<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

/ \ |

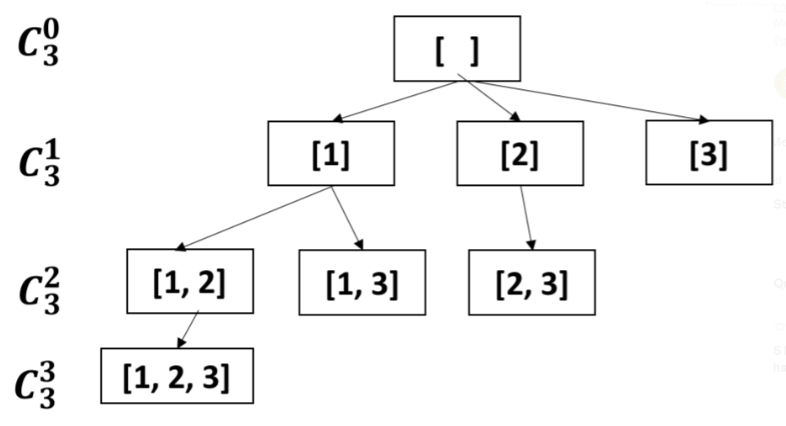
{1,2} {1,3} {2,3} index=2

/

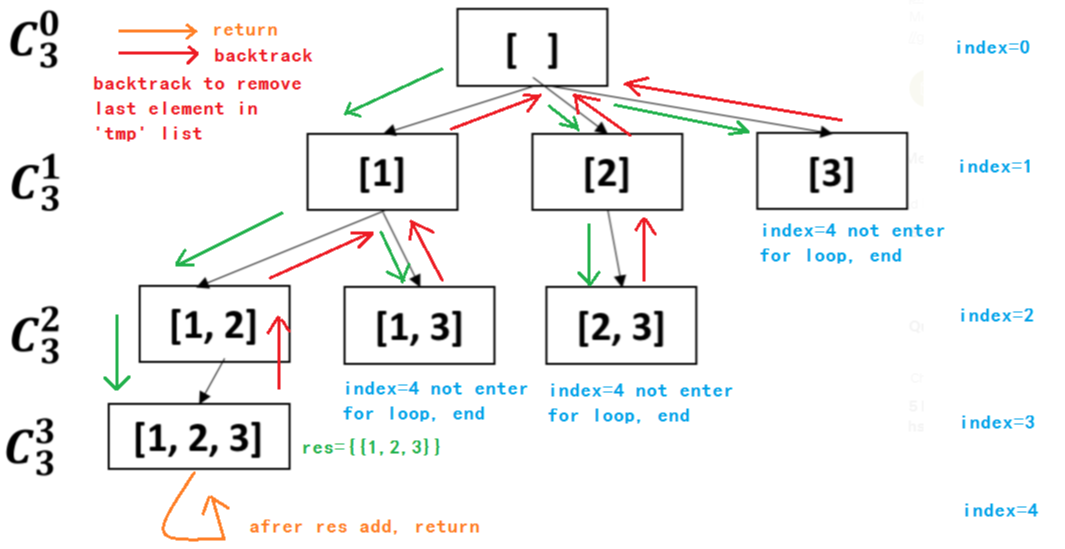
{1,2,3} index=3

**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)**

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.

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 number

T(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](https://www.youtube.com/watch?v=q0s6m7AiM7o) **[time: 4:30]**

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