的优化，我们注意到更新 dp [i] [*] 的时候，只用到 dp [i - 1] [*] 的情况，所以我们可以只用一个一维数组就够了。和72题解法二，以及5题，10题，53题等等优化思路一样，这里不详细说了。

```
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>>[] dp = new ArrayList[k + 1];
    // i 从 1 到 n
    dp[0] = new ArrayList<>();
    dp[0].add(new ArrayList<Integer>());
    for (int i = 1; i <= n; i++) {
        // j 从 1 到 i 或者 k
        List<List<Integer>> temp = new ArrayList<>(dp[0]);
        for (int j = 1; j <= i && j <= k; j++) {
            List<List<Integer>> last = temp;
            if(dp[j]!=null){
                temp = new ArrayList<>(dp[j]);
            }
            // 判断是否可以从 i - 1 里边选 j 个
            if (i <= j) {
                dp[j] = new ArrayList<>();
            }
            // 把 i - 1 里边选 j - 1 个的每个结果加上 i
            for (List<Integer> list : last) {
                List<Integer> tmpList = new ArrayList<>(list);
                tmpList.add(i);
                dp[j].add(tmpList);
            }
        }
    }
    return dp[k];
}
```

总

开始的时候直接用了动态规划，然后翻了一些 Discuss 感觉发现了新世界，把目前为止常用的思路都用到了，回溯，递归，迭代，动态规划，这道题也太经典了！值得细细回味。

在递归解法中为何用到return new LinkedList<>(Arrays.asList(row)) ？

Arrays.asList vs. new ArrayList(Arrays.asList())

Refer to

https://www.baeldung.com/java-arrays-aslist-vs-new-arraylist

1. Overview

In this short tutorial, we'll take a look at the differences between Arrays.asList(array) and ArrayList(Arrays.asList(array)).

2. Arrays.asList

Let's start with the Arrays.asList method.

Using this method, we can convert from an array to a fixed-size List object. This List is just a wrapper that makes the array available as a list. No data is copied or created.

Also, we can't modify its length because adding or removing elements is not allowed.

However, we can modify single items inside the array. Note that all the modifications we make to the single items of the List will be reflected in our original array:

String[] stringArray = new String[] { "A", "B", "C", "D" };

List stringList = Arrays.asList(stringArray);

Now, let's see what happens if we modify the first element of stringList:

stringList.set(0, "E");

assertThat(stringList).containsExactly("E", "B", "C", "D");

assertThat(stringArray).containsExactly("E", "B", "C", "D");

As we can see, our original array was modified, too. Both the list and the array now contain exactly the same elements in the same order.

Let's now try to insert a new element to stringList:

stringList.add("F");

java.lang.UnsupportedOperationException

at java.base/java.util.AbstractList.add(AbstractList.java:153)

at java.base/java.util.AbstractList.add(AbstractList.java:111)

As we can see, adding/removing elements to/from the List will throw java.lang.UnsupportedOperationException.

3. ArrayList(Arrays.asList(array))

Similar to the Arrays.asList method, we can use ArrayList<>(Arrays.asList(array)) when we need to create a List out of an array.

But, unlike our previous example, this is an independent copy of the array, which means that modifying the new list won't affect the original array. Additionally, we have all the capabilities of a regular ArrayList, like adding and removing elements:

```
String[] stringArray = new String[] { "A", "B", "C", "D" };
List stringList = new ArrayList<>(Arrays.asList(stringArray));
```

Now let's modify the first element of stringList:

```
stringList.set(0, "E");
assertThat(stringList).containsExactly("E", "B", "C", "D");
```

And now, let's see what happened with our original array:

```
assertThat(stringArray).containsExactly("A", "B", "C", "D");
```

As we can see, our original array remains untouched.

Before wrapping up, if we take a look at the JDK source code, we can see the Arrays.asList method returns a type of ArrayList that is different from java.util.ArrayList. The main difference is that the returned ArrayList only wraps an existing array — it doesn't implement the add and remove methods.

4. Conclusion

In this short article, we took a look at the differences between two ways of converting an array into an ArrayList. We saw how those two options behave and the difference between how they implement their internal arrays.