CS精英挑战营 2019

计算思维：可计算性与问题求解

徐 明 星
xumx@tsinghua.edu.cn

六教 6C 102

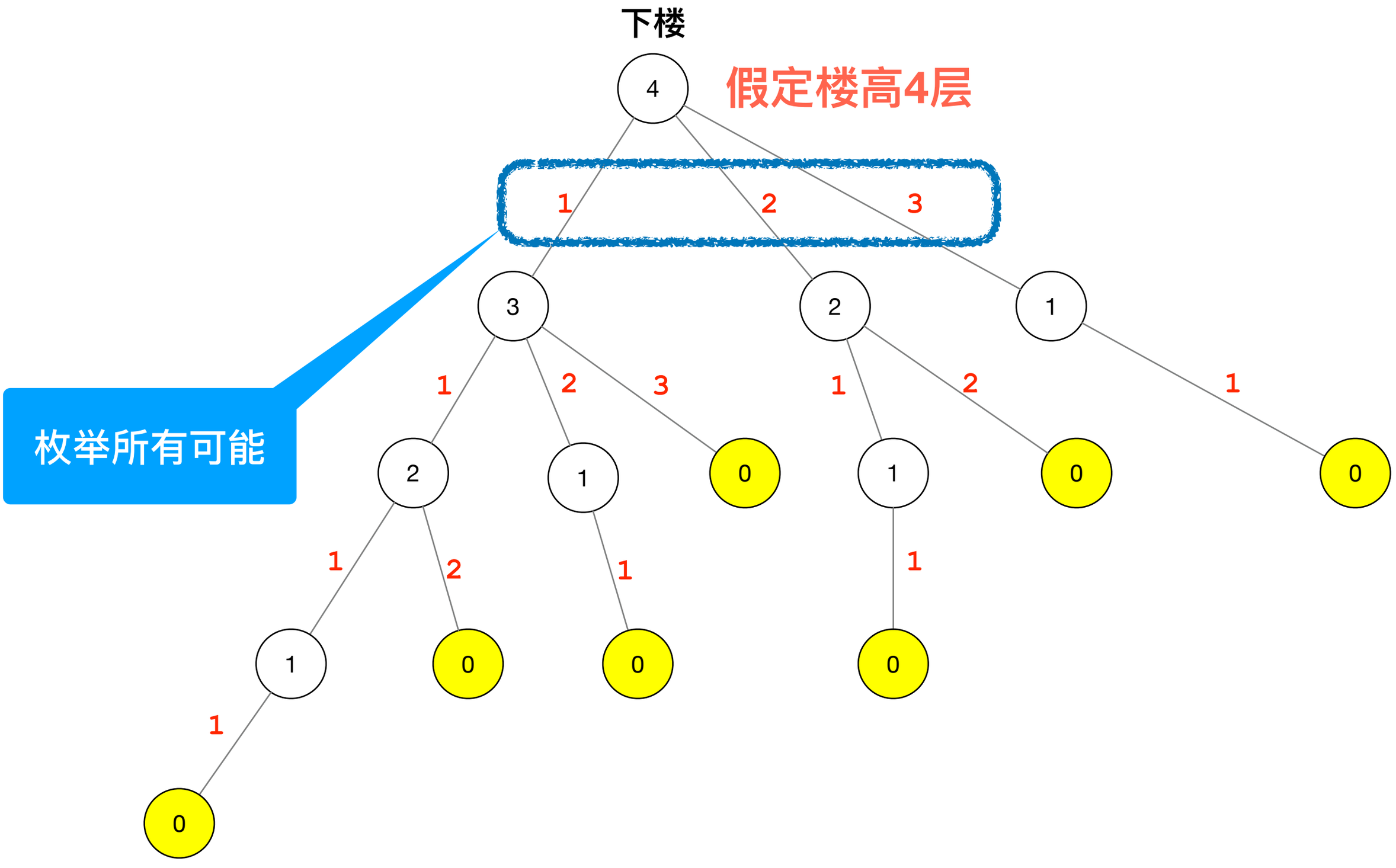
【任务1.1】下楼

从楼上走到楼下共有 h 个台阶，每一步有三种走法

- 走一个台阶；
- 走二个台阶；
- 走三个台阶。

问：一共可以走出多少种方案？即共要多少步？每一步走几级台阶？

用与或图在纸上模拟“下楼”操作



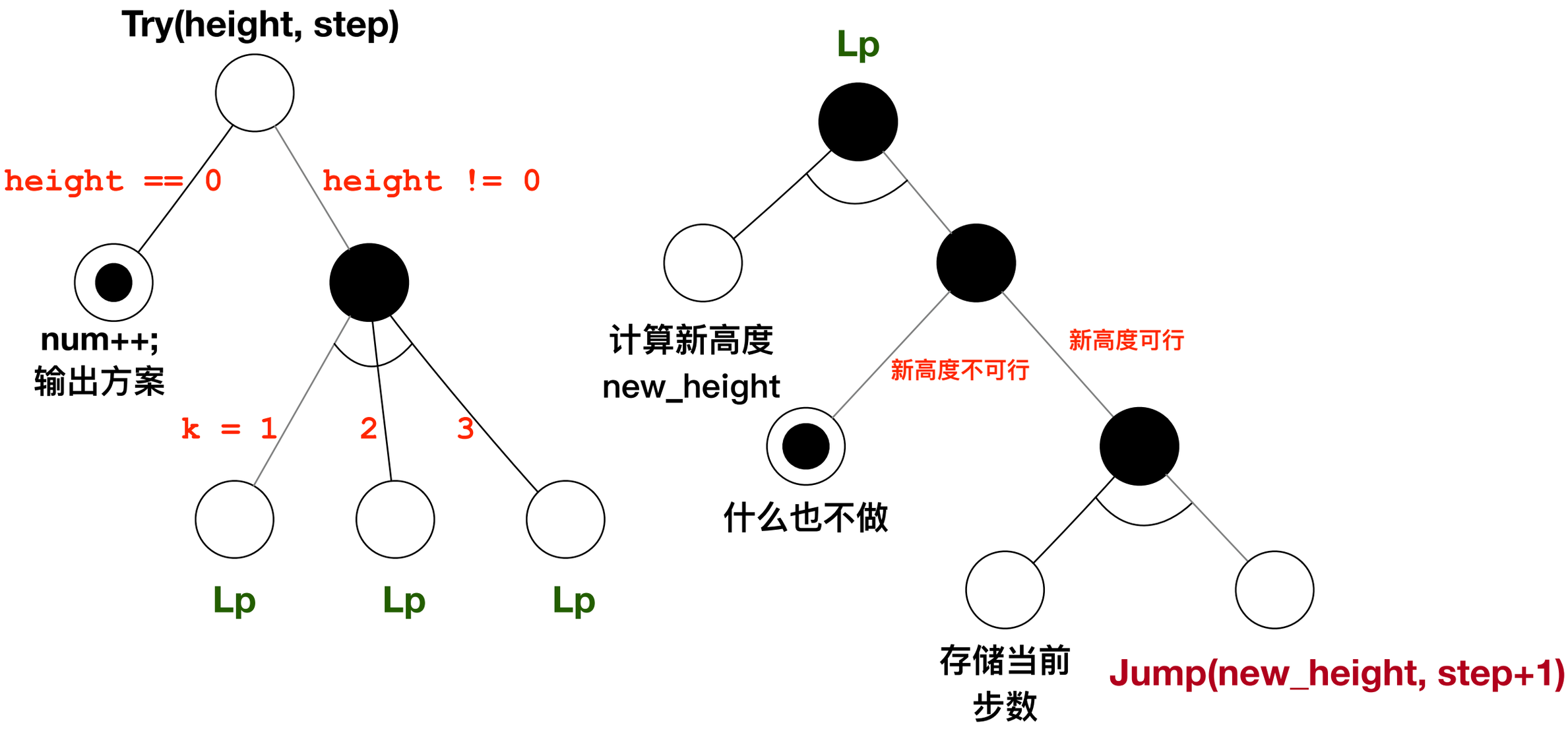
解题思路（伪代码）

1. 根据枚举思想，每一步需要尝试所有不同的步数j。j 或者是为1，或是为 2，或是为 3。这可用 for 循环结构来实现。
2. 试着一步一步地走，从高到低，让变量i 先取台阶数 h。从楼上到楼下，每走一步，变量 i 的值会减去每一步所走的台阶数 j。
   2.1. 开始时，i = h（初值）
   2.2. 以后 i = i-j，(j=1, 2, 3)
   2.3. 当 i = 0 时，剩余台阶数为0，这说明已走到楼下
3. 每一步走法策略都相同，故可以用递归算法。

```cpp
#include <iostream>     // cout
using namespace std;

// 方案细节记录在take中，方案数用num累计
int take[99], num = 0;

void Try(int i, int s); // 有i级台阶，从第s步开始

int main() {
    cout << "请输入楼梯台阶数：";
    int h;
    cin >> h;     // 输入楼梯的台阶数

    Try(h, 1);      // 从第h级，开始下第一步

    cout << "总方案数：" << num << endl;

    return 0;
}
```

```
void Try(int i, int s) { // 有i级台阶，从第s步开始
    for (int j=3; j>0; j--) {
        if (i >= j) {
            take[s] = j;     // 记录第s步走j个台阶
            if (i==j) {      // 如果已经到了楼下
                num++;       // 则方案数加1
                cout << "方案" << num << ": ";
                for (int k=1; k<=s; k++) cout << take[k];
                cout << endl;
            }
            else // 尚未走到楼下
                Try(i-j, s+1); // 再试剩下的台阶
        } // __if(i>=j)__
    } // __FOR_J__
}
```

解题思路2：先判断中止，再枚举递归



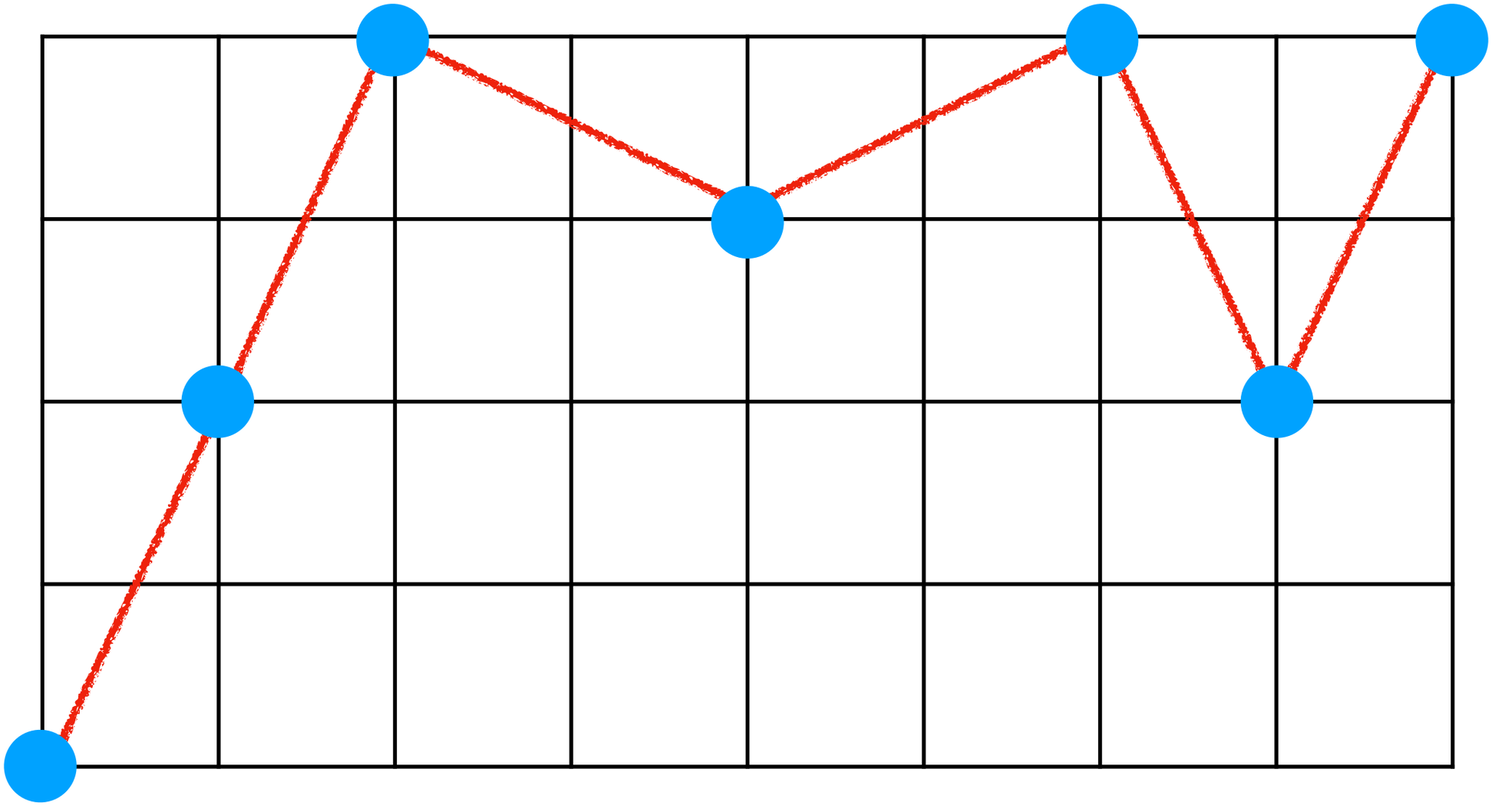
```cpp
#include <iostream>
using namespace std;

/// 楼梯高度(just for test)
const int TARGET_H = 5;

/// 方案总数、方案内容
int num, path[TARGET_H];

/// 第step步，从高度height开始，继续下楼
void Try(int height, int step);

int main() {
    /// 总方案数初值为0
    num = 0;

    /// 第0步，从高度TARGET_H出发
    Try(TARGET_H, 0);

    return 0;
}
```

```
/// 第step步，从高度height开始，继续下楼
void Try(int height, int step) {
    /// 递归中止条件：到达楼梯底层
    if (height == 0) {
        num ++;
        cout << num << ": ";
        for (int i = 0; i < step; i++) cout << path[i] << ' ';
        cout << endl;
        return;
    }
    /// 依次尝试不同的下楼步数(循环变量i是步数)
    for (int i = 1; i <= 3; i++) {
        /// 1. 计算新高度
        int new_height = height - i;
        /// 2. 高度是否可行?
        if (new_height < 0) continue;
        /// 3. 记录当前步数
        path[step] = i;
        /// 4. 继续向目标前进
        Try(new_height, step+1);
    }
}
```

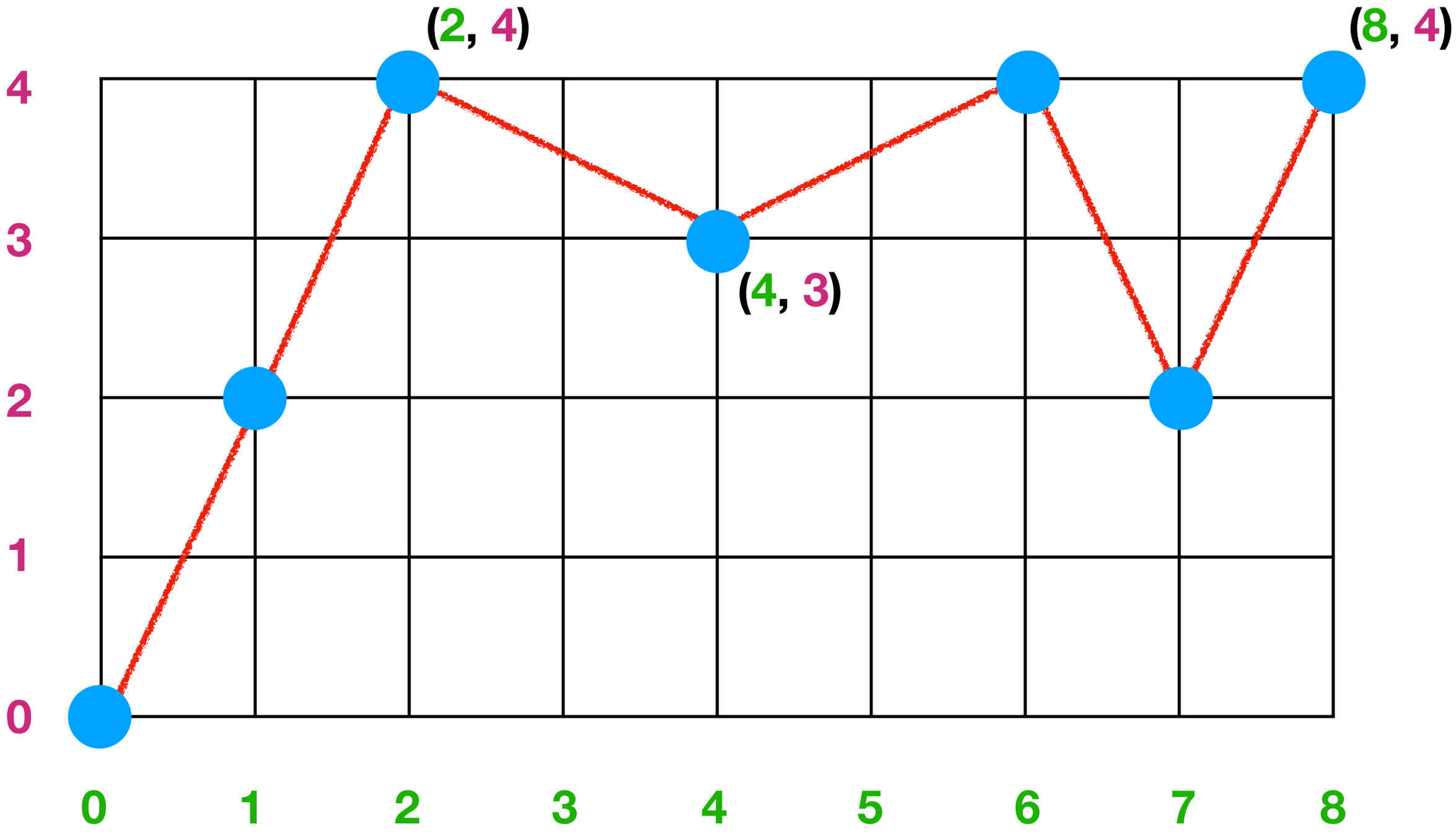
【任务1.2】跳马

在半张中国象棋的棋盘上，一只马从左下角跳到右上角，只允许往右跳，不允许往左跳，问能有多少种跳步方案。

要求：输出方案数和各方案的具体跳法。



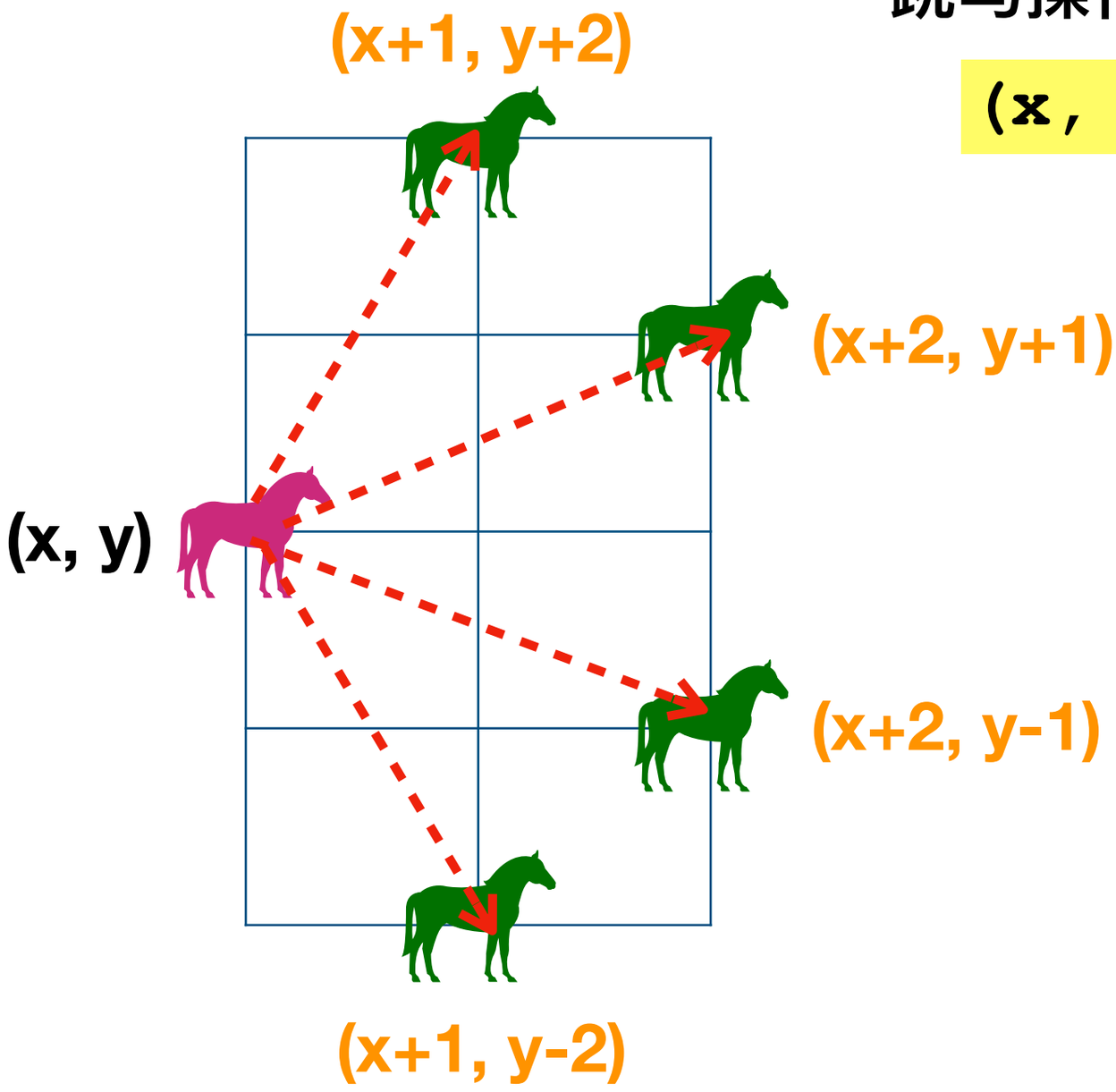
数字化棋盘，使位置可计算



数字化跳法，使操作可计算

跳马操作，转换成了位置（坐标）的变化

```
(x, y) -> (x + dx, y + dy)
```



跳法	dx	dy
0	1	2
1	2	1
2	2	-1
3	1	-2

解题思路：遍历当前所有可能的跳法

跳法	dx	dy
0	1	2
1	2	1
2	2	-1
3	1	-2

跳马操作，转换成了位置（坐标）的变化

```
(x, y) -> (x + dx, y + dy)
```

```
0: (x, y) -> (x + 1, y + 2)
```

```
1: (x, y) -> (x + 2, y + 1)
```

```
2: (x, y) -> (x + 2, y - 1)
```

```
3: (x, y) -> (x + 1, y - 2)
```

设计数据结构

1. 马的不同跳法：

跳法	dx	dy
0	1	2
1	2	1
2	2	-1
3	1	-2

设计数据结构

1. 马的不同跳法：

```
int dx[] = {1, 2, 2, 1};
```

跳法	dx	dy
0	1	2
1	2	1
2	2	-1
3	1	-2

设计数据结构

1. 马的不同跳法：

```
int dx[] = {1, 2, 2, 1}, dy[] = {2, 1, -1, -2};
```

跳法	dx	dy
0	1	2
1	2	1
2	2	-1
3	1	-2

设计数据结构

1. 马的不同跳法：使用“平行数组” -- 两个数组，成对使用

```
int dx[] = {1, 2, 2, 1}, dy[] = {2, 1, -1, -2};
```

跳法	dx	dy
0	1	2
1	2	1
2	2	-1
3	1	-2

```
0: (x, y) -> (x + dx[0], y + dy[0])
```

```
1: (x, y) -> (x + dx[1], y + dy[1])
```

```
2: (x, y) -> (x + dx[2], y + dy[2])
```

```
3: (x, y) -> (x + dx[3], y + dy[3])
```

【编程技巧】用数组存储所有跳法，用循环枚举实现遍历

```
0: (x, y) -> (x + dx[0], y + dy[0])
```

```
1: (x, y) -> (x + dx[1], y + dy[1])
```

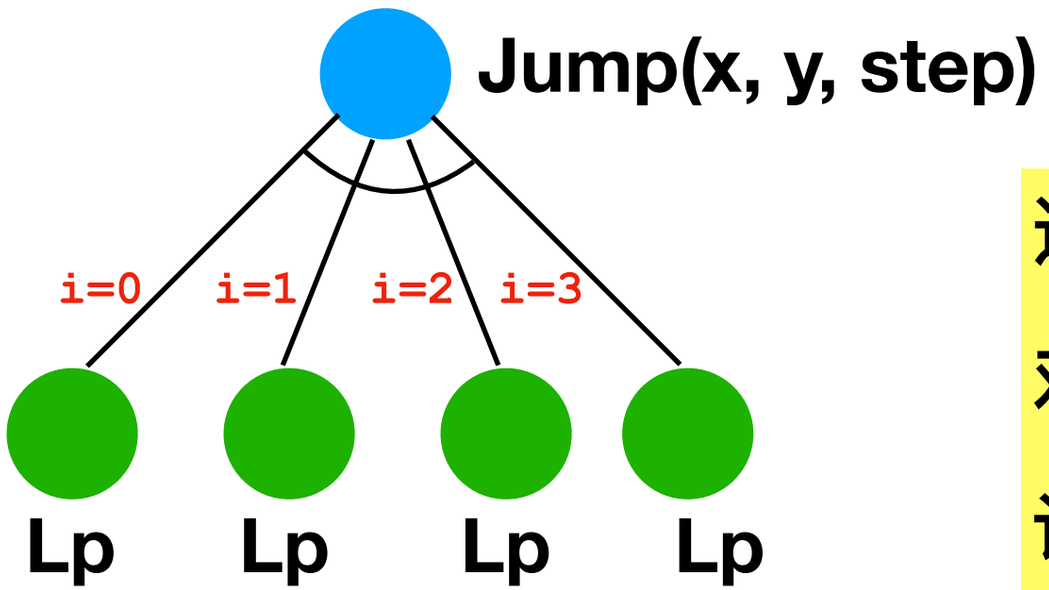
```
2: (x, y) -> (x + dx[2], y + dy[2])
```

```
3: (x, y) -> (x + dx[3], y + dy[3])
```

使用循环语句对所有可能的跳法进行**枚举**，也称**“遍历”**。

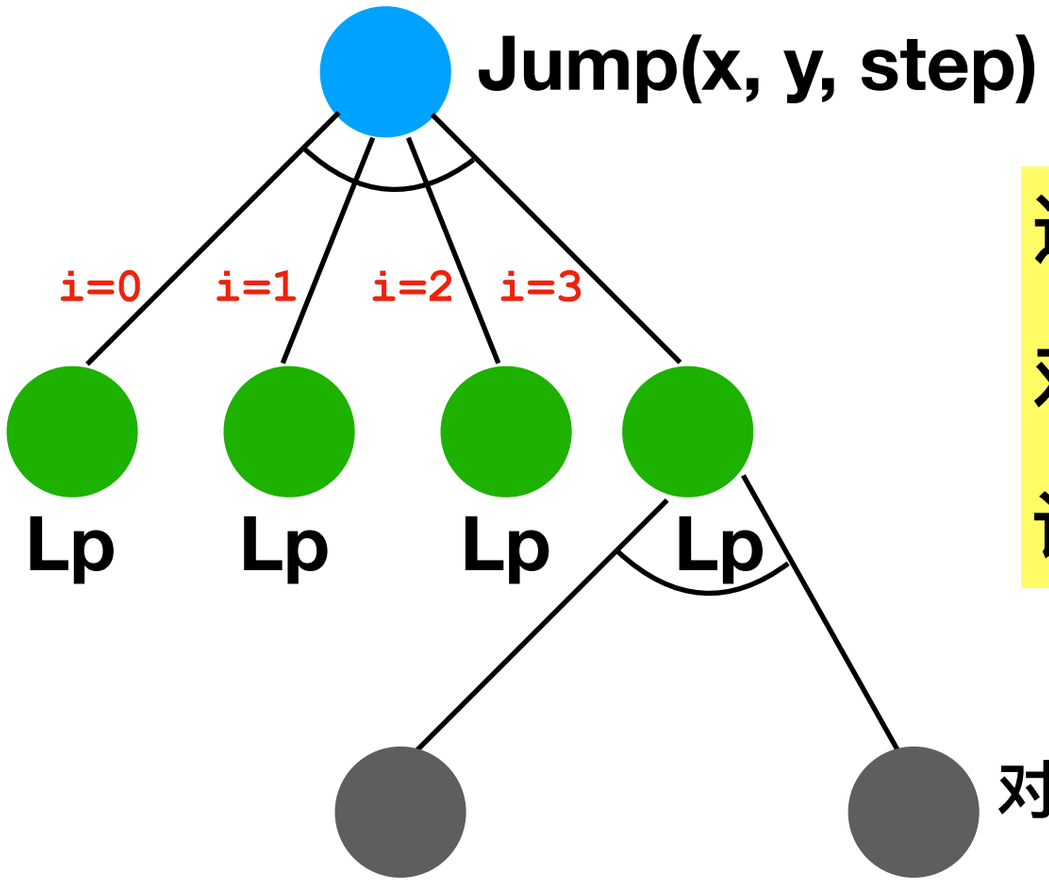
```
for i from 0 to 4 do:
    (x, y) -> (x + dx[i], y + dy[i])
    ......
```

解题思路1：第step步从(x,y)开始遍历



遍历从当前位置出发的所有可能性，并对各种可能性得到的方案进行记录，累计方案的总数。

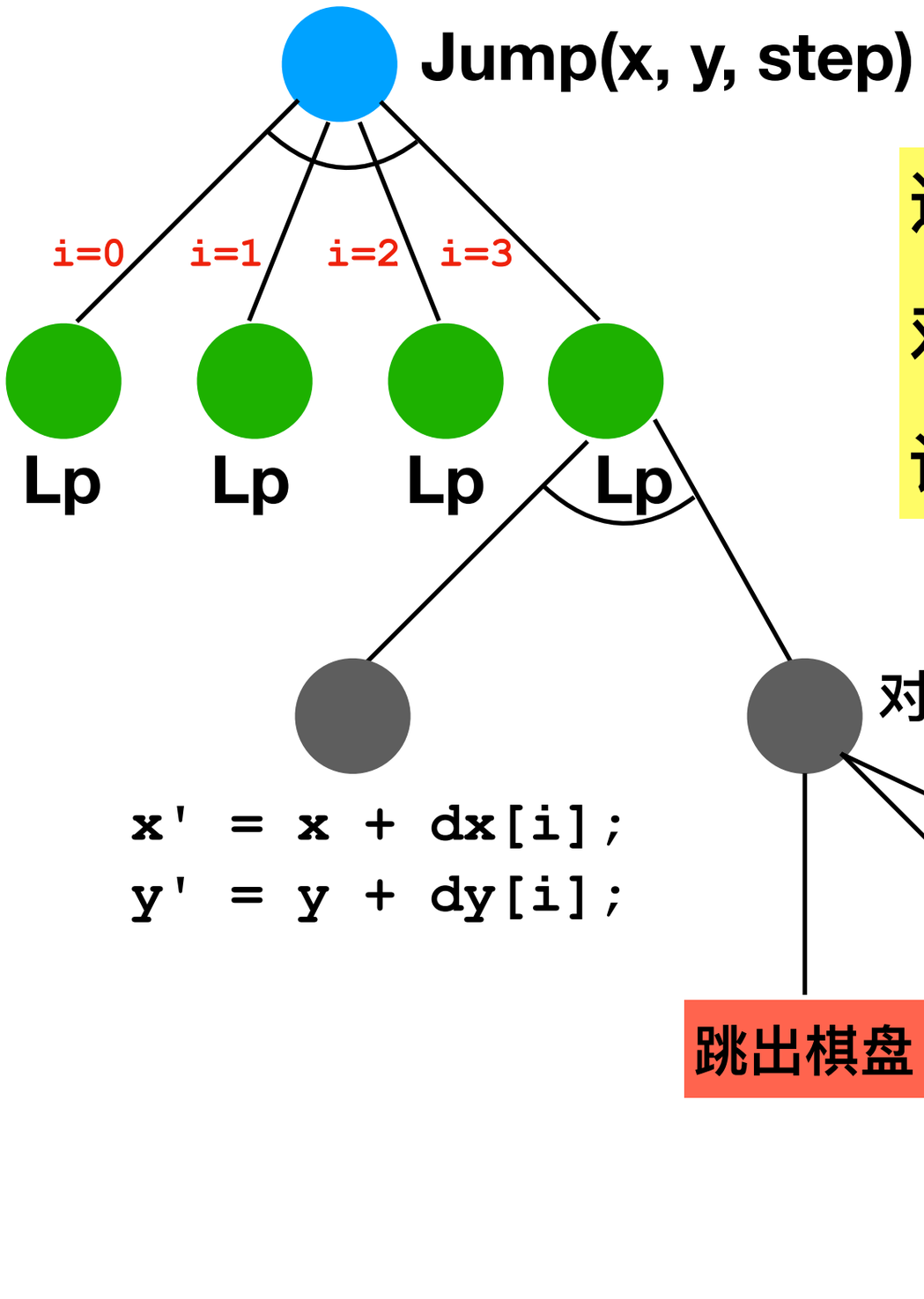
解题思路1：第step步从(x,y)开始遍历



遍历从当前位置出发的所有可能性，并对各种可能性得到的方案进行记录，累计方案的总数。

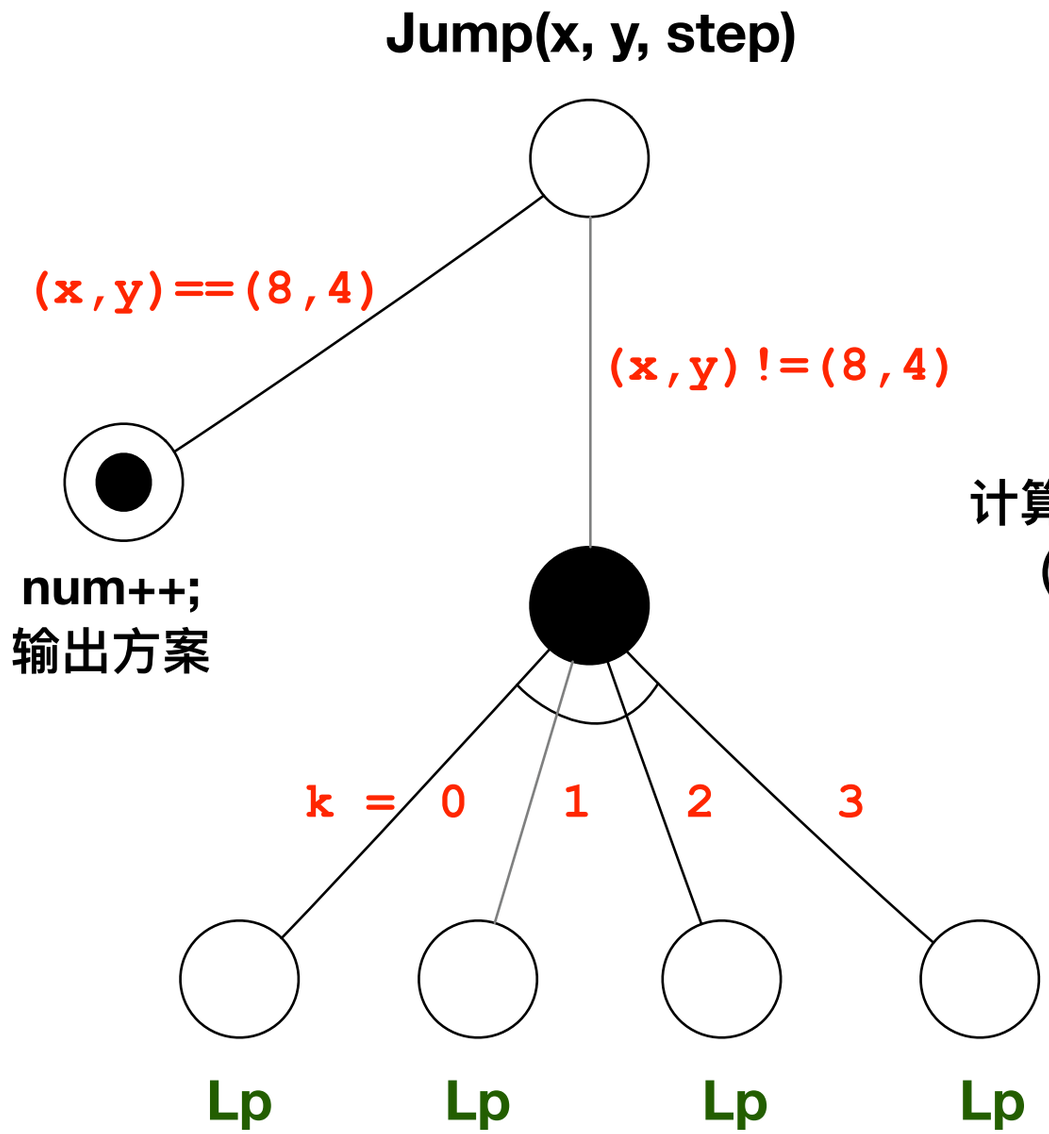
```
x' = x + dx[i];
y' = y + dy[i];
```

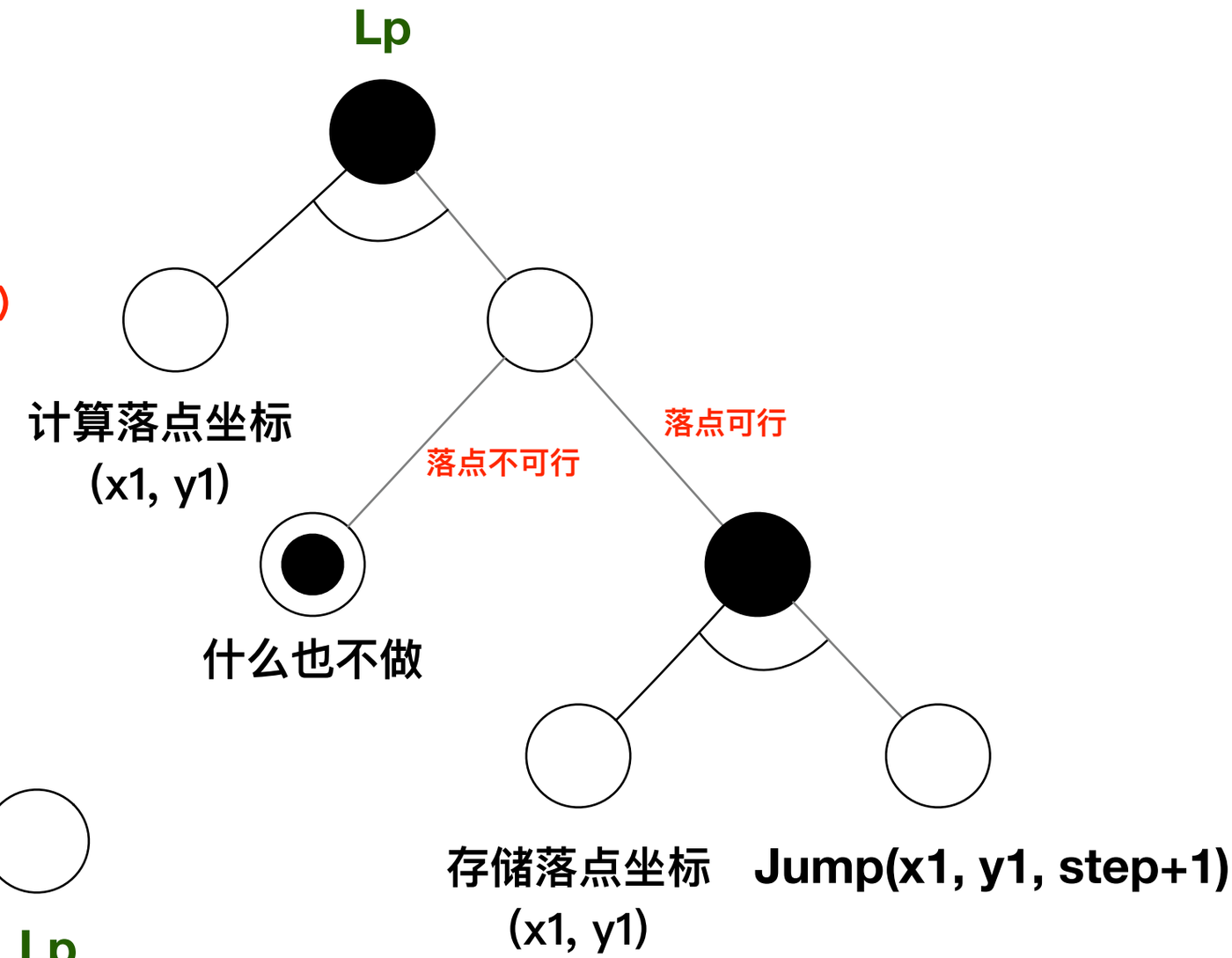
解题思路1：第step步从(x,y)开始遍历



遍历从当前位置出发的所有可能性，并对各种可能性得到的方案进行记录，累计方案的总数。

解题思路2：先判断中止，再枚举递归





设计数据结构

1. 马的不同跳法：使用“平行数组” -- 两个数组，成对使用

```
int dx[] = {1, 2, 2, 1}, dy[] = {2, 1, -1, -2};
```

2. 操作步骤记录：二维数组，一维是跳步的次序，一维是位置坐标

```
int path[100][2];  // 总步数取一个较大的值（估计值）
```

每一步记录两个值 path[step][0], path[step][1]，分别对应x和y

```
for i from 0 to 4 do:
    (x, y) -> (x + dx[i], y + dy[i])
    ......
```

path[step][0]　　path[step][1]

设计数据结构

1. 马的不同跳法：使用“平行数组” -- 两个数组，成对使用

```
int dx[] = {1, 2, 2, 1}, dy[] = {2, 1, -1, -2};
```

2. 操作步骤记录：二维数组，一维是跳步的次序，一维是位置坐标

```
int path[100][2];  // 总步数取一个较大的值（估计值）
```

每一步记录两个值 path[step][0], path[step][1]，分别对应x和y

3. 以上均定义为全局变量

4. 当前总的方案数也定义一个全局变量num来表示

```cpp
#include <iostream>          // cout
using namespace std;

int dx[] = {1, 2, 2, 1}, dy[] = {2, 1, -1, -2};
int num, path[100][2];

void Jump(int x, int y, int step);

int main() {
    // 初始方案数置0
    num = 0;

    // 第0步，从(0,0)出发
    Jump(0, 0, 0);

    return 0;
}
```

```
void Jump(int x, int y, int step) {
    if ((x == 8) && (y == 4)) {   // 是否到达目标?
        num++;     // 方案数加1
        cout << num << ": ";
        for (int i=0; i<step; i++) // 从起点开始输出各步的坐标
            cout << "(" << path[i][0] << ", " << path[i][1] << ") ";
        cout << endl;

        return;
    }

    // 遍历四种跳步方向
    for (int k=0; k<4; k++) {
        int x1 = x + dx[k], y1 = y + dy[k];

        // (x1, y1)是否可行?
        if ((x1 < 0) || (x1 > 8) || (y1 < 0) || (y1 > 4)) continue;

        path[step][0] = x1;
        path[step][1] = y1;

        /// 跳一步,探索不同的跳步方案
        Jump(x1, y1, step+1);
    }
}
```

运行结果

```
方案 1: (0, 0) (1, 2) (2, 4) (4, 3) (6, 4) (7, 2) (8, 4)
方案 2: (0, 0) (1, 2) (2, 4) (4, 3) (5, 1) (6, 3) (8, 4)
方案 3: (0, 0) (1, 2) (2, 4) (4, 3) (5, 1) (7, 2) (8, 4)
方案 4: (0, 0) (1, 2) (2, 4) (3, 2) (4, 4) (6, 3) (8, 4)
方案 5: (0, 0) (1, 2) (2, 4) (3, 2) (4, 4) (5, 2) (6, 4) (7, 2) (8, 4)
方案 6: (0, 0) (1, 2) (2, 4) (3, 2) (4, 4) (5, 2) (6, 0) (7, 2) (8, 4)
方案 7: (0, 0) (1, 2) (2, 4) (3, 2) (5, 3) (7, 2) (8, 4)
方案 8: (0, 0) (1, 2) (2, 4) (3, 2) (5, 1) (6, 3) (8, 4)
方案 9: (0, 0) (1, 2) (2, 4) (3, 2) (5, 1) (7, 2) (8, 4)
方案 10: (0, 0) (1, 2) (2, 4) (3, 2) (4, 0) (5, 2) (6, 4) (7, 2) (8, 4)
方案 11: (0, 0) (1, 2) (2, 4) (3, 2) (4, 0) (5, 2) (6, 0) (7, 2) (8, 4)
方案 12: (0, 0) (1, 2) (3, 3) (5, 2) (6, 4) (7, 2) (8, 4)
方案 13: (0, 0) (1, 2) (3, 3) (5, 2) (6, 0) (7, 2) (8, 4)
方案 14: (0, 0) (1, 2) (3, 3) (4, 1) (5, 3) (7, 2) (8, 4)
方案 15: (0, 0) (1, 2) (3, 3) (4, 1) (6, 0) (7, 2) (8, 4)
方案 16: (0, 0) (1, 2) (3, 1) (4, 3) (6, 4) (7, 2) (8, 4)
方案 17: (0, 0) (1, 2) (3, 1) (4, 3) (5, 1) (6, 3) (8, 4)
方案 18: (0, 0) (1, 2) (3, 1) (4, 3) (5, 1) (7, 2) (8, 4)
方案 19: (0, 0) (1, 2) (3, 1) (5, 2) (6, 4) (7, 2) (8, 4)
方案 20: (0, 0) (1, 2) (3, 1) (5, 2) (6, 0) (7, 2) (8, 4)
方案 21: (0, 0) (1, 2) (2, 0) (3, 2) (4, 4) (6, 3) (8, 4)
方案 22: (0, 0) (1, 2) (2, 0) (3, 2) (4, 4) (5, 2) (6, 4) (7, 2) (8, 4)
方案 23: (0, 0) (1, 2) (2, 0) (3, 2) (4, 4) (5, 2) (6, 0) (7, 2) (8, 4)
方案 24: (0, 0) (1, 2) (2, 0) (3, 2) (5, 3) (7, 2) (8, 4)
方案 25: (0, 0) (1, 2) (2, 0) (3, 2) (5, 1) (6, 3) (8, 4)
方案 26: (0, 0) (1, 2) (2, 0) (3, 2) (5, 1) (7, 2) (8, 4)
方案 27: (0, 0) (1, 2) (2, 0) (3, 2) (4, 0) (5, 2) (6, 4) (7, 2) (8, 4)
方案 28: (0, 0) (1, 2) (2, 0) (3, 2) (4, 0) (5, 2) (6, 0) (7, 2) (8, 4)
方案 29: (0, 0) (1, 2) (2, 0) (4, 1) (5, 3) (7, 2) (8, 4)
方案 30: (0, 0) (1, 2) (2, 0) (4, 1) (6, 0) (7, 2) (8, 4)
方案 31: (0, 0) (2, 1) (3, 3) (5, 2) (6, 4) (7, 2) (8, 4)
方案 32: (0, 0) (2, 1) (3, 3) (5, 2) (6, 0) (7, 2) (8, 4)
方案 33: (0, 0) (2, 1) (3, 3) (4, 1) (5, 3) (7, 2) (8, 4)
方案 34: (0, 0) (2, 1) (3, 3) (4, 1) (6, 0) (7, 2) (8, 4)
方案 35: (0, 0) (2, 1) (4, 2) (6, 3) (8, 4)
方案 36: (0, 0) (2, 1) (4, 0) (5, 2) (6, 4) (7, 2) (8, 4)
方案 37: (0, 0) (2, 1) (4, 0) (5, 2) (6, 0) (7, 2) (8, 4)
总方案数: 37
```

【编程经验】用结构数组代替平行数组

```
#include <iostream>          // cout
using namespace std;

struct position { int x, y; };
position dxy[4] = {{1,2}, {2,1}, {2, -1}, {1, -2}};
position start_pos = {0, 0};
position path[100];
int num;

void Jump(position pos, int step);

int main() {
    num = 0;                        // 初始方案数置0
    Jump(start_pos, 0);             // 跳第一步
    return 0;
}
```

【编程经验】用结构数组代替平行数组

```
void Jump(position pos, int step) {
    // 是否到达目标?
    if ((pos.x == 8) && (pos.y == 4)) {
        num++;    // 方案数加1
        cout << num << ": ";
        for (int i=0; i<step; i++) // 从起点开始输出各步的坐标
            cout << "(" << path[i].x << ", "<< path[i].y << ") ";
        cout << endl;
        return;
    }

    // 遍历四种跳步方向
    for (int k=0; k<4; k++) {
        position next_pos = {pos.x + dxy[k].x, pos.y + dxy[k].y};
        // 检查next_pos是否可行?
        if ((next_pos.x < 0) || (next_pos.x > 8) ||
            (next_pos.y < 0) || (next_pos.y > 4)) continue;

        // 记录方案! 结构变量可以直接赋值!
        path[step] = next_pos;

        // 跳下一步
        Jump(next_pos, step+1);
    }
}
```

【编程经验】定义函数检查落点坐标

```
#include <iostream>          // cout
using namespace std;

struct position { int x, y; };
position dxy[4] = {{1,2}, {2,1}, {2, -1}, {1, -2}};
position start_pos = {0, 0}, goal_pos = {8, 4};
position path[100];
int num;

void Jump(position pos, int step);

int main() {
    num = 0;                        // 初始方案数置0
    Jump(start_pos, 0);             // 跳第一步
    return 0;
}
```

【编程经验】定义函数检查落点坐标

```
bool IsValid(position pos) {
    return (pos.x >= 0) && (pos.x <= 8) && (pos.y >= 0) && (pos.y <= 4);
}

bool IsGoal(position pos) { return (pos.x == goal_pos.x) && (pos.y == goal_pos.y); }

void Jump(position pos, int step) {
    if (IsGoal(pos)) { // 是否到达目标?
        num++;    // 方案数加1
        cout << num << ": ";
        for (int i=0; i<step; i++) // 从起点开始输出各步的坐标
            cout << "(" << path[i].x << ", "<< path[i].y << ") ";
        cout << endl;
        return;
    }
    for (int k=0; k<4; k++) { // 遍历四种跳步方向
        position next_pos = {pos.x + dxy[k].x, pos.y + dxy[k].y};
        if (!IsValid(next_pos)) continue; // 检查next_pos是否可行?
        path[step] = next_pos;  // 记录这一步的方案
        Jump(next_pos, step+1); // 跳下一步
    }
}
```

【任务1.3】分书

有编号分别为 0、1、2、3、4 的五本书，准备分给A、B、C、D、E五个人。请你写一个程序，输出所有的分书方案，要求每个分书方案都能让每个人都皆大欢喜（即每人都分到感兴趣的书）。

假定这5个人对5本书的阅读兴趣如下表：

	0	1	2	3	4
A	0	0	1	1	0
B	1	1	0	0	1
C	0	1	1	0	1
D	0	0	0	1	0
E	0	1	0	0	1

解题思路（数据结构设计）

1、阅读兴趣用一个二维数组描述：

```
int like[5][5] = {{0, 0, 1, 1, 0},
                  {1, 1, 0, 0, 1},
                  {0, 1, 1, 0, 1},
                  {0, 0, 0, 1, 0},
                  {0, 1, 0, 0, 1}};
```

like[i][j]

- 1：编号 i 的人 喜欢编号 j 的书籍
- 0：编号 i 的人不喜欢编号 j 的书籍

解题思路（数据结构设计）

2、书籍状态用一个一维数组描述：

```
int assigned[5];
```

数组元素存储的是分配到下标对应书本的读者编号。

若 assigned[book_id] == -1 则表示book_id这本书没有分配。

注意：数组下标是书的编号。

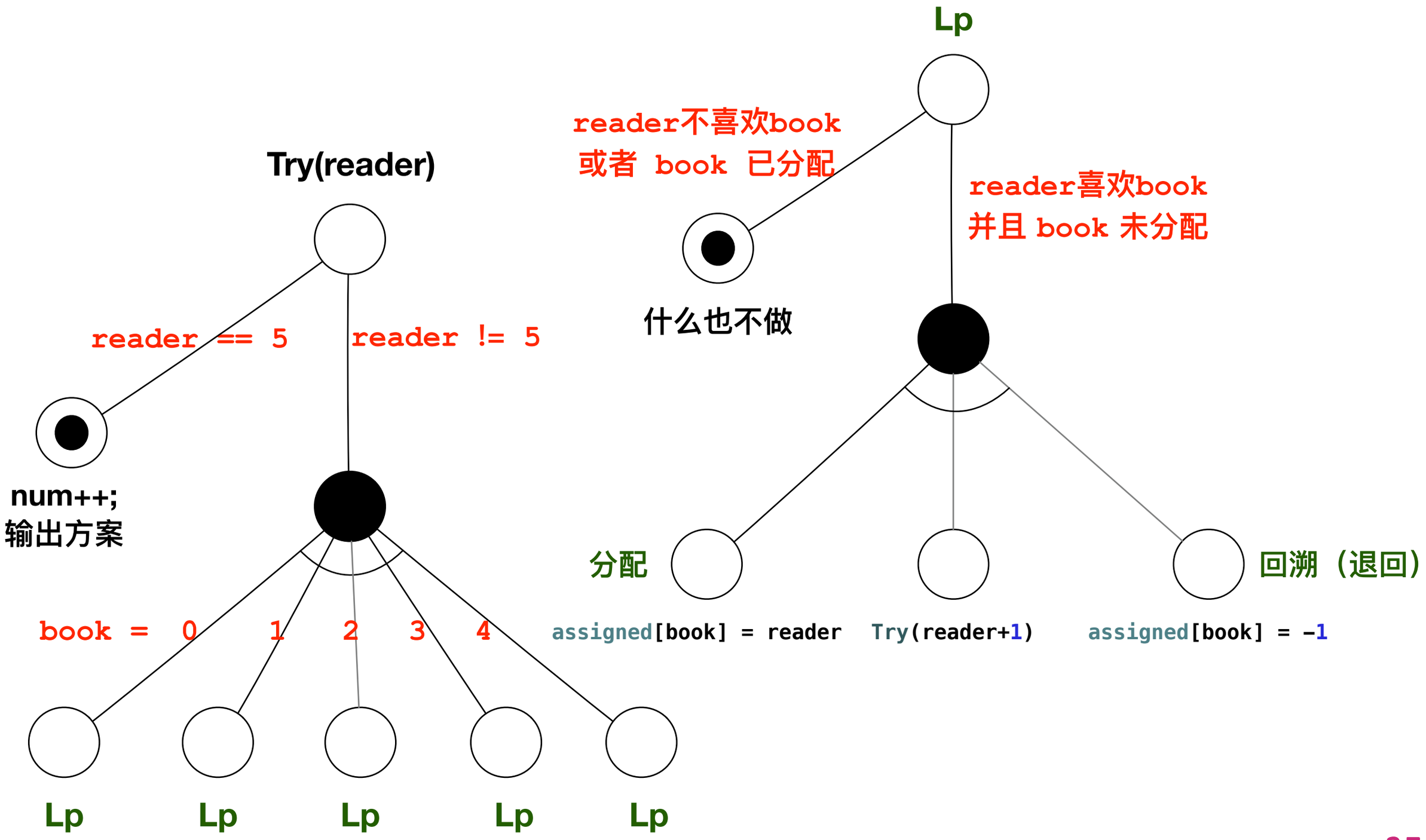
开始时，设置所有书本均未分配出去。有两种方法：

```
int assigned[5] = { -1, -1, -1, -1, -1 };
```

或者：

```
for (int i=0; i<5; i++) assigned[i] = -1;
```

解题思路（与或图）


Lp
reader不喜欢book
或者 book 已分配
reader喜欢book
并且 book 未分配
Try(reader)
什么也不做
reader == 5
reader != 5
num++;
输出方案
分配
回溯（退回）
book = 0 1 2 3 4
assigned[book] = reader
Try(reader+1)
assigned[book] = -1
Lp Lp Lp Lp Lp

```cpp
#include <iostream>          // cout
using namespace std;
/// 读者与书本的编号都是基于0的
int like[5][5] = {  {0, 0, 1, 1, 0},
                    {1, 1, 0, 0, 1},
                    {0, 1, 1, 0, 1},
                    {0, 0, 0, 1, 0},
                    {0, 1, 0, 0, 1}  };
int num; /// 方案数
int assigned[5]; /// assigned[book_id] = reader_id，值为-1表示没有被分配
void Try(int reader);

int main() {
    // 设置分书方案数初始值为0
    num = 0;

    /// 设置书本初始状态为未分配
    for (int book = 0; book < 5; book++) assigned[book] = -1;

    /// 从第0个读者开始，寻找所有分书方案
    Try(0);

    return 0;
}
```

```
void Try(int reader) {
    /// 递归中止条件：所有读者均已分配合适书籍
    if (reader == 5) {
        num++;
        cout << "第" << num << "个方案（5本书的读者编号）：";
        for (int i = 0; i < 5; i++) cout << assigned[i] << ' ';
        cout << endl;
        return;
    }
    /// 逐一为每本书找到合适的读者
    for (int book = 0; book < 5; book++) {
        /// 是否满足分书条件
        if ((like[reader][book] != 1) || assigned[book] != -1) continue;
        /// 记录当前这本书的分配情况
        assigned[book] = reader;
        /// 为下一位读者分配合适书籍
        Try(reader+1);
        /// 将书退还（回溯），尝试另一种方案
        assigned[book] = -1;
    }
}
```

【编程技巧】能否不使用回溯?

```
#include <iostream>          // cout
using namespace std;

int like[5][5] = {  {0, 0, 1, 1, 0},
                    {1, 1, 0, 0, 1},
                    {0, 1, 1, 0, 1},
                    {0, 0, 0, 1, 0},
                    {0, 1, 0, 0, 1}  };
/// 方案数
int num;
/// 分配方案：记录5本书分别分给谁（用户编号）
struct assign_state { int assigned[5]; } state;
void Try(int reader, assign_state state);

int main() {
    num = 0; // 分书方案数初始值
    for (int book = 0; book < 5; book++) state.assigned[book] = -1;
    Try(0, state); // 从第0个人（A）开始分书
    return 0;
}
```

```
void Try(int reader, assign_state state) {
    /// 递归中止条件：所有读者均已分配合适书籍
    if (reader == 5) {
        num++;
        cout << "第" << num << "个方案（5本书的读者编号）：";
        for (int i = 0; i < 5; i++) cout << state.assigned[i] << ' ';
        cout << endl;
        return;
    }
    /// 逐一为每本书找到合适的读者
    for (int book = 0; book < 5; book++) {
        /// 是否满足分书条件
        if (like[reader][book] != 1 || state.assigned[book] != -1)
            continue;

        /// 记录当前这本书的分配情况
        assign_state next_state = state; /// 产生新的状态变量
        next_state.assigned[book] = reader;

        /// 为下一位读者分配合适书籍
        Try(reader+1, next_state);
    }
}
```

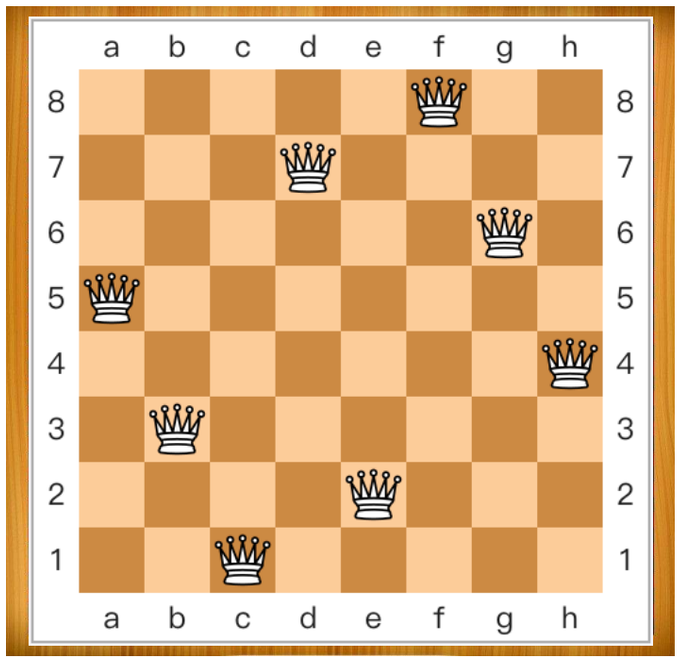
不用回溯了

【任务1.4】八皇后问题

在国际象棋的棋盘上，放置8个皇后（棋子），使皇后两两之间互不攻击。

八皇后问题的来源：国际西洋棋棋手马克斯·贝瑟尔于1848年提出。八皇后问题的第一个解在1850年由弗朗兹·诺克给出，他也是将问题推广到N皇后摆放的人之一。

Chess composer **Max Bezzel** published the eight queens puzzle in 1848. **Franz Nauck** published the first solutions in 1850. Nauck also extended the puzzle to the n queens problem, with n queens on a chessboard of n × n squares.



所谓互不攻击，是说任何两个皇后都要满足：

（1）不在棋盘的同一行；

（2）不在棋盘的同一列；

（3）不在棋盘的同一对角线上。

解题思路1：枚举八个皇后的位置?

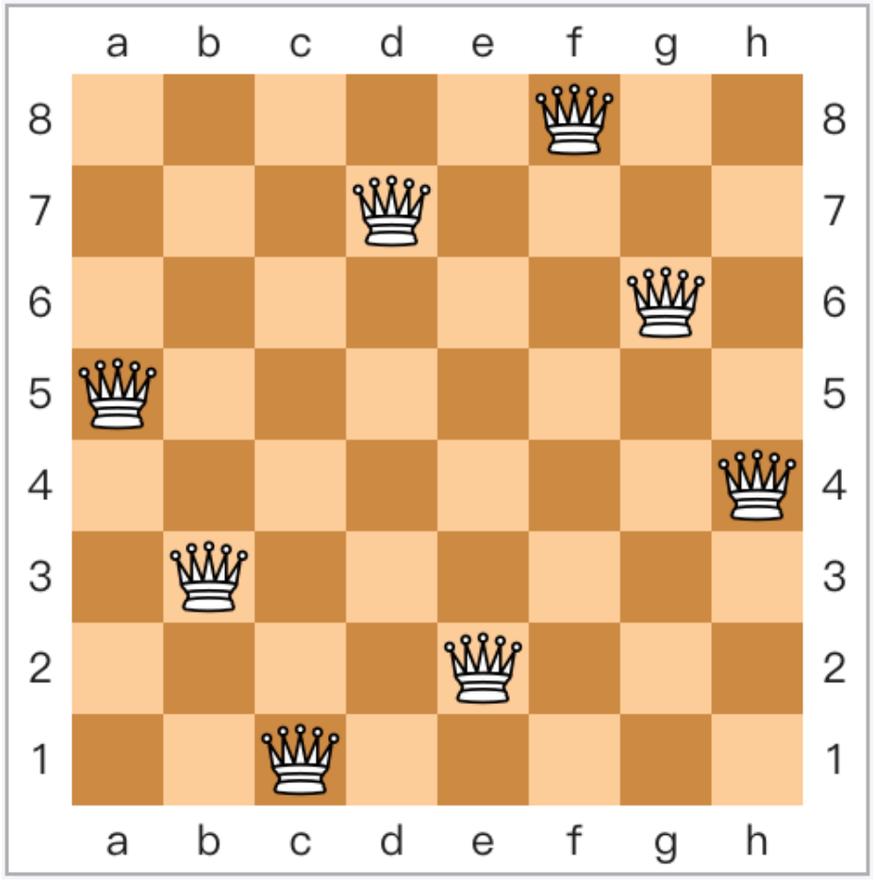
```
int main() {
    int q[9], num = 0;
    for (q[1] = 1; q[1] <= 8; q[1] ++) {
        for (q[2] = 1; q[2] <= 8; q[2] ++) {
            for (q[3] = 1; q[3] <= 8; q[3] ++) {
                for (q[4] = 1; q[4] <= 8; q[4] ++) {
                    for (q[5] = 1; q[5] <= 8; q[5] ++) {
                        for (q[6] = 1; q[6] <= 8; q[6] ++) {
                            for (q[7] = 1; q[7] <= 8; q[7] ++) {
                                for (q[8] = 1; q[8] <= 8; q[8] ++) {
                                    if (IsSafe(q)) {
                                        num ++;
                                        cout << num << ": ";
                                        for (int i = 1; i <= 8; i++) cout << q[i] << ' ';
                                        cout << endl;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return 0;
}
```

过于暴力了

解题思路1：枚举八个皇后的位置?

```
bool IsSafe(int q[9])
{
    /// ???
}
```

难点在这里!



已知8个皇后的位置，如何判断它们是否可以相互攻击？即是否有两个以上的皇后位于同一行、同一列、同一对角线上?

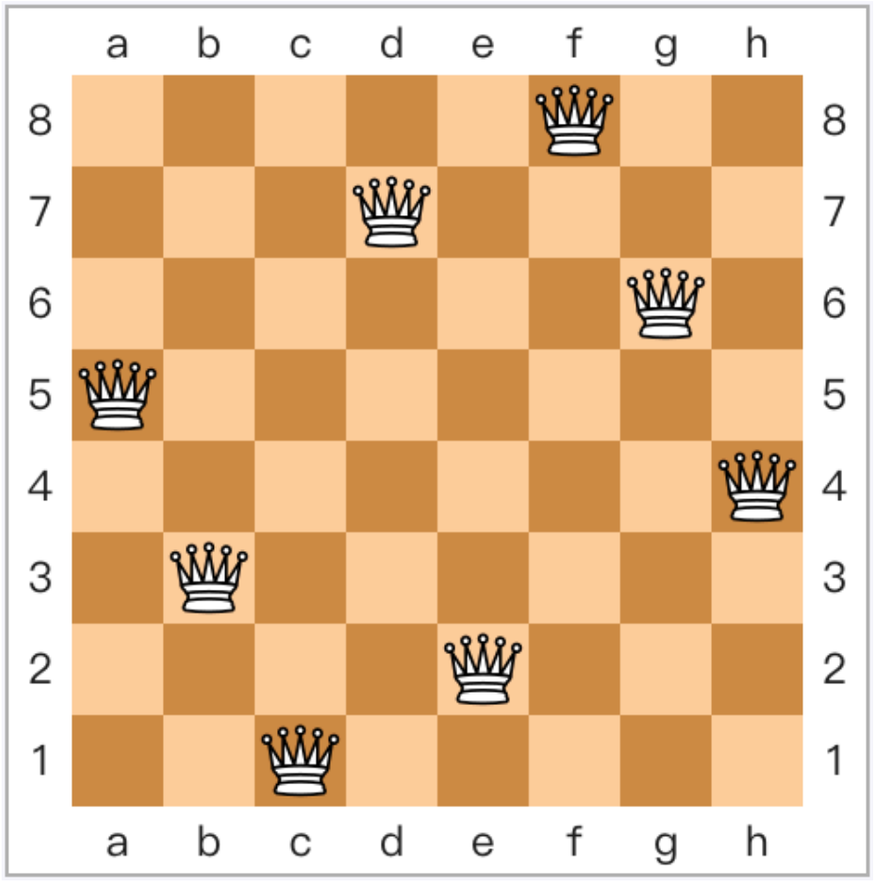
存在两种不同的算法（实现思路）：

1、以皇后为枚举对象，对皇后两两之间进行攻击性的判断，可以借用插入排序的思想（按行或列的次序，将后面的每一个皇后跟前面所有皇后依次进行攻击判断）。

2、以行、列、对角线为枚举对象，计算各方向上的皇后数量（类似于“词频统计”）。

解题思路2：何必暴力枚举?

- 从第一列开始逐一摆放皇后。存在8种可能的摆法，可放置在8行中的某一行上。接下来摆放后续7个皇后。
- 对于第i个皇后，检查第i列中的8个位置，找到不被任何前i-1个皇后攻击的位置。
- 当所有8个皇后都成功放入棋盘后，就找到了一个可行的解。
- 为了找到所有可能的解，需要记录下这一布局，然后继续检查其他可能的列。




```python
def solve():
    s = []
    stack = [[]]
    while stack != []:
        a = stack.pop()
        if len(a) == 8:
            s.append(a)
        else:
            for i in range(1, 9):
                if valid(i, a):
                    stack.append(a+[i])
    return s

def valid(x, a):
    y = len(a) + 1
    for i in range(1, y):
        if x == a[i-1] or abs(y - i) == abs(x-a[i-1]):
            return False
    return True

s = solve()
for r in s: print(r)
print("Total %d solutions." % len(s))
```

解题思路3：枚举思想+递归算法

假定按列摆放皇后。棋盘共有8列（横轴方向），每列能有且只能有一个皇后，至多能放8个皇后。因此，这8个皇后“**分别应该放在哪一行上（纵轴方向）？**”就成为求解任务要回答的问题。

我们采用试探方法：“向前走（放置皇后），碰壁回头（取走皇后）”的策略，即“回溯法”的解题思路。

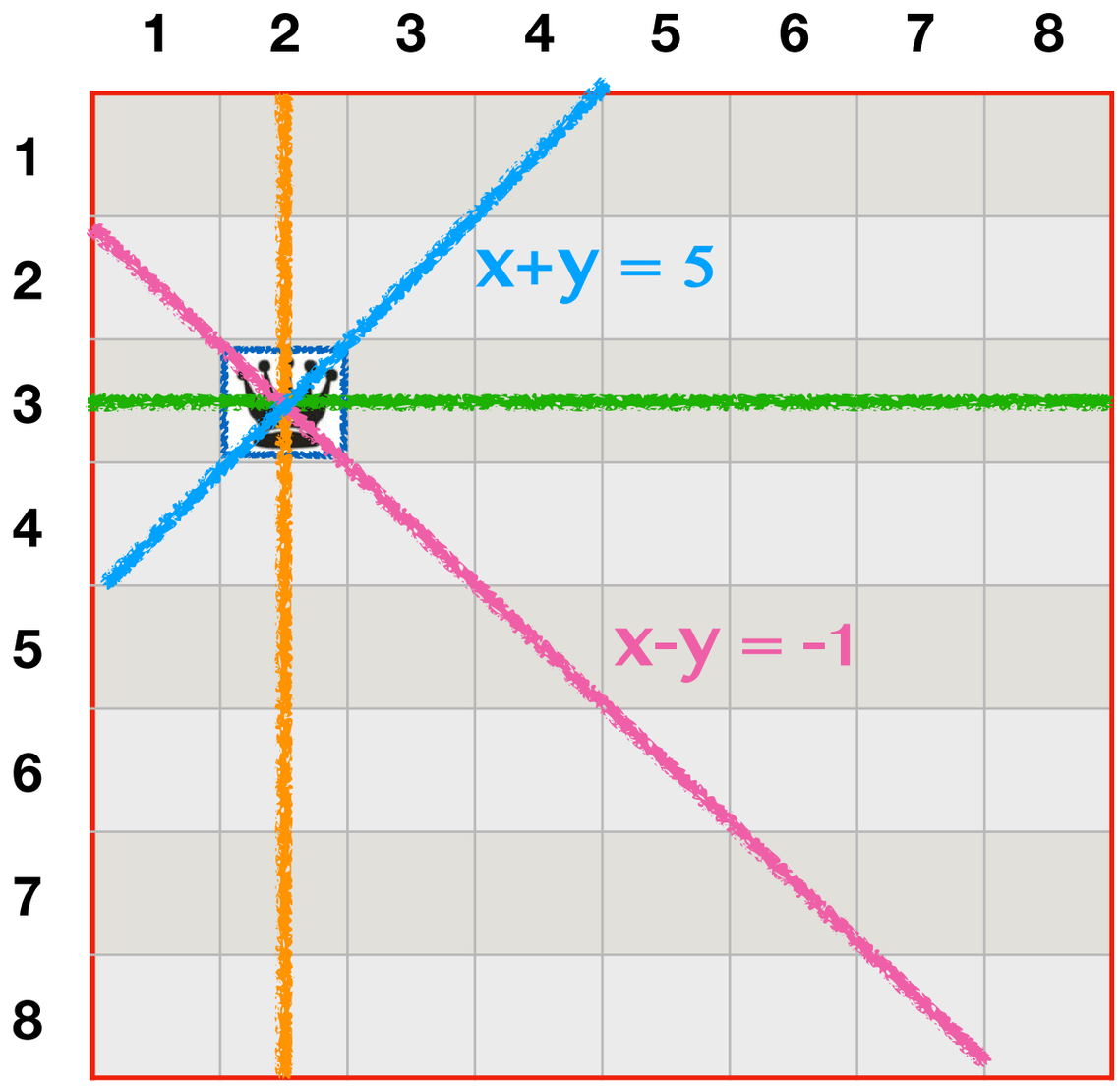
解题思路3：枚举思想+递归算法

定义函数Try(i)：将第i个皇后放到棋盘上。由于棋盘的对称性，我们假定是逐列放置皇后。

将第i列的皇后放在j行位置上之后（如果该位置是安全的话），棋盘各位置安全性将发生变化（对未来放置皇后过程有影响）。

在放第i列的皇后时，第i列上应还没有皇后（因为是按列依次来摆放的），不会在列上遭到其它皇后的攻击，因此只用考虑来自当前行和两个对角线上已有皇后的攻击。

对棋盘上指定位置的安全性进行判断

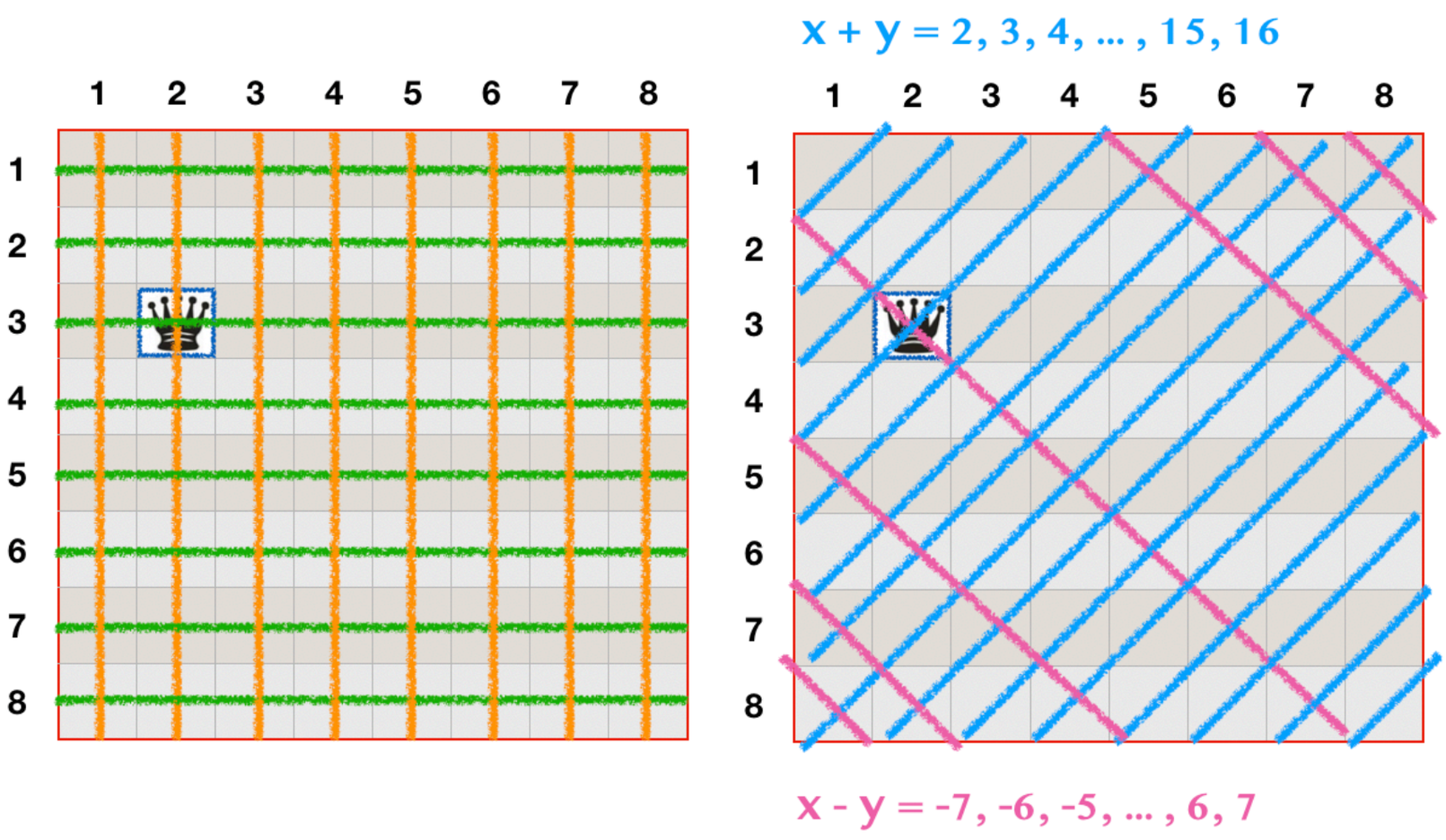


- 行：对比纵坐标
- 列：对比横坐标
- 对角线：比较横纵坐标的关系

$x+y=5$

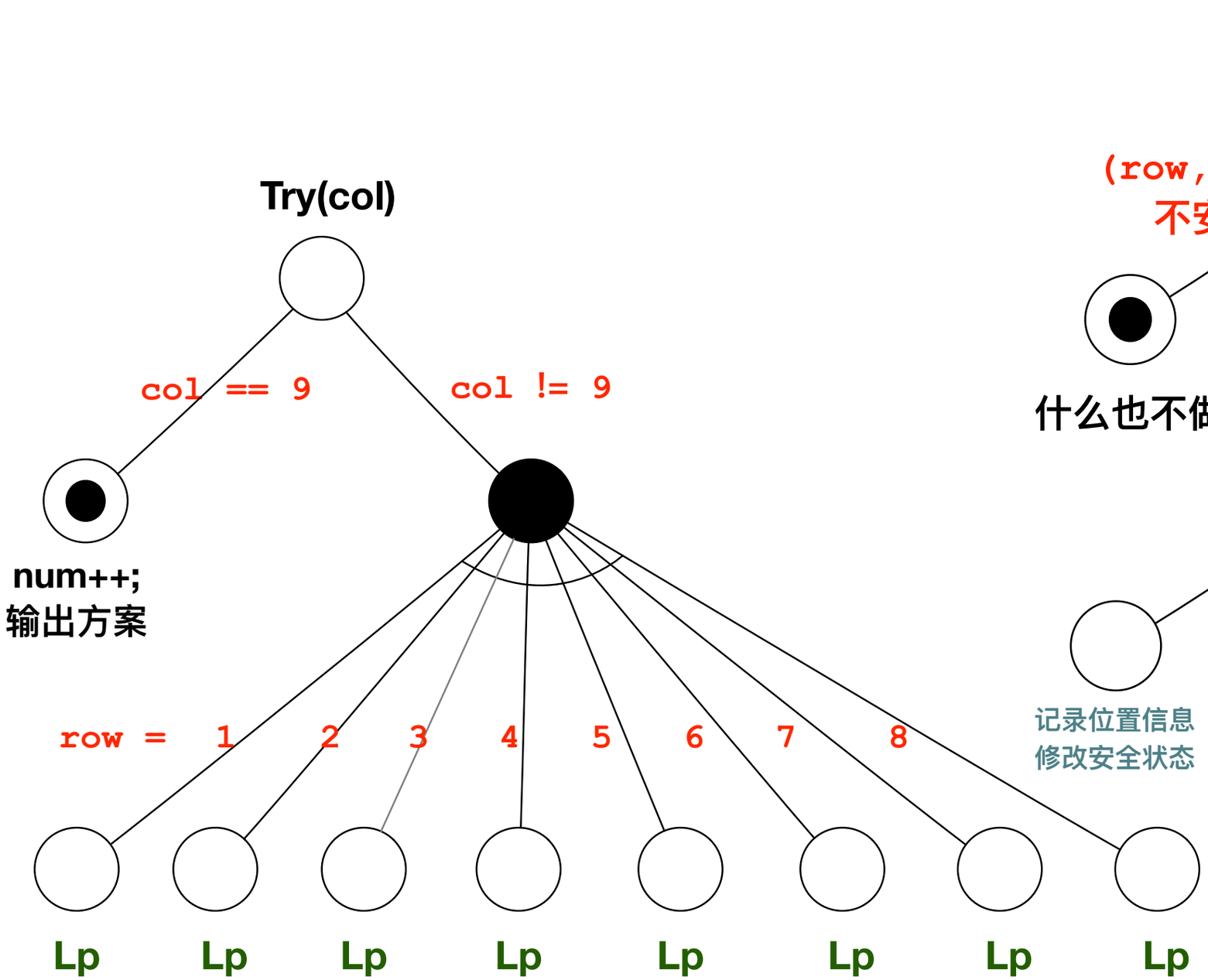
$x-y=-1$

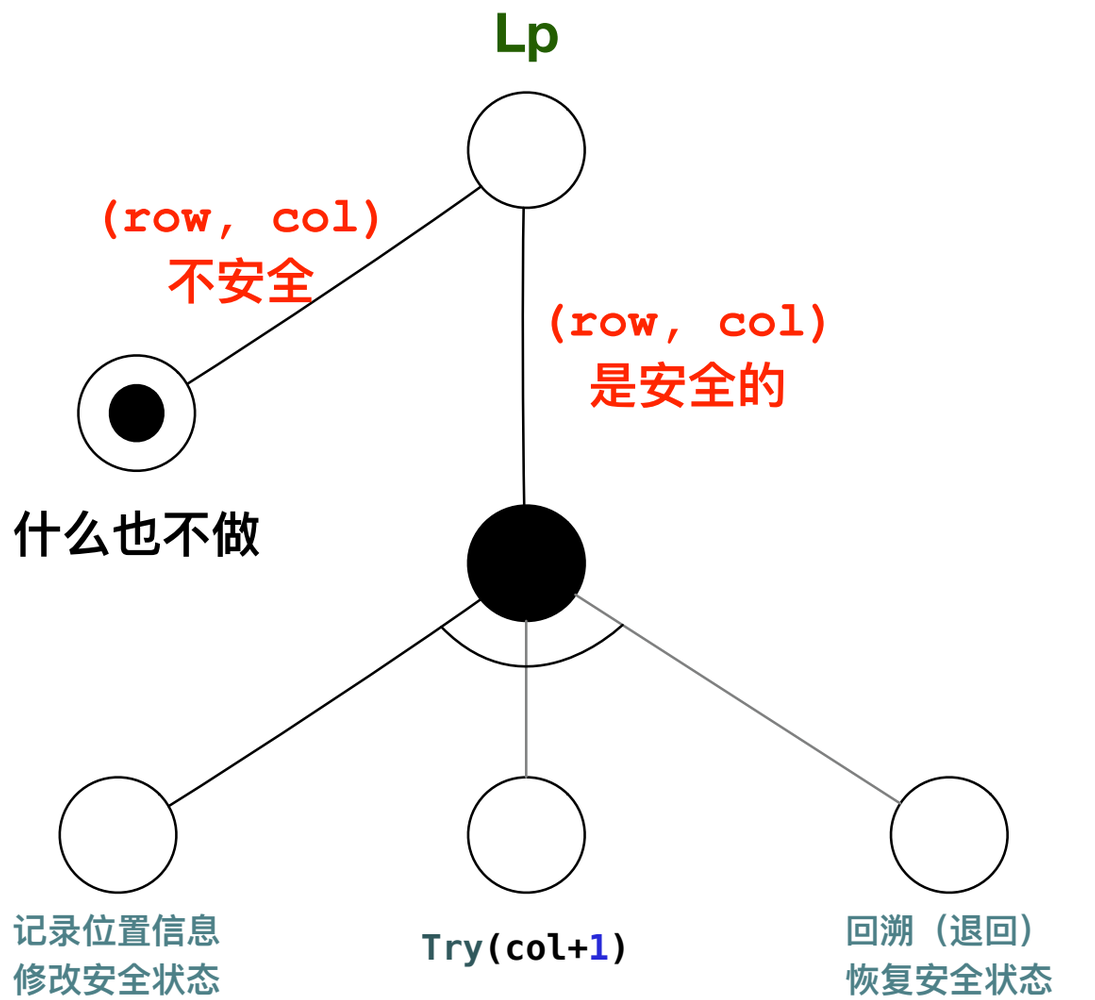
解题思路3（数据结构设计）




```
int Num; // 方案数

int Q[9]; // 8个皇后所占用的行号。下标是皇后的列号，值是皇后的行号。
bool S[9], L[17], R[17]; // 行、(i-j)对角线\、(i+j)对角线/是否安全
const int OFFSET = 9; // 用来统一数组下标范围为[2,3,...,16]
```

解题思路3（与或图）


Try(col)
col == 9
col != 9
num++;
输出方案
row = 1 2 3 4 5 6 7 8
Lp Lp Lp Lp Lp Lp Lp Lp


Lp
(row, col)
不安全
(row, col)
是安全的
什么也不做
记录位置信息
修改安全状态
Try(col+1)
回溯（退回）
恢复安全状态

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

int Num;          // 方案数
int Q[9];         // 8个皇后所占用的行号
bool S[9];        // S[1]~S[8]，当前行是否安全
bool L[17];       // L[2]~L[16]，(i-j)对角线是否安全
bool R[17];       // R[2]~R[16]，(i+j)对角线是否安全
const int OFFSET = 9;     // 用来统一数组下标范围[2,3,...,16]

void Try(int col);

int main() {
    Num = 0;
    for (int i = 0; i < 9; i++)  S[i] = true;
    for (int i = 0; i < 17; i++) L[i] = R[i] = true;

    Try(1);      /// 从第1列开始放皇后

    return 0;
}
```

```
void Try(int col) {
    /// 递归中止条件：所有列均已放上皇后了
    if (col == 9) {
        Num ++;

        cout << "方案" << Num << ": ";
        for (int k = 1; k <= 8; k++) cout << Q[k] << " ";
        cout << endl;

        return;
    }

    /// 依次尝试当前列的 8 行位置
    【详细代码见后续页】
}
```

```
    /// 依次尝试当前列的 8 行位置
    for (int row = 1; row <= 8; row++) {
        /// 判断拟放置皇后的位置是否安全
        if (!S[row] || !R[col + row] ||
            !L[col - row + OFFSET]) continue;

        Q[col] = row; /// 记录位置信息（行号）

        /// 修改三个方向的安全性标记
        S[row] = false;
        L[col - row + OFFSET] = false;
        R[col + row] = false;

        Try(col + 1); /// 递归尝试放下一列

        /// 回溯：恢复三个方向原有安全性
        S[row] = true;
        L[col - row + OFFSET] = true;
        R[col + row] = true;
    }
```

【编程技巧】能否不进行“回溯”?

```
int Num;                    // 方案数

struct place_state {
    int Q[9];                   // 8个皇后所占用的行号
    bool S[9];                  // S[1]~S[8]，当前行是否安全
    bool L[17];                 // L[2]~L[16]，(i-j)对角线是否安全
    bool R[17];                 // R[2]~R[16]，(i+j)对角线是否安全
};

const int OFFSET = 9;       // 调整一三象限对角线数组的下标范围
```

【编程技巧】能否不进行“回溯”?

```
void Try(int col, place_state state) {
    /// 递归中止条件：所有列均已放上皇后了
    if (col == 9) {
        Num ++;
        cout << "方案" << Num << ": ";
        for (int k = 1; k <= 8; k++) cout << state.Q[k] << " ";
        cout << endl;

        return;
    }
    /// 依次尝试当前列的 8 行位置
    【详细代码见后续页】
}
```

【编程技巧】能否不进行“回溯”?

```
/// 依次尝试当前列的 8 行位置
for (int row = 1; row <= 8; row++) {
    /// 判断拟放置皇后的位置是否安全
    if (!state.S[row] ||
        !state.R[col + row] || !state.L[col - row + OFFSET]) continue;

    place_state next_state = state;
    next_state.Q[col] = row;  /// 记录位置信息（行号）

    /// 修改三个方向的安全性标记
    next_state.S[row] = false;
    next_state.L[col - row + OFFSET] = false;
    next_state.R[col + row] = false;

    /// 递归尝试放下一列
    Try(col + 1, next_state);
}
```

【任务1.5】人鬼渡河

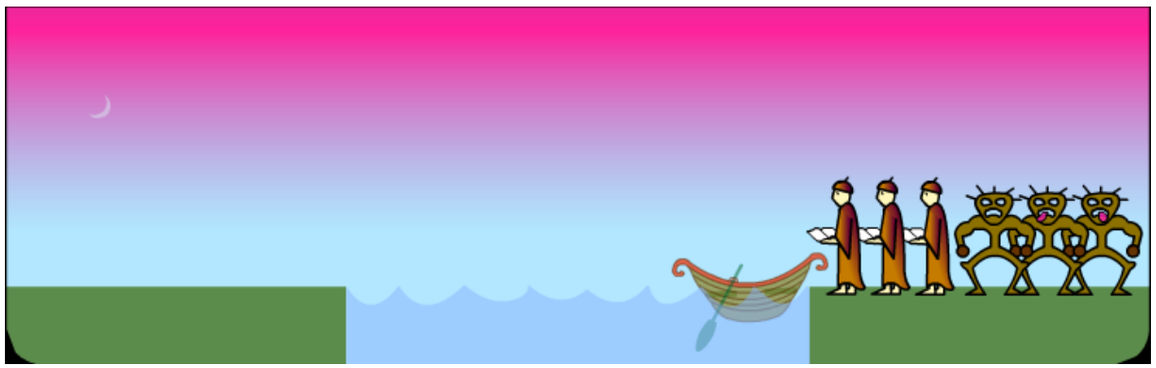
目标：将东岸的3人3鬼通过一只小船安全转移到西岸，希望摆渡次数尽可能少。

条件：

- 船的容量有限，一次最多只能坐2人（或2鬼或1人1鬼）。
- 无论是在河的东岸还是在河的西岸，一旦鬼数多于人数，则人将被鬼吃掉。
- 怎样渡河的大权掌握在人的手中。

说明：划船的时间忽略不计。船一靠岸即将船与岸视为一体，人和鬼即使还没有下船也视为已上岸。

任务：编写程序，求出一种渡河方案。



从实际操作中发现了哪些特点和规律?

目标：找一种“方案”，能将人鬼安全摆渡至对岸。

方案：一系列“指令”。

指令：？？？？？？？？

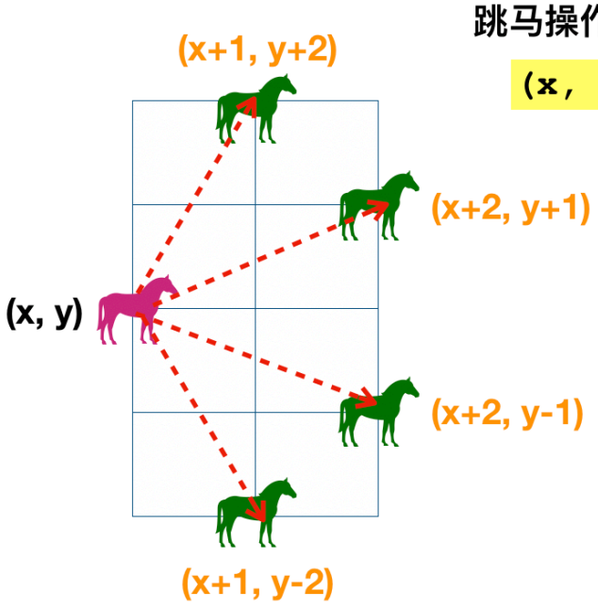
A ---> B :: 将一个事物，转变成另一个事物。

考虑到“计算机”的能力（与特长），指令必须是对“数”的运算（即将一个数转变成为另一个数）。

因此，确定“指令”之前，需要确定是哪些“数”在被计算！它们是什么?

【编程经验】学而习时之，温故而知新

数字化跳法，使操作可计算



跳马操作，转换成了位置（坐标）的变化

```
(x, y) -> (x + dx, y + dy)
```

跳法	dx	dy
0	1	2
1	2	1
2	2	-1
3	1	-2

解题思路（数据结构设计）

1、阅读兴趣用一个二维数组描述：

请注意二维数组的初始化方法（语法）

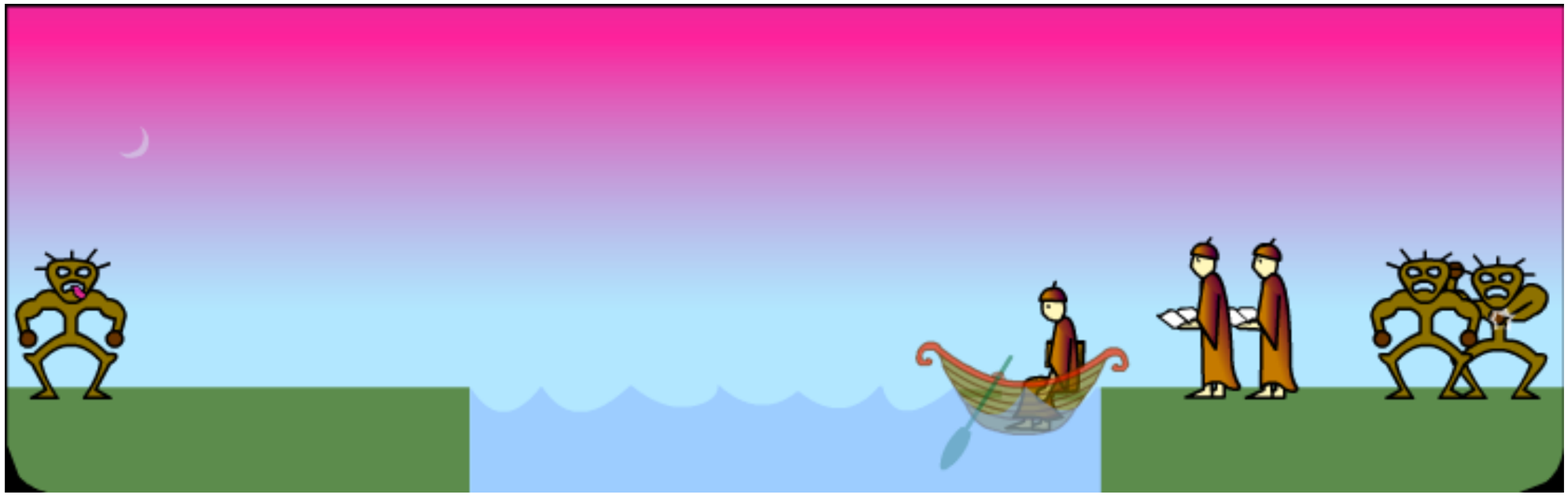
```
int like[5][5] = {{0, 0, 1, 1, 0},
                  {1, 1, 0, 0, 1},
                  {0, 1, 1, 0, 1},
                  {0, 0, 0, 1, 0},
                  {0, 1, 0, 0, 1}};
```

like[i][j]
- 1：编号 i 的人 喜欢编号 j 的书籍
- 0：编号 i 的人不喜欢编号 j 的书籍

在“跳马”和“分书”这两个问题中，解题的关键是：

- 将棋子移动过程转变成坐标平面上运动点的坐标变化， 以及将读者与书籍之间的约束条件（规则）用矩阵元素来表示。
- 求解过程是在所有可能的变化决策中，分阶段寻找满足要求的决策。这是一种特殊的枚举。

【编程技巧】思考问题的数学模型



对于上面的摆渡过程，**可被计算的到底是什么?**

或者说，玩游戏过程中的“操作”，对应的数学上的描述是什么?

再或者说：**操作过程中所改变的，能用什么数学概念来刻画?**

归根结底，问题的**数学模型**是什么?

考察变化的量，寻找可算的量

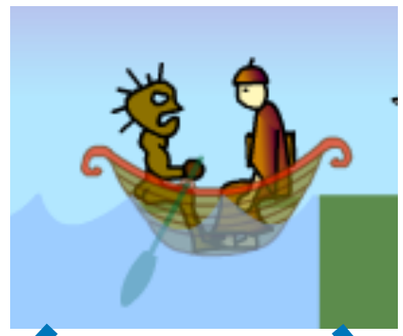
西岸的数量在变化

船中的数量在变化

东岸的数量在变化

考察变化的量，寻找可算的量

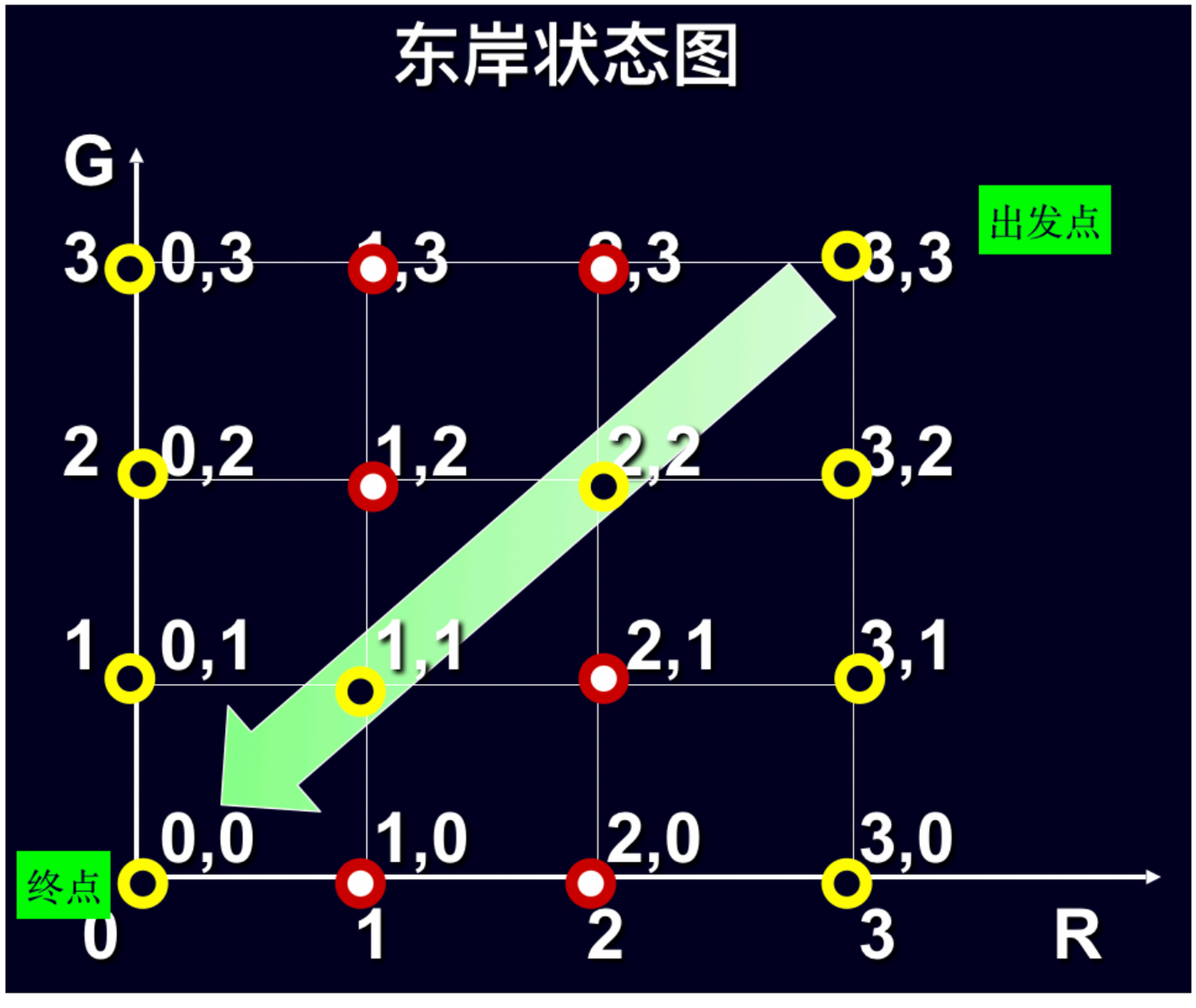
两岸人鬼数量的变化是由渡船上的人鬼数量与船的运动方向决定的！



A + B + C = D

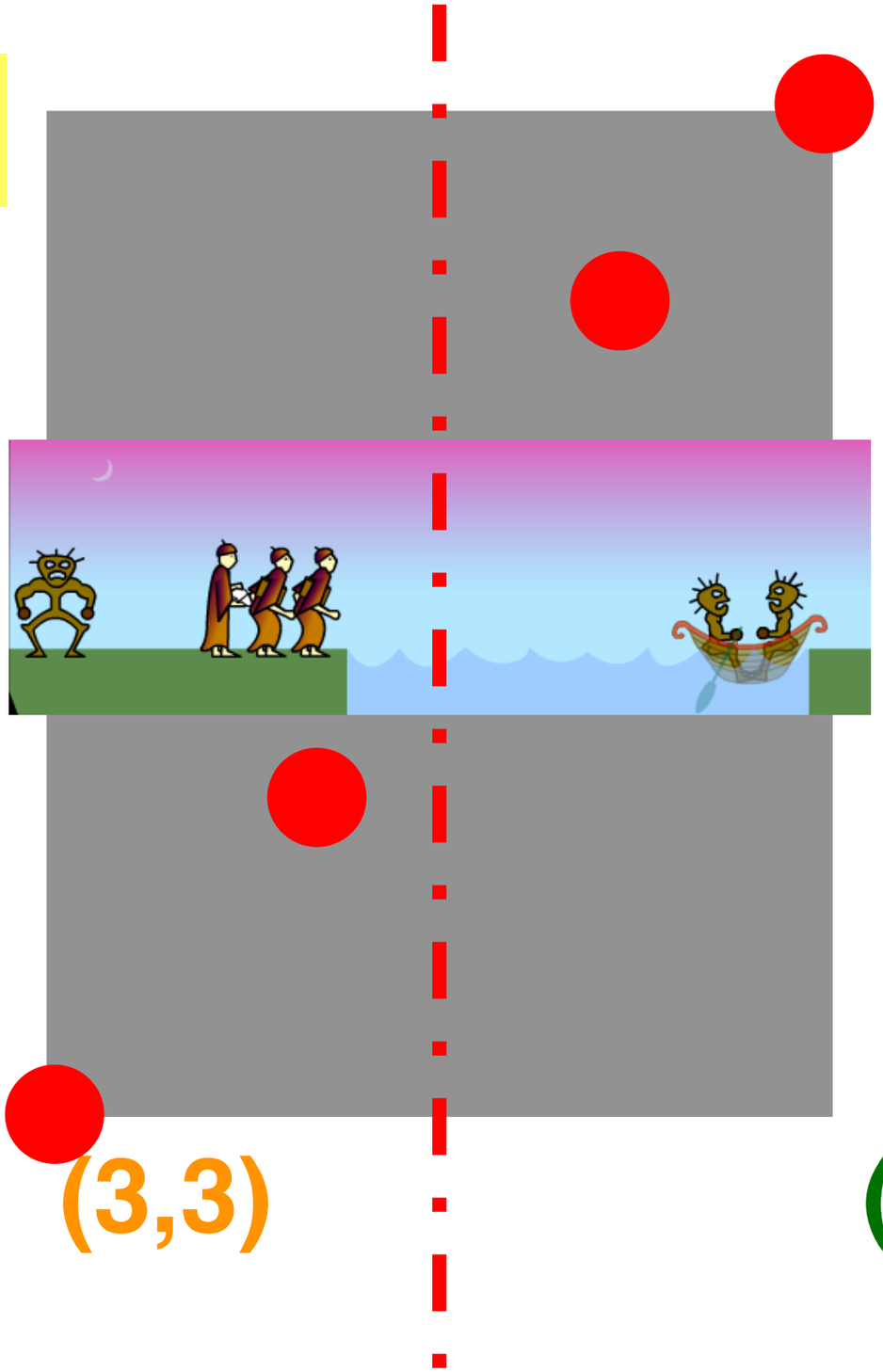
解题思路（数学模型）：状态数字化

设人数用R表示，鬼数用G表示，则每一个渡河中的“东岸场景”都与一个数对（R，G）相对应。东岸所有可能场景（人鬼数），会组成平面上的一个网格。



解题思路（数学模型）：操作数字化

$$S_{K+1} = S_K + \Delta_k$$



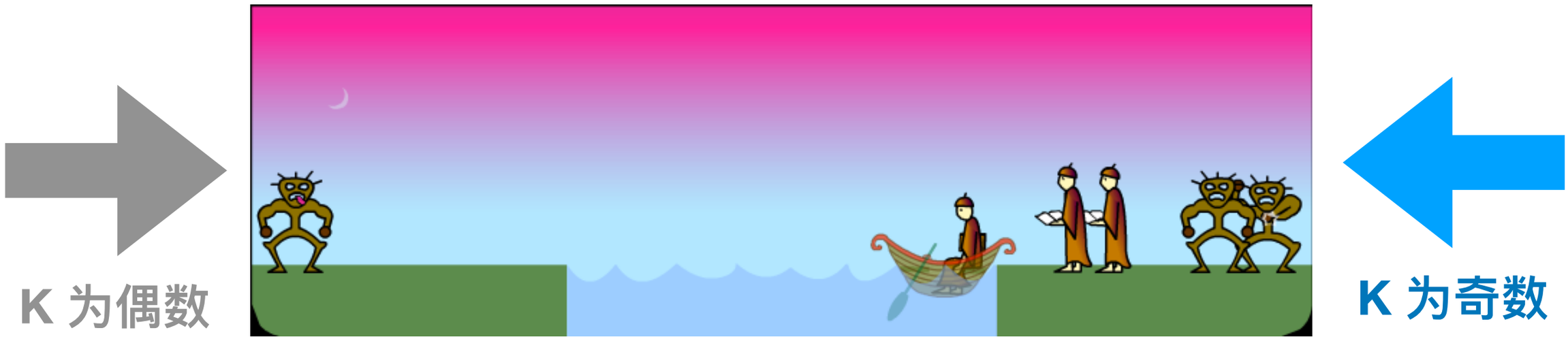
西岸

东岸

关于渡河的摆渡方向

设K为摆渡行船的次数（序号，从1开始计数），从东岸到西岸或从西岸到东岸记1次。显然，

- 船从东到西，K为奇数；
- 船从西到东，K为偶数。



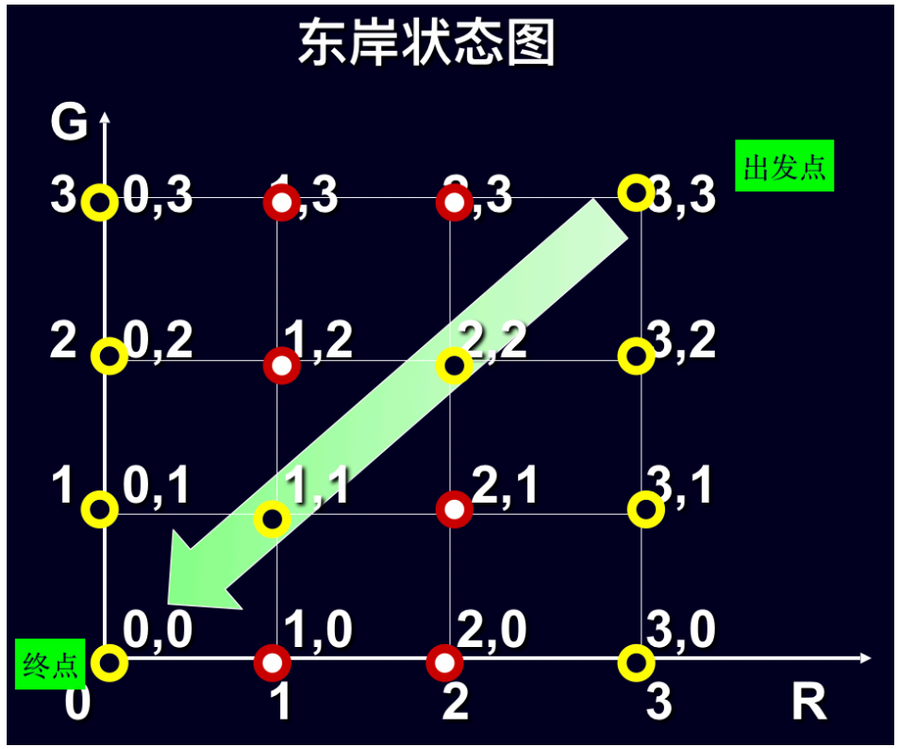
关于渡河的决策（操作）种类

定义一个2维向量d_K 为第K次渡河的摆渡策略：

$$d_K = (U_K, V_K)$$

其中，U_K为上船的人数，V_K为上船的鬼数，则所有合法（规则）的渡河摆渡决策集合D定义如下：

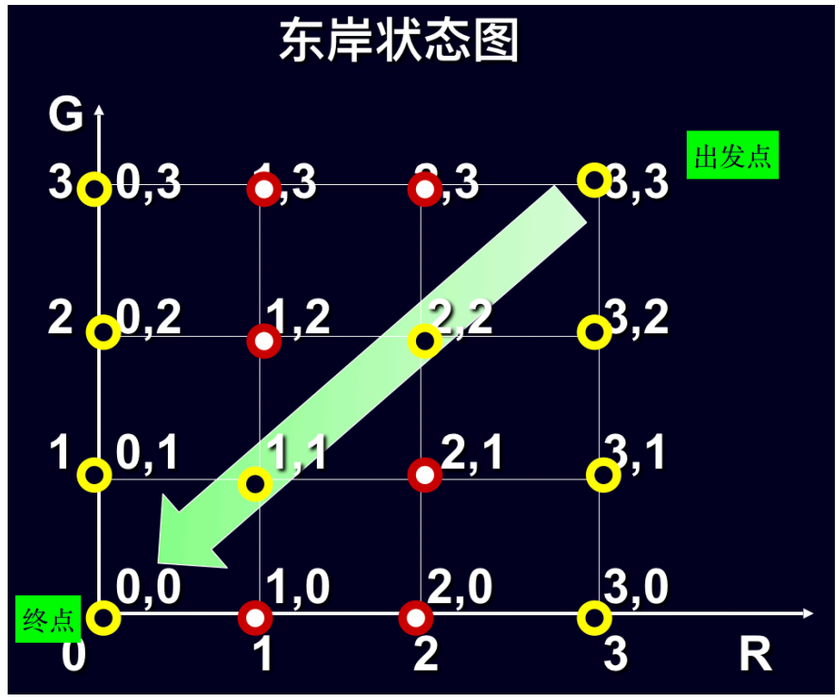
$$D = \{ (U,V) \mid U = 2,\ V = 0;$$
$$U = 1,\ V = 0;$$
$$U = 1,\ V = 1;$$
$$U = 0,\ V = 1;$$
$$U = 0,\ V = 2;$$
$$\}$$



关于渡河前东岸的安全状态

用2维向量$S_K = (R_K, G_K)$定义为第K次渡河前东岸的渡河状态，则安全的渡河东岸状态集合S定义如下：

S = { (R,G) | R = 0, G = 0, 1, 2, 3;
R = 3, G = 0, 1, 2, 3;
R = 1, G = 1;
R = 2, G = 2;
}

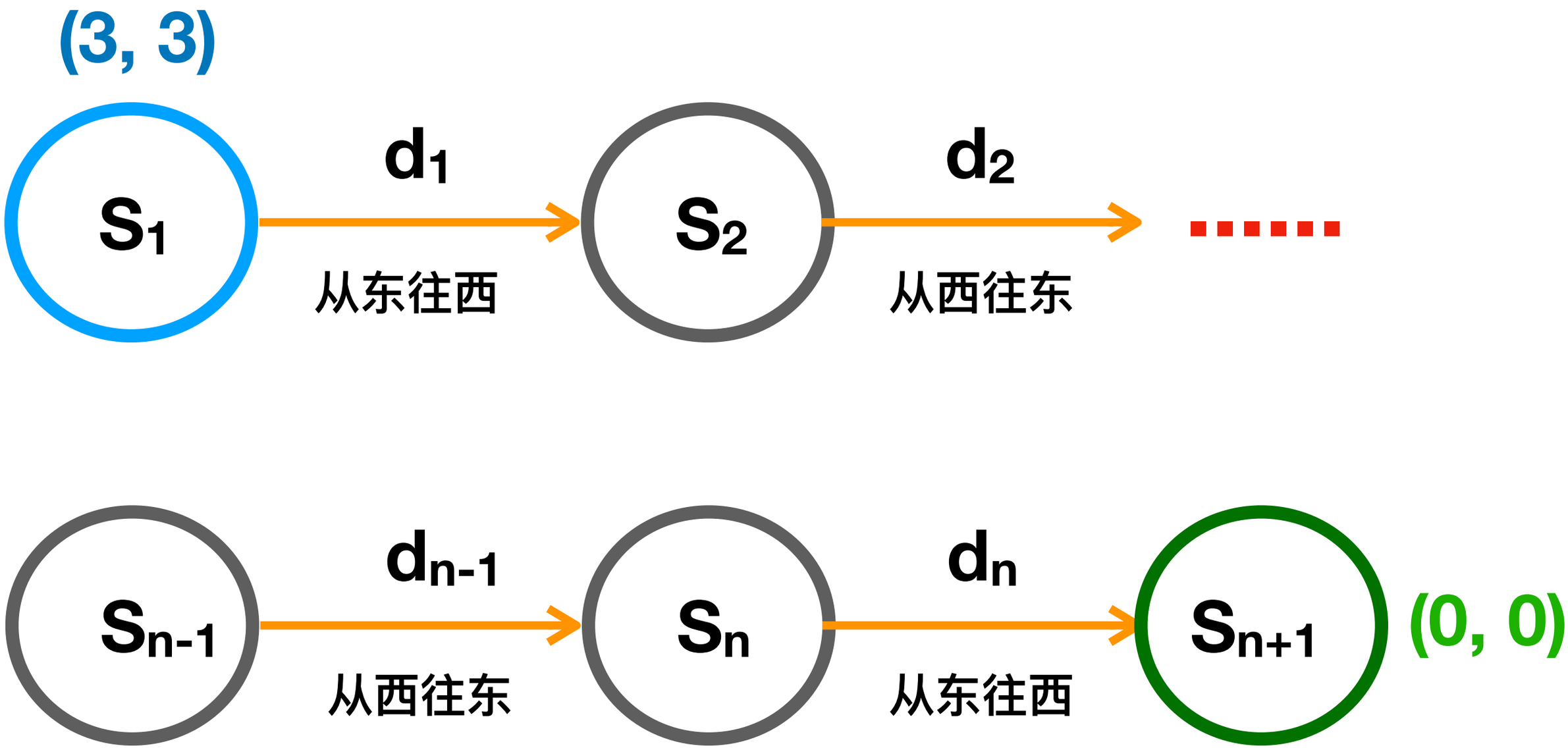


根据规则，推导出安全性检测标准：

在东岸，人数等于3或人数等于0，或人数鬼数相等

```
bool AQ = (u == 3) || (u == 0) || (u == v); //是否安全
if (!AQ) continue; //(2) 不安全，舍弃当前决策
```

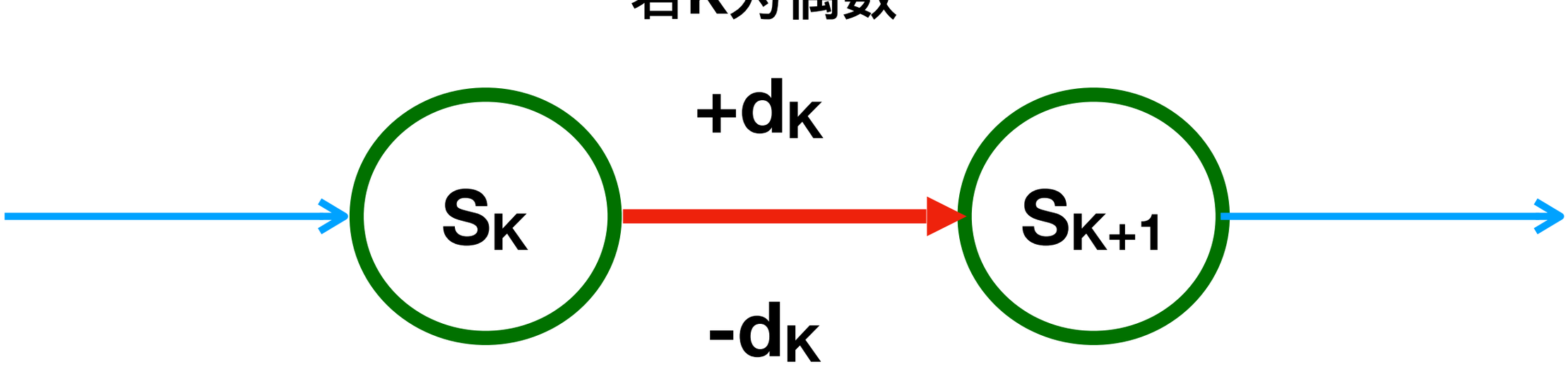
解题思路：多步决策，使用状态方程



解题思路：状态转移公式是关键

$$S_{K+1} = S_K + (-1)^K d_K$$

若K为偶数



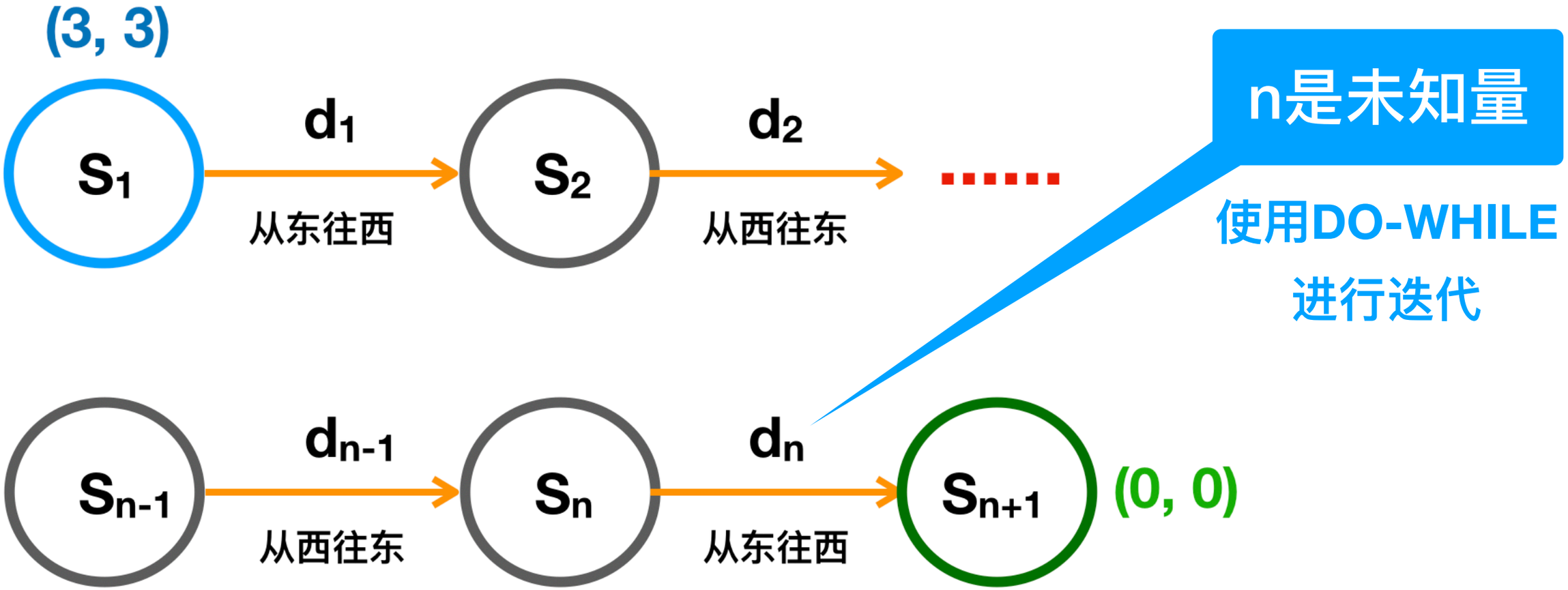
若K为奇数

注意：K的范围是[1..n]，从1开始，到未知待求的n结束。

算法实现：迭代 + 枚举 + 递推

$$S_{K+1} = S_K + (-1)^K d_K$$

- 迭代：从起始到结束
- 枚举：所有可能决策
- 递推：状态转移方程



解题思路（数据结构）

```
//定义描述渡河状态东岸人数与鬼数的结构变量。R: 状态中的人数, G: 状态中的鬼数
struct state { int R, G; };

state s[20];                    //结构数组记录渡河时的状态转移过程
int choice[20] = {0};    //记录状态转移过程的决策号，初始化都为0
int k;                          //状态号

//摆渡策略(数组)
state d[6] = {{0, 0}, //0号策略不用
              {2, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 2}};
```

决策编号：1　　2　　3　　4　　5

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

//定义描述渡河状态东岸人数与鬼数的结构变量。R: 状态中的人数, G: 状态中的鬼数
struct state { int R, G; };

state s[20];                //结构数组记录渡河时的状态转移过程
int choice[20] = {0};       //记录状态转移过程的决策号，初始化都为0
int k;                      //状态号

//摆渡策略(数组)
state d[6] = {{0, 0}, //0号策略不用
              {2, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 2}};

void display();             //输出渡河状态
void transfer_state();      //渡河状态转移函数

int main() {
    transfer_state();
    display();
    return 0;
}
```

```
//渡河状态转移函数
void transfer_state() {
    k = 1; //初始状态设为1
    s[1].R = 3; //初始状态东岸有3人
    s[1].G = 3; //初始状态东岸有3鬼

    do {
        int fx = 1; //摆渡方向，东向西或西向东
        if (k % 2 == 1) fx = -1; //奇数表明摆渡要从东岸到西岸
        int i; //决策号
        // 针对下一状态（次序为k+1，choice数组元素初值为0）
        // 依次尝试所有决策（决策号从1到5）
```

【此处详细代码见下一页】

```
        if (i > 5) {choice[k+1]=0; k--;} //所有摆渡决策都没成功，则需要回退
    } while (!(s[k].R == 0 && s[k].G == 0));    //目标是东岸既无人又无鬼
}     /// 持续迭代，直至到达终点（结束状态）
```

```
/// 针对下一状态（次序为k+1，choice数组元素初值为0），依次尝试所有决策（决策号从1到5）
 for (i = choice[k+1]+1; i <= 5; i++) {  //试探采用哪个决策能安全走1步
    int u = s[k].R + fx * d[i].R; //按第i号策略走1步东岸的人数
    int v = s[k].G + fx * d[i].G; //按第i号策略走1步东岸的鬼数

    if (u > 3 || v > 3 || u < 0 || v < 0) continue; //(1) 越界，舍弃当前决策

    bool AQ = (u == 3) || (u == 0) || (u == v); //是否安全
    if (!AQ) continue; //(2) 不安全，舍弃当前决策

    bool CHF = false; //是否重复
    //查历史信息（倒序），仅考虑摆渡方向一致的状态（增量为-2）
    for (int j = k - 1; j >= 1; j -= 2)
        if (s[j].R == u && s[j].G == v) CHF = true; //若人鬼数一致，则是重复状态
    if (CHF) continue; //(3) 重复，则舍弃当前决策，继续(continue)尝试下一决策

    k++; //按策略渡河，状态号加1
    s[k].R = u;   s[k].G = v;
    choice[k] = i; //记录决策号
    break; //已找到一个决策，跳出(break)循环，**暂停**尝试其他策略
}
```

```
void display() {
    for (int i = 1; i <= k; i++)
        cout << setw(2) << i << ": choice = " << choice[i] /// 决策号
             << " {" << d[choice[i]].R << "," << d[choice[i]].G << "}" /// 决策内容
             << " (" << s[i].R << "," << s[i].G << ") "/// 状态内容
             << endl;
}
```

程序输出结果

```
 1: choice = 0 {0,0} (3,3)
 2: choice = 1 {1,1} (2,2)
 3: choice = 4 {1,0} (3,2)
 4: choice = 5 {0,2} (3,0)
 5: choice = 2 {0,1} (3,1)
 6: choice = 3 {2,0} (1,1)
 7: choice = 1 {1,1} (2,2)
 8: choice = 3 {2,0} (0,2)
 9: choice = 2 {0,1} (0,3)
10: choice = 5 {0,2} (0,1)
11: choice = 2 {0,1} (0,2)
12: choice = 5 {0,2} (0,0)
Program ended with exit code: 0
```

【任务1.6】人鬼渡河2.0

目标：将东岸的3人3鬼通过一只小船安全转移到西岸，希望摆渡次数尽可能少。

条件：

- 船的容量有限，一次最多只能坐2人（或2鬼或1人1鬼）。
- 无论是在河的东岸还是在河的西岸，一旦鬼数多于人数，则人被鬼扔到河中。
- 怎样渡河的大权掌握在人的手中。

任务：编写程序，求出所有的渡河方案，要求没有多余的重复步骤。

解题思路：视渡河操作为特殊的“跳马”

视某一时刻东岸的（人，鬼）数对，是状态空间中的坐标点！

于是：

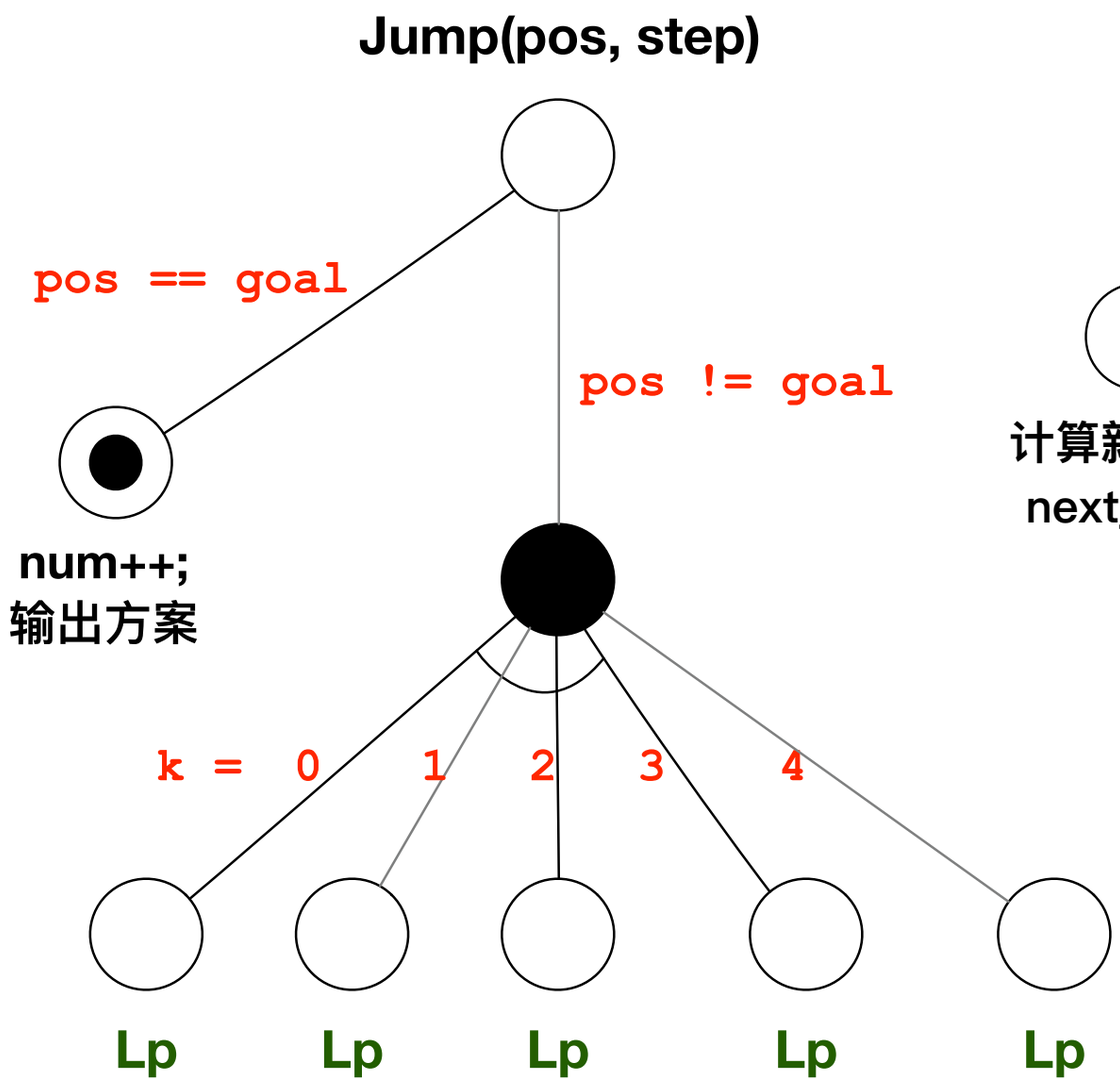
- 任务转换为从起点状态（3，3）“跳”到终点状态（0，0）
- 坐标变化操作规则按照题目关于安全性的要求，可得

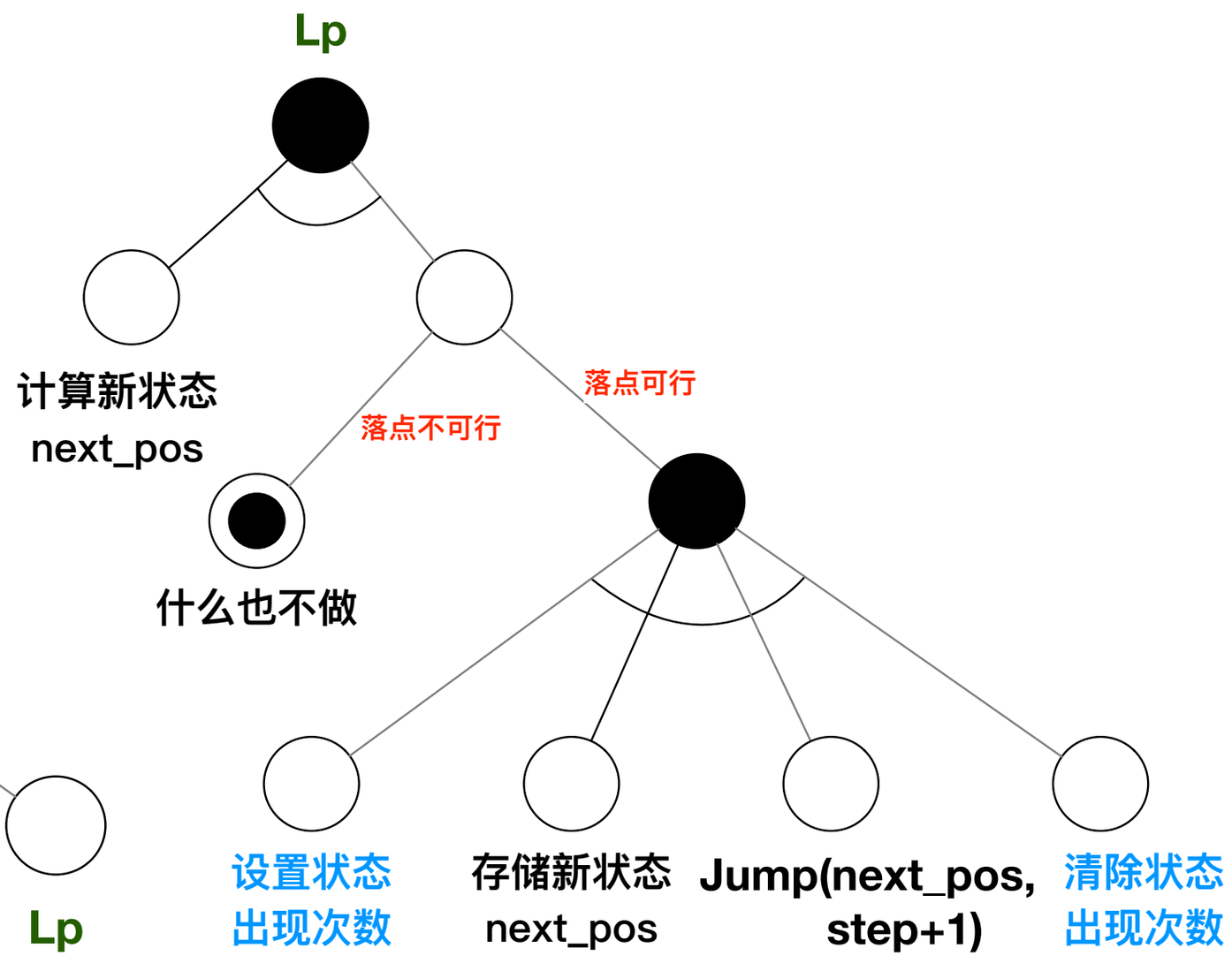
```
dxy[] = {{1,0}, {0,1}, {1,1}, {2,0}, {0,2}};
```

- 对每一个状态，枚举尝试所有可能的决策
- 若当前状态与终点状态一致，则输出此方案

使用 枚举+递归 来解决

解题思路（与或图）


Jump(pos, step)
pos == goal
num++;
输出方案
pos != goal
k = 0 1 2 3 4
Lp Lp Lp Lp Lp


Lp
计算新状态
next_pos
落点不可行
落点可行
什么也不做
设置状态
出现次数
存储新状态
next_pos
Jump(next_pos,
step+1)
清除状态
出现次数

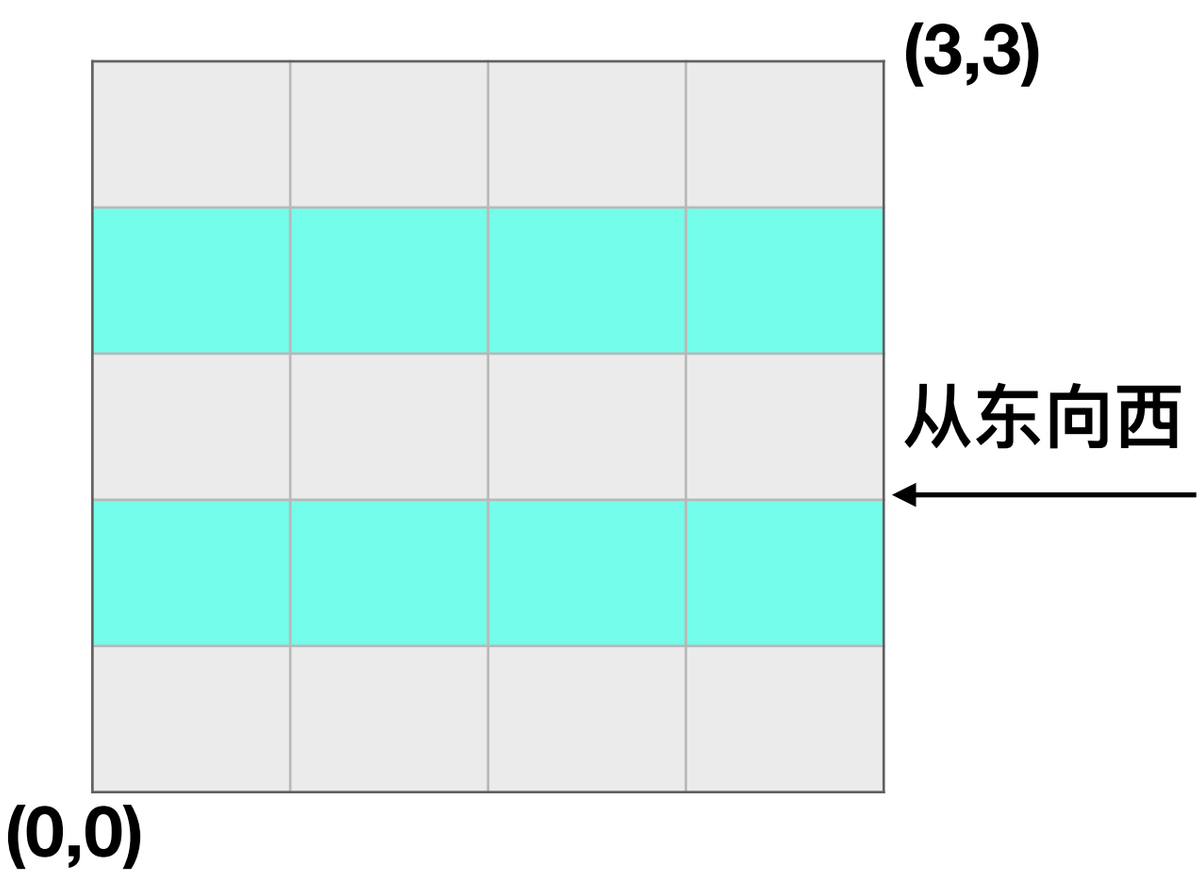
```cpp
#include <iostream>
using namespace std;

struct position { int x, y; };

// 不同决策对应的状态坐标变化
position dxy[] = {{1,0}, {0,1}, {1,1}, {2,0}, {0,2}};

// 起始状态，结束状态
position start_pos = {3, 3}, goal_pos = {0, 0};

// 决策序列的记录
position path[100];
int num; // 总的方案数

/// 记录出现次数，防止重复!!!
int pos_cnt[2][4][4] = {{{0}}};
```

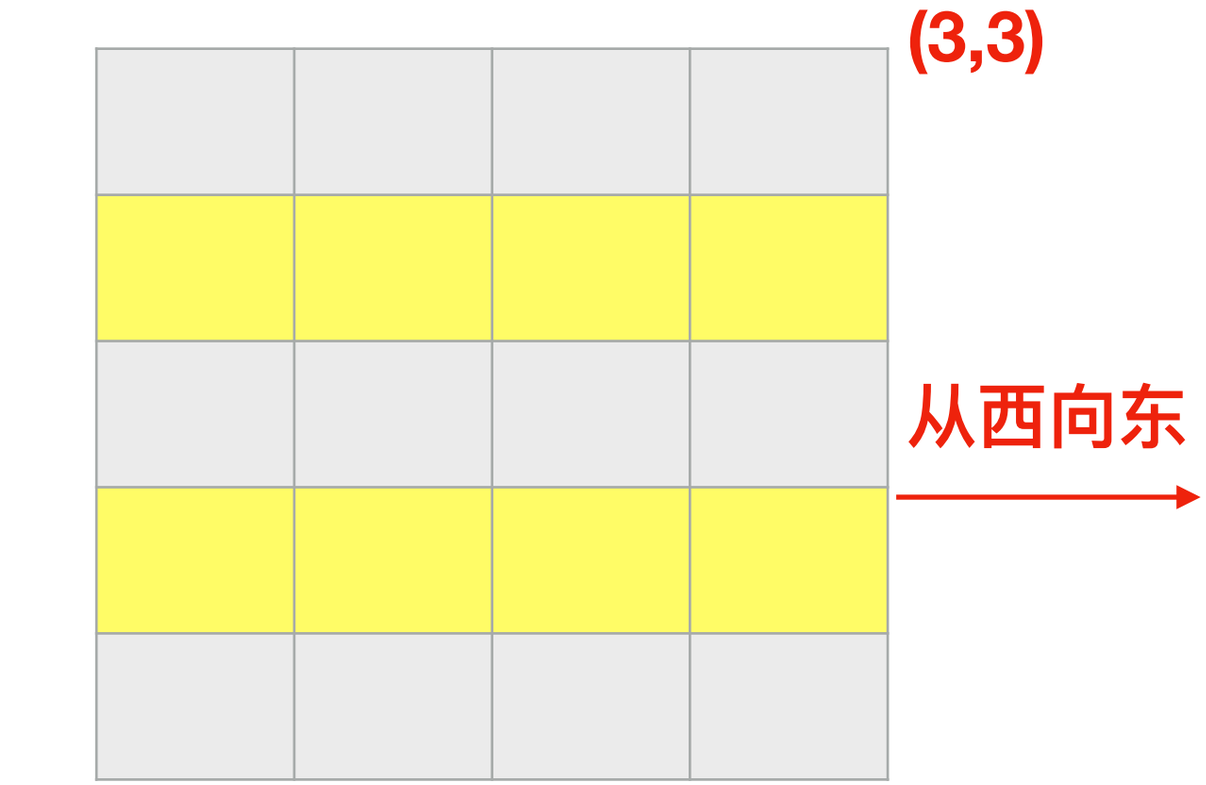
```
bool IsEq(position pos1, position pos2) {
    return (pos1.x == pos2.x) && (pos1.y == pos2.y);
}

bool IsDone(position pos) {
    return IsEq(pos, goal_pos);
}

bool IsValid(position pos, int step) {
    bool v, s, r;
    int dir = step % 2; /// 0: from left to right; 1: from right to left

    /// 1. 合法性检查
    v = (pos.x >= 0) && (pos.x <= 3) && (pos.y >= 0) && (pos.y <= 3);

    /// 2. 安全性检查
    s = (pos.x == pos.y || pos.x == 0 || pos.x == 3);  /// 根据游戏规则推导出来

    /// 3. 重复性检查
    r = pos_cnt[dir][pos.x][pos.y] == 0; /// 根据算法特点推导出来的

    return v && s && r;
}
```

【编程技巧】备忘录法（标记）防重复

```
/// 记录出现次数，防止重复!!!
int pos_cnt[2][4][4] = {{{0}}};
```

```
/// 3. 重复性检查
r = pos_cnt[dir][pos.x][pos.y] == 0; /// 根据算法特点推导出来
```

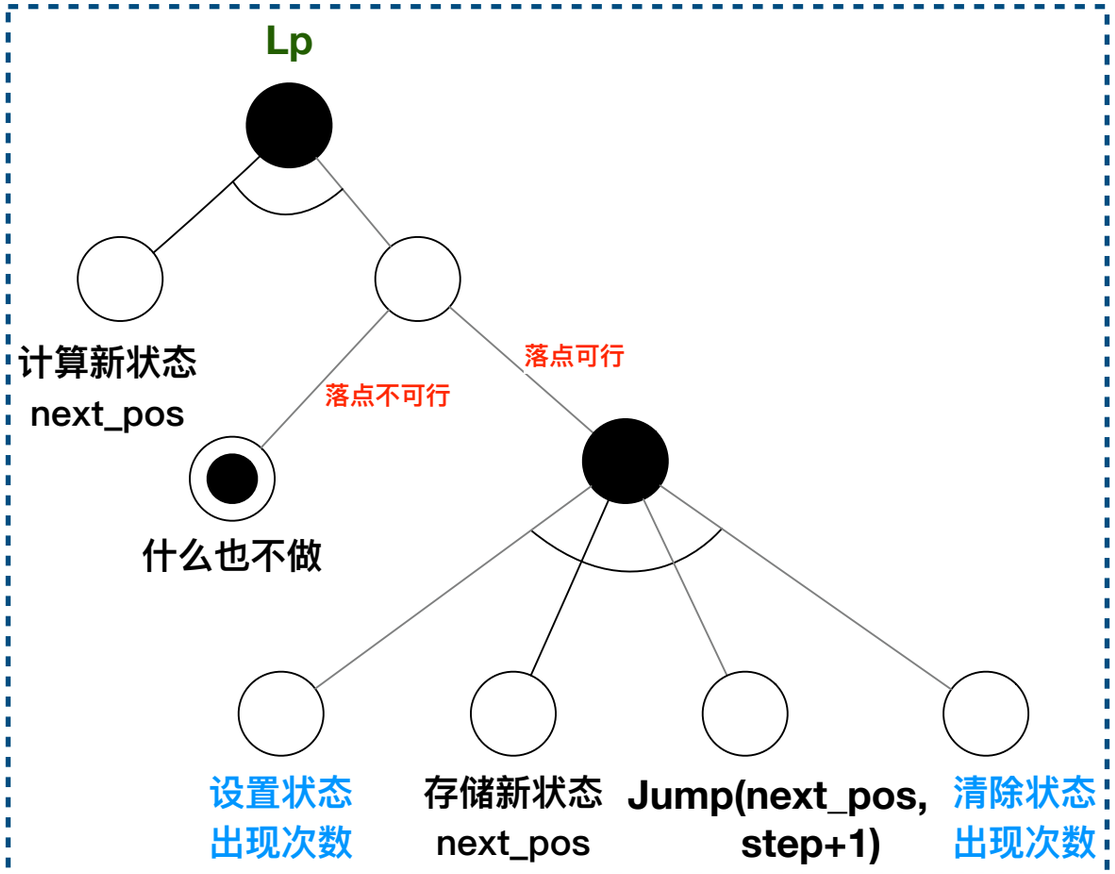




```
void SetCount(position pos, int step, int cnt) {
    int dir = step % 2; /// 0: from left to right; 1: from right to left
    pos_cnt[dir][pos.x][pos.y] = cnt;
}

position GetNewPos(position pos, int k, int step) {
    int dir = (step % 2 == 0) ? -1 : 1; /// 次序决定方向，方向决定加减 !!!
    /// 通过引入方向变量，使下面的新坐标计算公式成为通用的，对两个方向均适用
    position next_pos = {pos.x + dir * dxy[k].x, pos.y + dir * dxy[k].y};
    return next_pos;
}
```

```
void LogStep(position pos, int step) {
    path[step] = pos;
}

void OutStep(position pos) {
    cout << "(" << pos.x << ", "<< pos.y << ") ";
}

void OutAll(int step) {
    for (int i=0; i<=step; i++) OutStep(path[i]);
    cout << endl;
}
```

```
void Jump(position pos, int step) {
    if (IsDone(pos)) {  // 是否到达目标?
        num++;          // 方案数加1
        cout << num << ": ";
        OutAll(step);   // 输出方案
        return;
    }

    // 遍历N种方案
    for (int k=0; k<sizeof(dxy)/sizeof(dxy[0]); k++) {

        position next_pos = GetNewPos(pos, k, step);
        if (!IsValid(next_pos, step+1)) continue; // next_pos是否可行?

        SetCount(next_pos, step+1, 1); // 设置次数!!!
        LogStep(next_pos, step+1);     // 记录方案
        Jump(next_pos, step+1);        // 走下一步
        SetCount(next_pos, step+1, 0); // 清除次数!!! (回溯)

    }
}
```



```
int main() {
    num = 0;                          // 初始方案数置0
    SetCount(start_pos, 0, 1); // 第0步在起点（第一次）!!!
    LogStep(start_pos, 0);      // 记录起点
    Jump(start_pos, 0);          // 走第一步
    return 0;
}
```

```
1: (3, 3) (2, 2) (3, 2) (3, 0) (3, 1) (1, 1) (2, 2) (0, 2) (0, 3) (0, 1) (1, 1) (0, 0)
2: (3, 3) (2, 2) (3, 2) (3, 0) (3, 1) (1, 1) (2, 2) (0, 2) (0, 3) (0, 1) (0, 2) (0, 0)
3: (3, 3) (3, 1) (3, 2) (3, 0) (3, 1) (1, 1) (2, 2) (0, 2) (0, 3) (0, 1) (1, 1) (0, 0)
4: (3, 3) (3, 1) (3, 2) (3, 0) (3, 1) (1, 1) (2, 2) (0, 2) (0, 3) (0, 1) (0, 2) (0, 0)
Program ended with exit code: 0
```

能否不使用回溯?

```
SetCount(next_pos, step+1, 1); // 设置次数!!!
LogStep(next_pos, step+1);     // 记录方案
Jump(next_pos, step+1);        // 走下一步
SetCount(next_pos, step+1, 0); // 清除次数!!! (回溯)
```

解题思路2：将“方向”纳入状态坐标中

```
// 维持坐标定义不变
struct position { int x, y; };
position dxy[] = {{1,0}, {0,1}, {1,1}, {2,0}, {0,2}};

// 新增状态结构定义（包含坐标数据成员）
struct state {
    int dir;
    position pos;
};
state start = {-1, {3, 3}}, goal = {1, {0, 0}};
```

注意上面代码中结构变量的初始化形式

解题思路2：通过查找历史记录防重复

```
bool IsValid(state st, int step) {
    /// 1. 合法性检查
    if (st.pos.x < 0 || st.pos.x > 3 || st.pos.y < 0 || st.pos.y > 3)
        return false;

    /// 2. 安全性检查【根据游戏规则推导出来】
    if (st.pos.x != 0 && st.pos.x != 3 && st.pos.x != st.pos.y)
        return false;

    /// 3. 重复性检查
    for (int i=step-2; i>=0; i-=2)
        if (IsEq(st, path[i])) return false;

    return true;
}
```

说明：这种重复性检查方法，恰好可以使递归不需要回溯！

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

struct position { int x, y; };
position dxy[] = {{1,0}, {0,1}, {1,1}, {2,0}, {0,2}};

struct state {
    int dir;
    position pos;
};
state start = {-1, {3, 3}}, goal = {1, {0, 0}};

state path[100];
int num;

bool IsEq(state st1, state st2) {
    return (st1.dir == st2.dir) &&
           (st1.pos.x == st2.pos.x) && (st1.pos.y == st2.pos.y);
}

bool IsDone(state st) { return IsEq(st, goal); }
```

```
bool IsValid(state st, int step) {
    /// 1. 合法性检查
    if (st.pos.x < 0 || st.pos.x > 3 || st.pos.y < 0 || st.pos.y > 3)
        return false;
    /// 2. 安全性检查【根据游戏规则推导出来】
    if (st.pos.x != 0 && st.pos.x != 3 && st.pos.x != st.pos.y)
        return false;
    /// 3. 重复性检查
    for (int i=step-2; i>=0; i-=2)
        if (IsEq(st, path[i])) return false;
    return true;
}
```

```
state GetNewState(state st, int k, int step) {
    state next_st = {-st.dir, {st.pos.x + st.dir * dxy[k].x,
                               st.pos.y + st.dir * dxy[k].y}};
    return next_st;
}
```

注意代码的写法

```
void LogStep(state st, int step) { path[step] = st; }

void OutStep(state st) {
    cout << setw(2) << st.dir << " (" << st.pos.x << ", "<< st.pos.y << ") ";
}

void OutAll(int step) {
    for (int i=0; i<=step; i++) OutStep(path[i]);
    cout << endl;
}
```

```
void Jump(state st, int step) {
    // 是否到达目标?
    if (IsDone(st)) {
        num++;
        cout << num << ": ";
        OutAll(step);
        return;
    }

    // 遍历N种决策
    for (int k=0; k<sizeof(dxy)/sizeof(dxy[0]); k++) {
        state next_st = GetNewState(st, k, step);
        if (!IsValid(next_st, step+1)) continue;
        LogStep(next_st, step+1); // 记录该决策
        Jump(next_st, step+1);    // 转到下一状态
    }
}
```

```
int main() {
    num = 0;                  // 初始方案数置0
    LogStep(start, 0); // 记录起点
    Jump(start, 0);     // 从起点出发
    return 0;
}
```

思考题：在调用Jump函数之前，能否不记录起始状态?

程序输出结果

```
1: -1 (3, 3)  1 (2, 2) -1 (3, 2)  1 (3, 0) -1 (3, 1)  1 (1, 1) -1 (2, 2)  1 (0, 2) -1 (0, 3)  1 (0, 1) -1 (1, 1)  1 (0, 0)
2: -1 (3, 3)  1 (2, 2) -1 (3, 2)  1 (3, 0) -1 (3, 1)  1 (1, 1) -1 (2, 2)  1 (0, 2) -1 (0, 3)  1 (0, 1) -1 (0, 2)  1 (0, 0)
3: -1 (3, 3)  1 (3, 1) -1 (3, 2)  1 (3, 0) -1 (3, 1)  1 (1, 1) -1 (2, 2)  1 (0, 2) -1 (0, 3)  1 (0, 1) -1 (1, 1)  1 (0, 0)
4: -1 (3, 3)  1 (3, 1) -1 (3, 2)  1 (3, 0) -1 (3, 1)  1 (1, 1) -1 (2, 2)  1 (0, 2) -1 (0, 3)  1 (0, 1) -1 (0, 2)  1 (0, 0)
Program ended with exit code: 0
```

【思考题】

其他变化形式：http://www.robspuzzlepage.com/jumping.htm

【思考题】

请编程输出完成【华容道】

任务的操作步骤

【思考题】

两个水瓶，一个容量是9升，一个容量是4升，如何才能从河中取出6升水?

请编程输出操作步骤。

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

int main()
{
    cout << "END. See you later!" << endl;
    return 0;
}
```