LeetCode 17. 电话号码的字母组合

17. 电话号码的字母组合

难度 中等 608

给定一个仅包含数字 `2-9` 的字符串，返回所有它能表示的字母组合。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

示例:

> **输入：** "23"
> **输出：** ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

(递归) $O(4^l)$

1. 可以通过手工或者循环的方式预处理每个数字可以代表哪些字母。
2. 通过递归尝试拼接一个新字母。
3. 递归到目标长度，将当前字母串加入到答案中。

注意，有可能数字串是空串，需要特判。

时间复杂度

- 由于使用了递归的方式，时间复杂度与答案个数相同。
- 设数字串长度为 l，则最坏时间复杂度为 $O(4^l)$。

```
class Solution {
public:
    vector<char> digit[10];
    vector<string> res;

    void init() {
        char cur = 'a';
        for (int i = 2; i < 10; i++) {
            for (int j = 0; j < 3; j++)
                digit[i].push_back(cur++);
            if (i == 7 || i == 9)
                digit[i].push_back(cur++);
        }
    }
```

```
    void solve(string digits, int d, string cur) {
        if (d == digits.length()) {
            res.push_back(cur);
            return;
        }

        int cur_num = digits[d] - '0';

        for (int i = 0; i < digit[cur_num].size(); i++)
            solve(digits, d + 1, cur + digit[cur_num][i]);
    }

    vector<string> letterCombinations(string digits) {
        if (digits == "")
            return res;
        init();
        solve(digits, 0, "");
        return res;
    }
};
*************************************************************************
/*
state{}
for 每个数字
    for c = 当前数字的所有备选字母
        for s = state{}中的所有字符串
            s += c
            将s加入到新的集合中去
*/
class Solution {
public:
    string chars[8] = {"abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};

    vector<string> letterCombinations(string digits) {
        if(digits.empty()) return vector<string>();

        vector<string> state(1,"");
        for(auto u : digits)
        {
            vector<string> now;
            for(auto c : chars[u - '2'])
                for(auto s : state)
                    now.push_back(s + c);
            state = now;
        }
        return state;
    }
};
```

LeetCode 79.单词搜索

79. 单词搜索

难度 中等 337

给定一个二维网格和一个单词，找出该单词是否存在于网格中。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中"相邻"单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

示例:

```
board =
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]

给定 word = "ABCCED", 返回 true.
给定 word = "SEE", 返回 true.
给定 word = "ABCB", 返回 false.
```

算法

(DFS) $O(n^2 3^k)$

在深度优先搜索中，最重要的就是考虑好搜索顺序。

我们先枚举单词的起点，然后依次枚举单词的每个字母。
过程中需要将已经使用过的字母改成一个特殊字母，以避免重复使用字符。

时间复杂度分析：单词起点一共有 n^2 个，单词的每个字母一共有上下左右四个方向可以选择，但由于不能走回头路，所以除了单词首字母外，仅有三种选择。所以总时间复杂度是 $O(n^2 3^k)$。

```
/*
1.枚举起点
2.从起点开始，依次搜索下一个点的位置
3.在枚举过程中，要保证和目标单词匹配
*/
class Solution {
public:
    bool exist(vector<vector<char>>& board, string str) {
        for (int i = 0; i < board.size(); i ++ )
            for (int j = 0; j < board[i].size(); j ++ )
                if (dfs(board, str, 0, i, j))
                    return true;
        return false;
    }

    bool dfs(vector<vector<char>> &board, string &str, int u, int x, int y) {
        if (board[x][y] != str[u]) return false;
        if (u == str.size() - 1) return true;
        int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
        char t = board[x][y];
        board[x][y] = '*';
        for (int i = 0; i < 4; i ++ ) {
```

```
            int a = x + dx[i], b = y + dy[i];
            if (a >= 0 && a < board.size() && b >= 0 && b < board[a].size()) {
                if (dfs(board, str, u + 1, a, b)) return true;
            }
        }
        board[x][y] = t;
        return false;
    }
};
*************************************************************************
class Solution {
public:
    int n,m;
    int dx[4] = {-1,0,1,0},dy[4] = {0,1,0,-1};
    bool exist(vector<vector<char>>& board, string word) {
        if(board.empty() || board[0].empty()) return false;//矩阵是空的
        n = board.size(),m = board[0].size();

        for(int i = 0;i < n;i ++)   //枚举起点
            for(int j = 0;j < m;j ++)
                if(dfs(board,i,j,word,0))
                    return true;
        return false;
    }
    //x，y是当前走到哪个格子了，word是要找的目标单词，u是目标单词的第几位
    bool dfs(vector<vector<char>> &board,int x,int y,string &word,int u)
    {
        if(board[x][y] != word[u]) return false;
        if(u == word.size() - 1) return true;//所有位都匹配完且成功了

        board[x][y] = '.';
        for(int i = 0;i < 4;i ++)
        {
            int a = x + dx[i],b = y + dy[i];
            if(a >= 0 && a < n && b >= 0 && b < m)
                if(dfs(board,a,b,word,u + 1))
                    return true;
        }
        board[x][y] = word[u];//回溯，恢复现场
        return false;
    }
};
```

LeetCode 46. 全排列

46. 全排列

难度 中等 560

给定一个**没有重复**数字的序列，返回其所有可能的全排列。

示例:

```
输入: [1,2,3]
输出:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

算法

(回溯) $O(n \times n!)$

我们从前往后，一位一位枚举，每次选择一个没有被使用过的数。
选好之后，将该数的状态改成"已被使用"，同时将该数记录在相应位置上，然后递归。
递归返回时，不要忘记将该数的状态改成"未被使用"，并将该数从相应位置上删除。

时间复杂度分析：

- 搜索树中最后一层共 $n!$ 个叶节点，在叶节点处记录方案的计算量是 $O(n)$，所以叶节点处的计算量是 $O(n \times n!)$。
- 搜索树一共有 $n! + \frac{n!}{2!} + \frac{n!}{3!} + \ldots = n!(1 + \frac{1}{2!} + \frac{1}{3!} + \ldots) \leq n!(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots) = 2n!$ 个内部节点，在每个内部节点内均会for循环 n 次，因此内部节点的计算量也是 $O(n \times n!)$。 所以总时间复杂度是 $O(n \times n!)$。

```cpp
/*
两种不同方法:
枚举每个位置上放哪个数


枚举每个数放到哪个位置上
*/
class Solution {
public:
    vector<vector<int>> ans;//所有方案
    vector<bool> st;
    vector<int> path;//当前方案

    vector<vector<int>> permute(vector<int>& nums) {

        for (int i = 0; i < nums.size(); i ++ ) st.push_back(false);//初始化
        dfs(nums, 0);
        return ans;
    }

    void dfs(vector<int> &nums, int u)//u是第几个数
    {
```

```
        if (u == nums.size())
        {
            ans.push_back(path);
            return ;
        }

        for (int i = 0; i < nums.size(); i ++ )
            if (!st[i])
            {
                st[i] = true;
                path.push_back(nums[i]);
                dfs(nums, u + 1);
                st[i] = false;
                path.pop_back();
            }
    }
};
```

LeetCode 47. 全排列 II

47. 全排列 II

难度 中等 239

给定一个可包含重复数字的序列，返回所有不重复的全排列。

示例:

```
输入：[1,1,2]
输出：
[
  [1,1,2],
  [1,2,1],
  [2,1,1]
]
```

算法

(回溯) $O(n!)$

由于有重复元素的存在，这道题的枚举顺序和 Permutations 不同。

1. 先将所有数从小到大排序，这样相同的数会排在一起；
2. 从左到右依次枚举每个数，每次将它放在一个空位上；
3. 对于相同数，我们人为定序，就可以避免重复计算：我们在dfs时记录一个额外的状态，记录上一个相同数存放的位置 $start$，我们在枚举当前数时，只枚举 $start+1, start+2, \dots, n$ 这些位置。
4. 不要忘记递归前和回溯时，对状态进行更新。

时间复杂度分析：搜索树中最后一层共 $n!$ 个节点，前面所有层加一块的节点数量相比于最后一层节点数是无穷小量，可以忽略。且最后一层节点记录方案的计算量是 $O(n)$，所以总时间复杂度是 $O(n \times n!)$。

```
class Solution {
public:
    vector<bool> st;
    vector<int> path;
    vector<vector<int>> ans;
```

```
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        st = vector<bool>(nums.size(), false);
        path = vector<int>(nums.size());
        dfs(nums, 0, 0);
        return ans;
    }
    void dfs(vector<int>& nums, int u, int start)//u表示枚举到了哪个数字，start表示从哪个位置从哪儿开始搜
    {
        if (u == nums.size())
        {
            ans.push_back(path);
            return;
        }

        for (int i = start; i < nums.size(); i ++ )
            if (!st[i])
            {
                st[i] = true;
                path[i] = nums[u];
                if (u + 1 < nums.size() && nums[u + 1] != nums[u])
                    dfs(nums, u + 1, 0);
                else
                    dfs(nums, u + 1, i + 1);
                st[i] = false;
            }
    }
};
```

LeetCode 78. 子集

78. 子集

难度 中等 490

给定一组**不含重复元素**的整数数组 *nums*，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集。

示例:

```
输入: nums = [1,2,3]
输出:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

算法

(集合的二进制表示) $O(2^n n)$

假设集合大小是 n，我们枚举 $0 \dots 2^n - 1$，一共 2^n 个数。
每个数表示一个子集，假设这个数的二进制表示的第 i 位是1，则表示该子集包含第 i 个数，否则表示不包含。

另外，如果 $n \geq 30$，则 $2^n \geq 10^9$，肯定会超时，所以我们可以断定 $n \leq 30$，可以用 `int` 型变量来枚举。

时间复杂度分析：一共枚举 2^n 个数，每个数枚举 n 位，所以总时间复杂度是 $O(2^n n)$。

```
/*
二进制表示
0~n的每一位上的0、1表示这个数是否被选了！！！

i >> j & 1 ----i的二进制表示的第j位是否为1 ！！！
1 << n -----表示2的n次方！！！
*/
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> res;
        int n = nums.size();
        for (int i = 0; i < (1 << n); i ++ )
        {
            vector<int> temp;
            for (int j = 0; j < n; j ++ )
                if (i >> j & 1) //i的二进制表示的第j位是否为1
```

```
                    temp.push_back(nums[j]);
                res.push_back(temp);
            }
            return res;
        }
};
```

LeetCode 90. 子集 II

90. 子集 II

难度 中等 167

给定一个可能包含重复元素的整数数组 ***nums***，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集。

示例:

```
输入: [1,2,2]
输出:
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

算法

(暴力枚举) $O(n2^n)$

为了方便处理，我们先将数组排序，这样相同元素就会排在一起。

然后暴力搜索所有方案，搜索顺序是这样的：
我们先枚举每个不同的数，枚举到数 x 时，我们再求出 x 的个数 k，然后我们枚举在集合中放入 $0, 1, 2, \dots k$ 个 x，共 $k+1$ 种情况。
当枚举完最后一个数时，表示我们已经选定了一个集合，将该集合加入答案中即可。

时间复杂度分析：不同子集的个数最多有 2^n 个，另外存储答案时还需要 $O(n)$ 的计算量，所以时间复杂度是 $O(n2^n)$。

```
/*
每个数可多选
*/
class Solution {
public:
    vector<vector<int>> ans;
    vector<int> path;

    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        sort(nums.begin(), nums.end());
```

```
        dfs(0, nums);
        return ans;
    }

    void dfs(int u, vector<int>&nums)
    {
        if (u == nums.size())
        {
            ans.push_back(path);
            return;
        }
        int k = u;//计算一段区间一共有多少个相同的数字
        while (k < nums.size() && nums[k] == nums[u]) k ++ ;
        dfs(k, nums);
        for (int i = u; i < k; i ++ )
        {
            path.push_back(nums[i]);
            dfs(k, nums);
        }
        path.erase(path.end() - (k - u), path.end());//恢复现场
    }
};
*****************************************************************************
class Solution {
public:
    vector<vector<int>> ans;
    vector<int> path;

    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        sort(nums.begin(),nums.end());
        dfs(nums,0);
        return ans;
    }

    void dfs(vector<int> &nums,int u)
    {
        if(u == nums.size())
        {
            ans.push_back(path);
            return;
        }

        int k = 0;
        while(u + k < nums.size() && nums[u + k] == nums[u]) k ++;

        for(int i = 0; i <= k;i ++)
        {
            dfs(nums,u + k);
            path.push_back(nums[u]);
        }

        for(int i = 0;i <= k;i ++)path.pop_back();

    }
};
```

LeetCode 216. 组合总和 III

216. 组合总和 III

难度 中等 87

找出所有相加之和为 ***n*** 的 ***k*** 个数的组合。组合中只允许含有 1 - 9 的正整数，并且每种组合中不存在重复的数字。

说明：

- 所有数字都是正整数。
- 解集不能包含重复的组合。

示例 1:

```
输入: k = 3, n = 7
输出: [[1,2,4]]
```

示例 2:

```
输入: k = 3, n = 9
输出: [[1,2,6], [1,3,5], [2,3,4]]
```

(DFS) $O(C_9^k \times k)$

暴力搜索出所有从9个数中选 k 个的方案，记录所有和等于 n 的方案。
为了避免重复计数，比如 {1, 2, 3} 和 {1, 3, 2} 是同一个集合，我们对集合中的数定序，每次枚举时，要保证同一方案中的数严格递增，即如果上一个选的数是 x，那我们从 $x+1$ 开始枚举当前数。

时间复杂度分析：从9个数中选 k 个总共有 C_9^k 个方案，将每个方案记录下来需要 $O(k)$ 的时间，所以时间复杂度是 $O(C_9^k \times k)$。

```
/*
依次枚举每个数从哪个位置上选
dfs(枚举到了第几个数字，当前选择的所有数的和，开始枚举的位置)
倒着枚举
*/

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

    vector<vector<int>> combinationSum3(int k, int n) {
        dfs(k, n, 1);
        return ans;
    }

    void dfs(int k, int n, int start)
    {
        if (!k)
        {
            if (!n) ans.push_back(path);
            return;
```

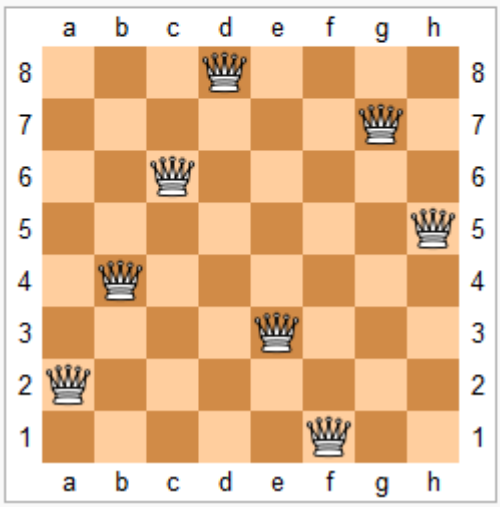
```
        }

        for (int i = start; i <= 10 - k; i ++ )
            if (n >= i)
            {
                path.push_back(i);
                dfs(k - 1, n - i, i + 1);
                path.pop_back();
            }
    }
};
*************************************************************************
class Solution {
public:
    vector<vector<int>> ans;
    vector<int> path;
    vector<vector<int>> combinationSum3(int k, int n) {
        dfs(k,1,n);
        return ans;
    }

    void dfs(int k,int start,int n)
    {
        if(!k)//枚举完所有数
        {
            if(!n) ans.push_back(path);//总和也是倒着来
            return;
        }

        for(int i = start;i <= 9;i ++)
        {
            path.push_back(i);
            dfs(k - 1,i + 1,n - i);
            path.pop_back();
        }
    }
};
```

LeetCode 52. N皇后 II

52. N皇后 II

难度 困难 101

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。



One solution to the eight queens puzzle

上图为 8 皇后问题的一种解法。

给定一个整数 *n*，返回 *n* 皇后不同的解决方案的数量。

示例:

```
输入：4
输出：2
解释：4 皇后问题存在如下两个不同的解法。
[
 [".Q..",  // 解法 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // 解法 2
  "Q...",
  "...Q",
  ".Q.."]
```

```cpp
//暴力
class Solution {
public:
    int ans;
    vector<bool> row, col, diag, anti_diag;

    int totalNQueens(int n) {
        row = col = vector<bool>(n, false);
        diag = anti_diag = vector<bool>(2 * n, false);
        ans = 0;
        dfs(0, 0, 0, n);
        return ans;
    }
```

```
    void dfs(int x, int y, int s, int n)
    {
        if (y == n) x ++ , y = 0;
        if (x == n)
        {
            if (s == n) ++ ans;
            return ;
        }

        dfs(x, y + 1, s, n);
        if (!row[x] && !col[y] && !diag[x + y] && !anti_diag[n - 1 - x + y])
        {
            row[x] = col[y] = diag[x + y] = anti_diag[n - 1 - x + y] = true;
            dfs(x, y + 1, s + 1, n);
            row[x] = col[y] = diag[x + y] = anti_diag[n - 1 - x + y] = false;
        }
    }
};
********************************************************************************
******
/*
依次枚举每一行皇后的位置，且不与上面冲突
精确覆盖问题，Dancing Link算法
*/
class Solution {
public:

    int ans = 0,n;
    vector<bool> col,d,ud;
    int totalNQueens(int _n) {
        n = _n;//把n做成全局变量
        col = vector<bool>(n);
        d = ud = vector<bool>(n * 2);
        dfs(0);

        return ans;
    }

    void dfs(int u)
    {
        if(u == n )//找到了一个方案
        {
            ans ++;
            return;
        }

        for(int i = 0;i < n;i ++)
            if(!col[i] && !d[u + i] && !ud[u - i + n])
            {
                col[i] = d[u + i] = ud[u - i + n] = true;
                dfs(u + 1);
                col[i] = d[u + i] = ud[u - i + n] = false;
            }
    }
};
```

LeetCode 37. 解数独

37. 解数独

难度 困难 341

编写一个程序，通过已填充的空格来解决数独问题。

一个数独的解法需遵循如下规则：

1. 数字 `1-9` 在每一行只能出现一次。
2. 数字 `1-9` 在每一列只能出现一次。
3. 数字 `1-9` 在每一个以粗实线分隔的 `3x3` 宫内只能出现一次。

空白格用 `'.'` 表示。

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

一个数独。

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5

(深度优先搜索——DFS)

1. 首先按照Valid Sudoku的方法，预处理出 `col` 、 `row` 和 `squ` 数组。
2. 从 `(0,0)` 位置开始深度优先搜索，遇到 `.` 时，枚举可以填充的数字，然后判重并加入 `col` 、 `row` 和 `squ` 数组中。
3. 如果成功到达结尾，则返回 `true` ，告知搜索可以终止。

```
class Solution {
public:
    bool dfs(int x, int y, vector<vector<char>>& board,
             vector<int>& row, vector<int>& col, vector<int>& squ) {
```

```
        if (y == 9) {
            x++;
            y = 0;
        }
        if (x == 9)
            return true;
        if (board[x][y] == '.') {
            for (int i = 1; i <= 9; i++)
                if (! (
                    (row[x] & (1 << i)) ||
                    (col[y] & (1 << i)) ||
                    (squ[(x / 3) * 3 + (y / 3)] & (1 << i))
                    )
                   ) {
                    row[x] |= (1 << i);
                    col[y] |= (1 << i);
                    squ[(x / 3) * 3 + (y / 3)] |= (1 << i);
                    board[x][y] = i + '0';
                    if(dfs(x, y + 1, board, row, col, squ))
                        return true;
                    board[x][y] = '.';
                    row[x] -= (1 << i);
                    col[y] -= (1 << i);
                    squ[(x / 3) * 3 + (y / 3)] -= (1 << i);
                }
        }
        else {
            if (dfs(x, y + 1, board, row, col, squ))
                return true;
        }
        return false;
    }

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

        vector<int> row(9), col(9), squ(9);
        for (int i = 0; i < 9; i++)
            for (int j = 0; j < 9; j++) {
                if (board[i][j] == '.')
                    continue;
                int num = board[i][j] - '0';
                row[i] |= (1 << num);
                col[j] |= (1 << num);
                squ[(i / 3) * 3 + (j / 3)] |= (1 << num);
            }

        dfs(0, 0, board, row, col, squ);
    }
};
**********************************************
/*从前往后枚举每个空格该填哪个数
状态：row[9][9],col[9][9],cell[3][3][9]
精确覆盖问题，Dancing Links算法
*/
class Solution {
public:
    bool row[9][9] = {0},col[9][9] = {0},cell[3][3][9] = {0};//初始化
    void solveSudoku(vector<vector<char>>& board) {
```

```
        for(int i = 0;i < 9;i ++)
            for(int j = 0;j < 9;j ++)
            {
                char c = board[i][j];
                if(c != '.')
                {
                    int t = c - '1';
                    row[i][t] = col[j][t] = cell[i / 3][j / 3][t] = true;
                }
            }

        dfs(board,0,0);//左上角开始走
    }

    bool dfs(vector<vector<char>> &board,int x,int y)//board是引用类型
    {
        if(y == 9) x ++ ,y = 0;//出界就去下一行
        if(x == 9) return true;//整个数都做完了
        if(board[x][y] != '.') return dfs(board,x,y + 1);//已经填了数，调到下一个

        for(int i = 0;i < 9;i ++)
        {
            if(!row[x][i] && !col[y][i] && !cell[x / 3][y / 3][i])
            {
                board[x][y] = '1' + i;//更新状态
                row[x][i] = col[y][i] = cell[x / 3][y / 3][i] = true;
                if(dfs(board,x,y + 1)) return true;
                row[x][i] = col[y][i] = cell[x / 3][y / 3][i] = false;
                board[x][y] = '.';//恢复状态
            }
        }
        return false;
    }
};
```

LeetCode 473. 火柴拼正方形

473. 火柴拼正方形

难度 中等 72

还记得童话《卖火柴的小女孩》吗？现在，你知道小女孩有多少根火柴，请找出一种能使用所有火柴拼成一个正方形的方法。不能折断火柴，可以把火柴连接起来，并且每根火柴都要用到。

输入为小女孩拥有火柴的数目，每根火柴用其长度表示。输出即为是否能用所有的火柴拼成正方形。

示例 1:

```
输入: [1,1,2,2,2]
输出: true

解释: 能拼成一个边长为2的正方形，每边两根火柴。
```

示例 2:

```
输入: [3,3,3,3,4]
输出: false

解释: 不能用所有火柴拼成一个正方形。
```

注意:

1. 给定的火柴长度和在 `0` 到 `10^9` 之间。
2. 火柴数组的长度不超过15。

算法

(深度优先搜索) $O(4^n)$

很明显正方形四条边长一样。
因此我们可以求出边长。
接下来设置一个SubSum的数组，长度为4，表明当前状态下每条边长的长度。
我们在深度优先搜索的过程中枚举每根木棍放在第几个边长上。 最后只要前三个边长的长度等于给定的边长，那么剩下的第四个边长一定也和一样。（当然木棍总和如果不是4的倍数要先剔除）
一点优化： 深度优先搜索在寻找的过程中， 深度尽可能的浅时间比较短， 那么我们就将长的木棍放在前面，短的放在后面即可。

时间复杂度分析：每次枚举长度为4，共有n层，故复杂度为$O(4^n)$

```
bool cmp(int x,int y){
    return x>y;
}
class Solution {
public:
    bool makesquare(vector<int>& nums) {
        if (nums.size() == 0) return false;
        int sum = 0;
        for (int i = 0;i<nums.size();++i){
            sum += nums[i];
        }
        int len = sum/4;
```

```
        if (len*4 != sum)
            return false;


        sort(nums.begin(),nums.end(),cmp);
        vector<int> SubSum(4);
        bool res = dfs(nums,SubSum,len,0);
        return res;

    }
    bool dfs(vector<int>&nums,vector<int>SubSum,int len,int index){
        if (SubSum[0] == len && SubSum[1] == len && SubSum[2] == len){
            return true;
        }
        for (int i = 0;i<4;++i){
            if (SubSum[i]+nums[index]> len) continue;
            SubSum[i] += nums[index];
            if (dfs(nums,SubSum,len,index+1)) return true;
            SubSum[i] -= nums[index];
        }
        return false;


    }
};
**********************************************************************************
/*
依次构造正方形的每条边
剪枝：
1.从大到小枚举所有边
2.每条内部的木棒长度规定成从大到小
3.如果当前木棒拼接失败，则跳过接下来所有长度相同的木棒
4.如果当前木棒拼接失败，且是当前边的第一个，则直接减掉当前分支
5.如果当前木棒拼接失败，且是当前边的最后一个，则直接减掉当前分支
*/
class Solution {
public:
    vector<bool> st;

    bool makesquare(vector<int>& nums) {
        int sum = 0;
        for(auto u : nums) sum += u;

        if(!sum || sum % 4) return false;

        sort(nums.begin(),nums.end());//第一、二个剪枝
        reverse(nums.begin(),nums.end());

        st = vector<bool>(nums.size());
        return dfs(nums,0,0,sum / 4);
    }

    bool dfs(vector<int> &nums,int u, int cur,int length)
    {
        if(cur == length) u ++,cur = 0;
        if(u == 4) return true;

        for(int i = 0;i < nums.size();i ++)
```

```
        {
            if(!st[i] && cur + nums[i] <= length)
            {
                st[i] = true;
                if(dfs(nums,u,cur + nums[i],length)) return true;
                st[i] = false;
                if(!cur) return false;//第四个剪枝
                if(cur + nums[i] == length) return false;//第五个剪枝
                while(i + 1 < nums.size() && nums[i + 1] == nums[i]) i ++;//第三
个剪枝
            }

        }
    return false;
    }
};
```