77-组合

题述

77. 组合

难度 中等 957

给定两个整数 `n` 和 `k`，返回范围 `[1, n]` 中所有可能的 `k` 个数的组合。

你可以按 **任何顺序** 返回答案。

示例 1：

```
输入：n = 4, k = 2
输出：
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

示例 2：

```
输入：n = 1, k = 1
输出：[[1]]
```

提示：

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

思路

本题这是回溯法的经典题目。

直接的解法当然是使用for循环，例如示例中k为2，很容易想到 用两个for循环，这样就可以输出 和示例中一样的结果。

代码如下：

```
int n = 4;
for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++) {
        cout << i << " " << j << endl;
    }
}
```

输入：n = 100, k = 3 那么就三层for循环，代码如下：

```
int n = 100;
for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++) {
        for (int u = j + 1; u <= n; n++) {
            cout << i << " " << j << " " << u << endl;
        }
    }
}
```

如果n为100，k为50呢，那就50层for循环，是不是开始窒息。

此时就会发现虽然想暴力搜索，但是用for循环嵌套连暴力都写不出来！

咋整？

回溯搜索法来了，虽然回溯法也是暴力，但至少能写出来，不像for循环嵌套k层让人绝望。

那么回溯法怎么暴力搜呢？

上面我们说了**要解决 n为100，k为50的情况，暴力写法需要嵌套50层for循环，那么回溯法就用递归来解决嵌套层数的问题。**

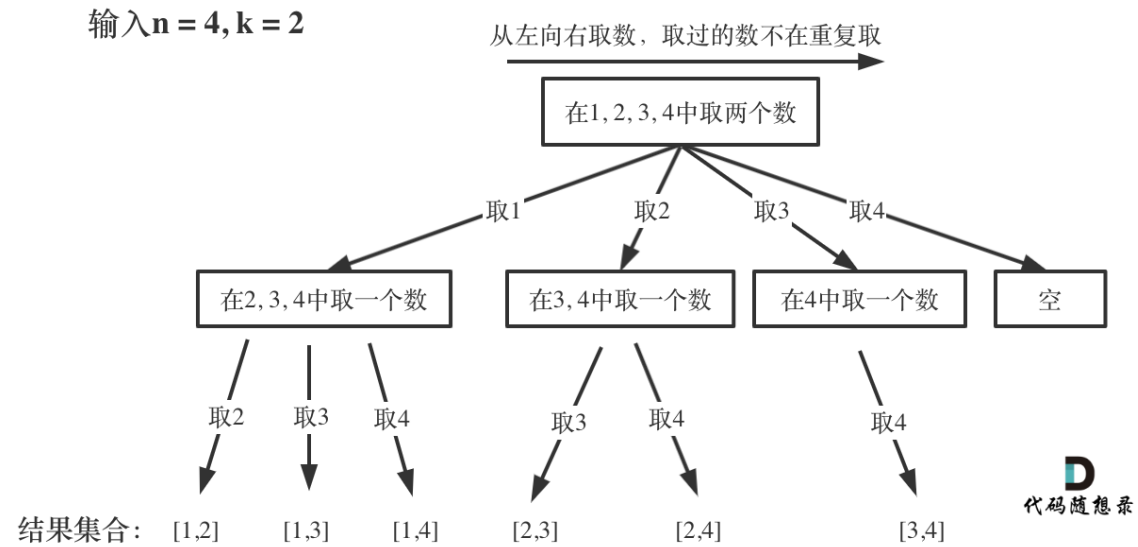
递归来做层叠嵌套（可以理解是开k层for循环），**每一次的递归中嵌套一个for循环，那么递归就可以用于解决多层嵌套循环的问题了。**

此时递归的层数大家应该知道了，例如：n为100，k为50的情况下，就是递归50层。

一些同学本来对递归就懵，回溯法中递归还要嵌套for循环，可能就直接晕倒了！

如果脑洞模拟回溯搜索的过程，绝对可以让人窒息，所以需要抽象图形结构来进一步理解。

那么把组合问题抽象为如下树形结构：



可以看出这个棵树，一开始集合是 1，2，3，4， 从左向右取数，取过的数，不在重复取。

第一次取1，集合变为2，3，4 ，因为k为2，我们只需要再取一个数就可以了，分别取2，3，4，得到集合[1,2] [1,3] [1,4]，以此类推。

每次从集合中选取元素，可选择的范围随着选择的进行而收缩，调整可选择的范围。

图中可以发现n相当于树的宽度，k相当于树的深度。

那么如何在这个树上遍历，然后收集到我们要的结果集呢?

图中每次搜索到了叶子节点，我们就找到了一个结果。

相当于只需要把达到叶子节点的结果收集起来，就可以求得 n个数中k个数的组合集合。

回溯

- 递归函数的返回值以及参数

在这里要定义两个全局变量，一个用来存放符合条件单一结果，一个用来存放符合条件结果的集合。

代码如下：

```
vector<vector<int>> result; // 存放符合条件结果的集合
vector<int> path; // 用来存放符合条件结果
```

其实不定义这两个全局遍历也是可以的，把这两个变量放进递归函数的参数里，但函数里参数太多影响可读性，所以我定义全局变量了。

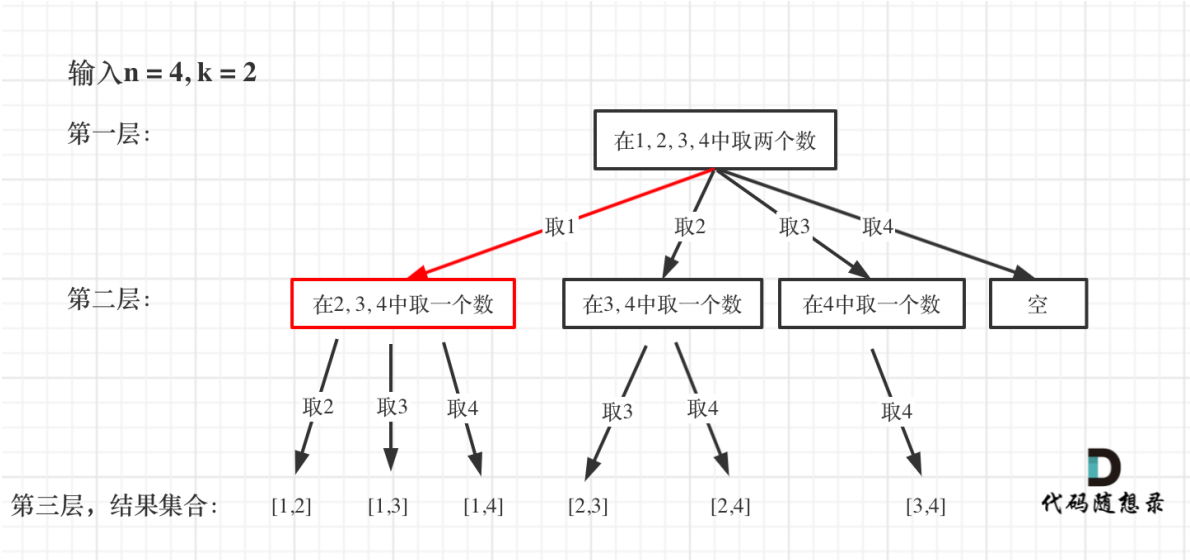
函数里一定有两个参数，既然是集合n里面取k的数，那么n和k是两个int型的参数。

然后还需要一个参数，为int型变量startIndex，这个参数用来记录本层递归的中，集合从哪里开始遍历（集合就是[1,...,n] ）。

为什么要有这个startIndex呢?

每次从集合中选取元素，可选择的范围随着选择的进行而收缩，调整可选择的范围，就是要靠startIndex。

从下图中红线部分可以看出，在集合[1,2,3,4]取1之后，下一层递归，就要在[2,3,4]中取数了，那么下一层递归如何知道从[2,3,4]中取数呢，靠的就是startIndex。



所以需要startIndex来记录下一层递归，搜索的起始位置。

那么整体代码如下：

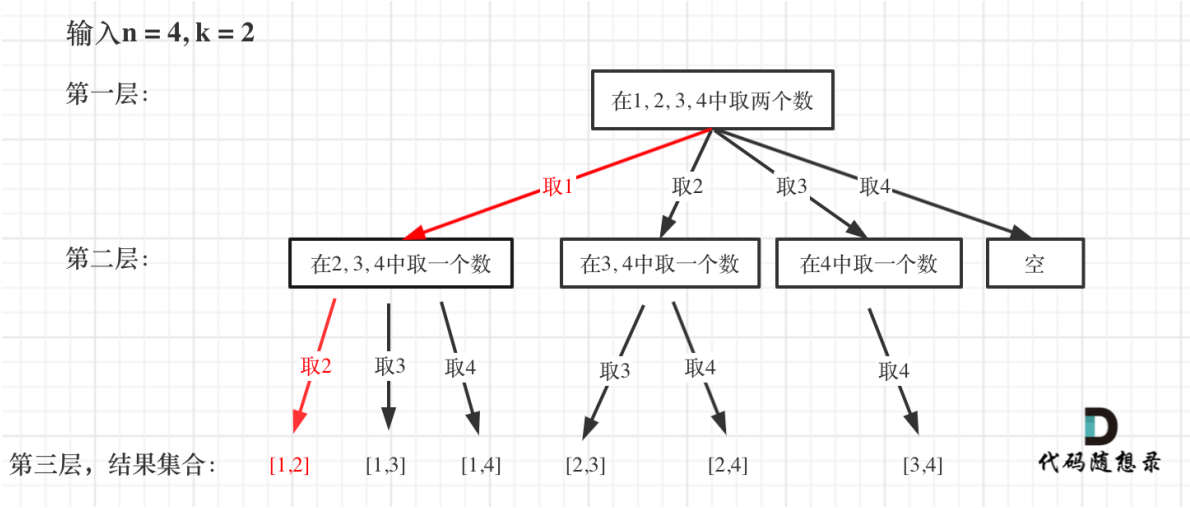
```
vector<vector<int>> result; // 存放符合条件结果的集合
vector<int> path; // 用来存放符合条件单一结果
void backtracking(int n, int k, int startIndex)
```

- 回溯函数终止条件

什么时候到达所谓的叶子节点了呢?

path这个数组的大小如果达到k，说明我们找到了一个子集大小为k的组合了，在图中path存的就是根节点到叶子节点的路径。

如图红色部分：



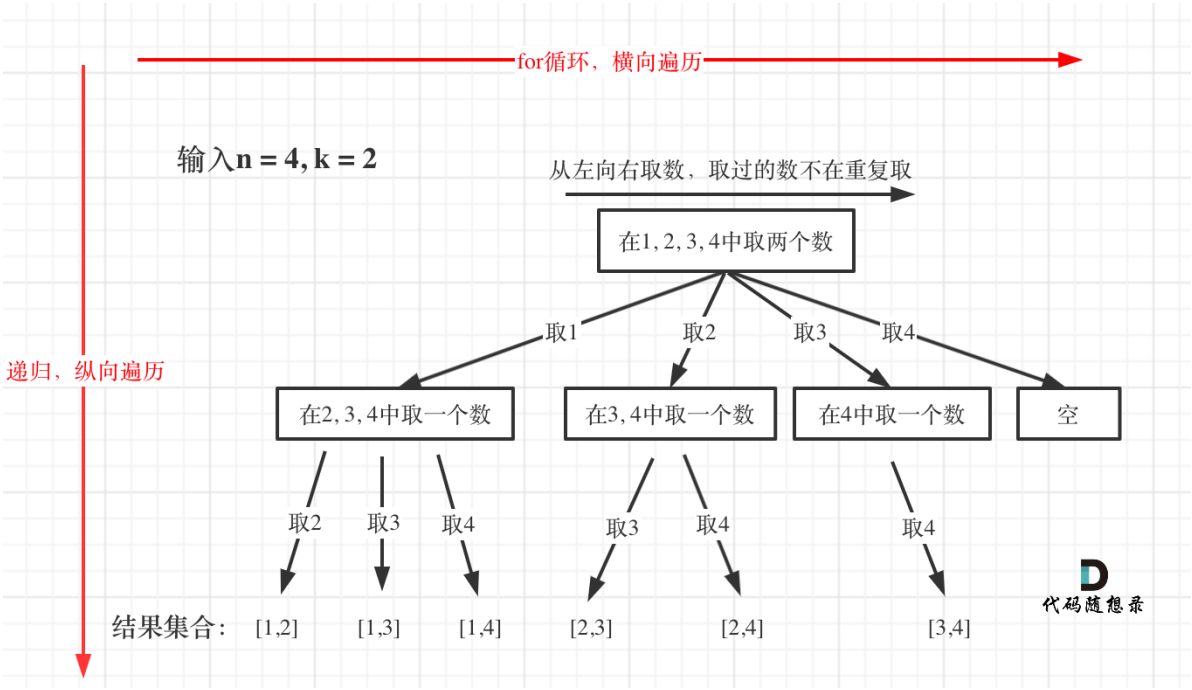
此时用result二维数组，把path保存起来，并终止本层递归。

所以终止条件代码如下：

```
if (path.size() == k) {
    result.push_back(path);
    return;
}
```

- 单层搜索的过程

回溯法的搜索过程就是一个树型结构的遍历过程，在如下图中，可以看出for循环用来横向遍历，递归的过程是纵向遍历。



如此我们才遍历完图中的这棵树。

for循环每次从startIndex开始遍历，然后用path保存取到的节点i。

代码如下：

```
for (int i = startIndex; i <= n; i++) { // 控制树的横向遍历
    path.push_back(i); // 处理节点
    backtracking(n, k, i + 1); // 递归：控制树的纵向遍历，注意下一层搜索要从i+1开始
    path.pop_back(); // 回溯，撤销处理的节点
}
```

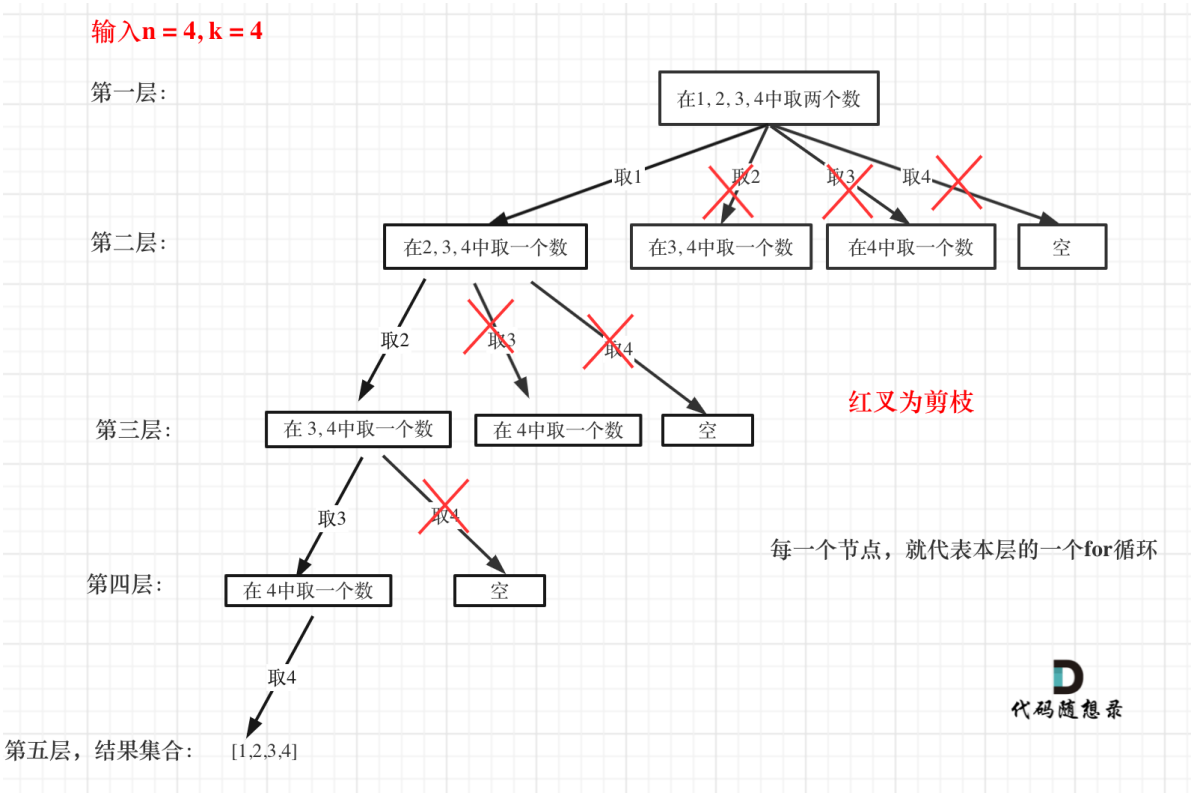
可以看出backtracking（递归函数）通过不断调用自己一直往深处遍历，总会遇到叶子节点，遇到了叶子节点就要返回。

backtracking的下面部分就是回溯的操作了，撤销本次处理的结果。

剪枝优化

回溯法虽然是暴力搜索，但也有时候可以有点剪枝优化一下的。

来举一个例子，n = 4，k = 4的话，那么第一层for循环的时候，从元素2开始的遍历都没有意义了。 在第二层for循环，从元素3开始的遍历都没有意义了。



所以，可以剪枝的地方就在递归中每一层的for循环所选择的起始位置。

如果for循环选择的起始位置之后的元素个数 已经不足 我们需要的元素个数了，那么就没有必要搜索了。

接下来看一下优化过程如下：

1. 已经选择的元素个数：path.size();
2. 还需要的元素个数为: k - path.size();
3. 在集合n中至多要从该起始位置 : n - (k - path.size()) + 1，开始遍历

为什么有个+1呢，因为包括起始位置，我们要是一个左闭的集合。

举个例子，n = 4，k = 3， 目前已经选取的元素为0（path.size为0），n - (k - 0) + 1 即 4 - (3 - 0) + 1 = 2。

从2开始搜索都是合理的，可以是组合[2, 3, 4]。

这里大家想不懂的话，建议也举一个例子，就知道是不是要+1了。

所以优化之后的for循环是：

```
for (int i = startIndex; i <= n - (k - path.size()) + 1; i++) // i为本次搜索的起始位置
```

题解

C++ - 回溯

```
class Solution {
private:
    vector<vector<int>> result; //存放符合条件结果的集合
    vector<int> path;   //用来存放符合条件结果
    void backtracking(int n,int k,int startIndex)
    {
```

```
        if(path.size() == k)
        {
            result.push_back(path);
            return;
        }
        for(int i = startIndex; i <= n; i++)
        {
            path.push_back(i);  //处理节点
            backtracking(n,k,i+1);  //递归
            path.pop_back();    //回溯，撤销处理的节点
        }
    }
public:
    vector<vector<int>> combine(int n, int k)
    {
        result.clear();
        path.clear();
        backtracking(n,k,1);
        return result;
    }
};
```

C++剪枝

```
class Solution {
private:
    vector<vector<int>> result; //存放符合条件结果的集合
    vector<int> path;   //用来存放符合条件结果
    void backtracking(int n,int k,int startIndex)
    {
        if(path.size() == k)
        {
            result.push_back(path);
```

```
            return;
        }
        for(int i = startIndex; i <= n - (k-path.size()) + 1; i++)
        {
            path.push_back(i);  //处理节点
            backtracking(n,k,i+1);  //递归
            path.pop_back();    //回溯，撤销处理的节点
        }
    }
public:
    vector<vector<int>> combine(int n, int k)
    {
        backtracking(n,k,1);
        return result;
    }
};
```

提交结果	执行用时	内存消耗	语言	提交时间	备注
通过	4 ms	9.8 MB	C++	2022/04/27 10:10	
通过	16 ms	9.6 MB	C++	2022/04/27 09:30	

Python

```
class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        res = []
        path =[]
        def backTrack(n,k,startIndex):
            if len(path) == k:
                res.append(path[:])
                return
            for i in range(startIndex,n+1):
                path.append(i)
                backTrack(n,k,i+1)
```

```
                path.pop()
        backTrack(n,k,1)
        return res
```

Python剪枝

```python
class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        res=[]  #存放符合条件结果的集合
        path=[]  #用来存放符合条件结果
        def backtrack(n,k,startIndex):
            if len(path) == k:
                res.append(path[:])
                return
            for i in range(startIndex,n-(k-len(path))+2):  #优化的地方
                path.append(i)  #处理节点
                backtrack(n,k,i+1)  #递归
                path.pop()  #回溯，撤销处理的节点
        backtrack(n,k,1)
        return res
```

思考

组合问题是回溯法解决的经典问题，我们开始的时候给大家列举一个很形象的例子，就是n为100，k为50的话，直接想法就需要50层for循环。

从而引出了回溯法就是解决这种k层for循环嵌套的问题。

然后进一步把回溯法的搜索过程抽象为树形结构，可以直观的看出搜索的过程。

接着用回溯法三部曲，逐步分析了函数参数、终止条件和单层搜索的过程。