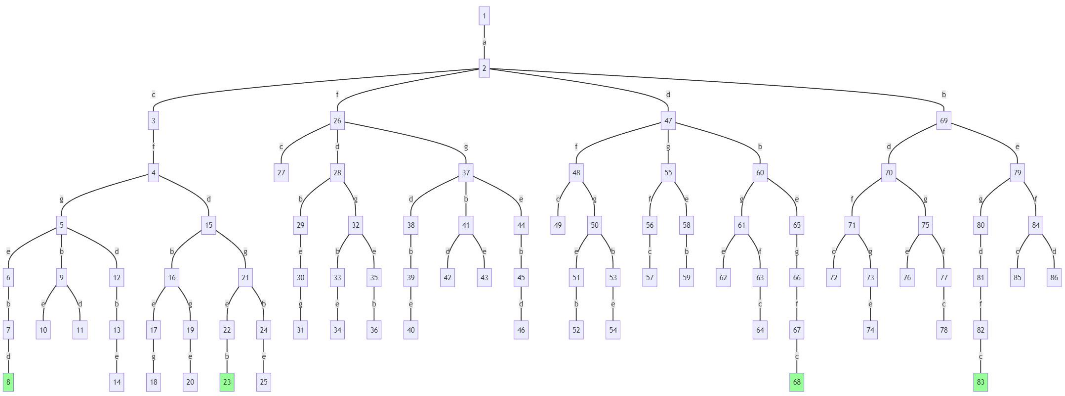
1. 解空间树如下：



其中方框中的状态表示状态变化的顺序，绿色标记即为可行解

acfdgeba/acgebda/adbegfca/abegdfca

搜索过程为：从状态图 1 到状态图 2 依次往下搜索，并且设置标记表示走过了，直到没路可走，就回退到上一步检查还没走过的路，并且清除标记。以此类推完成整个回溯。

例如：

a->c->f->g->e->b->d，找到一个解

回溯到 g->b->e

回溯到 g->b->d

回溯到 g->d->b->e

等等

2. 动态规划求解找零问题，有无限张 1 元、2 元和 3 元

设置状态表如下：

面额	组合
1	1 元*1
2	1 元*2
3	3 元*1
4	1 元*1+3 元*1
5	5 元*1
6	1 元*1+5 元*1， 3 元*2
7	1 元*2+5 元*1， 1 元*1+3 元*2
8	3 元*1+5 元*1
9	1 元*1+3 元*1+5 元*1， 3 元*3

故解为：1 张 1 元与一张 3 元与一张 5 元、3 张 3 元

伪代码：

```
Algorithm note_change(notes[0..n-1], target)
    // 用动态规划求解找零问题所有解
    // 输入：找零面额 notes[0..n-1]，升序排列，找零目标 target
    // 输出：面额为 target 的找零结果组合的集合
    // 首先构建状态表，初始化 1 至 5 元的最少张数组合
    ans[1] <- [[0, 1, 0, 0, 0, 0]]
    ans[2] <- [[0, 2, 0, 0, 0, 0]]
```

```
    ans[3] <- [[0, 0, 0, 1, 0, 0]]
    ans[4] <- [[0, 1, 0, 1, 0, 0]]
    ans[5] <- [[0, 0, 0, 0, 0, 1]]
    // 从目前最大面额的下一个开始，直到目标面额
    for i <- note[n - 1] + 1 to target do
        temp <- ∞
        k <- 0
        // 找到最小组合张数 temp，并记录组合的索引
        for j <- 0 to n - 1 do
            // 遍历 i - note[j]的每种组合
            for x in ans[i - note[j]] do
                cur <- count(x)
                if cur < temp then
                    temp <- cur

        // 根据最小组合的张数更新状态表
        k <- 0
        for j <- 0 to n - 1 do
            // 遍历 i - note[j]的每种组合
            for x in ans[i - note[j]] do
                // 如果组合最小
                if count(x) = temp then
                    // 增加一张 note[j]面额的
                    y = x
                    y[note[j]] <- y[note[j]] + 1
                    // 没有重复
                    if y not in ans[i] then
                        ans[i, k] <- y
                        k <- k + 1

    return ans[target]
Algorithm count(a[0..n - 1])
    // 输入：某一面额需要的面额组合
    // 输出：组合面额张数
    ans <- 0
    for i <- 0 to n - 1 do
        ans <- ans + a[i]
    return ans
```

代码如下：

```
# 面额种类
NOTE = [1, 3, 5]
# 需要兑换的零钱
```

```python
target = 9

def func(NOTE, target):
    dp = [None] * (target + 1)
    # dp 表建立，每个对应一个集合
    for i in range(target + 1):
        dp[i] = set()
    # 将 list 转为 tuple 才有 hash，初始化
    dp[1].add(tuple([1, 0, 0]))
    dp[2].add(tuple([2, 0, 0]))
    dp[3].add(tuple([0, 1, 0]))
    dp[4].add(tuple([1, 1, 0]))
    dp[5].add(tuple([0, 0, 1]))
    for i in range(6, target + 1):
        # 一个很大的数
        minSum = 99999999
        # 得到最小组合
        for note in NOTE:
            for x in dp[i - note]:
                tempSum = sum(x)
                minSum = min(minSum, tempSum)
        # 通过最小组合，可能不止一个，找到新的组合
        for note in NOTE:
            for x in dp[i - note]:
                if minSum == sum(x):
                    temp = list(x)
                    temp[NOTE.index(note)] += 1
                    dp[i].add(tuple(temp))

    return dp[target]
print(func(NOTE, target))
```

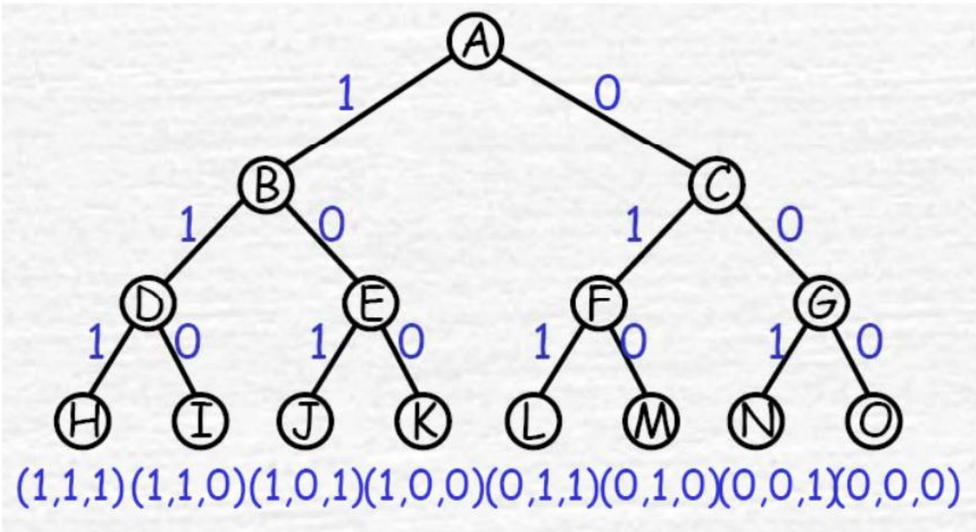
算法过程：

设计 ans[i]记录面额为 i 所需要的最少的张数的组合，利用二维数组记录组合，ans[i][j][k]表示第 j 中组合的面额为 k 需要多少张，考虑到所给出的面额 1，3，5，做出初始化 ans[1..5]，对于面额 n，我们取 n - 1, n - 3, n - 5 面额的组合，在此基础上增加一张对应面额，从所有组合中取最小的组合，注意该过程中需要取最小

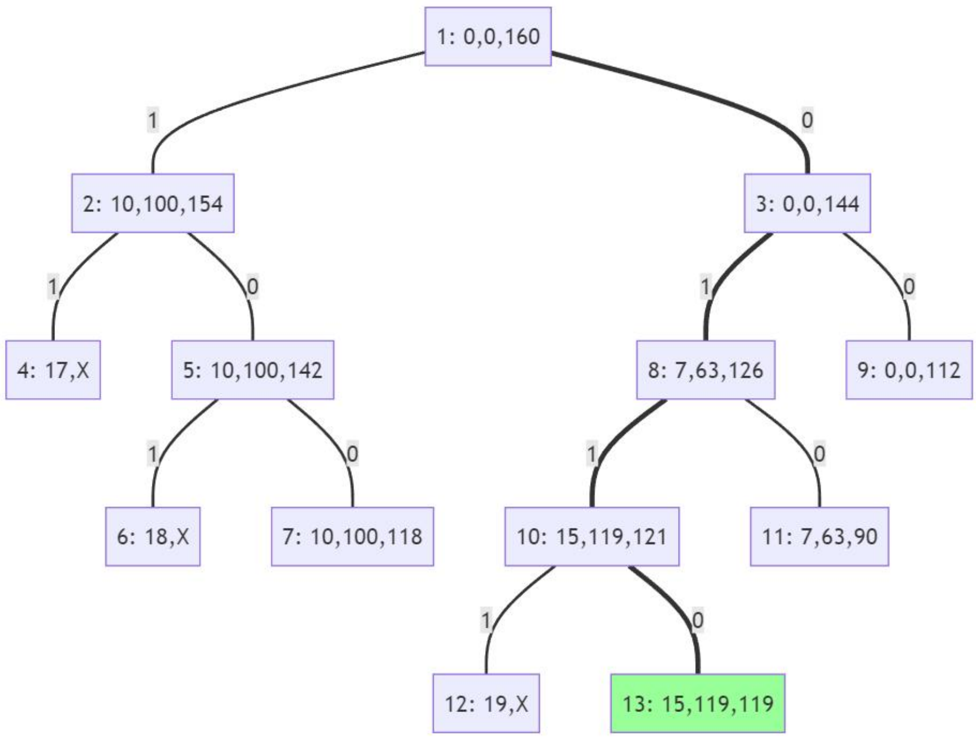
张数和去重。

3.

(1) 解空间树如下：



(2) 搜索过程：



其中方框中的内容为：第 i 个状态即第 i 个节点（表示搜索顺序），背包当前重量，背包当前价值以及 UP_BOUND。解空间树的分支为 1 或 0，表示装或不装当前物品。

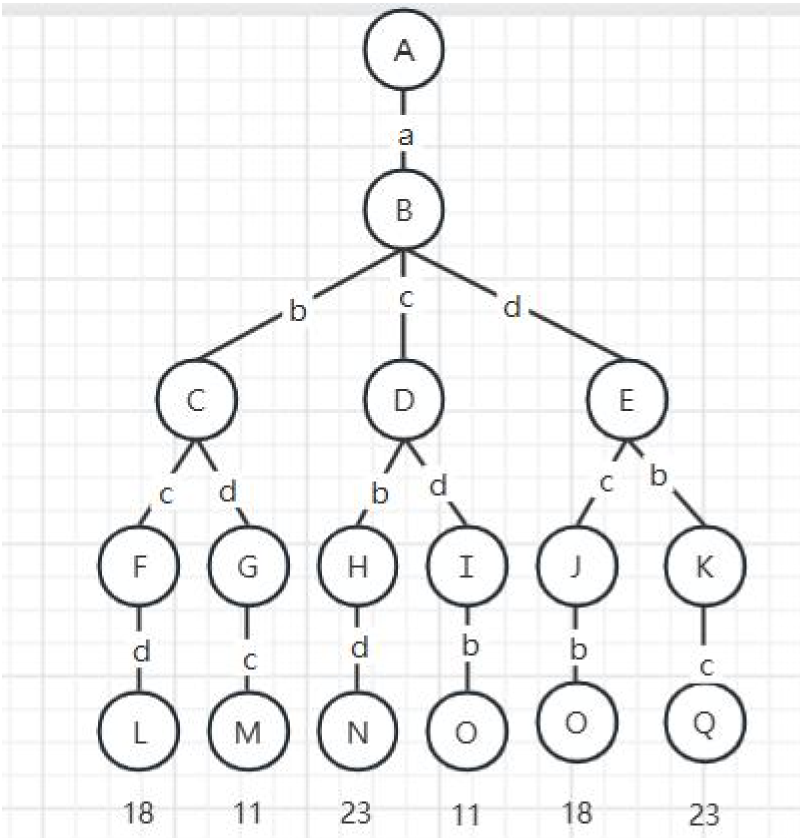
扩展结点	活结点	优先队列	可行解	解值
1	2，3	2，3		

2	4（死节点），5	3，5		
3	8，9	5，8，9		
5	6（死节点），7	8，7，9		
8	10，11	10，7，9，11		
10	12（死节点），13	13，7，9，11	13	119
13				

(3) 结果为选择第 2，3 个物品，解值为 119

4. TSP

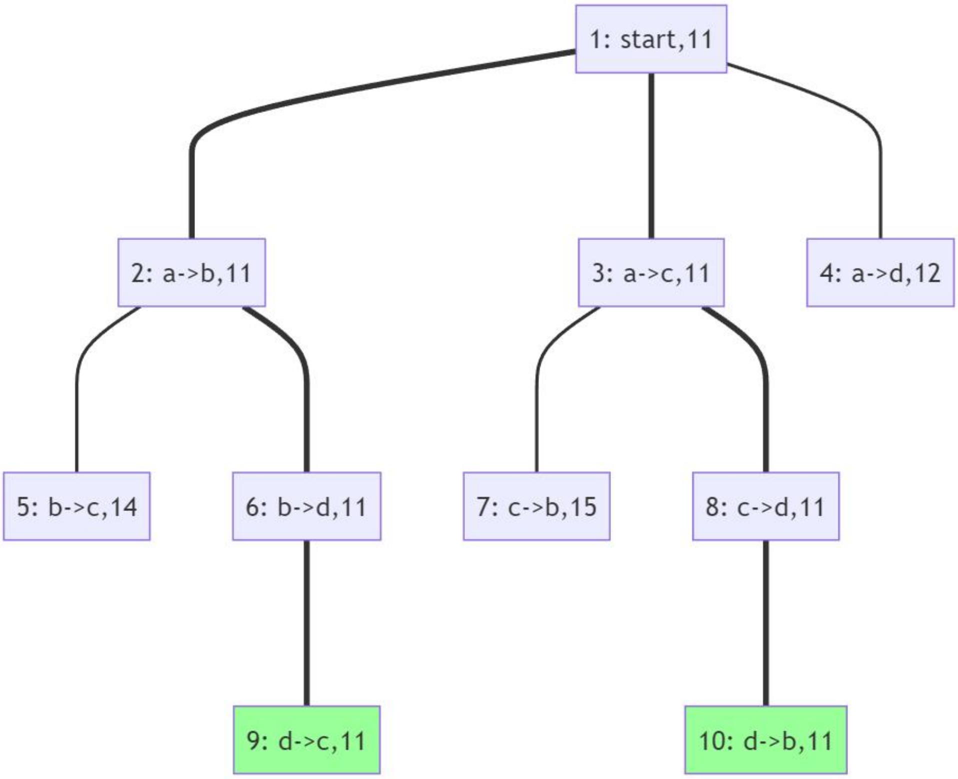
(1) 解空间树如下：



(2)

① LOW_BOUND 设计为邻接矩阵每行尽量取最小两个数，表示一个节点两条边尽量小。并且使用贪心算法计算出“上界”为 11

- -



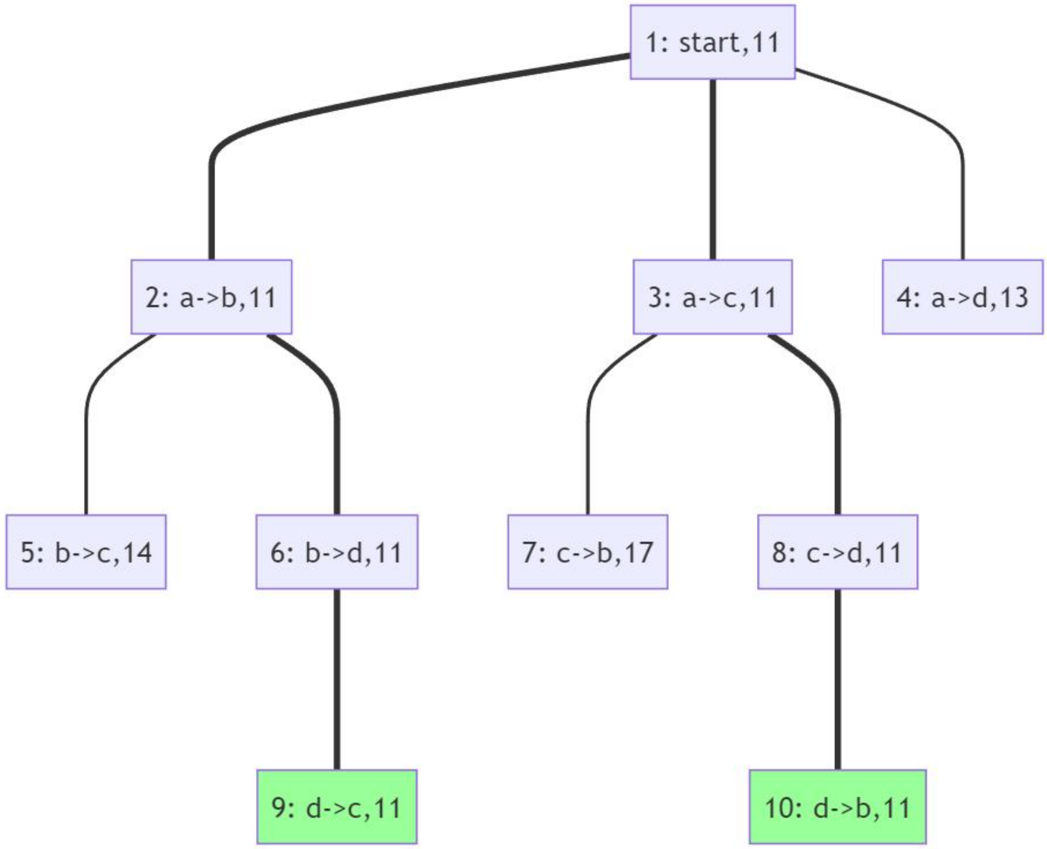
② 搜索过程

扩展结点	活结点	优先队列	可行解	解值
1	2，3，4（剪枝）	2，3		
2	5（剪枝），6	3，6		
3	7（剪枝），8	6，8		
6	9	8，9	9	11
8	10	9，10	10	11
9		10		
10				

③ 最优解为 a->b->d->c 或 a->c->d->b，解值均为 11

(3)

① LOW_BOUND 设计为在边集中选择还未选择的最小的边的加合。并且使用贪心算法计算出“上界”为 11



② 搜索过程同上

③ 最优解同上

5. 插棒

  a. 剩下插棒位置不限

    i. 主要思路

对于棋盘上当前的空位，尝试遍历周围的六个方向，若该方向上邻居的邻居存在且邻居也存在，则可以将插棒移动至当前空位，并将被跳过的邻居移走，其中是否为空位可以用一个布尔型数组表示 true 表示有插棒，false 则表示空位。

然后进入递归函数重复上述步骤。其中可以利用剪枝函数判断若当前步数已经大于 13 则可以停止。递归函数结束的条件是棋盘上只剩下一个插棒。

退出递归函数的时候，将之前的 false 重新置为 true，以便下一次的回溯。

count 为全局变量，初始值为 0，记录移动步数

board 记录棋盘是否为空

DIRECTION 数组记录 6 个方向邻居的邻居的索引 offset

NEIGHTBOR 数组记录 6 个方向上邻居的索引 offset

    i. 伪代码

```
Algorithm triangle_problem(board[0...14])
    // 剪枝，当前步数大于 13 就不用继续了
```

```
    if count > 13 then
        return
    // 只剩下一个插棒结束递归
    num <- 0
    for i <- 0 to 14 do
        if board[i] = true then
            num <- num + 1
    if num = 1 then
        record()
        return
    // 找出空位
    for i <- 0 to 14 do
        if board[i] = true then
            continue

        // 从空位找到六个方向上的邻居的邻居
        for j <- 0 to 4 do
            // 邻居的邻居
            new_empty = i + DIRECTION[j]
            // 出界或空位就跳过
            if new_empty < 0 or new_empty > 14 or board[new_empty]
= true then
                continue
            // 得到 i 的邻居
            neighbor = i + NEIGHBOR[j]
            // 更新期盼
            board[i] <- true
            board[new_empty] <- false
            board[neighbor] <- false
            // 增加当前步数
            count <- count + 1
            // 进入递归函数
            triangle_problem(board)
            // 回溯
            board[i] <- false
            board[new_empty] <- true
            board[neighbor] <- true
            // 减少当前步数
            count <- count - 1
```

b. 剩下插棒的位置在最初空位上

ii. 主要思路

对于棋盘上当前的空位，尝试遍历周围的六个方向，若该方向上邻居的邻居存在且邻居也存在，则可以将插棒移动至当前空位，并将被跳

过的邻居移走，其中是否为空位可以用一个布尔型数组表示 true 表示有插棒，false 则表示空位。

然后进入递归函数重复上述步骤。其中可以利用剪枝函数判断若当前步数已经大于 13 则可以停止。递归函数结束的条件是棋盘上只剩下一个插棒，并且该插棒位置与初始空位相同我们才将问题的解记录下来。

退出递归函数的时候，将之前的 false 重新置为 true，以便下一次的回溯。

count 为全局变量，初始值为 0，记录移动步数

board 记录棋盘是否为空

DIRECTION 数组记录 6 个方向邻居的邻居的索引 offset

NEIGHTBOR 数组记录 6 个方向上邻居的索引 offset

iii. 伪代码，其中 count 为全局变量，初始值为 0，移动步数

```
Algorithm triangle_problem(board[0...14])
    // 剪枝，当前步数大于 13 就不用继续了
    if count > 13 then
        return
    // 只剩下一个插棒结束递归
    num <- 0
    idx <- -1
    for i <- 0 to 14 do
        if board[i] = true then
            num <- num + 1
            idx <- i
    if num = 1 then
        // 初始位置才记录
        if idx = INIT_LOCATION then
            record()
        return
    // 找出空位
    for i <- 0 to 14 do
        if board[i] = true then
            continue

        // 从空位找到六个方向上的邻居的邻居
        for j <- 0 to 5 do
            // 邻居的邻居
            new_empty = i + DIRECTION[j]
            // 出界或空位就跳过
            if new_empty < 0 or new_empty > 14 or board[new_empty]
= true then
                continue
            // 得到 i 的邻居
            neighbor = i + NEIGHBOR[j]
            // 更新棋盘
```

```
            board[i] <- true
            board[new_empty] <- false
            board[neighbor] <- false
            // 增加当前步数
            count <- count + 1
            // 进入递归函数
            triangle_problem(board)
            // 回溯
            board[i] <- false
            board[new_empty] <- true
            board[neighbor] <- true
            // 减少当前步数
            count <- count - 1
```

iv. 搜索结果如下

v. 代码如下：

```
import sys
NEI = []
NEI.append([-1, -1, 2, 1, -1, -1])
NEI.append([0, 2, 4, 3, -1, -1])
NEI.append([-1, -1, 5, 4, 1, 0])
NEI.append([1, 4, 7, 6, -1, -1])
NEI.append([2, 5, 8, 7, 3, 1])
NEI.append([-1, -1, 9, 8, 4, 2])
NEI.append([3, 7, 11, 10, -1, -1])
NEI.append([4, 8 ,12, 11, 6, 3])
NEI.append([5, 9, 13, 12, 7, 4])
NEI.append([-1, -1, 14, 13, 8, 5])
NEI.append([6, 11, -1, -1, -1, -1])
NEI.append([7, 12, -1, -1, 10, 6])
NEI.append([8, 13, -1, -1, 11, 7])
NEI.append([9, 14, -1, -1, 12, 8])
NEI.append([-1, -1, -1, -1, 13, 9])


NEINEI = []
NEINEI.append([-1, -1, 5, 3, -1, -1])
NEINEI.append([-1, -1, 8, 6, -1, -1])
NEINEI.append([-1, -1, 9, 7, -1, -1])
NEINEI.append([0, 5, 12, 10, -1, -1])
NEINEI.append([-1, -1, 13, 11, -1, -1])
NEINEI.append([-1, -1, 14, 12, 3, 0])
NEINEI.append([1, 8, -1, -1, -1, -1])
NEINEI.append([2, 9, -1, -1, -1, -1])
NEINEI.append([-1, -1, -1, -1, 6, 1])
NEINEI.append([-1, -1, -1, -1, 7, 2])
```

```python
NEINEI.append([3, 12, -1, -1, -1, -1])
NEINEI.append([4, 13, -1, -1, -1, -1])
NEINEI.append([5, 14, -1, -1, 10, 3])
NEINEI.append([-1, -1, -1, -1, 11, 4])
NEINEI.append([-1, -1, -1, -1, 12, 5])
board = [True] * 15
board[12] = False
ans = []
def func(count):
    # 步数超过 13
    if count > 13:
        return
    # 只剩一个
    # num = 0
    # idx = -1
    # for i, x in enumerate(board):
    #     if x:
    #         num += 1
    #         idx = i
    # if num  == 1 and idx == 12:
    #     print("success")
    #     print(ans)
    #     sys.exit()
    #     return
    if board.count(True) == 1:
        print("success")
        print(ans)
        sys.exit() # 发现答案数量过多，得到一个答案就结束程序
        return
    print(count, board.count(False))
    # 遍历整个棋盘
    for idx, x in enumerate(board):
        # 空的
        if x == False:
            # 遍历六个方向
            for i in range(6):
                # 有邻居的邻居且有邻居并且都有棒
                if NEINEI[idx][i] != -1 and NEI[idx][i] != 1 and board[NEINEI[idx][i]] == True and board[NEI[idx][i]] == True:
                    board[NEINEI[idx][i]] = False
                    board[NEI[idx][i]] = False
                    board[idx] = True
                    # 记录当前棋盘状态
                    ans.append(board[0:15])
```

```
                    func(count + 1)
                    # 退出当前棋盘状态
                    ans.pop()
                    # 回溯
                    board[NEINEI[idx][i]] = True
                    board[NEI[idx][i]] = True
                    board[idx] = False
func(0)
```

vi. 搜索结果如下

```
    1
   1 1
  1 1 0
 1 1 0 1
1 1 1 1 1
    1
   1 1
  1 1 1
 1 1 0 0
1 1 1 1 0
    1
   1 1
  1 1 1
 0 0 1 0
1 1 1 1 0
    1
   0 1
  0 1 1
 1 0 1 0
1 1 1 1 0
    1
   0 1
  1 0 0
 1 0 1 0
1 1 1 1 0
    1
   1 1
  0 0 0
 0 0 1 0
1 1 1 1 0
    1
   1 1
  0 1 0
 0 0 0 0
```

1 1 1 0 0
0
1 0
0 1 1
0 0 0 0
1 1 1 0 0
0
0 0
0 0 1
0 0 1 0
1 1 1 0 0
0
0 0
0 0 1
0 0 1 0
1 0 0 1 0
0
0 0
0 0 0
0 0 0 0
1 0 1 1 0
0
0 0
0 0 0
0 0 0 0
1 1 0 0 0
0
0 0
0 0 0
0 0 0 0
0 0 1 0 0

vii. 经过代码统计发现：若将棋盘位置从 0-14 编号，从上到下、从左到右，则最后一个插棒只会回到编号为 0、4、6、9、12 几个位置，并且回到原来位置的方法数最多。

编程题，金矿

```
    int dx[4] = {0, 0, -1, 1};
    int dy[4] = {-1, 1, 0, 0};
    int m, n;
    int ans = 0;
    int temp = 0;
```

```
    void help(vector<vector<int>>& grid, vector<vector<bool>>& 
vis, int x, int y) {
        ans = max(ans, temp);
        // 出界
        if (x < 0 || y < 0 || x >= m || y >= n) {
            return;
        }
        // 没东西或已经访问过
        if (grid[x][y] == 0 || vis[x][y]) {
            return;
        }
        // 当前收获
        temp += grid[x][y];
        // 访问过
        vis[x][y] = true;
        // 遍历四个方向
        for (int i = 0; i < 4; ++i) {
            int new_x = x + dx[i];
            int new_y = y + dy[i];
            help(grid, vis, new_x, new_y);
        }
        vis[x][y] = false;
        temp -= grid[x][y];
    }
    int getMaximumGold(vector<vector<int>>& grid) {
        m = grid.size();
        n = grid[0].size();
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                vector<vector<bool>> vis(m, vector<bool>(n, 
false));
                help(grid, vis, i, j);
            }
        }
        return ans;
    }
```
算法思路

利用一个变量 temp 记录当前开采总量，ans 记录解值，一个首先枚举网格内所有点作为起点，进行开采。

利用回溯与递归的方法。递归结束条件为出界或开采到 0 个金矿或访问过的地方。

递归体中首先将 ans 更新为 temp 和 ans 中的最大值，然后将 temp 增加当前开采的金矿，并将当前位置设置为访问过 vis[x][y] = true，遍历上下左右四个方向进入递归函数，跳出递归函数之后，将该位置设置为没访问过 vis[x][y] = false，

以便下一次访问，temp 减去当前开采的金矿。

最后即可得到解值。

复杂度分析：

空间复杂度：每轮递归都需要一个和原来 grid 大小相同的 vis 数组记录是否访问过，因此复杂度为 O(mn)

时间复杂度：共有 mn 个起点，每个起点对应一轮递归，注意到仅仅一开始进入递归时可能有四个方向进入函数，而之后最多三个方向，因此复杂度为 $O(mn * 3^{mn})$

运行截图