DFS回溯算法

回溯算法，本质就是深度优先搜索，就是环顾看一下四周邻居，然后每个挨个DFS递归遍历一下，

只要清楚

1. 在哪个树形图上递归
2. 进递归之前做什么事情，从递归出来后做什么事情，以及递归停止条件是什么

那么算法实现就没什么难度了。

当然，组合问题就是上面讲的，对于棋盘问题，要遍历的东西都给你了，只需要知道2就ok了。

2是递归，遍历算法的灵魂，核心，需要对递归非常深入理解才能在写算法的过程知道算法执行到树的哪里了，

以及树的状态是什么样的，以及你收集结果的向量里面是什么样的，也就是代码和树形图在你脑子里有机融合在一起理解了，就好比我说一段下象棋的话“先把左边的炮向上走5格”，你脑子里就能把棋盘上的炮走到相应的位置，不管我说什么，你都能知道棋盘上发生了什么变化。

然后递归就是我说出的话，棋盘就是树形图，如果你能轻松理解递归在树形图上的执行过程，说明理解很到位了，

后面的遍历这类题目理解上最困难的地方基本就没了。

如果理解不了，无法把代码放在树形图上理解，虽然也能按照套路套模板写，但是如果哪里出错了，你基本很难排查错误，如果做对了，也肯定总觉得哪里不爽，没有打通任督二脉这题就结束了的感觉。

题型一：求不重复元素集合中的子集

77. 组合

中等 相关标签 相关企业 A文

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]]`

https://leetcode.cn/problems/combinations

思路分析：

题目意思就是1-n里面拿k个数，输入所有k个数的子集，不能重复。

另外，1-n中每个数字只有一个，不重复。

如果用for循环解决这个问题，先用第一层for拿第一个数字，然后再用第二层拿第二个数字，...，最后再用第k层for拿第k个数字，总共需要写k层for循环，假如k=100，显然不现实。

比如对于k=3的情形，三层for循环如下

```
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;
        }
    }
}
```

每层怎么拿的，这里就不详细赘述了，就是正常从一个集合里拿k个元素的拿法，高中就知道怎么拿了。

这里可以直接把上面的代码转为大致意思的递归算法，先给个大致雏形，

第一层，for (int i = 1; i <= n; i++) 其实就是挨个问候 1~n ， DFS（1~n）

第二层，for (int j = i + 1; j <= n; j++)，就是挨个问候i+1~n，DFS（i+1~n）

也就是上一层i固定下来后，开始问候下一层的时候，从i+1开始问候。

大致伪代码如下：

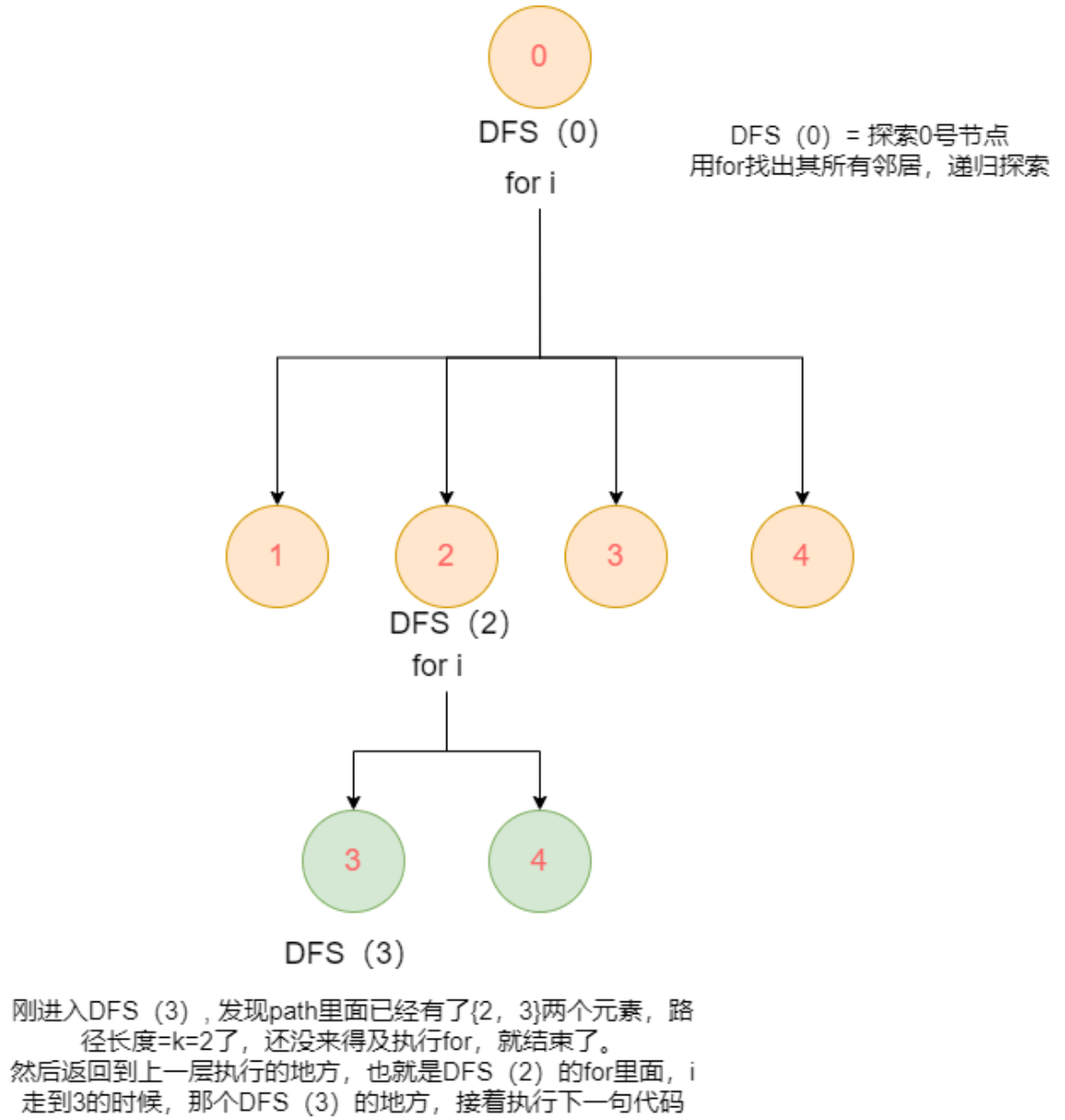
```
定义一些必要的东西
void DFS（审问张三）{
    判断是否问候到了第k层，到了就返回
    for（张三把他的同伙全供出来了）{
        把当前同伙标记为嫌疑人
        DFS（当前的同伙）
        把当前同伙从嫌疑人名单上删除
    }
}
```

上面的代码也可以用栈实现，就是审问root张三的时候，把他的同伙全部压栈，这是第一步，然后执行下面的循环

```
while(栈非空){
    弹出栈顶的那个同伙李四；
    用for把李四的下一层同伙全部压入栈；
}
```

不过这样做的话，由于这种方式每次入栈和出栈不清楚当前走到哪一层了，收集路径的逻辑没有递归清晰，不好维护需要的结果，估计也能实现，可能特别复杂，所以这里就不使用迭代形式写了。

下面这张图告诉你如何用树形图收集结果，就是收集走过的路径，也就是我们要的结果必须沿着深度方向访问，如果挨层访问完再访问下一层



然后这张图告诉你执行DFS的过程，

执行DFS（0）的时候，i 分别 遍历 1，2，3，4，然后当i遍历到2的时候，进入DFS（3），在DFS（3）里面又接着用for从3开始遍历。

到达当前这点的时候，也就是DFS进来这个节点的时候，需要把节点收集起来，因为这就是路径，出去的时候需要把节点弹出，然后才能收集其他路径。如果路径收集到k=2层的节点，也就是路径长度为k的时候，递推结束。

```
class Solution {
public:
    vector<int> path;
    vector<vector<int>> result;
    void DFS(int n, int k, int node){
        if (path.size()==k) {
            result.push_back(path);
            return;
        }
        for (int i = node + 1; i <= n; i++){
            path.push_back(i);
            DFS(n,k,i);
            path.pop_back();
```

```
        }
    }
    vector<vector<int>> combine(int n, int k) {
        DFS(n,k,0);
        return result;
    }
};
```

其实这题就是简单的DFS搜索，知道当前层节点是谁，然后每个节点的下一层是哪些节点，就很容易了。

78. 子集

中等 相关标签 相关企业 A文

给你一个整数数组 `nums` ，数组中的元素 **互不相同** 。返回该数组所有可能的子集（幂集）。

解集 **不能** 包含重复的子集。你可以按 **任意顺序** 返回解集。

示例 1：

> **输入：** nums = [1,2,3]
> **输出：** [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

示例 2：

> **输入：** nums = [0]
> **输出：** [[],[0]]

这题是收集所有子集，而不是包含k个元素的子集，因此递归条件就是走到底的时候，也就是路径长度达到最长的时候。

其次，路径收集的不是nums的下标，而是数组中的元素；

最后，注意nums下标不要越界。

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

class Solution {
public:
    vector<int> path;
    vector<vector<int>> result;

    void DFS(vector<int>& nums, int n, int node) {
        if (node == n) {
            return;
        }
        for (int i = node + 1; i <= n; i++) {
            path.push_back(nums[i-1]);  // 这里root设为0，其下一层为1，2，3，作为下标需要-1
                                        // 如果root设为-1，就没问题了
            result.push_back(path); // 在每个节点处都要收集路径
```

```
            DFS(nums, n, i);
            path.pop_back();
        }
    }

    vector<vector<int>> subsets(vector<int>& nums) {
        result.push_back(path); // 添加空集
        DFS(nums, nums.size(), 0);
        return result;
    }
};
```

题型二： 集合中元素重复，还是求子集，需要剪枝

90. 子集 II

中等 相关标签 相关企业

给你一个整数数组 `nums` ，其中可能包含重复元素，请你返回该数组所有可能的 子集（幂集）。

解集 **不能** 包含重复的子集。返回的解集中，子集可以按 **任意顺序** 排列。

示例 1：

> **输入：** nums = [1,2,2]
> **输出：** [[],[1],[1,2],[1,2,2],[2],[2,2]]

示例 2：

> **输入：** nums = [0]
> **输出：** [[],[0]]

思路分析：

简单来说就是遍历当前一层的所有邻居时，有的邻居重复了，不需要，需要定义一个只在本轮for循环中生效的一个记录，记录当前层遍历时已经用过的元素。那么这个记录需要定义在哪里呢，其实就是DFS函数中for循环前面定义，当DFS其中一个节点时，会创建一个函数栈，在这个函数栈里，会用for遍历所有邻居，创建的记录也只会在当前函数栈中有效。

另外，所有元素需要排序，也就是重复的元素一定要挤在一起，中间不能夹杂其他元素，

比如[2,1,2]找子集的过程中，第一层可以取2或者1，第二层中，遍历2的邻居得到1，2，遍历1的邻居得到2，

于是两层中可以得到包含两个元素的子集分别为[2,1] , [2,2] , [1,2] , 不难看出 [1,2] 和 [2,1] 重了，究其原因，就是

[2,1,2]中两个2中间夹杂了1，第一个2后面可以配第二个1，然后第二个1后面又能配一个2，导致重了，所以必须排序。

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

    void DFS(vector<int>& nums,int n,int node){
        if (node==n-1) return;
        unordered_set<int> uset;  // 记录DFS(node)访问的邻居
        for (int i = node + 1; i < n; i++){
            if (uset.find(nums[i])!=uset.end()) continue;
            path.push_back(nums[i]);
            result.push_back(path);
            uset.insert(nums[i]); // 记录已经访问过的邻居
            DFS(nums,n,i);
            // 在当前层中,只有路径在访问下个邻居的时候需要弹出,uset不需要弹出
            path.pop_back();
        }
    }
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        sort(nums.begin(),nums.end());
        result.push_back(path);
        DFS(nums,nums.size(),-1);
        return result;
    }
};
```

注意这里遍历的邻居就用下标0，1，2，3，，，n-1表示，没有用1，2，3，，，n表示

注意与之前写的代码的区别

题型三： 组合问题的衍生问题

衍生1：组合求和

40. 组合总和 II

中等 相关标签 相关企业 A文

给定一个候选人编号的集合 `candidates` 和一个目标数 `target` ，找出 `candidates` 中所有可以使数字和为 `target` 的组合。

`candidates` 中的每个数字在每个组合中只能使用 **一次** 。

注意：解集不能包含重复的组合。

示例 1:

> **输入：** candidates = `[10,1,2,7,6,1,5]`, target = `8`,
> **输出：**
> [
> [1,1,6],
> [1,2,5],
> [1,7],
> [2,6]
>]

示例 2:

> **输入：** candidates = [2,5,2,1,2], target = 5,
> **输出：**
> [
> [1,2,2],
> [5]
>]

思路分析：

还是需要枚举所有路径，即所有子集，只不过这次是求路径之和，也就是每个子集中的元素之和，如果和等于实现给定的target才会收集这条路径。既然意思明确了，代码仅仅需要在前面枚举所有子集的代码基础上稍微修改即可。

先写出枚举所有子集的代码

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

    void DFS(vector<int>& nums, int n, int node) {

        if (node == n-1) {
            return;
        }
        for (int i = node + 1; i < n; i++) {
            path.push_back(nums[i]);
            result.push_back(path);
            DFS(nums, n, i);
            path.pop_back();
        }
    }
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        DFS(candidates,candidates.size(),-1);
        return result;
    }
```

```
};
```

然后加入求路径和的逻辑

```
class Solution {
public:
    vector<int> path;
    int pathSum = 0;    // 新添加的变量，用于记录路径和
    vector<vector<int>> result;

    void DFS(vector<int>& nums, int target, int n, int node) {
        if (node == n-1) {
            return;
        }
        for (int i = node + 1; i < n; i++) {
            path.push_back(nums[i]);
            pathSum += nums[i];      // 写在path后面
            if (pathSum==target) result.push_back(path);
            DFS(nums, target, n, i);
            path.pop_back();
            pathSum -= nums[i];  // pathSum和path的行为保持同步
        }
    }
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        DFS(candidates,target,candidates.size(),-1);
        return result;
    }
};
```

然后加入去除同一层重复邻居的逻辑

```
class Solution {
public:
    vector<int> path;
    int pathSum = 0;
    vector<vector<int>> result;

    void DFS(vector<int>& nums, int target, int n, int node) {
        unordered_set<int> unset;   // 在当前层定义一个变量，仅存在于DFS(node)栈中
        if (node == n-1) {
            return;
        }
        for (int i = node + 1; i < n; i++) {
            if (unset.find(nums[i])!=unset.end()) continue; // 必须先检查这个邻居是
否重复
            unset.insert(nums[i]);
            path.push_back(nums[i]);
            pathSum += nums[i];
            if (pathSum==target) result.push_back(path);
            DFS(nums, target, n, i);
            path.pop_back();
            pathSum -= nums[i];  // pathSum和path的行为保持同步
        }
    }
```

```
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        sort(candidates.begin(),candidates.end());  // 必须排序，避免重复元素中间夹杂其他元素
        DFS(candidates,target,candidates.size(),-1);
        return result;
    }
};
```

最后优化代码，就是每条路径不一定需要走到底，当目标和满足后，后面的路不要再走了，继续看下一个邻居，

不过一定不要忘了把当前节点弹出。

还能加一句，就是和大于target的时候，后面的路也不需要走了，因为给的数都是正数，只会越加越大

```
if (pathSum==target) {
    result.push_back(path);
    path.pop_back();
    pathSum -= nums[i];
    continue;
    }
if (pathSum > target) {
    path.pop_back();
    pathSum -= nums[i];
    continue;
    }
```

216. 组合总和 III

中等 相关标签 相关企业 A文

找出所有相加之和为 `n` 的 `k` 个数的组合，且满足下列条件：

- 只使用数字1到9
- 每个数字 **最多使用一次**

返回 *所有可能的有效组合的列表* 。该列表不能包含相同的组合两次，组合可以以任何顺序返回。

示例 1:

> **输入:** ***k*** = 3, ***n*** = 7
> **输出:** [[1,2,4]]
> **解释:**
> 1 + 2 + 4 = 7
> 没有其他符合的组合了。

示例 2:

> **输入:** ***k*** = 3, ***n*** = 9
> **输出:** [[1,2,6], [1,3,5], [2,3,4]]
> **解释:**
> 1 + 2 + 6 = 9
> 1 + 3 + 5 = 9
> 2 + 3 + 4 = 9
> 没有其他符合的组合了。

思路分析：

这题更简单了，全集就是1-9这些数字，没有重复，不用去重了，然后是拿k个数相加，也就是必须走到路径长度为k的时候判断并返回。

```
class Solution {
public:
    vector<int> path;
    int pathSum = 0;    // 增加一个变量记录路径之和
    vector<vector<int>> result;
    void DFS(int n, int k, int node){
        if (path.size()==k) {
            if (pathSum == n) result.push_back(path); // 仅路径和为n的时候加入result
            return;
        }
        for (int i = node + 1; i <= 9; i++){
            path.push_back(i);
            pathSum += i;
            DFS(n,k,i);
            path.pop_back();
            pathSum -= i;
        }
    }
    vector<vector<int>> combinationSum3(int k, int n) {
```

```
        DFS(n,k,0);
        return result;
    }
};
```

实际上上面在路径长度到达k之前，如果pathSum大于等于n了，后面不需要走下去了，直接返回，由于上面的代码也没有运行超时，这里就不写了。

衍生2：分割回文串问题

和前面差不多，都是需要先求所有子集，然后对子集做一定的操作，题型三是求和，这里是判断是否是回文串。

131. 分割回文串

中等 相关标签 相关企业

给你一个字符串 `s`，请你将 `s` 分割成一些子串，使每个子串都是 **回文串** 。返回 `s` 所有可能的分割方案。

示例 1：

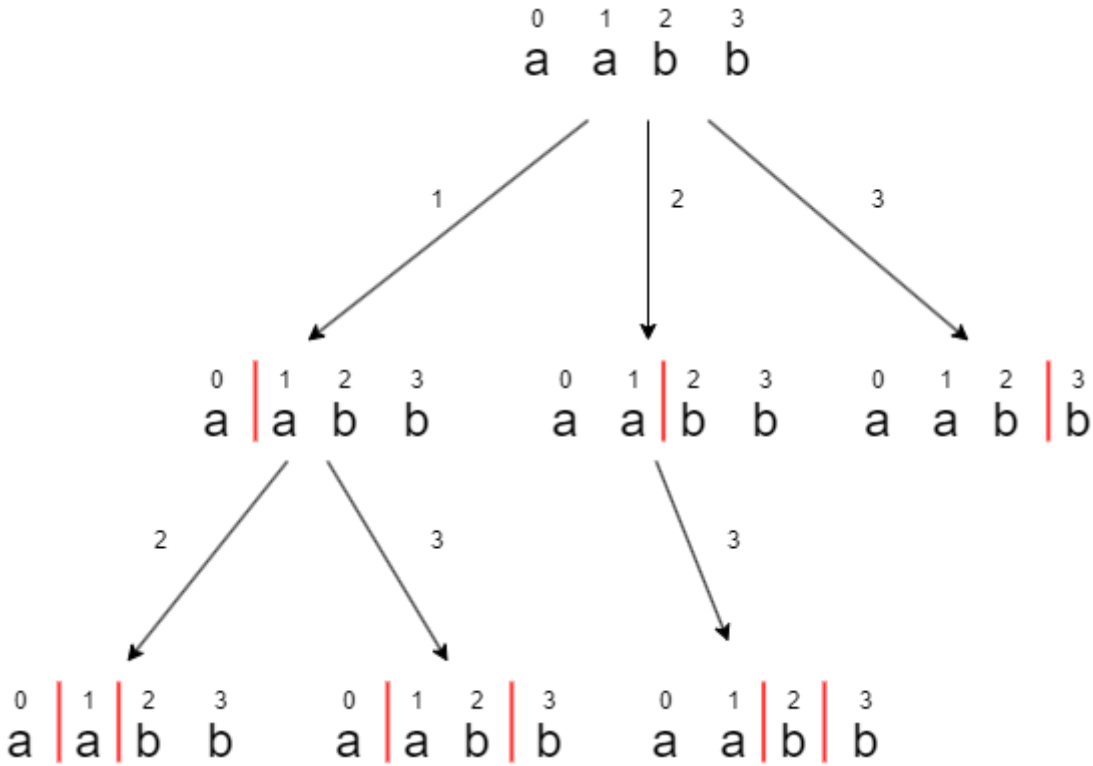
> **输入：** `s = "aab"`
> **输出：** `[["a","a","b"],["aa","b"]]`

示例 2：

> **输入：** `s = "a"`
> **输出：** `[["a"]]`

思路分析：

如下图，我们首先给出这个字符串的所有分割方案，是这样得出来的，首先分割时的每一刀可以落在下标1，2，3前面，所以你的刀的全集就是{1，2，3}，那么刀的分割方案就是这个刀集的全部子集，空集表示不分割。



也就是说上面每一个节点的位置，都需要判断这种分割是否可行，可行的话把分割后的子集加入result中。

整个算法可以分三步完成：

第一步：写出刀集的所有子集的DFS回溯算法，就是求子集的那个算法，这个是基础，包括前面求组合和的时候，

先把子集枚举出来，然后对每个子集做什么事之后再说。

```
class Solution {
public:
    vector<int> path;
    vector<vector<int>> result1;
    vector<vector<string>> result2;

    void DFS(string s, int node){
        if (node == s.size()-1) return;
        for (int i = node + 1; i < s.size(); i++){
            path.push_back(i);
            result1.push_back(path);
            DFS(s,i);
            path.pop_back();
        }
    }
    vector<vector<string>> partition(string s) {

    }
};
```

第二步：加入判断回文串的逻辑，如果当前分割方案是回文串，才把分割方案加入结果。

判断当前分割下是否是回文串，需要先看当前刀的前面是不是回文串，如果不是，后面不需要继续分割下去，

如果是，再看刀的后面是不是回文串，如果是，把path，也就是分割方案，加入结果。

如果是第2刀，也就是上面树形的下一层，由于第一刀的前面已经判断是回文串了，所以第二刀切的时候，也只需要看第2刀前面（第1刀后面）是不是回文串，然后再看第2刀后面是不是回文串。在代码里面，for遍历的邻居全都是当前刀放在哪里，而node就是上一层刀的位置。所以需要判断的是node和i之间是否是回文串，以及i之后是不是回文串。

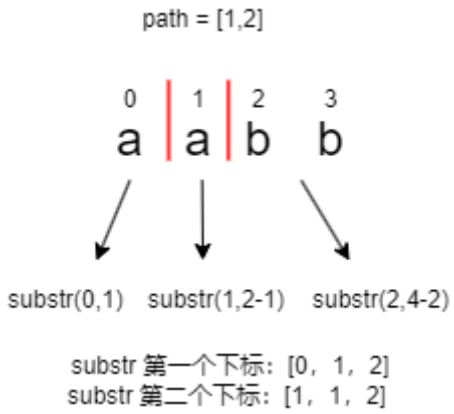
```
    bool isHuiWen(const string & s, int node, int i){
        for (int j = node, k = i -1; j < k; j++,k--){
            if (s[j]!=s[k]) return false;
        }
        return true;
    }
    bool isHuiWen(const string & s, int i){
        for (int j = i, k = s.size()-1; j < k; j++,k--){
            if (s[j]!=s[k]) return false;
        }
        return true;
    }
```

```
    vector<int> path;
    vector<vector<int>> result1;
    vector<vector<string>> result2;

    void DFS(string s, int node){
        if (node == s.size()-1) return;
        for (int i = node + 1; i < s.size(); i++){
            path.push_back(i);
            // 判断前串是不是回文
            if (isHuiWen(s,node,i)) {
                // 后串是回文加入结果,不是回文继续向后割
                if(isHuiWen(s,i)) {result1.push_back(path);}
            }
            // 前串不是回文,继续看下个邻居怎么割的
            else {
                path.pop_back();
                continue;
            }
            DFS(s,i);
            path.pop_back();
        }
    }
    vector<vector<string>> partition(string s) {

    }
};
```

第三步：我们把路径，也就是分割方案，转为分割后的真正字符串作为题目要求的结果形式返回，这步单独进行，不要和前面混在一起。所以我们的路径path第一个元素需要是0，需要把前面的path第一步初始化加入元素0.




```
vector<string> pathTransform(const string & s,const vector<int> & path){
    vector<string> path2str;
    vector<int> secondIndex;  // 记录第二个下标,也就是子串长度
    for (int i = 0; i < path.size()-1; i++){
        secondIndex.push_back(path[i+1]-path[i]);
    }
    secondIndex.push_back(s.size()-path[path.size()-1]);

    for (int i = 0; i < path.size(); i++){
        path2str.push_back(s.substr(path[i],secondIndex[i]));
    }

    return path2str;
}
```

```
void resultTransform(const vector<vector<int>> & result1,vector<vector<string>> &
result2){
    for (int i = 0; i < result1.size(); i++){
        result2.push_back(pathTransform(result1[i]));
    }
}
```

最后，补充一个不分割的情况，就是判断整个字符串是不是回文串，如果是就添加到result

```
    vector<vector<string>> partition(string s) {
        if (isHuiWen(s,0)) {
            vector<string> str;
            str.push_back(s);
            result2.push_back(str);
        }
        DFS(s,0);
        resultTransform(s,result1,result2);
        return result2;
    }
```

所有代码如下：

```
class Solution {
```

```
public:
    vector<int> path = {0};
    vector<vector<int>> result1;
    vector<vector<string>> result2;

    bool isHuiWen(const string & s, int node, int i){
        for (int j = node, k = i -1; j < k; j++,k--){
            if (s[j]!=s[k]) return false;
        }
        return true;
    }
    bool isHuiWen(const string & s, int i){
        for (int j = i, k = s.size()-1; j < k; j++,k--){
            if (s[j]!=s[k]) return false;
        }
        return true;
    }

    void DFS(string s, int node){
        if (node == s.size()-1) return;
        for (int i = node + 1; i < s.size(); i++){
            path.push_back(i);
            if (isHuiWen(s,node,i)) {
                // 后半串是回文加入结果,不是回文继续向后割
                if(isHuiWen(s,i)) {result1.push_back(path);}
            }
            // 前半串不是回文,继续看下个邻居怎么割的
            else {
                path.pop_back();
                continue;
            }
            DFS(s,i);
            path.pop_back();
        }
    }

    vector<string> pathTransform(const string & s,const vector<int> & path){
        vector<string> path2str;
        vector<int> secondIndex;  // 记录第二个下标,也就是子串长度
        for (int i = 0; i < path.size()-1; i++){
            secondIndex.push_back(path[i+1]-path[i]);
        }
        secondIndex.push_back(s.size()-path[path.size()-1]);

        for (int i = 0; i < path.size(); i++){
            path2str.push_back(s.substr(path[i],secondIndex[i]));
        }

        return path2str;
    }

    void resultTransform(const string & s,const vector<vector<int>> &
result1,vector<vector<string>> & result2){
        for (int i = 0; i < result1.size(); i++){
            result2.push_back(pathTransform(s,result1[i]));
```

```
        }
    }

    vector<vector<string>> partition(string s) {
        if (isHuiWen(s,0)) {
            vector<string> str;
            str.push_back(s);
            result2.push_back(str);
        }
        DFS(s,0);
        resultTransform(s,result1,result2);
        return result2;
    }
};
```

虽然执行效率不高，但是代码逻辑十分清晰，就是求解刀集的所有子集，然后对每个子集作为一种分割方案，看是不是回文串。最后把符合要求的路径，也就是分割方案，转换为题目要求的格式。

题型四： 棋盘问题

就是遍历棋盘的每一行，使用row把每一行记录下来，以便知道什么时候进入了最后一层，然后再使用一张二维数组记录遍历的时候的行为，并用来判断走下一行的时候改怎么走。

51. N 皇后

困难 相关标签 相关企业 A文

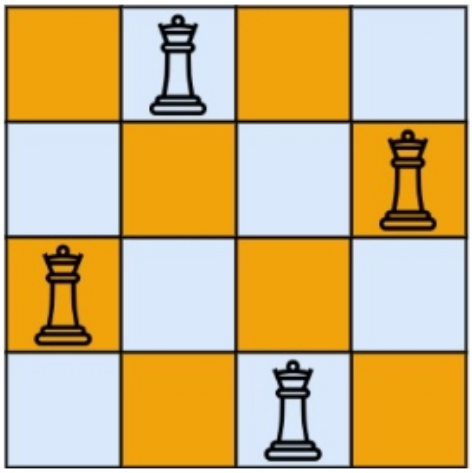
按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

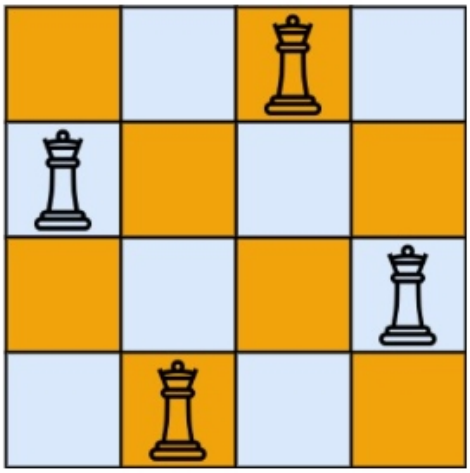
n 皇后问题 研究的是如何将 `n` 个皇后放置在 `n×n` 的棋盘上，并且使皇后彼此之间不能相互攻击。

给你一个整数 `n` ，返回所有不同的 **n 皇后问题** 的解决方案。

每一种解法包含一个不同的 **n 皇后问题** 的棋子放置方案，该方案中 `'Q'` 和 `'.'` 分别代表了皇后和空位。

示例 1：





> **输入：** `n = 4`
> **输出：** `[[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]`
> **解释：** 如上图所示，4 皇后问题存在两个不同的解法。

大致逻辑如下：

```
vector<vector<string>> result;
void DFS(int n, int row, vector<string> & chessboard){
    if (row==n){
        result.push_back(chessboard);
        return;
    }
    for (int col = 0; col < n; col++){
        if (isValid(n,row,col,chessboard)){
            chessboard[row][col] = 'Q';
            DFS(n,row+1,chessboard);
            chessboard[row][col] = '.';
        }
    }
}
```

就是用row记录当前遍历到哪一层了，chessboard记录当前棋盘状态，isVaild(row,col,chessboard)表示我在棋盘的当前这个位置能不能放棋子。当DFS深入递归完成后，回溯到当前层，则在试探当前层的下一个位置之前需要把刚刚下的棋子拿走。

对于isVaild(row,col,chessboard)函数，就是已知棋盘前面几行都落定棋子的情况下，已知chessboard前面几行的状态下，判断当前位置是否能下。

有两种简单的实现方式，第一种就是代码随想录里面，用for检查纵向，斜45°和斜-45°方向，是否存在棋子，如果存在就是冲突了。(水平方向我们只下一个棋子，不存在冲突情况)

另一种实现方式就是再定义一个0-1棋盘，初始全为1，表示每个格子都能下棋，然后在某个位置下棋后，会让这个棋子的前面讲到的各种方向上的位置设为false，检查合法性就是检查当前位置是否为true。

第一种方式，这一行的每一个位置都需要检查冲突，都会遍历前面几行的棋盘，时间成本大，第二种方式，这一行判断冲突几乎没有成本，虽然下棋后会遍历后面的盘面设置不能下的点，但是，这一行下棋的机会很小了，因为大部分位置都不能走了。所以时间成本很低，但是空间成本高，多了一个棋盘的存储。

听着似乎很正确，实际上存在一个问题，就是回溯的时候，检查合法性的那个棋盘怎么撤销标记，只能撤销走这一行前面那个位置的时候加的标记，不能撤销前面行走的时候加的标记，这是个难题。因此，还是选第一种实现方式好了。

下面写代码的时候，画个棋盘，然后检查上方，右上方，左上方是否有棋子，非常清晰。

```
bool isValid(int n, int row, int col, vector<string>& chessboard) {
    // 检查列
    for (int i = 0; i < row; i++) {
        if (chessboard[i][col] == 'Q') {
            return false;
        }
    }
    // 检查 45度角是否有皇后
    for (int i = row - 1, j = col - 1; i >=0 && j >= 0; i--, j--) {
        if (chessboard[i][j] == 'Q') {
            return false;
        }
    }
    // 检查 -45度角是否有皇后
    for(int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
        if (chessboard[i][j] == 'Q') {
            return false;
        }
    }
    return true;
}
```