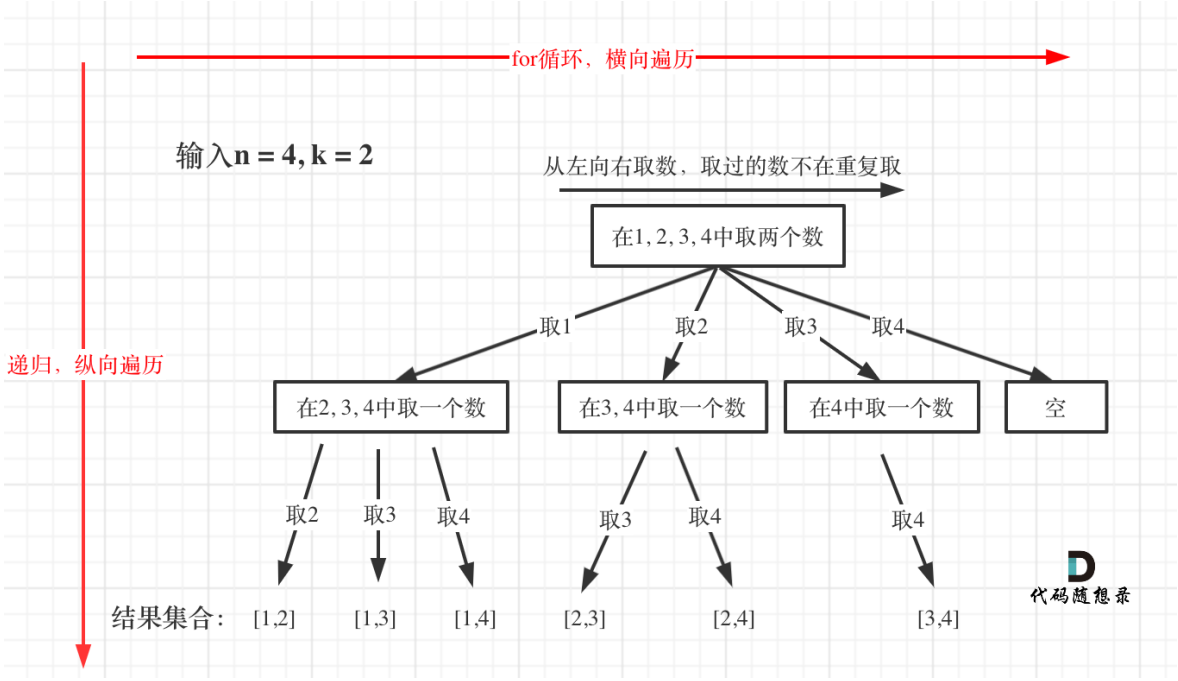
回溯算法

组合问题

未优化版本

组合问题模拟的是以下的过程。值得注意的是，组合不讲究顺序，需要去重，而排序问题需要讲究顺序




```
class Solution {
private:
    vector<int> path;
    vector<vector<int>> result;
    void backtracking(int n, int k, int startindex) {
        if (path.size() == k) {//符合题目要求的大小时
            result.push_back(path);
            return;
        }

        for (int i = startindex; i <= n;i++) {
        //这里的for循环模拟的就是第二行从取1到取4的操作
            path.push_back(i);
            backtracking(n, k, i + 1);
            path.pop_back();//回溯
        }
    }
```

```
public:
    vector<vector<int>> combine(int n, int k) {
        backtracking(n, k, 1);
        return result;
    }
};
```

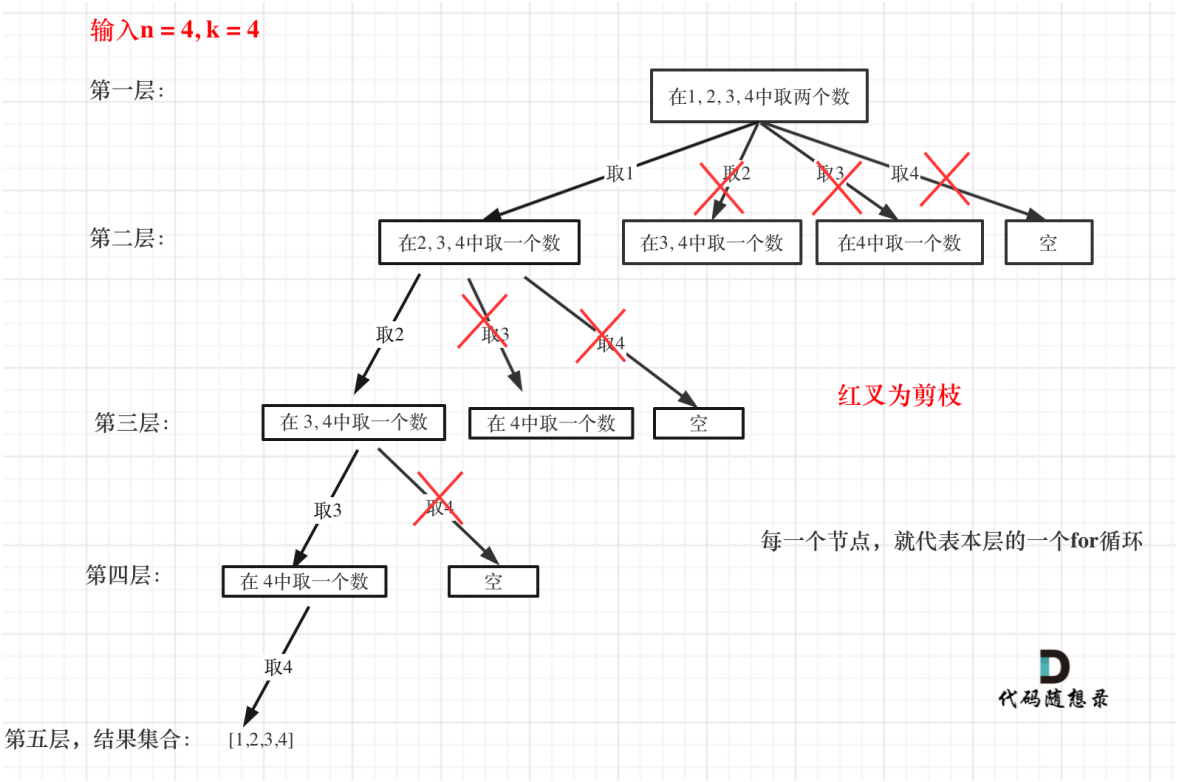
优化版本：进行剪枝操作

剪枝操作优化的是**for循环中终止位置**

当执行到path.size()==k时递归终止

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. 所需需要的元素个数为: k - path.size();
3. 列表中剩余元素（n-i） >= 所需需要的元素个数（k - path.size()）
4. 得出了至多开始的位置，也就是终止的条件，i<=n-(k-path.size())+1
5. 例如下面，当n=4,k=4，至多从1开始，只有[1,2,3,4]

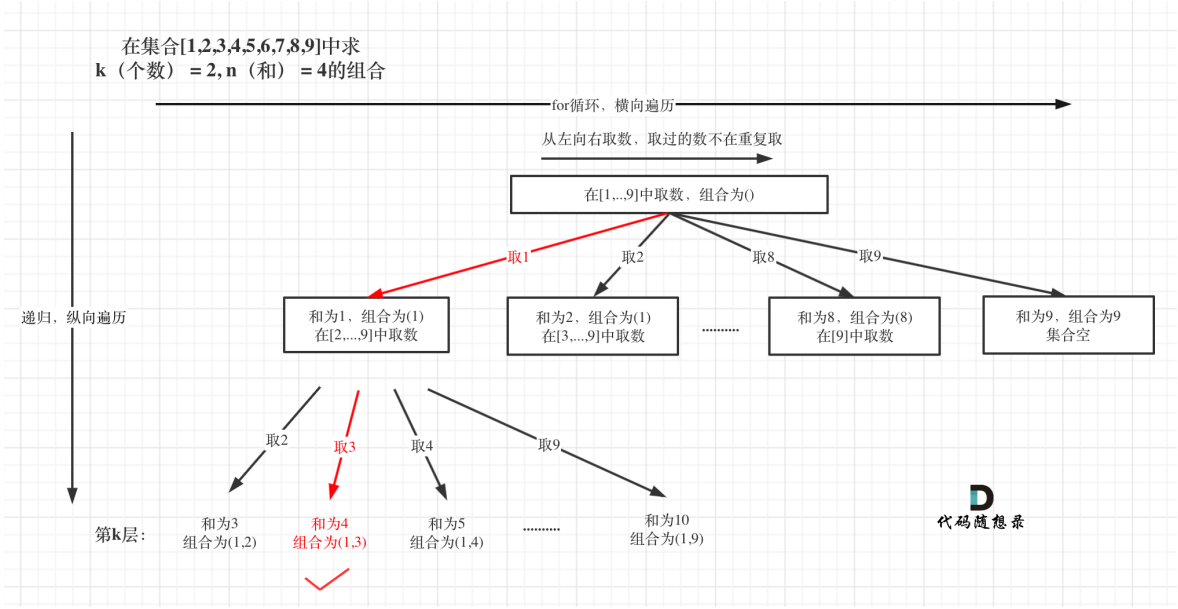


组合总和

数字1到9中选取k个数，结果等于n，不可以重复选取

从上到下可选元素的减少，是用来确保**某一个元素不会重复的选取，通过回溯参数start是否加1来控制**

从左到右可选元素的减少，是用来确保**集合不是重复**的{1,2}以及{2,1}。一般情况下这都是需要的，这是通过for循环中i++来控制



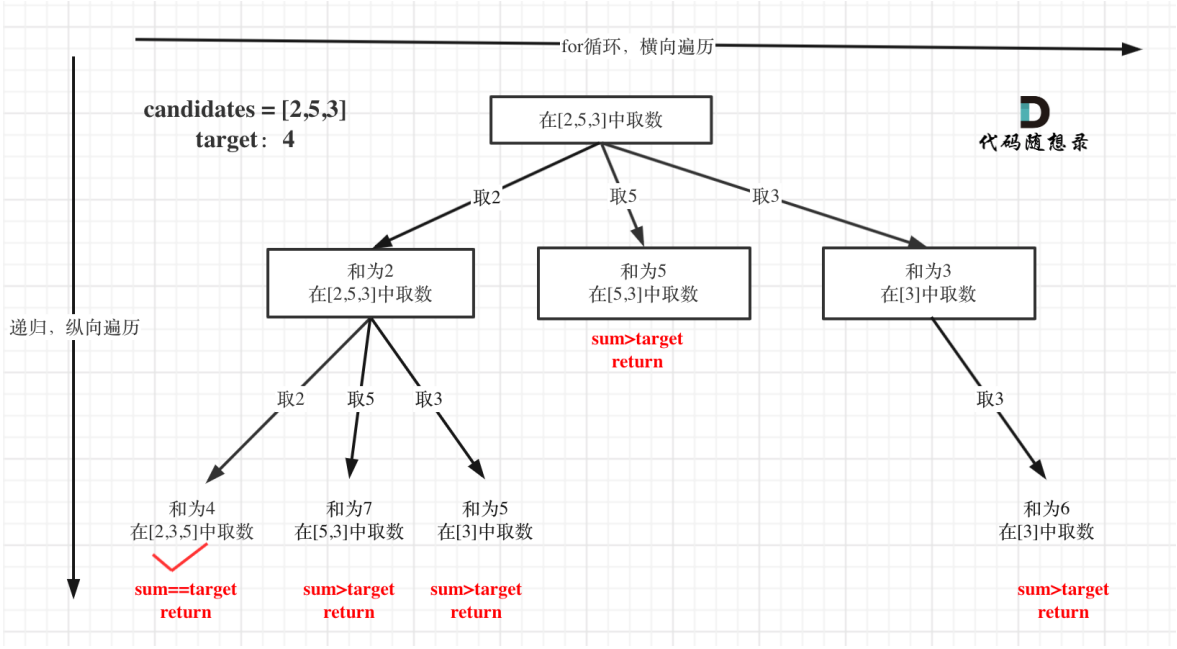
这一份待用使用到了两个剪枝操作：

一个是**总和的剪枝**，另一个是**集合个数的剪枝**

```
class Solution {
private:
    vector<int> path;
    vector<vector<int>> result;

    void backtracking(int n,int k,int curSum,int start) {

        if (curSum > n) {//总和部分的剪枝
            return;
        }

        if (path.size() == k) {
            if (curSum == n) {
                result.push_back(path);
            }
            return;//注意，无论满不满足条件，都要返回
        }
```

```
        for (int i = start;i <= 9-(k-path.size())+1;i++) {//个数的剪枝

            curSum += i;
            path.push_back(i);//插入数据

            backtracking(n, k, curSum, i + 1);//递归

            curSum -= i;
            path.pop_back();//回溯过程

        }
    }
public:
    vector<vector<int>> combinationSum3(int k, int n) {

        backtracking(n, k, 0, 1);
        return result;
    }
};
```

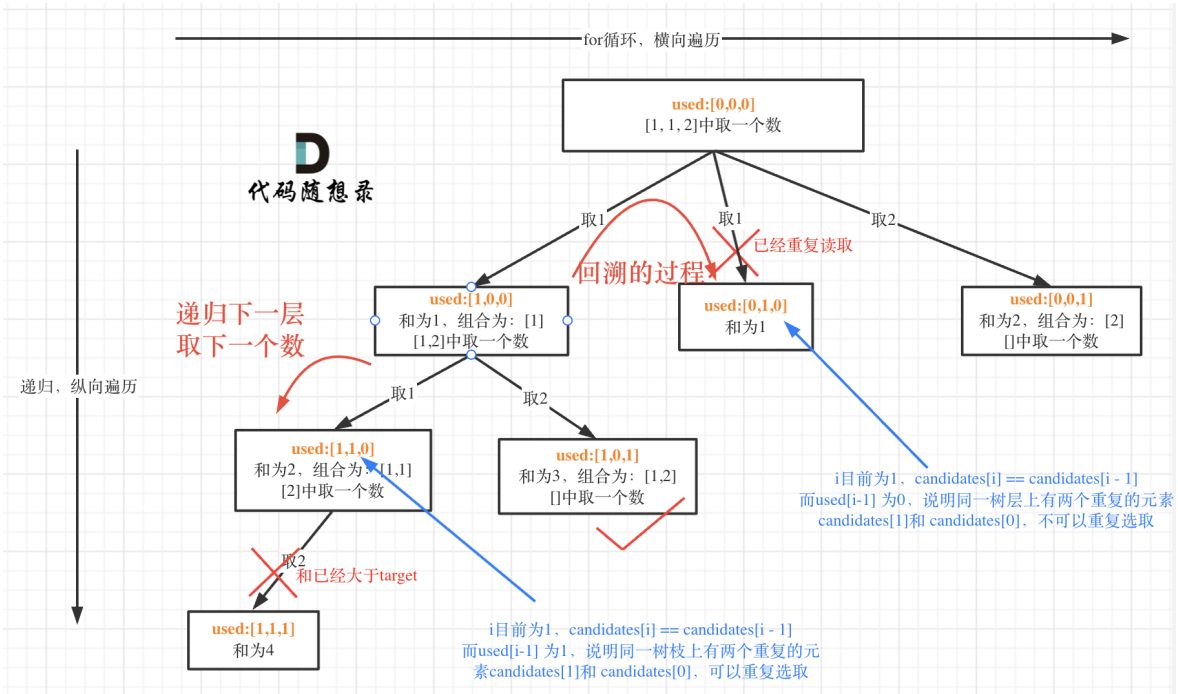
给定一个数组，总和结果为target。可以重复选取，数组中无重复数字




```
class Solution {
private:
    vector<int> path;;
    vector<vector<int>> result;
    void backtracking(vector<int>& candidates,int target,int start){
```

```
        if(target<0){
            return;
        }else if(target==0){
            result.push_back(path);
            return;
        }

        for(int i=start;i<candidates.size();i++){
            path.push_back(candidates[i]);
            backtracking(candidates,target-candidates[i],i);//元素可以重复使用，所以i
无需加1
            path.pop_back();
        }
    }
public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        backtracking(candidates,target,0);
        return result;
    }
};
```

给定一个数组，总和结果为target。不可以重复选取，且数组中出现了重复的数字



```
class Solution {
private:
    vector<bool> used;
    vector<int> path;
    vector<vector<int>> result;
    void backtracking(vector<int> candidates, int target, int start, vector<bool> u
sed) {
        if (target == 0) {
            result.push_back(path);
            return;
        }

        for (int i = start;i < candidates.size() && target - candidates[i] >= 0;i+
+) {//树层遍历

            if (i > 0 && candidates[i - 1] == candidates[i] && used[i] == false) {
                continue;
            }
            // used[i - 1] == true，说明同一树枝candidates[i - 1]使用过，树枝上可以重复
利用
            // used[i - 1] == false，说明同一树层candidates[i - 1]使用过，树层上不能重复
利用

            path.push_back(candidates[i]);
            target -= candidates[i];
            used[i] = true;

            backtracking(candidates, target, i + 1, used);

            path.pop_back();
            target += candidates[i];
            used[i] = false;
        }
    }
public:
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        sort(candidates.begin(), candidates.end());
        used.resize(candidates.size(), false);
        backtracking(candidates, target, 0, used);
        return result;
    }
};
```

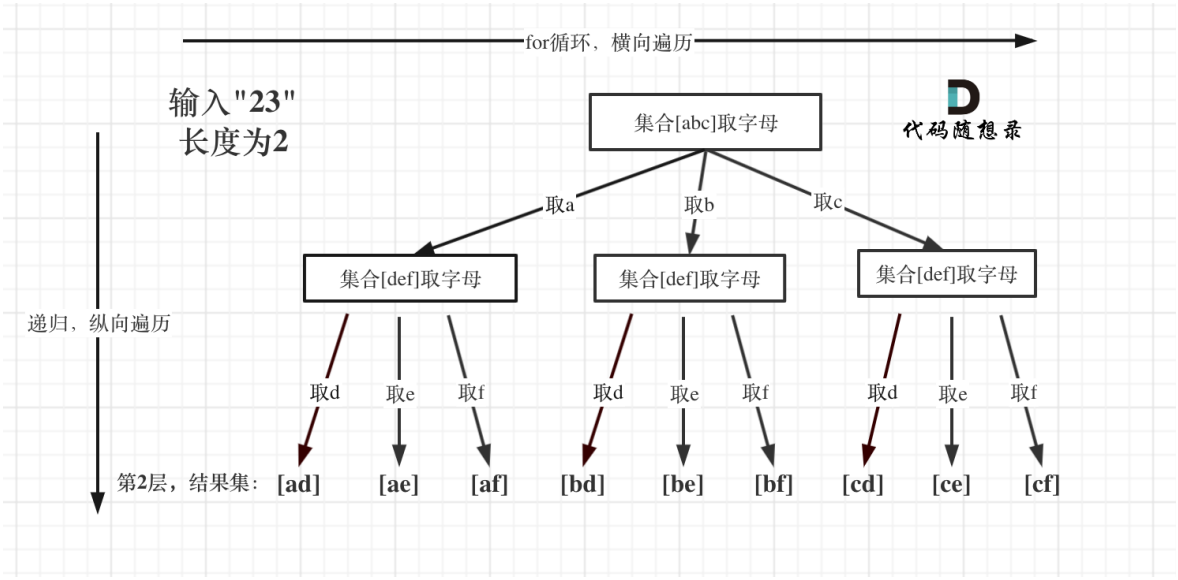
也可以不使用used数组

```cpp
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
private:
    vector<int> path;
    vector<vector<int>> result;
    void backtracking(vector<int> candidates, int target, int start) {
        if (target == 0) {
            result.push_back(path);
            return;
        }

        for (int i = start;i < candidates.size() && target - candidates[i] >= 0;i+
+) {
            if (i > start && candidates[i - 1] == candidates[i] ) {
            //如果之前同层已经出现过同样的数字
            //个人更加倾向于这个写法
            //排序后,同一层的相同元素只处理第一个;同一支的元素会受到index的约束,仍然会选取重
复元素
                continue;
            }

            path.push_back(candidates[i]);
            target -= candidates[i];

            backtracking(candidates, target, i + 1);

            path.pop_back();
            target += candidates[i];
        }
    }
public:
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        sort(candidates.begin(), candidates.end());
        backtracking(candidates, target, 0);
        return result;
    }
};
```

电话号码的字母组合

数字与字母的映射关系

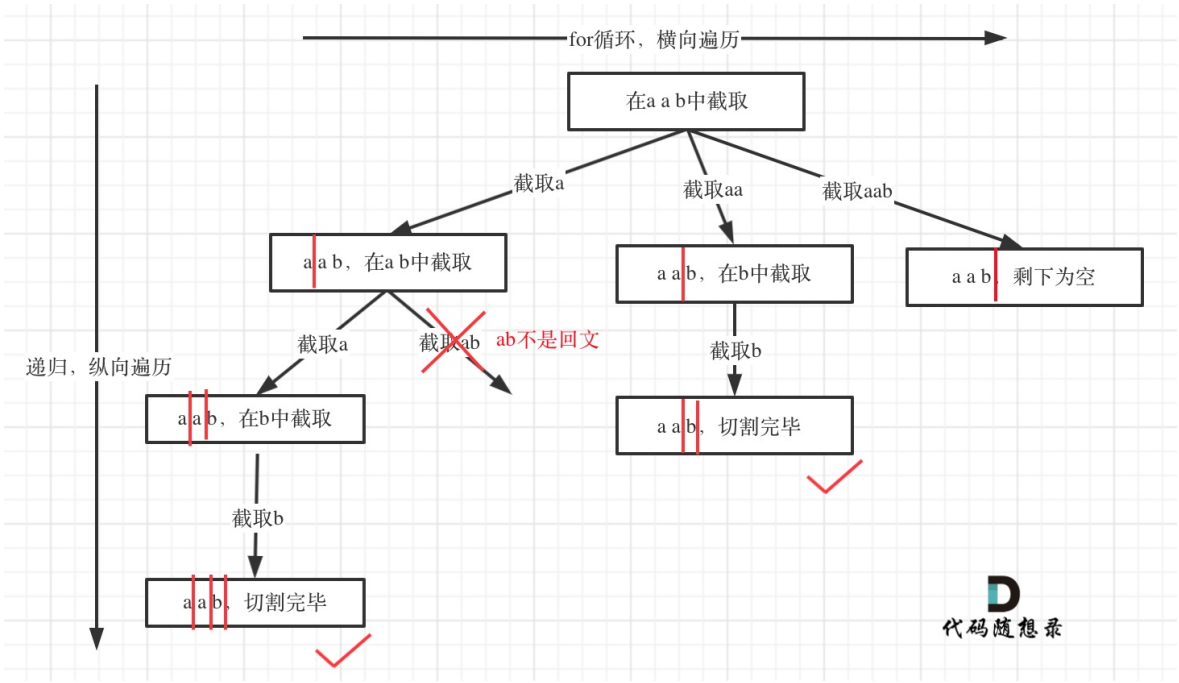
树形结构




```
class Solution {
private:
    string map[10] = {
        "",
        "",
        "abc",
        "def",
        "ghi",
        "jkl",
        "mno",
        "pqrs",
        "tuv",
        "wxyz"
    };
    string path;
    vector<string> result;
    void backtracking(string digits,int index) {
        if (index == digits.size()) {//当执行到最后一个字母时，仍然需要进行递归操作
```

```
            result.push_back(path);
            return;
        }

        int num = digits[index] - '0';//得到数字
        string alphabet = map[num];//得到当前的字符串

        for (int i = 0;i < alphabet.size();i++) {

            path.push_back(alphabet[i]);
            index += 1;

            backtracking(digits, index);//递归

            path.pop_back();
            index -= 1;//回溯算法
        }
    }
public:
    vector<string> letterCombinations(string digits) {
        if (digits.empty()) {
            return vector<string>();
        }
        backtracking(digits, 0);

        return result;
    }
};
```

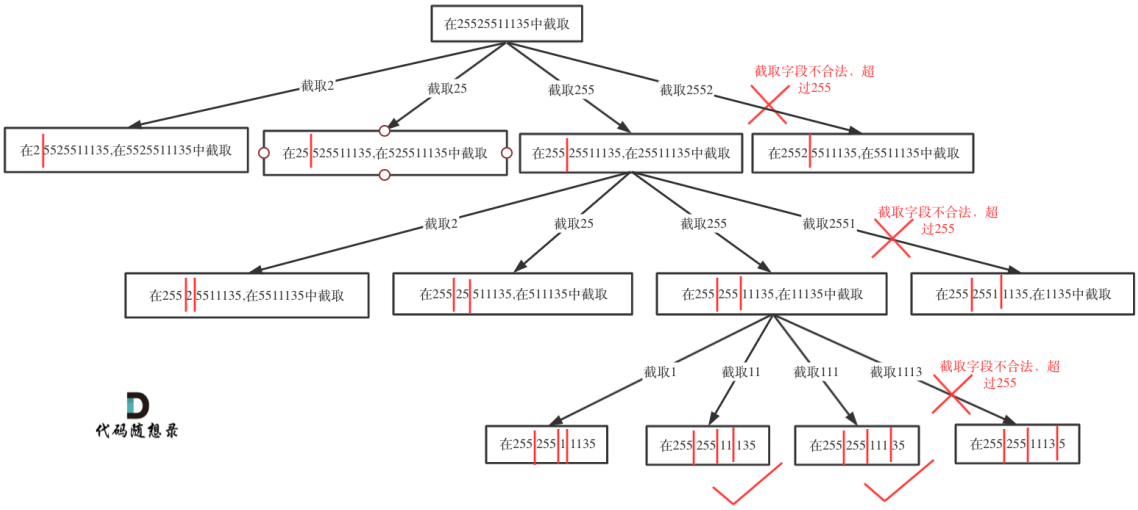
切割回文字符串




```
#include <vector>
#include <iostream>
using namespace std;

class Solution {
private:
    bool isPalindrome(string s,int start,int end) {//判断是否回文
        int left = 0;
        int right = end;
        while (left < right) {
            if (s[left] != s[right]) {
                return false;
            }

            left++;
            right--;
        }

        return true;
    }

    vector<string> path;
    vector<vector<string>> result;

    void backtracking(string s, int start) {//start表示分割点
        if (start == s.size()) {
            result.push_back(path);
            return;
```

```
        }

        for (int i = start;i < s.size();i++) {
            if (isPalindrome(s, start, i)) {
                string temp(s.begin() + start, s.begin() + i + 1);//回文传插入到path
中，左闭右开
                path.push_back(temp);
            }
            else {
                continue;
            }
            backtracking(s, i + 1);
            path.pop_back();
        }
    }
public:
    vector<vector<string>> partition(string s) {
        backtracking(s, 0);
        return result;
    }
};
```

有效的IP地址




```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

class Solution {
```

```
private:
    bool IsValid(string s, int start, int end) {

        if(start>s.size()-1||end<0){
            return false;
        }
        //防止出现这样的情况"10.10.23."


        if (s[start] == '0'&&start!=end) {//以0开头
            return false;
        }

        int num = 0;
        for (int i = start;i <= end;i++) {
            if (s[i] >= '0' && s[i] <= '9') {
                num = num * 10 + s[i]-'0';
                if (num > 255) {//大于255
                    return false;
                }
            }
            else {//非法字符
                return false;
            }
        }
        return true;
    }

    vector<string> result;

    void backtracking(string s, int start, int pointNum) {
        if (pointNum == 3) {
            if (IsValid(s, start, s.size() - 1)) {//最后一段的判断
                result.push_back(s);
                return;
            }
        }

        for (int i = start;i < s.size();i++) {
            if (IsValid(s, start, i)) {
                s.insert(s.begin() + i + 1, '.');
                pointNum++;

                backtracking(s, i + 2, pointNum);

                pointNum--;
                s.erase(s.begin() + i + 1);
            }
```

```
            else {
                break;
            }
        }
    }
public:
    vector<string> restoreIpAddresses(string s) {
        if(s.size()<4||s.size()>12){
            return vector<string>();
        }
        backtracking(s, 0, 0);
        return result;
    }
};
```

我的思路的优化版

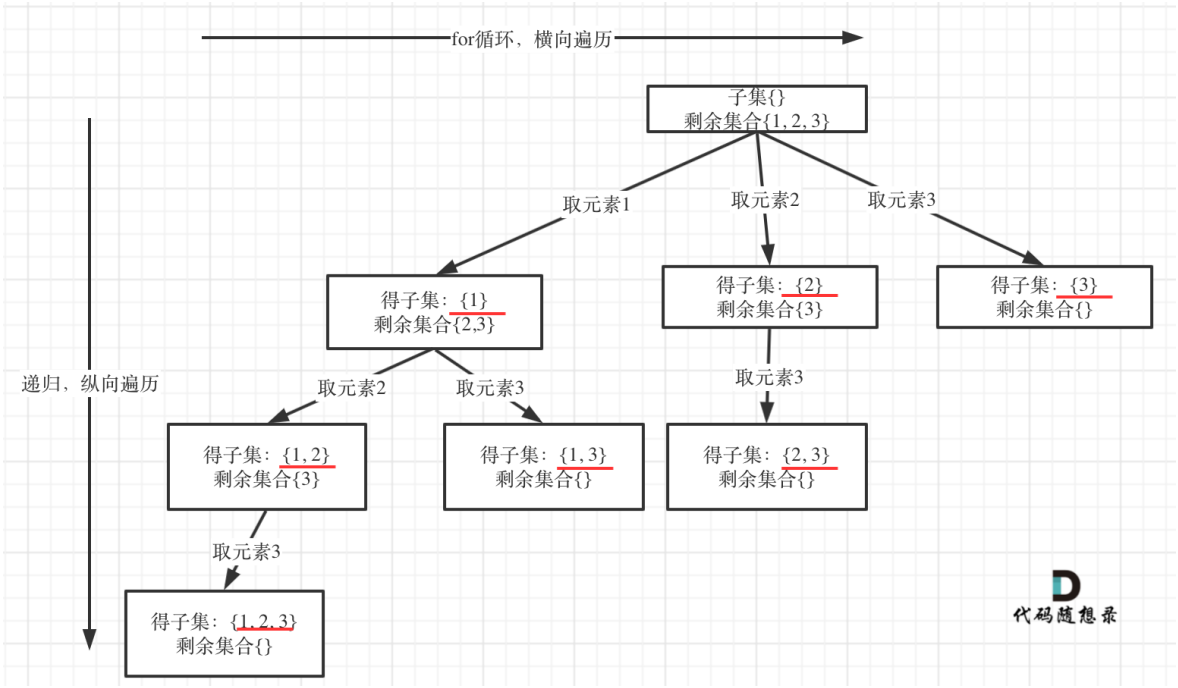
```
class Solution {
public:
    vector<string> restoreIpAddresses(string s) {
        vector<string> result;
        string path;
        backtracking(s, 0, 0, path, result);
        return result;
    }

private:
    bool IsValid(const string& s, int begin, int end) {
        if (begin > s.size() - 1 || end < 0) {
            return false;
        }

        if (s[begin] == '0' && begin != end) {
            return false;
        }

        if (end - begin > 2) {
            return false;
        }

        int num = 0;
        for (int i = begin; i <= end; i++) {
            if (s[i] >= '0' && s[i] <= '9') {
                num = num * 10 + s[i] - '0';
                if (num > 255) {
                    return false;
```

```
                }
            } else {
                return false;
            }
        }
        return true;
    }

    void backtracking(const string& s, int start, int point, string& path, vector<s
tring>& result) {
        if (point == 3) {
            if (IsValid(s, start, s.size() - 1)) {
                string temp = string(s.begin() + start, s.end());
                path += temp;
                result.push_back(path);
                return;
            }
        }

        for (int i = start; i < s.size(); i++) {
            if (IsValid(s, start, i)) {

                string temp = s.substr(start, i - start + 1);
                path += temp + '.';

                backtracking(s, i + 1, point + 1, path, result);

                path.erase(path.size() - temp.size() - 1);

            } else {
                break;
            }
        }
    }
};
```

子集

无重复元素

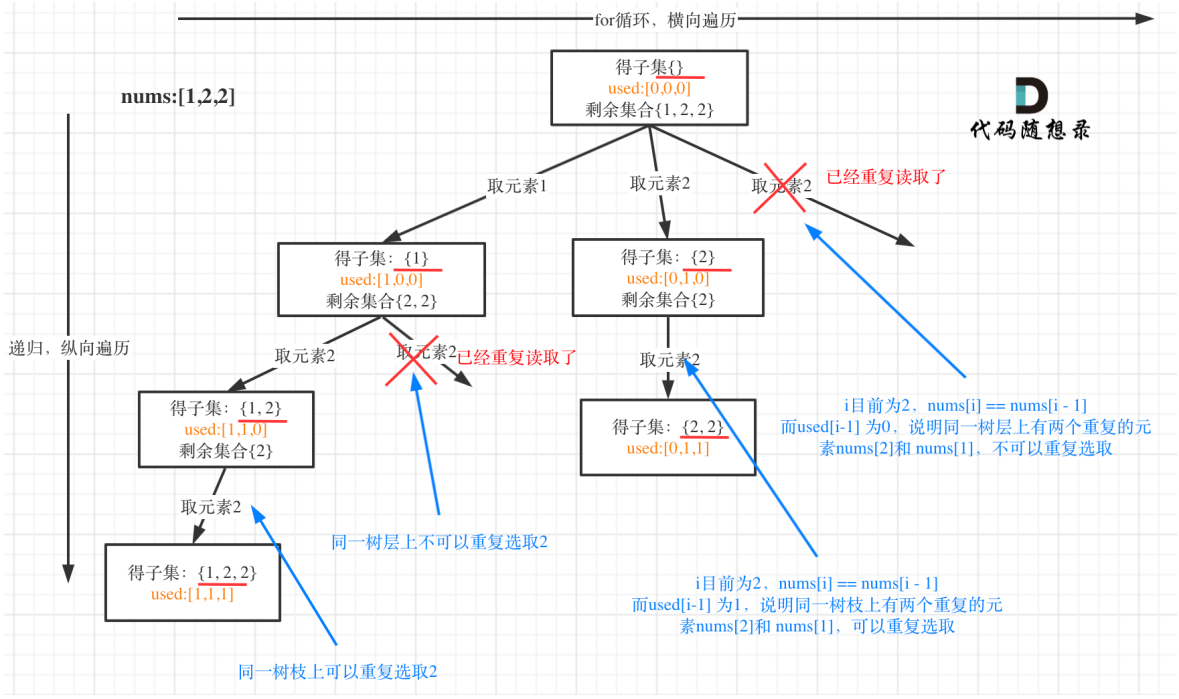
算法：每一次进行递归，都会将元素插入到result中，而不是递归到终点的时候才进行插入。（本体要求的是求出所有的子集，而不是符合相关条件的子集）




```
class Solution {
    vector<int> path;
    vector<vector<int>> result;

    void backtracking(vector<int>& nums, int start) {

        result.push_back(path);//这里不过是否终止，每一次都会push元素
        if (start >= nums.size()) {

            return;
        }

        for (int i = start;i < nums.size();i++) {
            path.push_back(nums[i]);

            backtracking(nums, i + 1);

            path.pop_back();
        }
    }

public:
    vector<vector<int>> subsets(vector<int>& nums) {
        backtracking(nums, 0);
        result.push_back(vector<int>());
        return result;
```

```
    }
};
```

有重复元素




```
class Solution {
private:
    vector<bool> used;
    vector<int> path;
    vector<vector<int>> result;

    void backtracking(vector<int>& nums, int start, vector<bool> used) {
        result.push_back(path);
        if (start == nums.size()) {
            return;
        }

        for (int i = start;i < nums.size();i++) {
            if (i>0&&nums[i] == nums[i - 1] && used[i-1] == true)//如果为true说明在树枝上，可以选取
            {
                continue;
            }
            else {
                path.push_back(nums[i]);
                used[i] = true;
```

```
                backtracking(nums, i+1, used);
                //这个地方是否加一取决于是否能重复取某一个元素

                path.pop_back();
                used[i] = false;
            }
        }
    }

public:
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        sort(nums.begin(), nums.end());//注意nums数组并没有提前排好序

        used.resize(nums.size(), false);
        backtracking(nums, 0, used);
        return result;
    }
};
```

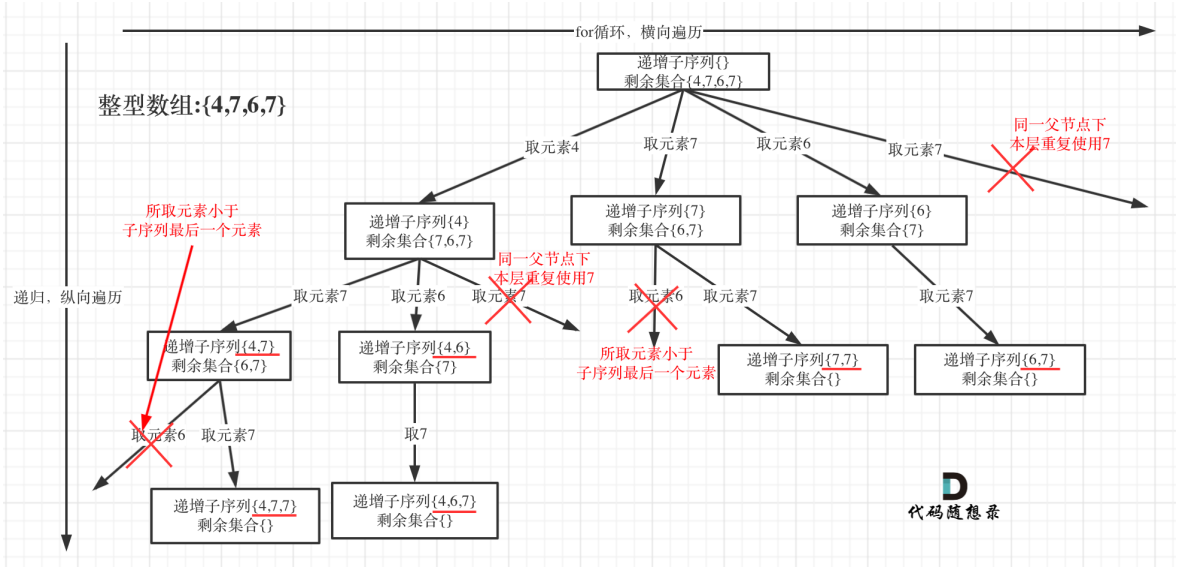
递增子序列

算法：每一层使用unorder_set来保存元素，如果出现过则跳过

每执行一遍函数，都是对于层的遍历，所以unorder_set放在递归函数里，记录的是同一层的数据

而执行到backtracking函数的时候，则是向**下一层去遍历**

如果将used放在外面，则used会被层、枝遍历到。此时，如果如果前一个used的话，说明在同一枝，没有used的话，在同一层



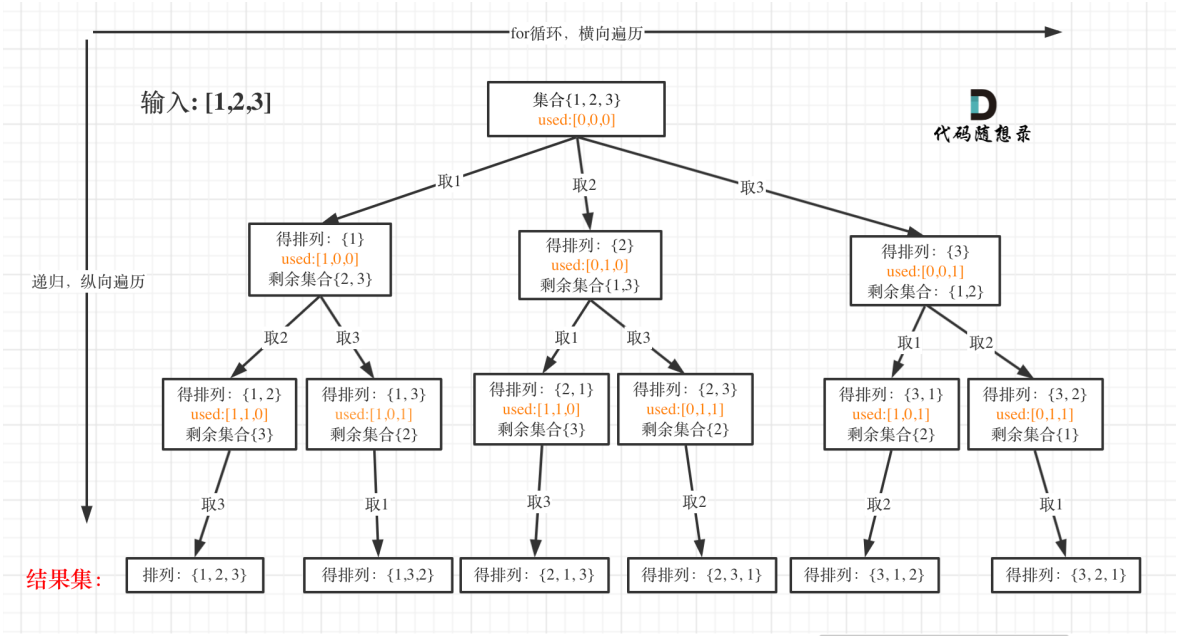
```
class Solution {
private:
    vector<int> path;
    vector<vector<int>> result;

    void backtracking(vector<int>& nums, int start) {
        if (path.size() > 1) {
            result.push_back(path);
        }

        if(start==nums.size()){
            return;
        }

        unordered_set<int> used;//用来记录同一层的元素，如果找到了，说明在同一层，必须被跳过
        for (int i = start;i < nums.size();i++) {
            if ((!path.empty() && path.back() > nums[i]) || used.count(nums[i])) {
                continue;
            }
            else {
                used.insert(nums[i]);
                path.push_back(nums[i]);

                backtracking(nums, i + 1);

                path.pop_back();
            }
        }
    }
public:
    vector<vector<int>> findSubsequences(vector<int>& nums) {
        backtracking(nums, 0);
        return result;
    }
};
```

全排列

排列与组合相比，元素顺序不同， 也是不同的情况。例如{1,2}与{2,1}，所以不使用start来控制起始下标去重

无重复元素

算法：在叶子节点收获，并且使用used来确保使用的不是使用过的元素。这里采用的是**树枝去重**




```
class Solution {
private:
    vector<int> path;
    vector<vector<int>> result;
    vector<bool> used;

    void backtracking(vector<int>& nums) {
        if (path.size() == nums.size()) {
            result.push_back(path);
            return;
        }

        for (int i = 0;i < nums.size();i++) {
            if (used[i]) {
                continue;
            }

            used[i] = true;
            path.push_back(nums[i]);

            backtracking(nums);

            used[i] = false;
            path.pop_back();
        }
    }
public:
    vector<vector<int>> permute(vector<int>& nums) {
        used.resize(nums.size(), false);
```

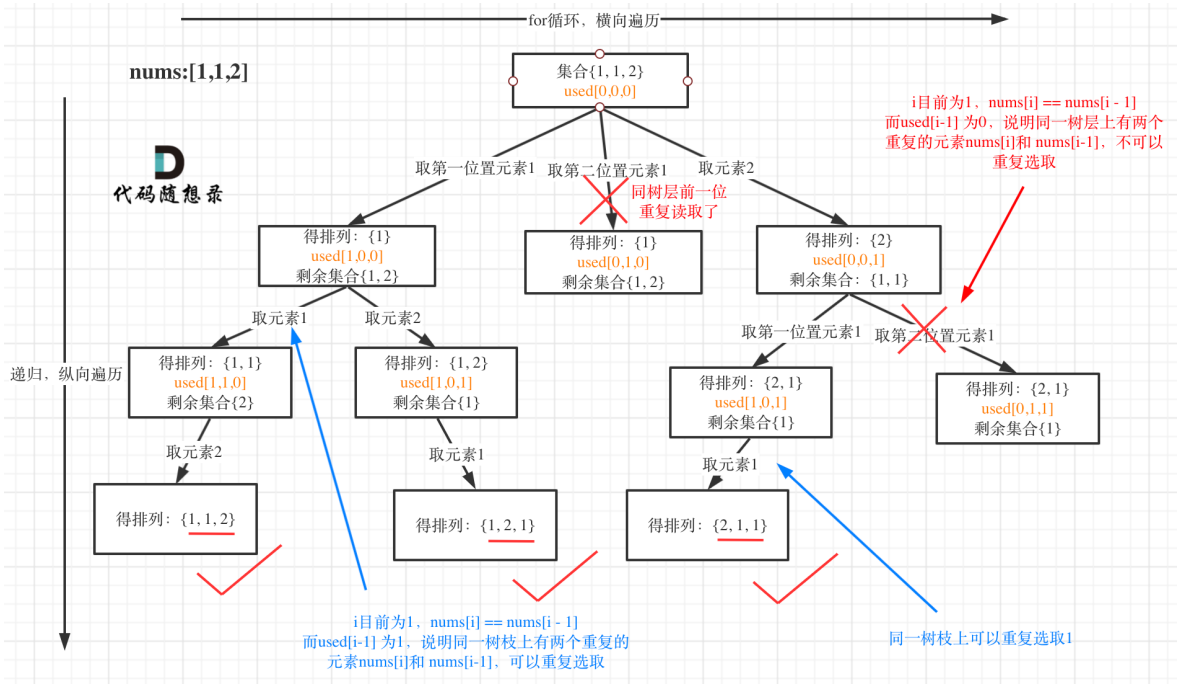
```
        backtracking(nums);
        return result;
    }
};
```

有重复元素

既要树层去重，又要防止重复选取元素

前者使用nums[i]==nums[i-1]&&used[i]=false判断

后者使用used[i]==true判断

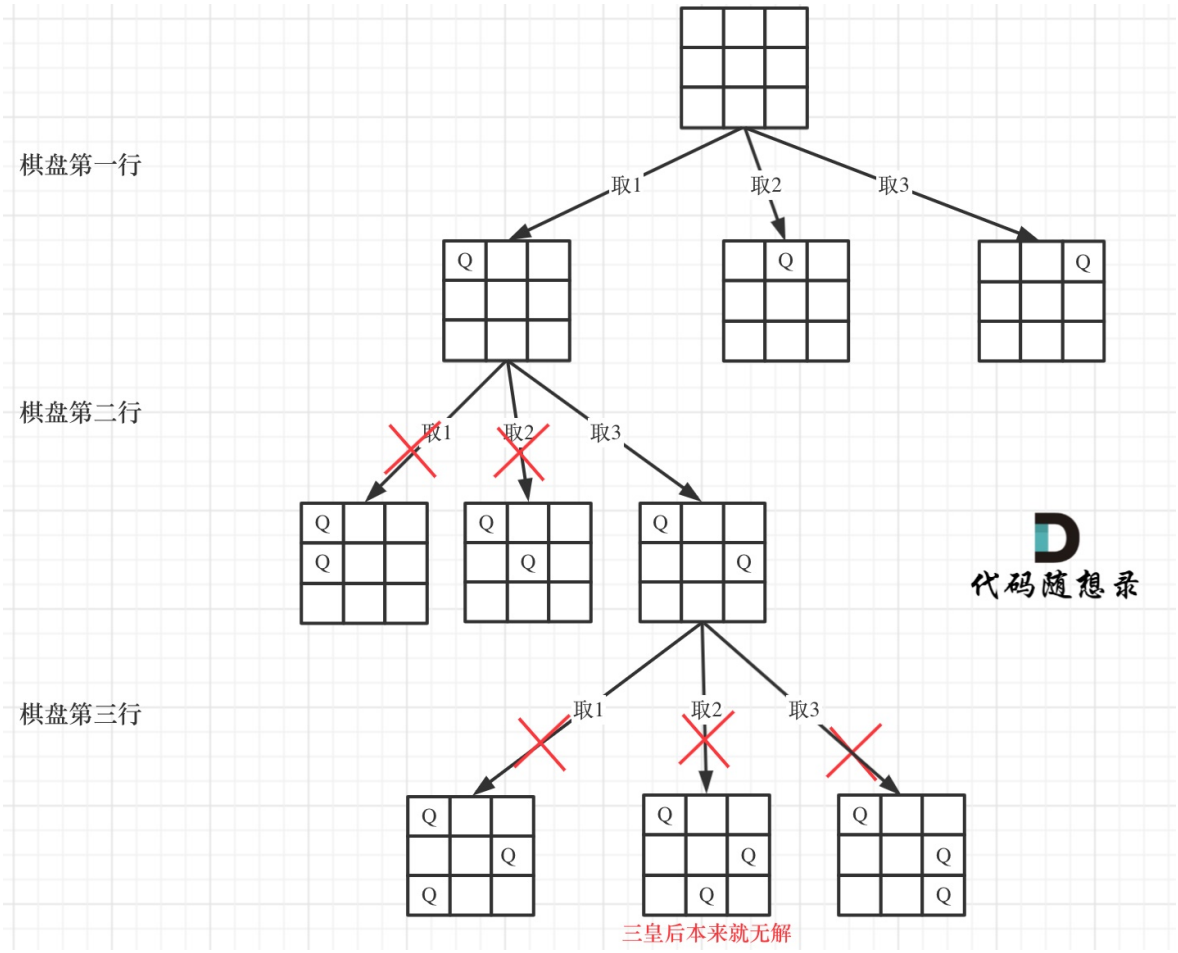



```
class Solution {
private:
    vector<int> path;
    vector<vector<int>> result;
    vector<bool> used;

    void backtracking(vector<int>& nums) {
        if (path.size() == nums.size()) {
            result.push_back(path);
            return;
        }

        for (int i = 0;i < nums.size();i++) {
            if (i > 0 && nums[i] == nums[i - 1] && used[i - 1] == false) {//数层去重
                continue;
```

```
            }
            else if (used[i] == true) {//树枝去重
                continue;
            }
            else {
                path.push_back(nums[i]);
                used[i] = true;

                backtracking(nums);

                path.pop_back();
                used[i] = false;
            }
        }
    }

public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        used.resize(nums.size(), false);
        sort(nums.begin(), nums.end());
        backtracking(nums);
        return result;
    }
};
```

N皇后

算法：本体难点二维数组回溯算法，使用row来控制开始的行数，类似于一维数组中的start。层上是row的列遍历的结果，枝上是row遍历的结果

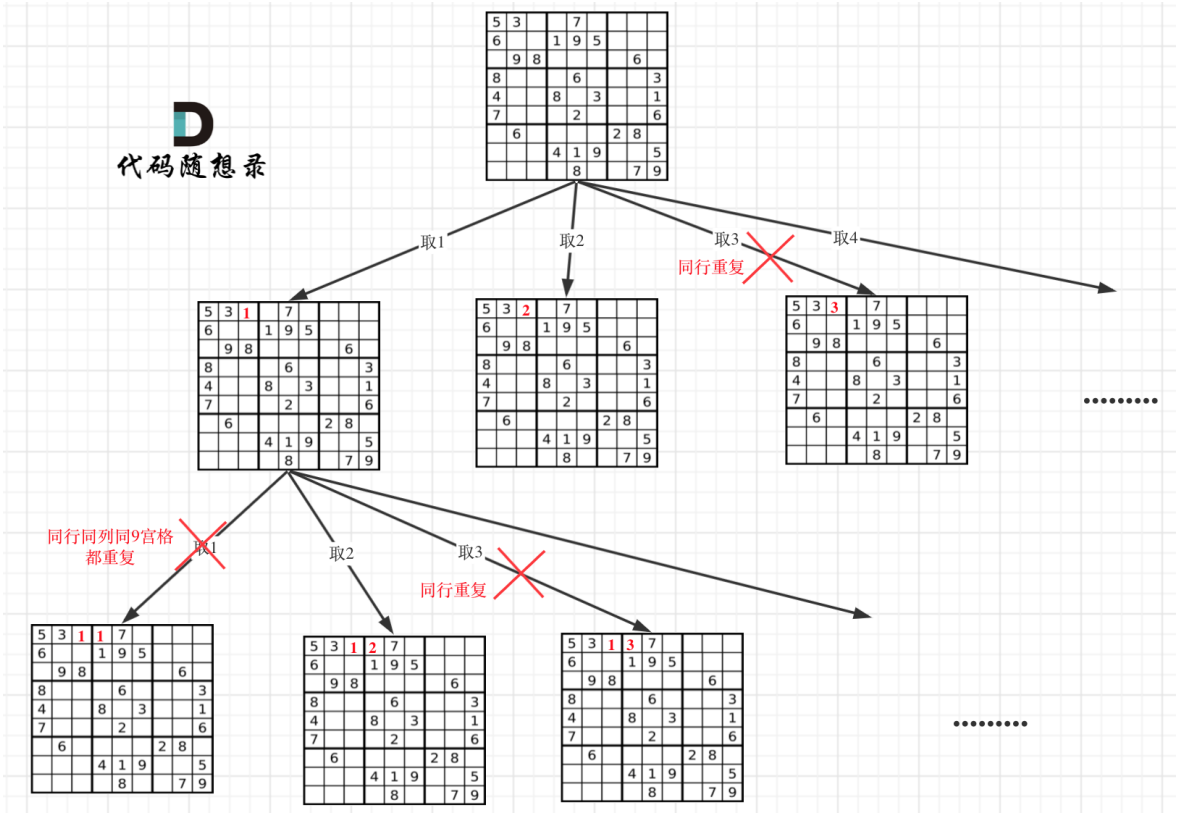



```
class Solution {
private:
    vector<vector<string>> result;
    vector<string> path;
    void backtracking( int row, int n) {
        if (row == n) {
            result.push_back(path);
            return;
        }

        for (int col = 0;col < n;col++) {
            if (isValid(row, col, n)) {
                path[row][col] = 'Q';
                backtracking(row + 1, n);
                path[row][col] = '.';
            }
            else {
                continue;
            }
        }
    }
```

```
    bool isValid(int row, int col, int n) {
        if (path[row][col] == 'Q') {
            return false;
        }

        int Direction_X[8] = {1,0,-1,0,1,1,-1,-1};
        int Direction_Y[8] = {0,1,0,-1,-1,1,1,-1};

        for (int i = 0;i < 8;i++) {
            int next_x = col + Direction_X[i];
            int next_y = row + Direction_Y[i];
            while(InRange(next_y, next_x, n)) {
                if (path[next_y][next_x] != 'Q') {
                    next_x += Direction_X[i];
                    next_y += Direction_Y[i];
                }
                else {
                    return false;
                }


            }
        }

        return true;
    }

    bool InRange(int row, int col, int n) {
        return row >= 0 && row < n && col >= 0 && col < n;
    }

public:
    vector<vector<string>> solveNQueens(int n) {
        path.resize(n, string(n, '.'));//初始化棋盘为kong
        backtracking(0, n);
        return result;
    }
};
```

解数独

只需要返回一种可能




```
class Solution {
private:

    bool backtracking(vector<vector<char>>& board) {

        for (int i = 0;i < board.size();i++) {
            for (int j = 0;j < board[0].size();j++) {
                if (board[i][j] == '.') {
                    for (char k = '1';k <= '9';k++) {
                        if (isValid(i, j, k, board)) {
                            board[i][j] = k;
                            if (backtracking(board)) {
                                return true;
                            }
                            board[i][j] = '.';
                        }
                    }
                    return false;//当遍历的字符1到字符9以后，没有返回true，说明不匹配
                }
            }
        }
    }

    bool isValid(int row, int col, char k, vector < vector<char>>& board) {
```

```
        for (int i = 0;i < 9;i++) {//行以及列的判断
            if (board[i][col] == k||board[row][i]==k) {
                return false;
            }
        }

        int startrow = (row / 3) * 3;
        int startcol = (col / 3) * 3;//九宫格中左上角的位置

        for (int i = startrow;i < startrow + 3;i++) {
            for (int j = startcol;j < startcol + 3;j++) {
                if (board[i][j] == k) {
                    return false;
                }
            }
        }

        return true;
    }

public:
    void solveSudoku(vector<vector<char>>& board) {
        backtracking(board);
    }
};
```

返回所有的可能

```
class Solution {
private:
    vector<vector<vector<char>>> result;
    vector <vector<char>> path;


    bool isValid(int row, int col, char k, vector < vector<char>>& board) {
        for (int i = 0;i < 9;i++) {//行以及列的判断
            if (board[i][col] == k || board[row][i] == k) {
                return false;
            }
        }

        int startrow = (row / 3) * 3;
        int startcol = (col / 3) * 3;//九宫格中左上角的位置

        for (int i = startrow;i < startrow + 3;i++) {
```

```
            for (int j = startcol;j < startcol + 3;j++) {
                if (board[i][j] == k) {
                    return false;
                }
            }
        }

        return true;
    }

    void backtracking(vector<vector<char>>& board) {

        for (int i = 0;i < board.size();i++) {
            for (int j = 0;j < board[0].size();j++) {
                if (board[i][j] == '.') {
                    for (char k = '1';k <= '9';k++) {
                        if (isValid(i, j, k, board)) {
                            board[i][j] = k;
                            backtracking(board);
                            board[i][j] = '.';
                        }
                        else {
                            return;
                        }
                    }
                }
            }
        }

        result.push_back(board);
    }
public:
    vector<vector<vector<char>>> solveSudoku(vector<vector<char>>& board) {
        backtracking(board);
        return result;
    }
};
```