第 2 章 搜索

1. **什么是搜索?**

搜索，是一种枚举，通过穷举所有的情况来找到最优解，或者统计合法解的个数。因此，搜索有时候也叫作暴搜。

搜索一般分为**深度优先搜索(DFS)与宽度优先搜索(BFS)**。

2. **深度优先遍历 vs 深度优先搜索，宽度优先遍历 vs 宽度优先搜索**

遍历是形式，搜索是目的。

不过，在一般情况下，我们不会去纠结概念的差异，两者可以等同。

3. **回溯与剪枝**

- 回溯：当在搜索的过程中，遇到走不通或者走到底的情况时，就回头。
- 剪枝：剪掉在搜索过程中，剪掉重复出现或者不是最优解的分支。

1. 深度优先搜索 - DFS

1.1 递归型枚举与回溯剪枝初识

1. 画决策树；
2. 根据决策树写递归。

1.1.1 枚举子集

> **题目来源：** 洛谷
>
> **题目链接：** B3622 枚举子集（递归实现指数型枚举）
>
> **难度系数：** ★

【题目描述】

> 今有 n 位同学，可以从中选出任意名同学参加合唱。
>
> 请输出所有可能的选择方案。

【输入描述】

> 仅一行，一个正整数 n 。

【输出描述】

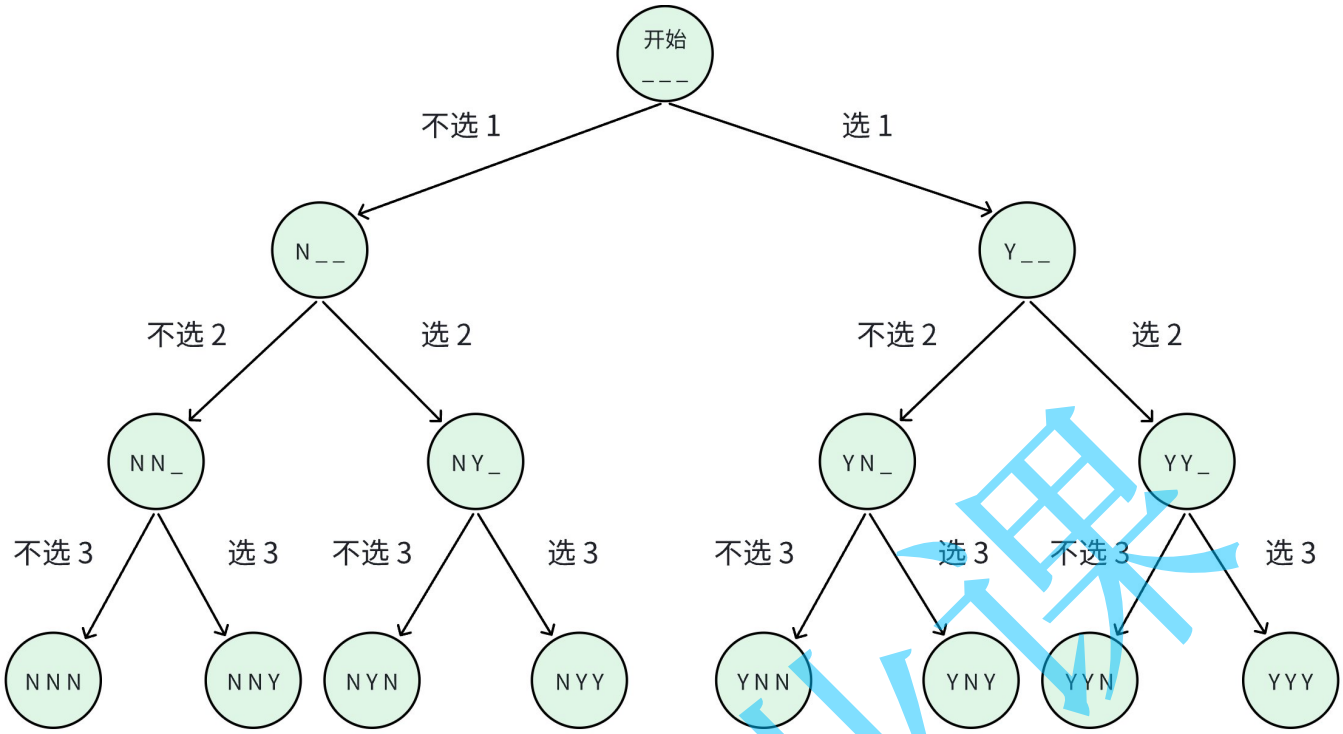
> 若干行，每行表示一个选择方案。
>
> 每一种选择方案用一个字符串表示，其中第 i 位为 `Y` 则表示第 i 名同学参加合唱；为 `N` 则表示不参加。
>
> 需要以字典序输出答案。
>
> 对于 100% 的数据，保证 $1 \leq n \leq 10$ 。

【示例一】

> 输入：
>
> ```
> 3
> ```
>
> 输出：
>
> ```
> NNN
> NNY
> NYN
> NYY
> YNN
> YNY
> YYN
> YYY
> ```

【解法】

设一共有 3 个数，分别是 $[1,2,3]$ 。「从前往后」考虑每一个数，针对当前这个数「选」或者「不选」，我们可以画出如下「决策树」：



设计递归函数：

- 重复子问题：针对某一位，「选」或者「不选」这个数。因为最终结果要按照「字典序」输出，我们可以「先考虑不选」，然后「再考虑选」；
- 实现方式参考代码和注释，结合「决策树」一起看会很清晰。

【参考代码】

```cpp
#include <iostream>

using namespace std;

int n;
string path; // 记录递归过程中，每一步的决策

void dfs(int pos)
{
    if(pos > n)
    {
        // path 就存着前 n 个人的决策
        cout << path << endl;
        return;
    }

```

```
    // 不选
    path += 'N';
    dfs(pos + 1);
    path.pop_back(); // 回溯，清空现场

    // 选
    path += 'Y';
    dfs(pos + 1);
    path.pop_back(); // 清空现场
}

int main()
{
    cin >> n;

    dfs(1);

    return 0;
}
```

1.1.2 组合型枚举

> **题目来源：** 洛谷
>
> **题目链接：** P10448 组合型枚举
>
> **难度系数：** ★

【题目描述】

> 从 $1 \sim n$ 这 n 个整数中随机选出 m 个，输出所有可能的选择方案。

【输入描述】

> 两个整数 n, m，在同一行用空格隔开。

【输出描述】

> 按照从小到大的顺序输出所有方案，每行 1 个。
>
> 首先，同一行内的数升序排列，相邻两个数用一个空格隔开。
>
> 其次，对于两个不同的行，对应下标的数一一比较，字典序较小的排在前面（例如 `1 3 5 7` 排在 `1 3 6 8` 前面）。
>
> 对于所有测试数据满足 $0 \leq m \leq n, n + (n - m) \leq 25$。

【示例一】

输入：

5 3

输出：

1 2 3

1 2 4

1 2 5

1 3 4

1 3 5

1 4 5

2 3 4

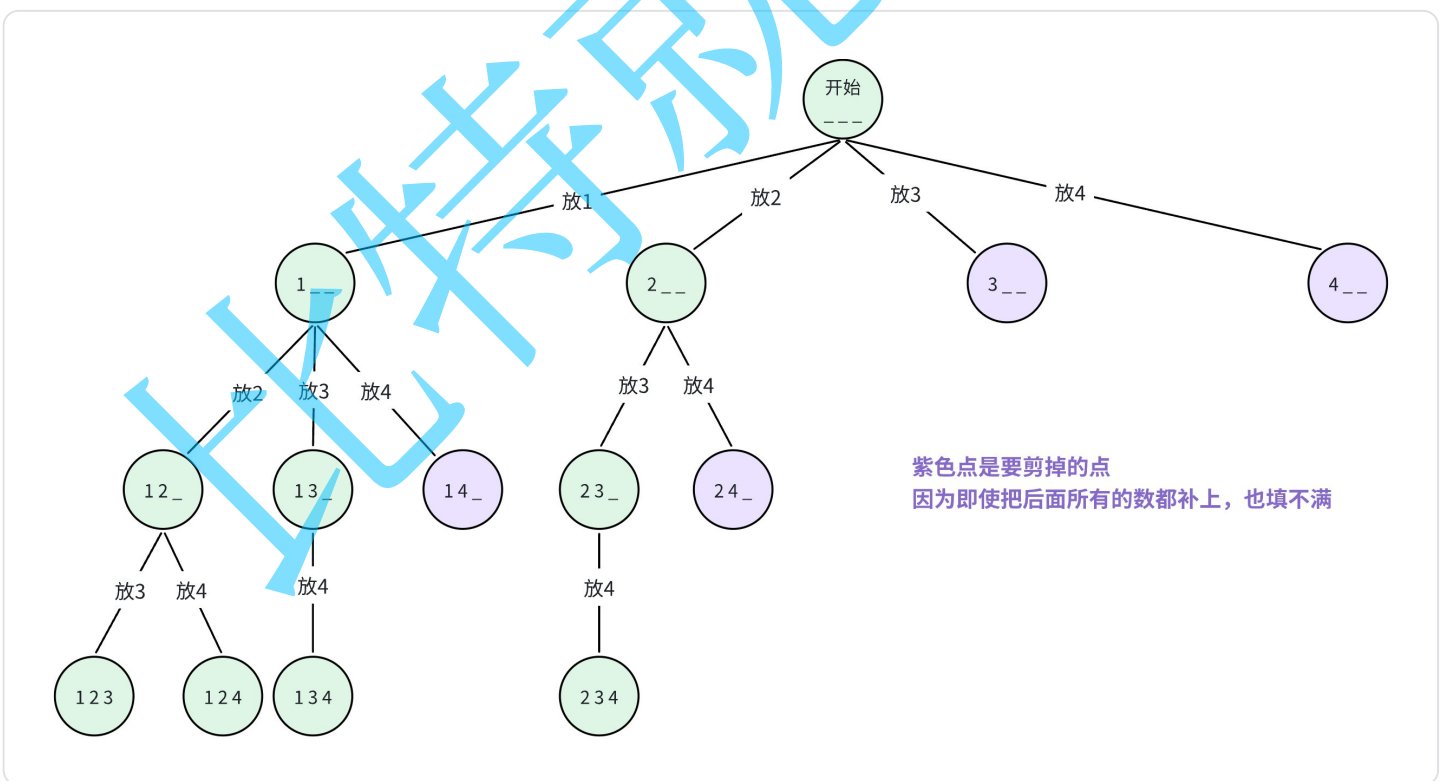
2 3 5

2 4 5

3 4 5

【解法】

设 $n=4, m=3$，「从前往后」考虑 3 个位置应该选哪个数，我们可以画出如下决策树：



设计递归函数：

- 重复子问题：当前这一位，应该放哪个数上去。因为这是一个「组合」问题，不涉及排列，所以我们当前位置开始放的数，应该是「上次决策的数的下一位」。

- 实现方式参考代码和注释，结合「决策树」一起看会很清晰。

【参考代码】

```cpp
#include <iostream>
#include <vector>

using namespace std;

int n, m;
vector<int> path;
// path.size();

void dfs(int begin)
{
    if(path.size() == m)
    {
        for(auto x : path) cout << x << " ";
        cout << endl;
        return;
    }

    for(int i = begin; i <= n; i++)
    {
        path.push_back(i);
        dfs(i + 1);
        path.pop_back(); // 清空现场
    }
}

int main()
{
    cin >> n >> m;

    dfs(1);

    return 0;
}
```


1.1.3 枚举排列

> **题目来源： 洛谷**

> **题目链接：** B3623 枚举排列（递归实现排列型枚举）
>
> **难度系数：** ★

【题目描述】

> 今有 n 名学生，要从中选出 k 人排成一列拍照。
>
> 请按字典序输出所有可能的排列方式。

【输入描述】

> 仅一行，两个正整数 n, k。
>
> 对于 100% 的数据， $1 \leq k \leq n \leq 10$。

【输出描述】

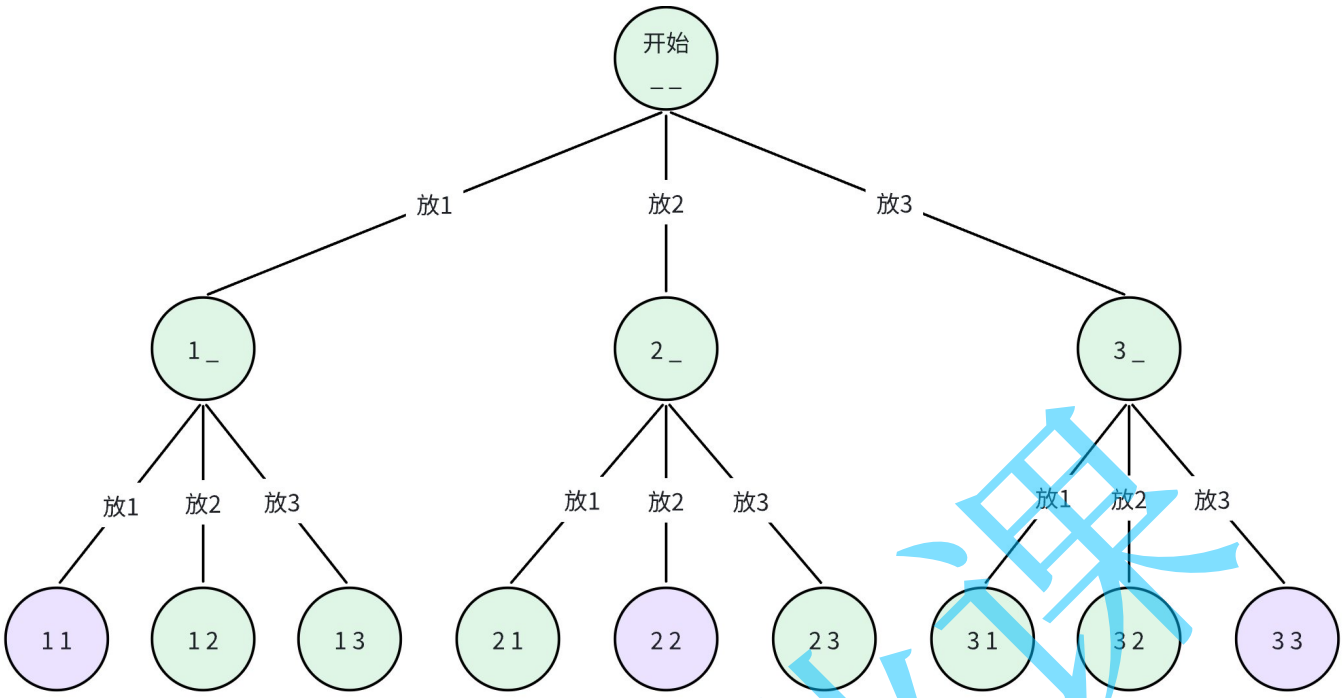
> 若干行，每行 k 个正整数，表示一种可能的队伍顺序。

【示例一】

> 输入：
>
> ```
> 3 2
> ```
>
> 输出：
>
> ```
> 1 2
> 1 3
> 2 1
> 2 3
> 3 1
> 3 2
> ```

【解法】

设 $n = 3, k = 2$，一共要选出两个数，可以依次「考虑要选出来的数」是谁，画出如下决策树：



紫色部分要剪掉，因为我们不能选择重复的数

设计递归函数：

- 重复子问题：考虑这一位要放上什么数。因为是「排列」问题，所以我们直接从 1 开始枚举要放的数。
- 剪枝：在这一条路径中，我们「不能选择之前已经选择过的数」。
- 实现方式参考代码和注释，结合「决策树」一起看会很清晰。

【参考代码】

```cpp
#include <iostream>
#include <vector>

using namespace std;

const int N = 15;

int n, k;
vector<int> path;
bool st[N]; // 标记一下哪些数已经选过了

void dfs()
{
```

```cpp
    if(path.size() == k)
    {
        for(auto x : path) cout << x << " ";
        cout << endl;
        return;
    }

    for(int i = 1; i <= n; i++)
    {
        if(st[i]) continue;
        path.push_back(i);
        st[i] = true;
        dfs();
        // 恢复现场
        path.pop_back();
        st[i] = false;
    }
}

int main()
{
    cin >> n >> k;

    dfs();

    return 0;
}
```

1.1.4 全排列问题

> **题目来源：** 洛谷
>
> **题目链接：** P1706 全排列问题
>
> **难度系数：** ★

【题目描述】

> 按照字典序输出自然数 1 到 n 所有不重复的排列，即 n 的全排列，要求所产生的任一数字序列中不允许出现重复的数字。

【输入描述】

> 一个整数 $n(1 \leq n \leq 9)$。

【输出描述】

> 由 $1 \sim n$ 组成的所有不重复的数字序列，每行一个序列。

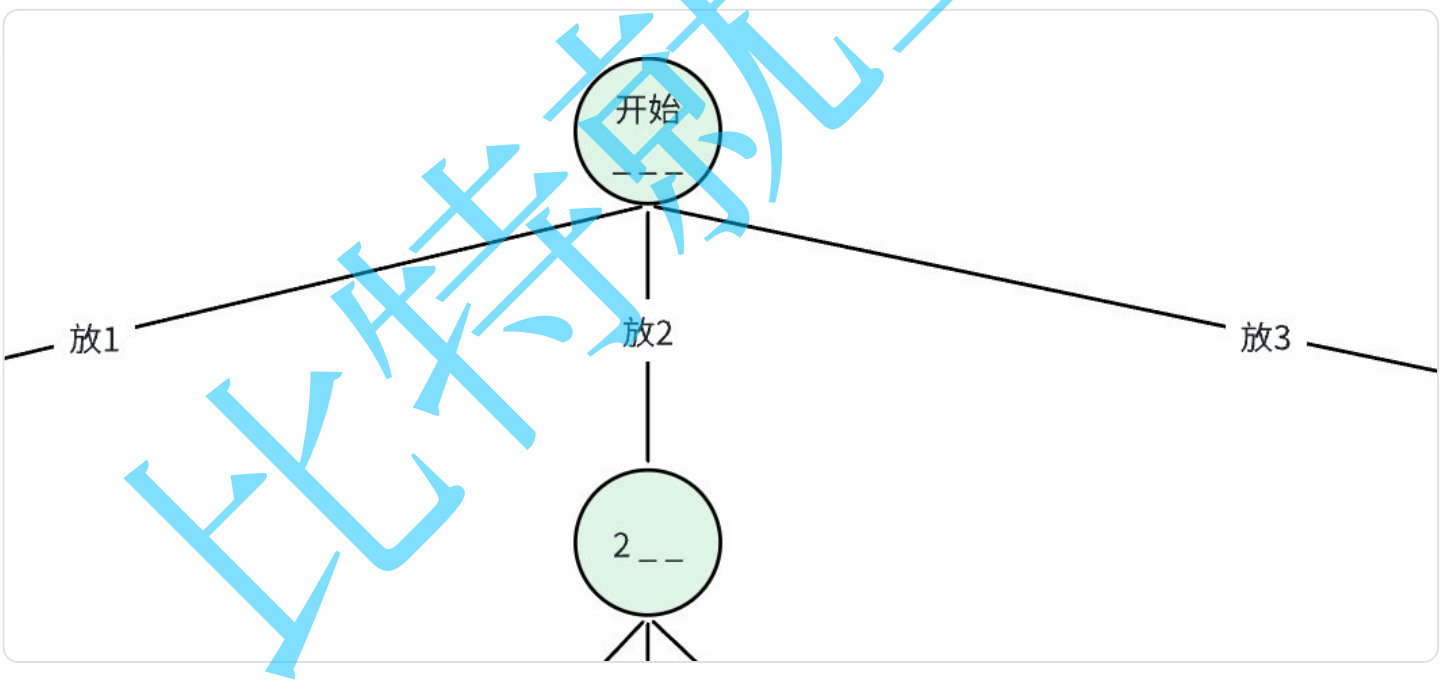
> 每个数字保留 5 个场宽。

【示例一】

> 输入：
>
> 3
>
> 输出：
>
> 1 2 3
>
> 1 3 2
>
> 2 1 3
>
> 2 3 1
>
> 3 1 2
>
> 3 2 1

【解法】

跟上一道题的决策一样，我们可以枚举每一位应该放上什么数，只不过少了 k 的限制。剪枝的策略还是一样的，那就是在路径中，「不能选择之前已经选过的数」。



关于「全排列问题」，我们也可以调用 STL 里面的「$next\_permutation$ 函数」。

【参考代码】

```
1 #include <iostream>
2 #include <vector>
```

```cpp

using namespace std;

const int N = 15;

int n;
bool st[N];
vector<int> path;

void dfs()
{
    if(path.size() == n)
    {
        for(auto x : path)
        {
            printf("%5d", x);
        }
        cout << endl;
        return;
    }

    for(int i = 1; i <= n; i++)
    {
        if(st[i]) continue;

        path.push_back(i);
        st[i] = true;
        dfs();
        // 恢复现场
        path.pop_back();
        st[i] = false;
    }
}

int main()
{
    cin >> n;

    dfs();

    return 0;
}
```


1.2 DFS

1.2.1 选数

> **题目来源：** 洛谷
>
> **题目链接：** P1036 [NOIP2002 普及组] 选数
>
> **难度系数：** ★

【题目描述】

> 已知 n 个整数 $x_1, x_2, x_3, ..., x_n$，以及 1 个整数 $k(k < n)$。从 n 个整数中任选 k 个整数相加，可分别得到一系列的和。例如当 $n = 4, k = 3$，4 个整数分别为 $3, 7, 12, 19$ 时，可得全部的组合与它们的和为：
>
> $3 + 7 + 12 = 22$
>
> $3 + 7 + 19 = 29$
>
> $7 + 12 + 19 = 38$
>
> $3 + 12 + 19 = 34$
>
> 现在，要求你计算出和为素数共有多少种。
>
> 例如上例，只有一种的和为素数：$3 + 7 + 19 = 29$

【输入描述】

> 第一行两个空格隔开的整数 $n, k(1 \le n \le 20, k < n)$。
>
> 第二行 n 个整数，分别为 $x_1, x_2, ..., x_n(1 \le x_i \le 5 \times 10^6)$。

【输出描述】

> 输出一个整数，表示种类数。

【示例一】

> 输入：
>
> ```
> 4 3
> 3 7 12 19
> ```
>
> 输出：
>
> ```
> 1
> ```

【解法】

「组合」型枚举，路径里面记录选择数的「总和」。在选出 k 个数之后，判断「是否是质数」。

【参考代码】

```
1 #include <iostream>
2
```

```cpp
using namespace std;

const int N = 25;

int n, k;
int a[N];

int ret;
int path; // 记录路径中所选择的数的和

bool isprime(int x)
{
    if(x <= 1) return false;
    // 试除法
    for(int i = 2; i <= x / i; i++)
    {
        if(x % i == 0) return false;
    }
    return true;
}

void dfs(int pos, int begin)
{
    if(pos > k)
    {
        if(isprime(path)) ret++;
        return;
    }

    for(int i = begin; i <= n; i++)
    {
        path += a[i];
        dfs(pos + 1, i + 1);
        path -= a[i]; // 恢复现场
    }
}

int main()
{
    cin >> n >> k;
    for(int i = 1; i <= n; i++) cin >> a[i];

    dfs(1, 1);

    cout << ret << endl;

    return 0;
```

```
50 }
```

1.2.2 飞机降落

题目来源： 洛谷

题目链接： P9241 [蓝桥杯 2023 省 B] 飞机降落

难度系数： ★★

【题目描述】

N 架飞机准备降落到某个只有一条跑道的机场。其中第 i 架飞机在 T_i 时刻到达机场上空，到达时它的剩余油料还可以继续盘旋 D_i 个单位时间，即它最早可以于 T_i 时刻开始降落，最晚可以于 $T_i + D_i$ 时刻开始降落。降落过程需要 L_i 个单位时间。

一架飞机降落完毕时，另一架飞机可以立即在同一时刻开始降落，但是不能在前一架飞机完成降落前开始降落。

请你判断 N 架飞机是否可以全部安全降落。

【输入描述】

输入包含多组数据。

第一行包含一个整数 T，代表测试数据的组数。

对于每组数据，第一行包含一个整数 N。

以下 N 行，每行包含三个整数：T_i, D_i, L_i。

对于 100% 的数据，$1 \leq T \leq 10$，$1 \leq N \leq 10$，$0 \leq T_i, D_i, Li \leq 10^5$。

【输出描述】

对于每组数据，输出 YES 或者 NO，代表是否可以全部安全降落。

【示例一】

输入：

```
2
3
0 100 10
10 10 10
0 2 20
3
```

> 0 10 20
>
> 10 10 20
>
> 20 10 20
>
> 输出：
>
> YES
>
> NO

【解法】

枚举所有飞机的「全排列」，判断是否存在一种排列，使的全部的飞机都能安全降落。

剪枝：

- 当前路径里面只能选没有选过的飞机；
- 如果这架飞机不能正常降落，剪掉；
- 如果已经找到一种安全降落的方式，停止枚举，可以通过「递归的返回值」判断是否搜索成功。

【参考代码】

```
#include <iostream>
#include <cstring>

using namespace std;

const int N = 15;

int n;
int t[N], d[N], l[N];

bool st[N]; // 标记路径中哪些飞机已经摆放过

bool dfs(int pos, int end)
{
    if(pos > n)
    {
        return true;
    }

    for(int i = 1; i <= n; i++)
    {
        if(st[i] == true) continue; // 剪枝
        if(end > t[i] + d[i]) continue; // 剪枝
        int newend = max(t[i], end) + l[i];
```

```
25          st[i] = true;
26          if(dfs(pos + 1, newend)) return true;
27          st[i] = false; // 回复现场
28      }
29
30      return false;
31  }
32
33  int main()
34  {
35      int T; cin >> T;
36
37      while(T--) // 多组测试数据的时候，一定要注意清空数据
38      {
39          memset(st, 0, sizeof st);
40          cin >> n;
41          for(int i = 1; i <= n; i++) cin >> t[i] >> d[i] >> l[i];
42
43          if(dfs(1, 0)) cout << "YES" << endl;
44          else cout << "NO" << endl;
45      }
46
47
48      return 0;
49  }
```

1.2.3 八皇后

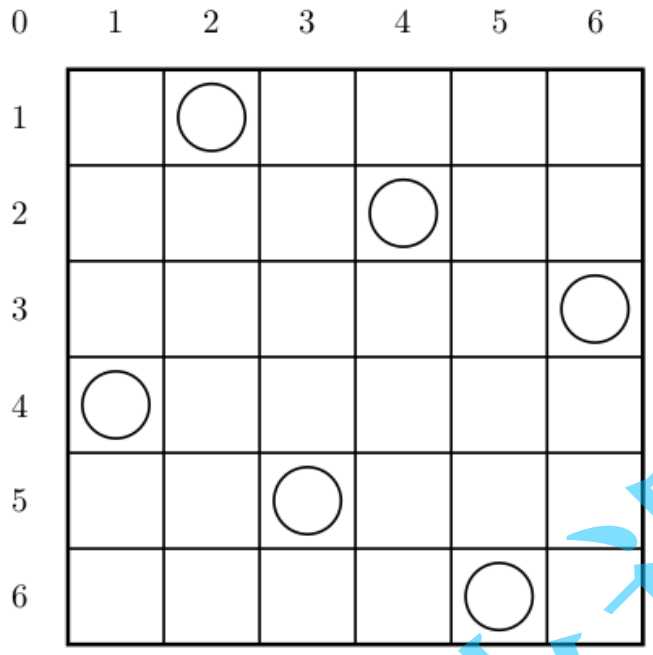
题目来源： 洛谷

题目链接： P1219 [USACO1.5] 八皇后 Checker Challenge

难度系数： ★★

【题目描述】

一个如下的 6×6 的跳棋棋盘，有六个棋子被放置在棋盘上，使得每行、每列有且只有一个，每条对角线（包括两条主对角线的所有平行线）上至多有一个棋子。



洛谷

上面的布局可以用序列 $2\,4\,6\,1\,3\,5$ 来描述，第 i 个数字表示在第 i 行的相应位置有一个棋子，如下：

行号 $1\,2\,3\,4\,5\,6$

列号 $2\,4\,6\,1\,3\,5$

这只是棋子放置的一个解。请编一个程序找出所有棋子放置的解。

并把它们以上面的序列方法输出，解按字典顺序排列。

请输出前 3 个解。最后一行是解的总个数。

【输入描述】

一行一个正整数 n，表示棋盘是 $n \times n$ 大小的。

对于 100% 的数据，$6 \leq n \leq 13$。

【输出描述】

前三行为前三个解，每个解的两个数字之间用一个空格隔开。第四行只有一个数字，表示解的总数。

【示例一】

输入：

```
6
```

输出：

```
2 4 6 1 3 5
3 6 2 5 1 4
```

> 4 1 5 2 6 3
>
> 4

【解法】

枚举策略：

- 「一行一行」的放皇后：从第一行开始，尝试在每一列上放皇后；
- 如果当前列放上皇后之后「没有冲突」，就标记一下这个「放法」，记录一下当前行的决策，然后「递归」考虑下一行；
- 等到「所有行」都放置完毕之后，输出本次枚举的决策。

枚举策略应该是比较容易想到的，这道题的难点在于如何判断「在这一列放上这个皇后之后，是否冲突」。当我们一行一行放的时候，「行是不会产生冲突的」。产生冲突的只有「列」，「主对角线」，以及「副对角线」。我们可以用三个数组分别标记：

- $col[i] = true$，表示第 i 行放置了一个皇后；
- $dig1[j - i + n] = true$，表示 $y = x + (j - i)$ 这条「主对角线」上放置了一个皇后；
- $dig2[j + i] = true$，表示 $y = -x + (j + i)$ 这条「副对角线」上放置了一个皇后。

【参考代码】

```cpp
#include <iostream>
#include <vector>

using namespace std;

const int N = 15;

int n;
bool col[N], st1[N * 2], st2[N * 2];

int ret;
vector<int> path;

void dfs(int x)
{
    if(x > n)
    {
        ret++;
        if(ret <= 3)
        {
```

```
            for(auto x : path) cout << x << " ";
            cout << endl;
        }
        return;
    }

    for(int y = 1; y <= n; y++)
    {
        // 判断能不能摆在这一列
        if(col[y] || st1[y - x + n] || st2[y + x]) continue; // 剪枝
        col[y] = st1[y - x + n] = st2[y + x] = true;
        path.push_back(y);
        dfs(x + 1);
        col[y] = st1[y - x + n] = st2[y + x] = false;
        path.pop_back();
    }
}

int main()
{
    cin >> n;

    dfs(1);

    cout << ret << endl;

    return 0;
}
```


1.2.4 数独

> **题目来源：** 洛谷
>
> **题目链接：** P1784 数独
>
> **难度系数：** ★★

【题目描述】

> 数独是根据 9×9 盘面上的已知数字，推理出所有剩余空格的数字，并满足每一行、每一列、每一个粗线宫内的数字均含 $1 \sim 9$ ，不重复。每一道合格的数独谜题都有且仅有唯一答案，推理方法也以此为基础，任何无解或多解的题目都是不合格的。
>
> 芬兰一位数学家号称设计出全球最难的“数独游戏”，并刊登在报纸上，让大家去挑战。
>
> 这位数学家说，他相信只有“智慧最顶尖”的人才有可能破解这个“数独之谜”。

> 据介绍，目前数独游戏的难度的等级有一到五级，一是入门等级，五则比较难。不过这位数学家说，他所设计的数独游戏难度等级是十一，可以说是所有数独游戏中，难度最高的等级。他还表示，他目前还没遇到解不出来的数独游戏，因此他认为“最具挑战性”的数独游戏并没有出现。

【输入描述】

> 一个未填的数独。

【输出描述】

> 填好的数独。

【示例一】

> 输入：
>
> 800000000
> 003600000
> 070090200
> 050007000
> 000045700
> 000100030
> 001000068
> 008500010
> 090000400
>
> 输出：
>
> 812753649
> 943682175
> 675491283
> 154237896
> 369845721
> 287169534
> 521974368
> 438526917
> 796318452

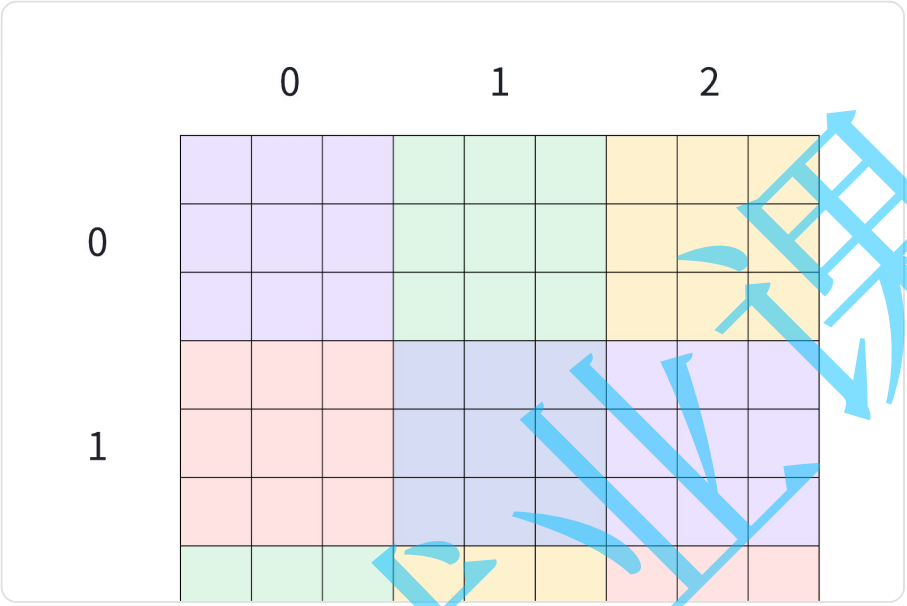
【解法】

枚举策略：

- 「一个格子一个格子」往里面填数。

- 从第一行的第一个格子开始，填上一个「没有冲突」的数，然后「递归」到下一个格子；
- 当某一行填满之后，递归到「下一行的起始位置」继续填数。

可以创建三个数组，用来帮助判断填上某一个数之后，是否会发生冲突。对于 3×3 方格，我们可以给每一个格子编上号，快读定位，如下图：



- $row[i][num] = true$ 表示：第 i 行已经放上了 num 这个数；
- $col[j][num] = true$ 表示：第 j 列已经放上了 num 这个数；
- $st[i/3][j/3][num] = true$ 表示：$[i/3, j/3]$ 的 3×3 方格里，已经放上了 num 这个数。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 10;

int n = 9;
int a[N][N];
bool row[N][N], col[N][N], st[N][N][N];

bool dfs(int i, int j)
{
    if(j == n)
    {
        // 当这一行填满之后
        i++;
```

```
17          j = 0;
18      }
19
20      if(i == n) return true; // 找到一种合法的情况，就停止递归
21      if(a[i][j]) return dfs(i, j + 1);
22
23      for(int x = 1; x <= 9; x++)
24      {
25          if(row[i][x] || col[j][x] || st[i / 3][j / 3][x]) continue; // 剪枝
26
27          row[i][x] = col[j][x] = st[i / 3][j / 3][x] = true;
28          a[i][j] = x;
29
30          if(dfs(i, j + 1)) return true;
31
32          // 恢复现场
33          row[i][x] = col[j][x] = st[i / 3][j / 3][x] = false;
34          a[i][j] = 0;
35      }
36
37      return false;
38  }
39
40  int main()
41  {
42      for(int i = 0; i < n; i++)
43      {
44          for(int j = 0; j < n; j++)
45          {
46              cin >> a[i][j];
47              int x = a[i][j];
48              if(x)
49              {
50                  // 标记一下
51                  row[i][x] = col[j][x] = st[i / 3][j / 3][x] = true;
52              }
53          }
54      }
55
56      dfs(0, 0);
57
58      for(int i = 0; i < n; i++)
59      {
60          for(int j = 0; j < n; j++)
61          {
62              cout << a[i][j] << " ";
63          }
```

```
64          cout << endl;
65      }
66
67      return 0;
68  }
```

1.3 剪枝与优化

剪枝，形象得看，就是剪掉搜索树的分支，从而减小搜索树的规模，排除掉搜索树中没有必要的分支，优化时间复杂度。

在深度优先遍历中，有几种常见的剪枝方法：

1. **排除等效冗余**

> 如果在搜索过程中，通过某一个节点往下的若干分支中，存在最终结果等效的分支，那么就只需要搜索其中一条分支。

2. **可行性剪枝**

> 如果在搜索过程中，发现有一条分支是无论如何都拿不到最终解，此时就可以放弃这个分支，转而搜索其它的分支。

3. **最优性剪枝**

> 在最优化的问题中，如果在搜索过程中，发现某一个分支已经超过当前已经搜索过的最优解，那么这个分支往后的搜索，必定不会拿到最优解。此时应该停止搜索，转而搜索其它情况。

4. **优化搜索顺序**

> 在有些搜索问题中，搜索顺序是不影响最终结果的，此时搜索顺序的不同会影响搜索树的规模。
>
> 因此，应当先选择一个搜索分支规模较小的搜索顺序，快速拿到一个最优解之后，用最优性剪枝剪掉别的分支。

5. **记忆化搜索**

> 记录每一个状态的搜索结果，当下一次搜索到这个状态时，直接找到之前记录过的搜索结果。
>
> 记忆化搜索，有时也叫动态规划。

1.3.1 数的划分

> **题目来源：** 洛谷
>
> **题目链接：** P1025 [NOIP2001 提高组] 数的划分
>
> **难度系数：** ★★

【题目描述】

> 将整数 n 分成 k 份，且每份不能为空，任意两个方案不相同（不考虑顺序）。
>
> 例如： $n=7,k=3$，下面三种分法被认为是相同的。
>
> $1,1,5$
>
> $1,5,1$
>
> $5,1,1$
>
> 问有多少种不同的分法。

【输入描述】

> n,k $(6<n\leq 200，2\leq k\leq 6)$

【输出描述】

> 1 个整数，即不同的分法。

【示例一】

> 输入：
>
> 7 3
>
> 输出：
>
> 4

【解法】

搜索策略：

- 将 $[1,n]$ 个数放在 k 个坑里面，使的坑里面的所有数的总和是 n；
- 其中，不同的坑里面的数可能相同；
- 但是 $[1,2]$ 与 $[2,1]$ 是同一种分法，因此，应该是一种组合型枚举。针对每一个坑里面的数应该放谁的时候，应该从上一个坑里面的数开始枚举。

剪枝策略：

- 当我们填了 cnt 个坑时，此时总和是 sum，如果后续坑位全部都填上最小值都会超过 n。说明我们之前填的数太大了，导致后面怎么填都会超过 n，直接剪掉。

注意剪枝位置的不同，而导致搜索树的不同：

- 如果在进入递归之前剪枝，我们不会进入非法的递归函数中；
- 但是如果在进入递归之后剪枝，我们就会多进入很多不合法的递归函数中。

【参考代码】

```cpp
#include <iostream>

using namespace std;

int n, k;
int path, ret;

void dfs(int pos, int begin)
{
    if(pos == k)
    {
        if(path == n) ret++;
        return;
    }

    // 可行性剪枝
    // if(path + begin * (k - pos) > n) return;

    for(int i = begin; i <= n; i++)
    {
        // 可行性剪枝
        if(path + i * (k - pos) > n) return;

        path += i;
        dfs(pos + 1, i);
        path -= i;
    }
}

int main()
{
    cin >> n >> k;

    dfs(0, 1);

    cout << ret << endl;

    return 0;
}
```

1.3.2 小猫爬山

> **题目来源：** 洛谷
>
> **题目链接：** P10483 小猫爬山
>
> **难度系数：** ★★

【题目描述】

> Freda 和 rainbow 饲养了 $N(N \leq 18)$ 只小猫，这天，小猫们要去爬山。经历了千辛万苦，小猫们终于爬上了山顶，但是疲倦的它们再也不想徒步走下山了
>
> Freda 和 rainbow 只好花钱让它们坐索道下山。索道上的缆车最大承重量为 W，而 N 只小猫的重量分别是 $C_1, C_2, \ldots, C_N$。当然，每辆缆车上的小猫的重量之和不能超过 $W(1 \leq C_i \leq W \leq 10^8)$。每租用一辆缆车，Freda 和 rainbow 就要付 1 美元，所以他们想知道，最少需要付多少美元才能把这 N 只小猫都运送下山？

【输入描述】

> 第一行包含两个用空格隔开的整数， N 和 W。接下来 N 行每行一个整数，其中第 $i+1$ 行的整数表示第 i 只小猫的重量 C_i。

【输出描述】

> 输出一个整数，最少需要多少美元，也就是最少需要多少辆缆车。

【示例一】

> 输入：
>
> ```
> 5 1996
> 1
> 2
> 1994
> 12
> 29
> ```
>
> 输出：
>
> ```
> 2
> ```

【解法】

搜索策略：依次处理每一只猫，对于每一只猫，我们都有两种处理方式：

- 要么把这只猫放在已经租好的缆车上；
- 要么重新租一个缆车，把这只猫放上去。

剪枝：

- 在搜索过程中，我们用全局变量记录已经搜索出来的最小缆车数量。如果当前搜索过程中，已经用的缆车数量大于全局记录的最小缆车数量，那么这个分支一定不会得到最优解，剪掉。
- 优化枚举顺序一：从大到小安排每一只猫
  - 重量较大的猫能够快速把缆车填满，较快得到一个最小值；
  - 通过这个最小值，能够提前把分支较大的情况提前剪掉。
- 优化枚举策略二：先考虑把小猫放在已有的缆车上，然后考虑重新租一辆车
  - 因为如果反着来，我们会先把缆车较大的情况枚举出来，这样就起不到剪枝的效果了。

【参考代码】

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 20;

int n, w;
int c[N]; // 小猫的信息

int cnt; // 当前用了多少车
int s[N]; // 每一辆车目前的总重

int ret = N; // 最优解

bool cmp(int a, int b)
{
    return a > b;
}

void dfs(int pos)
{
    // 策略二：最优性剪枝
    if(cnt >= ret) return;

    if(pos > n)
    {
        ret = cnt;
        return;
```

```
    }

    // 策略三：优化搜索顺序
    // 先安排在已有的车辆上
    for(int i = 1; i <= cnt; i++)
    {
        // 策略一：可行性剪枝
        if(s[i] + c[pos] > w) continue;
        s[i] += c[pos];
        dfs(pos + 1);
        s[i] -= c[pos]; // 恢复现场
    }

    // 重开一辆车
    cnt++;
    s[cnt] = c[pos];
    dfs(pos + 1);
    // 恢复现场
    s[cnt] = 0;
    cnt--;
}

int main()
{
    cin >> n >> w;
    for(int i = 1; i <= n; i++) cin >> c[i];

    // 策略三：优化搜索顺序
    sort(c + 1, c + 1 + n, cmp);

    dfs(1);

    cout << ret << endl;

    return 0;
}
```


1.4 记忆化搜索

记忆化搜索也是一种剪枝策略。

通过一个"备忘录"，记录第一次搜索到的结果，当下一次搜索到这个状态时，直接在"备忘录"里面找结果。

记忆化搜索，有时也叫动态规划。

【案例】

> **题目来源：** 力扣
>
> **题目链接：** 509. 斐波那契数
>
> **难度系数：** ★

【题目描述】

> **斐波那契数** （通常用 `F(n)` 表示）形成的序列称为 **斐波那契数列** 。该数列由 `0` 和 `1` 开始，后面的每一项数字都是前面两项数字的和。也就是：
>
> F(0) = 0，F(1) = 1
>
> F(n) = F(n - 1) + F(n - 2)，其中 n > 1
>
> 给定 `n` ，请计算 `F(n)` 。

【算法原理】

在搜索的过程中，如果发现特别多完全相同的子问题，就可以添加一个备忘录，将搜索的结果放在备忘录中。下一次在搜索到这个状态时，直接在备忘录里面拿值。

【代码实现】

```
class Solution
{
    int f[35]; // 搞一个备忘录

public:
    int fib(int n)
    {
        memset(f, -1, sizeof f); // 先初始化成一定不会存在的值

        return dfs(n);
    }

    int dfs(int n)
    {
        // 搜索之前先往备忘录里面瞅瞅
        if(f[n] != -1) return f[n];

        if(n == 0 || n == 1) return n;

        // 返回之前，把结果记录在备忘录中
```

```
21          f[n] = dfs(n - 1) + dfs(n - 2);
22          return f[n];
23      }
24  };
```

1.4.1 Function

题目来源： 洛谷

题目链接： P1464 Function

难度系数： ★

【题目描述】

对于一个递归函数 $w(a,b,c)$

- 如果 $a \le 0$ 或 $b \le 0$ 或 $c \le 0$ 就返回值 1 。
- 如果 $a > 20$ 或 $b > 20$ 或 $c > 20$ 就返回 $w(20,20,20)$
- 如果 $a < b$ 并且 $b < c$ 就返回 $w(a,b,c-1)+w(a,b-1,c-1)-w(a,b-1,c)$
- 其它的情况就返回
  $w(a-1,b,c)+w(a-1,b-1,c)+w(a-1,b,c-1)-w(a-1,b-1,c-1)$

这是个简单的递归函数，但实现起来可能会有些问题。当 a,b,c 均为 15 时，调用的次数将非常的多。你要想个办法才行。

注意：例如 $w(30,-1,0)$ 又满足条件 1 又满足条件 2 ，请按照最上面的条件来算，答案为 1 。

【输入描述】

会有若干行。

并以 $-1,-1,-1$ 结束。

保证输入的数在 $[-9223372036854775808, 9223372036854775807]$ 之间，并且是整数。

保证不包括 $-1,-1,-1$ 的输入行数 T 满足 $1 \le T \le 10^5$ 。

【输出描述】

输出若干行，每一行格式：

`w(a, b, c) = ans`

注意空格。

【示例一】

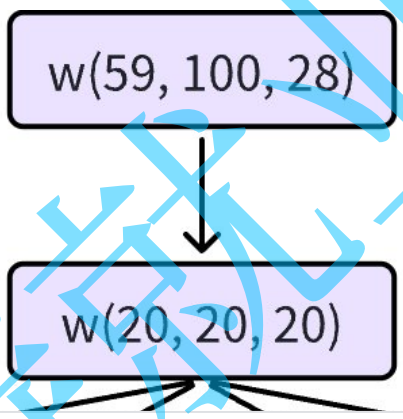
输入：

> 1 1 1
>
> 2 2 2
>
> -1 -1 -1
>
> 输出：
>
> w(1, 1, 1) = 2
>
> w(2, 2, 2) = 4

【解法】

题目叙述的非常清楚，我们仅需按照「题目的要求」把「递归函数」写出来即可。但是，如果不做其余处理的话，结果会「超时」。因为我们递归的「深度」和「广度」都非常大。

通过把「递归展开图」画出来，我们发现，在递归过程中会遇到大量「一模一样」的问题，如下图（因为递归展开过于庞大，这里只画出了一部分）：



因此，可以在递归的过程中，把每次算出来的结果存在一张「备忘录」里面。等到下次递归进入「一模一样」的问题之后，就「不用傻乎乎的展开计算」，而是在「备忘录里面直接把结果拿出来」，起到大量剪枝的效果。

【参考代码】

```
#include <iostream>

using namespace std;

typedef long long LL;

const int N = 25;

LL a, b, c;
LL f[N][N][N]; // 备忘录

```

```cpp
LL dfs(LL a, LL b, LL c)
{
    if(a <= 0 || b <= 0 || c <= 0) return 1;
    if(a > 20 || b > 20 || c > 20) return dfs(20, 20, 20);

    if(f[a][b][c]) return f[a][b][c];

    if(a < b && b < c) return f[a][b][c] = dfs(a, b, c - 1) + dfs(a, b - 1, c - 1) - dfs(a, b - 1, c);
    else return f[a][b][c] = dfs(a - 1, b, c) + dfs(a - 1, b - 1, c) + dfs(a - 1, b, c - 1) - dfs(a - 1, b - 1, c - 1);
}

int main()
{
    while(cin >> a >> b >> c)
    {
        // 多组测试数据：不需要清空
        if(a == -1 && b == -1 && c == -1) break;

        printf("w(%lld, %lld, %lld) = %lld\n", a, b, c, dfs(a, b, c));
    }

    return 0;
}
```

1.4.2 天下第一

题目来源： 洛谷

题目链接： P5635 【CSGRound1】天下第一

难度系数： ★★

【题目描述】

游戏是这样的：

给定两个数 x, y ，与一个模数 p 。

cbw 拥有数 x ，zhouwc 拥有数 y 。

第一个回合： $x \leftarrow (x + y) \bmod p$

第二个回合： $y \leftarrow (x + y) \bmod p$

第三个回合： $x \leftarrow (x + y) \bmod p$

第四个回合： $y \leftarrow (x + y) \bmod p$

> 以此类推....
>
> 如果 x 先到 0 ，则 cbw 胜利。如果 y 先到 0 ，则 zhouwc 胜利。如果 x,y 都不能到 0 ，则为平局。
>
> cbw 为了捍卫自己主席的尊严，想要提前知道游戏的结果，并且可以趁机动点手脚，所以他希望你来告诉他结果。

【输入描述】

> 有多组数据。
>
> 第一行： T 和 p 表示一共有 T 组数据且模数都为 p 。
>
> 以下 T 行，每行两个数 x,y 。
>
> $1 \leq T \leq 200$
>
> $1 \leq x,y,p \leq 10000$

【输出描述】

> 共 T 行
>
> 1 表示 cbw 获胜， 2 表示 zhouwc 获胜， `error` 表示平局。

【示例一】

> 输入：
>
> 1 10
>
> 1 3
>
> 输出：
>
> error

【示例二】

> 输入：
>
> 1 10
>
> 4 5
>
> 输出：
>
> 1

【解法】

用递归模拟整个游戏过程： $dfs(x,y)$ 的结果可以由 $dfs((x+y)\%p,(x+y+y)\%p)$ 得到。

因为测试数据是多组的，并且模数都是 p ，再加上递归的过程中会递归的相同的问题，所以可以把递归改写成记忆化搜索。其中：

- $f[x][y] = 1$，表示 cbw 赢；
- $f[x][y] = 2$，表示 $zhouwc$ 赢；
- $f[x][y] = 3$ 表示这个位置已经被访问过，如果没被修改成 1 或者 2 ，那就表明平局。

注意事项：

- 这道题的数据范围很大，用 int 类型创建二维数组空间会溢出。但是我们的最终结果仅有三种情况，所以可以用 $char$ 类型来存储最终结果，节省空间。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 1e4 + 10;

int x, y, p;
char f[N][N]; // 备忘录

char dfs(int x, int y)
{
    if(f[x][y]) return f[x][y]; // 剪枝

    f[x][y] = '3'; // 这个状态已经访问过了，之后再遇到时，表示平局

    if(x == 0) return f[x][y] = '1';
    if(y == 0) return f[x][y] = '2';
    return f[x][y] = dfs((x + y) % p, (x + y + y) % p);
}

int main()
{
    int T; cin >> T >> p;

    while(T--)
    {
        cin >> x >> y;
        char ret = dfs(x, y);
        if(ret == '1') cout << 1 << endl;
        else if(ret == '2') cout << 2 << endl;
        else cout << "error" << endl;
    }
```

```
33
34      return 0;
35  }
```

1.4.3 滑雪

题目来源： 洛谷

题目链接： P1434 [SHOI2002] 滑雪

难度系数： ★★

【题目描述】

Michael 喜欢滑雪。这并不奇怪，因为滑雪的确很刺激。可是为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，你不得不再次走上坡或者等待升降机来载你。Michael 想知道在一个区域中最长的滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。下面是一个例子：

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

一个人可以从某个点滑向上下左右相邻四个点之一，当且仅当高度会减小。在上面的例子中，一条可行的滑坡为 $24-17-16-1$（从 24 开始，在 1 结束）。当然 $25-24-23-\ldots-3-2-1$ 更长。事实上，这是最长的一条。

【输入描述】

输入的第一行为表示区域的二维数组的行数 R 和列数 C。下面是 R 行，每行有 C 个数，代表高度(两个数字之间用 1 个空格间隔)。

对于 100% 的数据，$1 \le R, C \le 100$。

【输出描述】

输出区域中最长滑坡的长度。

【示例一】

输入：

```
5 5
1 2 3 4 5
16 17 18 19 6
```

> 15 24 25 20 7
>
> 14 23 22 21 8
>
> 13 12 11 10 9
>
> 输出：
>
> 25

【解法】

暴力枚举：遍历整个矩阵，看看以当前位置为起点，最远能滑行多远的距离。在所有情况里面，取最大值即可。

如何求出以 $[i,j]$ 为起点的最大距离？

- 从 $[i,j]$ 位置上下左右瞅一瞅，如果能滑过去，就看看以下一个位置为起点，最远能滑行多远的距离；
- 找出四个方向上的最远距离，然后 $+1$。

因为出现相同子问题，所以可以用 dfs 来解决。又因为在搜索的过程中会遇到一模一样的问题，因此可以把递归改成记忆化搜索的方式。

【参考代码】

```
#include <iostream>

using namespace std;

const int N = 110;

int n, m;
int a[N][N];
int f[N][N]; // 备忘录

int dx[] = {0, 0, 1, -1};
int dy[] = {1, -1, 0, 0};

int dfs(int i, int j)
{
    if(f[i][j]) return f[i][j];

    int len = 1;
    // 上下左右四个方向搜
    for(int k = 0; k < 4; k++)
```

```cpp
    {
        int x = i + dx[k], y = j + dy[k];
        if(x < 1 || x > n || y < 1 || y > m) continue;
        if(a[i][j] <= a[x][y]) continue;

        len = max(dfs(x, y) + 1, len);
    }
    return f[i][j] = len;
}

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++)
            cin >> a[i][j];

    int ret = 1;
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++)
            ret = max(ret, dfs(i, j));

    cout << ret << endl;

    return 0;
}
```


2. 宽度优先搜索 - BFS

宽度优先搜索的过程中，每次都会从当前点向外扩展一层，所以会具有一个最短路的特性。因此，宽搜不仅能搜到所有的状态，而且还能找出起始状态距离某个状态的最小步数。

但是，前提条件是每次扩展的代价都是 1 ，或者都是相同的数。宽搜常常被用于解决边权为 1 的最短路问题。

2.1 BFS

2.1.1 马的遍历

> **题目来源：** 洛谷
>
> **题目链接：** P1443 马的遍历
>
> **难度系数：** ★★

【题目描述】

> 有一个 $n \times m$ 的棋盘，在某个点 (x, y) 上有一个马，要求你计算出马到达棋盘上任意一个点最少要走几步。

【输入描述】

> 输入只有一行四个整数，分别为 n, m, x, y。

【输出描述】

> 一个 $n \times m$ 的矩阵，代表马到达某个点最少要走几步（不能到达则输出 -1）。

【示例一】

> 输入：
>
> ```
> 3 3 1 1
> ```
>
> 输出：
>
> ```
> 0  3  2
> 3  -1 1
> 2  1  4
> ```

【解法】

题目要求到达某个点最少要走几步，因此可以用 bfs 解决。因为当权值为 1 时，bfs 每次都是扩展距离起点等距离的一层，天然具有最短性。

那就从起点开始，一层一层的往外搜，用一个 $dist$ 数组记录最短距离。

【参考代码】

```cpp
#include <iostream>
#include <queue>
#include <cstring>

using namespace std;

typedef pair<int, int> PII;

const int N = 410;

int n, m, x, y;
int dist[N][N];

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

```cpp

void bfs()
{
    memset(dist, -1, sizeof dist);

    queue<PII> q;
    q.push({x, y});
    dist[x][y] = 0;

    while(q.size())
    {
        auto t = q.front(); q.pop();
        int i = t.first, j = t.second;
        for(int k = 0; k < 8; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x < 1 || x > n || y < 1 || y > m) continue;
            if(dist[x][y] != -1) continue;

            dist[x][y] = dist[i][j] + 1; // 更细结果
            q.push({x, y});
        }
    }
}

int main()
{
    cin >> n >> m >> x >> y;

    bfs();

    for(int i = 1; i <= n; i++)
    {
        for(int j = 1; j <= m; j++)
        {
            cout << dist[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```


2.1.2 kotori和迷宫

> **题目来源：** 牛客网
>
> **题目链接：** kotori和迷宫
>
> **难度系数：** ★★

【题目描述】

> $kotori$ 在一个 $n \times m$ 迷宫里，迷宫的最外层被岩浆淹没，无法涉足，迷宫内有 k 个出口。
> $kotori$ 只能上下左右四个方向移动。她想知道有多少出口是她能到达的，最近的出口离她有多远？

【输入描述】

> 第一行为两个整数 n 和 m，代表迷宫的行和列数 $(1 \leq n, m \leq 30)$
>
> 后面紧跟着 n 行长度为 m 的字符串来描述迷宫。`'k'` 代表 $kotori$ 开始的位置，`'.'` 代表道路，`'*'` 代表墙壁，`'e'` 代表出口。保证输入合法。

【输出描述】

> 若有出口可以抵达，则输出 2 个整数，第一个代表 $kotori$ 可选择的出口的数量，第二个代表 $kotori$ 到最近的出口的步数。（注意， $kotori$ 到达出口一定会离开迷宫）
> 若没有出口可以抵达，则输出 -1。

【示例一】

> 输入：
>
> ```
> 6 8
> e.*.e.
> .**.*.e
> ..*k**..
> ***.**.e
> .:*.
> *.....e
> ```
>
> 输出：
>
> ```
> 2 7
> ```

【解法】

经典 bfs 问题。

从迷宫的起点位置逐层开始搜索，每搜到一个点就标记一下最短距离。当把整个迷宫全部搜索完毕之后，扫描整个标记数组，求出出口的数量以及最短的距离。

【参考代码】

```cpp
#include <iostream>
#include <queue>
#include <cstring>

using namespace std;

typedef pair<int, int> PII;

const int N = 35;

int n, m, x, y;
char a[N][N];
int dist[N][N];

int dx[] = {0, 0, 1, -1};
int dy[] = {1, -1, 0, 0};

void bfs()
{
    memset(dist, -1, sizeof dist);
    queue<PII> q;
    q.push({x, y});
    dist[x][y] = 0;

    while(q.size())
    {
        auto t = q.front(); q.pop();
        int i = t.first, j = t.second;
        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 1 && x <= n && y >= 1 && y <= m && a[x][y] != '*' && dist[x][y] == -1)
            {
                dist[x][y] = dist[i][j] + 1;

                if(a[x][y] == 'e')
                {
                    continue;
                }
                q.push({x, y});
            }
        }
    }
}

int main()
```

```cpp
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++)
    {
        for(int j = 1; j <= m; j++)
        {
            cin >> a[i][j];
            if(a[i][j] == 'k')
            {
                x = i; y = j;
            }
        }
    }

    bfs();

    // 统计结果
    int cnt = 0, ret = 1e9;
    for(int i = 1; i <= n; i++)
    {
        for(int j = 1; j <= m; j++)
        {
            if(a[i][j] == 'e' && dist[i][j] != -1)
            {
                cnt++;
                ret = min(ret, dist[i][j]);
            }
        }
    }

    if(cnt == 0) cout << -1 << endl;
    else cout << cnt << " " << ret << endl;

    return 0;
}
```


2.1.3 Catch That Cow

> **题目来源：** 洛谷
>
> **题目链接：** [USACO07OPEN] Catch That Cow S
>
> **难度系数：** ★★

【题目描述】

> FJ 丢失了他的一头牛，他决定追回他的牛。已知 FJ 和牛在一条直线上，初始位置分别为 x 和 y，假定牛在原地不动。FJ 的行走方式很特别：他每一次可以前进一步、后退一步或者直接走到 $2 \times x$ 的位置。计算他至少需要几步追上他的牛。

【输入描述】

> 第一行为一个整数 $t(1 \leq t \leq 10)$，表示数据组数；
>
> 接下来每行包含一个两个正整数 $x, y(0 < x, y \leq 10^5)$，分别表示 FJ 和牛的坐标。

【输出描述】

> 对于每组数据，输出最少步数。

【示例一】

> 输入：
>
> 1
>
> 5 17
>
> 输出：
>
> 4

【解法】

可以暴力枚举出所有的行走路径，因为是求最少步数，所以可以用 bfs 解决：

- 从起点位置开始搜索，每次向外扩展三种行走方式；
- 当第一次搜到牛的位置时，就是最短距离。

如果不做任何处理，时间和空间都会超。因为我们会搜索到很多无效的位置，所以我们要加上剪枝策略：

1. 当 -1 减到负数的时候，剪掉；

   因为如果走到负数位置，还是需要回头走到正数位置，一定不是最优解。

2. 当 $+1$ 操作越过 y 的时候，剪掉；

   如果 $+1$ 之后大于 y，说明本身就在 y 位置或者 y 的右侧，你再往右走还是需要再向左走回去。一定不是最优解，剪掉。

3. 当 y 是偶数，并且当 $\times 2$ 操作之后大于 y 的时候，剪掉，因为不如先减到 y 的一半然后再乘；

   设当前数是 x，那么：

   - 先乘后减，总的步数 $t_1 = 2x - y + 1$；
   - 先减后乘，总的步数 $t_2 = x - y/2 + 1$；
   - $t_1 - t_2 = 2x - y + 1 - (x - y/2 + 1) = x - y/2 > 0$；
   - 因此，先乘后减不如先减后乘。

4. 设 y 是奇数的时候，那么 $y+1$ 就是偶数，根据 3 可得， $\times 2$ 操作不能超过 $y+1$。

【参考代码】

```cpp
#include <iostream>
#include <queue>
#include <cstring>

using namespace std;

const int N = 1e5 + 10;

int n = 1e5;
int x, y;
int dist[N];

void bfs()
{
    queue<int> q;
    q.push(x);
    dist[x] = 0;

    while(q.size())
    {
        auto t = q.front(); q.pop();
        int a = t + 1, b = t - 1, c = t * 2;

        if(a <= n && dist[a] == -1)
        {
            dist[a] = dist[t] + 1;
            q.push(a);
        }

        if(b > 0 && dist[b] == -1)
        {
            dist[b] = dist[t] + 1;
            q.push(b);
        }

        if(c <= n && dist[c] == -1)
        {
            dist[c] = dist[t] + 1;
            q.push(c);
        }

```

```
        // 剪枝
        if(a == y || b == y || c == y) return;
    }
}

int main()
{
    int T; cin >> T;

    while(T--)
    {
        // 注意清空数据
        memset(dist, -1, sizeof dist);
        cin >> x >> y;
        bfs();
        cout << dist[y] << endl;
    }


    return 0;
}
```

2.1.4 八数码难题

> **题目来源：** 洛谷
>
> **题目链接：** P1379 八数码难题
>
> **难度系数：** ★★★

【题目描述】

> 在 3×3 的棋盘上，摆有八个棋子，每个棋子上标有 1 至 8 的某一数字。棋盘中留有一个空格，空格用 0 来表示。空格周围的棋子可以移到空格中。要求解的问题是：给出一种初始布局（初始状态）和目标布局（为了使题目简单,设目标状态为 123804765），找到一种最少步骤的移动方法，实现从初始布局到目标布局的转变。

【输入描述】

> 输入初始状态，一行九个数字，空格用 0 表示。

【输出描述】

> 只有一行，该行只有一个数字，表示从初始状态到目标状态需要的最少移动次数。保证测试数据中无特殊无法到达目标状态数据。

【示例一】

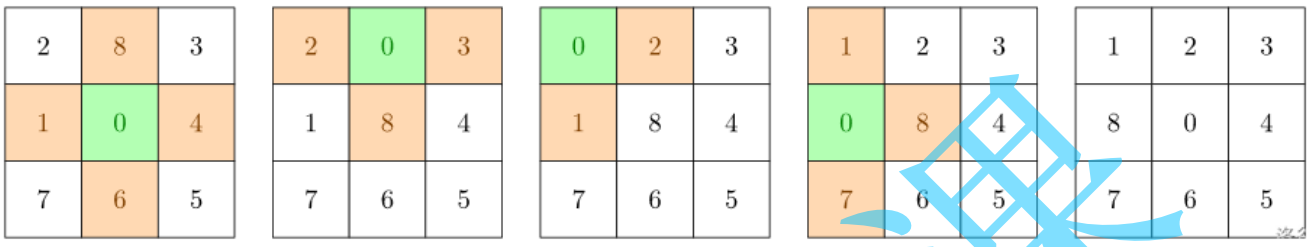
输入：

283104765

输出：

4

样例解释：



图中标有 0 的是空格。绿色格子是空格所在位置，橙色格子是下一步可以移动到空格的位置。如图所示，用四步可以达到目标状态。

【解法】

经过之前那么多题的铺垫，这道题的解法还是容易想到的。因为要求的是最短步数，因此可以用 bfs 解决。

- 从起始状态开始，每次扩展上下左右交换后的状态；
- 在搜索的过程中，第一次遇到最终状态就返回最短步数。

算法原理虽然容易，但是实现起来比较麻烦，我们要想办法处理下面几件事情：

1. 如何记录一个 3×3 的棋盘？

   可以用字符串。从上往下，从左往右将棋盘内的数依次存到一个字符串里，来标记棋盘的状态。

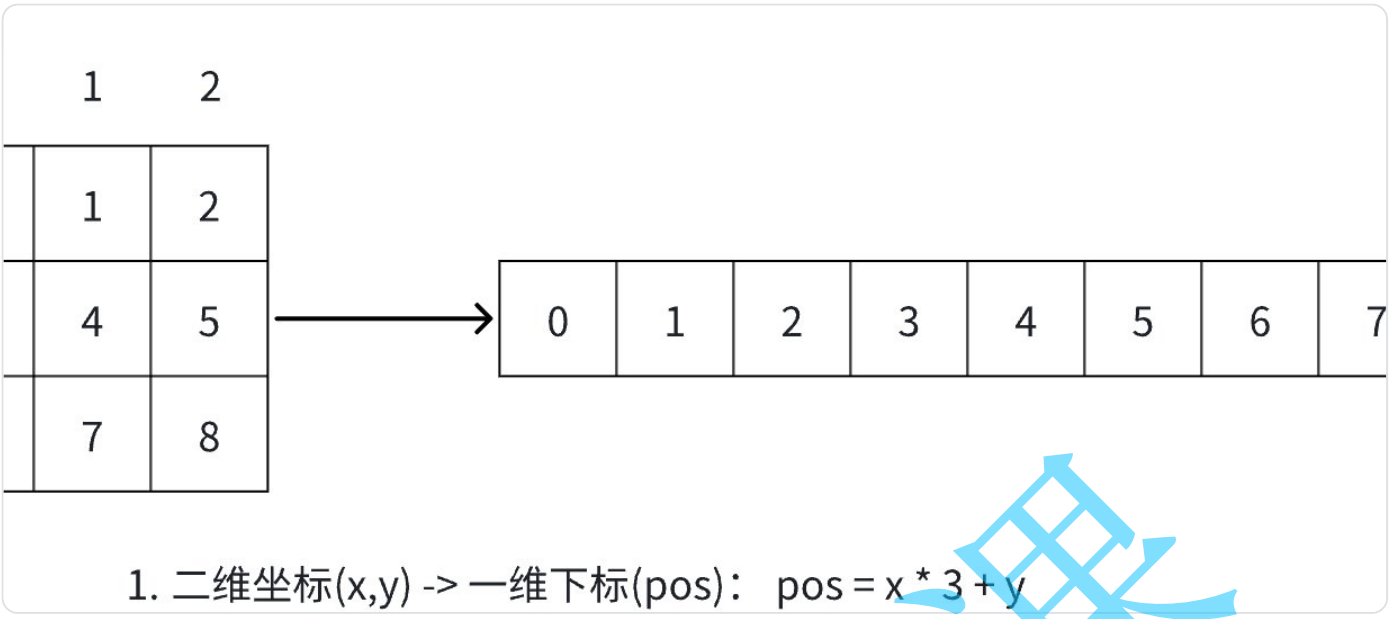
2. 如何记录最短路？

   可以用 $unordered\_map < string, int >$ 来标记最短距离；

3. 如何通过一个字符串找到交换之后的字符串？

   策略一：先把字符串还原成二维矩阵，然后交换 0 与四周的数字，最后再把交换之后的棋盘还原成字符串。

   虽然可行，但是太过于麻烦。我们其实可以通过计算，快速得出二维坐标与一维下标相互转换前后的值。如下图：



这个技巧特别常用，我们可以推广到 $n \times m$ 的矩阵坐标 (x, y)，映射成一个数 pos，可以起到空间优化的效果。后续做题中我们就会遇到。

因此，我们可以直接在字符串中，找出交换前后的下标，直接交换字符串对应位置，就能得到交换之后的状态。

【参考代码】

```cpp
#include <iostream>
#include <unordered_map>
#include <queue>

using namespace std;

string s;
string aim = "123804765";
unordered_map<string, int> dist;

int dx[] = {0, 0, 1, -1};
int dy[] = {1, -1, 0, 0};

void bfs()
{
    queue<string> q;
    q.push(s);
    dist[s] = 0;

    while(q.size())
    {
        string t = q.front(); q.pop();
```

```
23
24          int pos = 0;
25          while(t[pos] != '0') pos++;
26          int x = pos / 3, y = pos % 3; // 计算二维矩阵中对应的位置
27
28          for(int i = 0; i < 4; i++)
29          {
30              int a = x + dx[i], b = y + dy[i];
31              if(a >= 0 && a <= 2 && b >= 0 && b <= 2)
32              {
33                  string next = t;
34                  // (x, y) 与 (a, b) 做交换
35                  int p = 3 * a + b;
36                  swap(next[p], next[pos]);
37                  if(dist.count(next)) continue;
38
39                  dist[next] = dist[t] + 1;
40                  q.push(next);
41
42                  if(next == aim) return; // 剪枝
43              }
44          }
45      }
46  }
47
48  int main()
49  {
50      cin >> s;
51
52      bfs();
53
54      cout << dist[aim] << endl;
55
56      return 0;
57  }
```


2.2 多源 BFS

1. 单源最短路问题 vs 多源最短路问题

- 当问题中只存在一个起点时，这时的最短路问题就是单源最短路问题。
- 当问题中存在多个起点而不是单一起点时，这时的最短路问题就是多源最短路问题。

2. 多源 BFS

多源最短路问题的边权都为 1 时，此时就可以用多源 BFS 来解决。

3. 解决方式

把这些源点汇聚在一起，当成一个"超级源点"。然后从这个"超级源点"开始，处理最短路问题。落实到代码上时：

> 1. 初始化的时候，把所有的源点都加入到队列里面；
> 2. 然后正常执行 bfs 的逻辑即可。
>
> 也就是初始化的时候，比普通的 bfs 多加入几个起点。

2.2.1 矩阵距离

> **题目来源：** 牛客网
>
> **题目链接：** 矩阵距离
>
> **难度系数：** ★★

【题目描述】

> 给定一个 N 行 M 列的 01 矩阵 A，$A[i][j]$ 与 $A[k][l]$ 之间的曼哈顿距离定义为：$dist(A[i][j],A[k][l])=|i-k|+|j-l|$。
>
> 输出一个 N 行 M 列的整数矩阵 B，其中：
> $B[i][j]=min(1\le x\le N,1\le y\le M,A[x][y]=1)\{dist(A[i][j],A[x][y])\}$
>
> 即求与每个位置曼哈顿距离最近的 1，$N,M\le 1000$

【输入描述】

> 第一行两个整数 n,m。
>
> 接下来一个 n 行 m 列的 01 矩阵，数字之间没有空格。

【输出描述】

> 一个 n 行 m 列的矩阵 B，相邻两个整数之间用一个空格隔开。

【示例一】

> 输入：
>
> 3 4
>
> 0001
>
> 0011

> 0110
>
> 输出：
>
> 3210
>
> 2100
>
> 1001

【解法】

正难则反：

- 如果针对某一个点，直接去找最近的 1 ，我们需要对所有的 0 都来一次 bfs，这个时间复杂度是接受不了的。
- 但是我们如果反着来想，从 1 开始向外扩展，每遍历到一个 0 就更新一下最短距离。这样仅需一次 bfs，就可以把所有点距离 1 的最短距离更新出来。

正难则反是很重要的思想，后续还有很多题可以用到这个思想。

由于 1 的数量很多，因此可以把所有的 1 看成一个超级源点，从这个超级源点开始一层一层的向外扩展。实现起来也很简单，就是在初始化阶段把所有 1 的坐标加入到队列中，然后正常 bfs。

【参考代码】

```
#include <iostream>
#include <queue>
#include <cstring>

using namespace std;

typedef pair<int, int> PII;

const int N = 1010;

int n, m;
char a[N][N];
int dist[N][N];

int dx[] = {0, 0, 1, -1};
int dy[] = {1, -1, 0, 0};

void bfs()
{
    memset(dist, -1, sizeof dist);
```

```
21
22      queue<PII> q;
23      // 1. 所有的起点加入到队列里面
24      for(int i = 1; i <= n; i++)
25          for(int j = 1; j <= m; j++)
26              if(a[i][j] == '1')
27              {
28                  q.push({i, j});
29                  dist[i][j] = 0;
30              }
31
32      // 2. 正常的 bfs
33      while(q.size())
34      {
35          auto t = q.front(); q.pop();
36          int x = t.first, y = t.second;
37          for(int i = 0; i < 4; i++)
38          {
39              int a = x + dx[i], b = y + dy[i];
40              if(a >= 1 && a <= n && b >= 1 && b <= m && dist[a][b] == -1)
41              {
42                  dist[a][b] = dist[x][y] + 1;
43                  q.push({a, b});
44              }
45          }
46      }
47  }
48
49  int main()
50  {
51      cin >> n >> m;
52      for(int i = 1; i <= n; i++)
53          for(int j = 1; j <= m; j++)
54              cin >> a[i][j];
55
56      bfs();
57
58      for(int i = 1; i <= n; i++)
59      {
60          for(int j = 1; j <= m; j++)
61          {
62              cout << dist[i][j] << " ";
63          }
64          cout << endl;
65      }
66
67      return 0;
```

```
68 }
```

2.2.2 刺杀大使

题目来源： 洛谷

题目链接： P1902 刺杀大使

难度系数： ★★★

【题目描述】

某组织正在策划一起对某大使的刺杀行动。他们来到了使馆，准备完成此次刺杀，要进入使馆首先必须通过使馆前的防御迷阵。

迷阵由 $n \times m$ 个相同的小房间组成，每个房间与相邻四个房间之间有门可通行。在第 n 行的 m 个房间里有 m 个机关，这些机关必须全部打开才可以进入大使馆。而第 1 行的 m 个房间有 m 扇向外打开的门，是迷阵的入口。除了第 1 行和第 n 行的房间外，每个房间都被使馆的安保人员安装了激光杀伤装置，将会对进入房间的人造成一定的伤害。第 i 行第 j 列造成的伤害值为 $p_{i,j}$（第 i 行和第 n 行的 p 值全部为 0 ）。

现在某组织打算以最小伤害代价进入迷阵，打开全部机关，显然，他们可以选择任意多的人从任意的门进入，但必须到达第 n 行的每个房间。一个士兵受到的伤害值为他到达某个机关的路径上所有房间的伤害值中的最大值，整个部队受到的伤害值为所有士兵的伤害值中的最大值。现在，这个恐怖组织掌握了迷阵的情况，他们需要提前知道怎么安排士兵的行进路线可以使得整个部队的伤害值最小。

【输入描述】

第一行有两个整数 n, m，表示迷阵的大小。

接下来 n 行，每行 m 个数，第 i 行第 j 列的数表示 $p_{i,j}$。

对于 100% 的数据，$n, m \leq 1000$，$p_{i,j} \leq 1000$。

【输出描述】

输出一个数，表示最小伤害代价。

【示例一】

输入：

```
4 2
0 0
3 5
2 4
```

> 0 0
>
> 输出：
>
> 3

【解法】

直接找答案显然是不现实的，因为能走的路径实在是太多了，如果全都枚举出来时间上吃不消。但是题目要求的是最大值最小化，可以尝试用二分来优化枚举。

设最终结果是 x ，会发现一个性质：

- 当规定搜索过程中的最大值大于等于 x 时，我们一定可以从第一行走到最后一行；
- 当规定搜索过程中的最大值小于 x 时，我们一定不能走到最后一行。

因此，我们可以二分最终结果，通过 bfs 或者 dfs 来判断是否能走到最后一行。

如果用 dfs ，那就从第一行的每一列开始，全都搜索一遍。如果用 bfs ，可以看成多源 bfs 问题，直接把所有的源点加入队列中，然后正常搜索即可。

【参考代码】

```
#include <iostream>
#include <queue>
#include <cstring>

using namespace std;

typedef pair<int, int> PII;

const int N = 1010;

int n, m;
int p[N][N];
bool st[N][N]; // 在宽搜的过程中，不走重复路

int dx[] = {0, 0, 1, -1};
int dy[] = {1, -1, 0, 0};

// 伤害不超过 mid 的情况下，能够到达第 n 行
bool bfs(int mid)
{
    if(n == 1) return true;
    memset(st, 0, sizeof st);

```

```
24      queue<PII> q;
25      // 1. 把所有的源点加入到队列里面
26      for(int j = 1; j <= m; j++)
27      {
28          q.push({1, j});
29          st[1][j] = true;
30      }
31
32      while(q.size())
33      {
34          auto t = q.front(); q.pop();
35          int x = t.first, y = t.second;
36
37          for(int i = 0; i < 4; i++)
38          {
39              int a = x + dx[i], b = y + dy[i];
40
41              if(a >= 1 && a <= n && b >= 1 && b <= m && p[a][b] <= mid && st[a]
    [b] == false)
42              {
43                  st[a][b] = true;
44                  q.push({a, b});
45
46                  if(a == n) return true;
47              }
48          }
49      }
50
51      return false;
52  }
53
54  int main()
55  {
56      cin >> n >> m;
57      int l = 0, r = 0;
58      for(int i = 1; i <= n; i++)
59      {
60          for(int j = 1; j <= m; j++)
61          {
62              cin >> p[i][j];
63              r = max(r, p[i][j]);
64          }
65      }
66
67      // 二分答案
68      while(l < r)
69      {
```

```
70          int mid = (l + r) / 2;
71          if(bfs(mid)) r = mid;
72          else l = mid + 1;
73      }
74
75      cout << l << endl;
76
77      return 0;
78  }
```

2.3 01 BFS

01 BFS 又称双端队列 BFS。

在最短路问题中，边权值可能为 1 也可能为 0。那么，在 BFS 的过程中，可以将边权为 0 扩展出来的点放到队首，边权为 1 扩展出来的点放到队尾。这样就能保证像普通 BFS 一样整个队列队首到队尾权值单调不下降。

注意：

> 不同于普通的 bfs 问题，01bfs 问题在遍历的过程中，如果遇到之前已经遍历过的结点，有可能会找到一条更优的路线。在写代码的时候，要注意判断这一点。

2.3.1 小明的游戏

> **题目来源：** 洛谷
>
> **题目链接：** P4554 小明的游戏
>
> **难度系数：** ★★

【题目描述】

> 小明最近喜欢玩一个游戏。给定一个 $n \times m$ 的棋盘，上面有两种格子 `#` 和 `@`。游戏的规则很简单：给定一个起始位置和一个目标位置，小明每一步能向上，下，左，右四个方向移动一格。如果移动到同一类型的格子，则费用是 0 ，否则费用是 1 。请编程计算从起始位置移动到目标位置的最小花费。

【输入描述】

> 输入文件有多组数据。
>
> 输入第一行包含两个整数 n, m ，分别表示棋盘的行数和列数。
>
> 输入接下来的 n 行，每一行有 m 个格子（使用 `#` 或者 `@` 表示）。

> 输入接下来一行有四个整数 x_1, y_1, x_2, y_2 ，分别为起始位置和目标位置。
>
> 当输入 n，m 均为 0 时，表示输入结束。
>
> 对于 100% 的数据满足： $1 \le n, m \le 500$。

【输出描述】

> 对于每组数据，输出从起始位置到目标位置的最小花费。每一组数据独占一行。

【示例一】

> 输入：
>
> ```
> 2 2
> @#
> #@
> 0 0 1 1
> 2 2
> @@
> @#
> 0 1 1 0
> 0 0
> ```
>
> 输出：
>
> ```
> 2
> 0
> ```

【解法】

$01bfs$ 模板题。

- 如果走到相同格子，权值为 0 ，更新最短距离，然后加入到队头；
- 如果走到不同格子，权值为 1 ，更新最短距离，然后加入到队尾。

其余的搜索方式与常规 bfs 一模一样，但是注意松弛操作。

【参考代码】

```cpp
#include <iostream>
#include <deque>
#include <cstring>

```

```cpp
using namespace std;

typedef pair<int, int> PII;

const int N = 510;

int n, m, x1, y1, x2, y2;
char a[N][N];
int dist[N][N];

int dx[] = {0, 0, 1, -1};
int dy[] = {1, -1, 0, 0};

void bfs()
{
    if(x1 == x2 && y1 == y2)
    {
        dist[x2][y2] = 0;
        return;
    }

    memset(dist, -1, sizeof dist);
    deque<PII> q;
    q.push_back({x1, y1});
    dist[x1][y1] = 0;

    while(q.size())
    {
        auto t = q.front(); q.pop_front();
        int i = t.first, j = t.second;
        if(i == x2 && j == y2) return; // 剪枝

        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < n && y >= 0 && y < m)
            {
                char cur = a[i][j], next = a[x][y];
                int w = (cur == next ? 0 : 1);

                if(dist[x][y] == -1)
                {
                    dist[x][y] = dist[i][j] + w;

                    // 01 BFS
                    if(w == 0) q.push_front({x, y});
                    else q.push_back({x, y});
```

```
                }
                else if(dist[i][j] + w < dist[x][y]) // 虽然是第二次遇到，但是这种情况更优
                {
                    // 松弛操作
                    dist[x][y] = dist[i][j] + w;
                }

                // if(x == x2 && y == y2) return;
            }
        }
    }
}

int main()
{
    while(cin >> n >> m, n && m)
    {
        for(int i = 0; i < n; i++)
            for(int j = 0; j < m; j++)
                cin >> a[i][j];
        cin >> x1 >> y1 >> x2 >> y2;
        bfs();
        cout << dist[x2][y2] << endl;
    }

    return 0;
}
```


2.3.2 Three States

> **题目来源：** 洛谷
>
> **题目链接：** Three States
>
> **难度系数：** ★★★★

【题目描述】

> 给你一个 $n \times m$ 的地图，`.` 是荒地，`#` 是石头（不能走），数字是国家（编号为 `1`、`2`、`3`），求最少把多少荒地修成路可以使得三个国家连通，无解输出 -1。
>
> $1 \le n, m \le 10^3$。

【输入描述】

> 第一行：两个整数 n,m
>
> 接下来 n 行 m 列个字符，表示地图

【输出描述】

> 一行，表示最少路

【示例一】

> 输入：
>
> 4 5
>
> 11..2
>
> #..22
>
> #.323
>
> .#333
>
> 输出：
>
> 2

【示例二】

> 输入：
>
> 1 5
>
> 1#2#3
>
> 输出：
>
> -1

【解法】

正难则反：

- 直接找出结果点是很麻烦的，需要枚举所有的点，然后每个点都要来一次 bfs，这样是会超时的。
- 可以依次从三个国家出发，用 bfs 计算出来到达所有点的最短距离。求出所有距离之后，重新遍历所有的点，分情况求出到三个国家的最短距离。

算法原理很简单，但是里面有很多细节需要注意：

1. 因为每个国家可能有很多点，并且国家的点与点之间不连通，因此我们要用多源 bfs 求某个国家到所有点的最短距离；
2. 国家与国家之间的点如果相连，它们之间的距离是 0 （无论是不是同一个国家，在我们这道题里面，它们的距离都是 0 ，因为都不需要修路）。那么我们这道题里面边的权值要么是 0 ，要么是 1 。因此需要用 $01bfs$ 来更新所有的距离；

3. 计算某一个点到三个国家的最短距离时，应该分情况讨论。设 a,b,c 分别表示三个国家到该点的最短距离：

   a. 如果该点是一个国家，最短距离为 $a+b+c$；

   b. 如果该点是一个荒地，最短距离为 $a+b+c-2$，因为我们计算最短距离的时候，荒地会被计算三次，所以要减去两次。

【参考代码】

```cpp
#include <iostream>
#include <deque>
#include <cstring>

using namespace std;

typedef pair<int, int> PII;

const int N = 1010;

int n, m;
char a[N][N];
int dist[4][N][N];
// dist[1] dist[2] dist[3]

int dx[] = {0, 0, 1, -1};
int dy[] = {1, -1, 0, 0};

void bfs(int num)
{
    memset(dist[num], -1, sizeof dist[num]);
    // 多源bfs
    deque<PII> q;
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++)
        {
            if(a[i][j] - '0' == num)
            {
                q.push_back({i, j});
                dist[num][i][j] = 0;
            }
        }

    // 01bfs
    while(q.size())
    {
```

```
37          auto t = q.front(); q.pop_front();
38          int i = t.first, j = t.second;
39          for(int k = 0; k < 4; k++)
40          {
41              int x = i + dx[k], y = j + dy[k];
42              if(x >= 1 && x <= n && y >= 1 && y <= m && a[x][y] != '#')
43              {
44                  int w = (a[x][y] == '.' ? 1 : 0);
45
46                  if(dist[num][x][y] == -1) // 第一次遇到
47                  {
48                      dist[num][x][y] = dist[num][i][j] + w;
49                      if(w) q.push_back({x, y});
50                      else q.push_front({x, y});
51                  }
52                  else if(dist[num][i][j] + w < dist[num][x][y])
53                  {
54                      dist[num][x][y] = dist[num][i][j] + w;
55                  }
56              }
57          }
58      }
59  }
60
61  int main()
62  {
63      cin >> n >> m;
64      for(int i = 1; i <= n; i++)
65          for(int j = 1; j <= m; j++)
66              cin >> a[i][j];
67
68      bfs(1); bfs(2); bfs(3);
69
70      int ret = 0x3f3f3f3f;
71      for(int i = 1; i <= n; i++)
72          for(int j = 1; j <= m; j++)
73          {
74              if(a[i][j] == '#') continue;
75              int x = dist[1][i][j], y = dist[2][i][j], z = dist[3][i][j];
76
77              if(x == -1 || y == -1 || z == -1) continue;
78
79              if(a[i][j] == '.') ret = min(ret, x + y + z - 2);
80              else ret = min(ret, x + y + z);
81          }
82
83      if(ret == 0x3f3f3f3f) cout << -1 << endl;
```

```
84      else cout << ret << endl;
85
86      return 0;
87  }
```

3. Floodfill 问题

Floodfill 算法，又称漫水填充，像素法，本质是在寻找具有相同性质的联通块。

3.1 Lake Counting

> **题目来源：** 洛谷
>
> **题目链接：** P1596 [USACO10OCT] Lake Counting S
>
> **难度系数：** ★

【题目描述】

> 由于近期的降雨，雨水汇集在农民约翰的田地不同的地方。我们用一个 $N \times M (1 \leq N, M \leq 100)$ 的网格图表示。每个网格中有水（`W`）或是旱地（`.`）。一个网格与其周围的八个网格相连，而一组相连的网格视为一个水坑。约翰想弄清楚他的田地已经形成了多少水坑。给出约翰田地的示意图，确定当中有多少水坑。

【输入描述】

> 输入第 1 行：两个空格隔开的整数：N 和 M。
>
> 第 2 行到第 $N+1$ 行：每行 M 个字符，每个字符是 `W` 或 `.`，它们表示网格图中的一排。字符之间没有空格。

【输出描述】

> 输出一行，表示水坑的数量。

【示例一】

> 输入：
>
> 10 12
>
> W........WW.
>
> .WWW.....WWW
>
> ....WW...WW.
>
> ....WW.
>
> ....W..

> ..W.....W..
>
> .W.W.....WW.
>
> W.W.W.....W.
>
> .W.W.....W.
>
> ..W.....W.
>
> 输出：
>
> 3

【解法】

遍历整个矩阵，当遇到一个没有标记过的水坑时：

- 计数；
- 然后用 bfs 或者 dfs 将整个湖全都标记一下。

整个矩阵遍历一遍，就能得出湖的数量。

【参考代码】

```
#include <iostream>
#include <queue>
#include <cstring>

using namespace std;

const int N = 110;

int n, m;
char a[N][N];
bool st[N][N]; // 标记哪些区域已经被搜索过了

int dx[] = {0, 0, 1, -1, 1, 1, -1, -1};
int dy[] = {1, -1, 0, 0, 1, -1, 1, -1};

void dfs(int i, int j)
{
    st[i][j] = true;

    for(int k = 0; k < 8; k++)
    {
        int x = i + dx[k], y = j + dy[k];
        if(x >= 1 && x <= n && y >= 1 && y <= m && a[x][y] == 'W' && st[x][y]
== false)
```

```
24          {
25              dfs(x, y);
26          }
27      }
28  }
29
30  void bfs(int i, int j)
31  {
32      queue<pair<int, int>> q;
33      q.push({i, j});
34      st[i][j] = true;
35
36      while(q.size())
37      {
38          auto t = q.front(); q.pop();
39          int i = t.first, j = t.second;
40
41          for(int k = 0; k < 8; k++)
42          {
43              int x = i + dx[k], y = j + dy[k];
44              if(x >= 1 && x <= n && y >= 1 && y <= m && a[x][y] == 'W' && st[x]
    [y] == false)
45              {
46                  st[x][y] = true;
47                  q.push({x, y});
48              }
49          }
50      }
51  }
52
53  int main()
54  {
55      cin >> n >> m;
56      for(int i = 1; i <= n; i++)
57          for(int j = 1; j <= m; j++)
58              cin >> a[i][j];
59
60      int ret = 0;
61      for(int i = 1; i <= n; i++)
62      {
63          for(int j = 1; j <= m; j++)
64          {
65              if(a[i][j] == 'W' && st[i][j] == false)
66              {
67                  ret++;
68                  // dfs(i, j); // 把这块区域打上标记
69                  bfs(i, j); // 把这块区域打上标记
```

```
            }
        }
    }

    cout << ret << endl;

    return 0;
}
```

3.2 填涂颜色

> **题目来源：** 洛谷
>
> **题目链接：** P1162 填涂颜色
>
> **难度系数：** ★★

【题目描述】

> 由数字 0 组成的方阵中，有一任意形状的由数字 1 构成的闭合圈。现要求把闭合圈内的所有空间都填写成 2 。例如： 6×6 的方阵 $(n=6)$，涂色前和涂色后的方阵如下：
>
> 如果从某个 0 出发，只向上下左右 4 个方向移动且仅经过其他 0 的情况下，无法到达方阵的边界，就认为这个 0 **在闭合圈内**。闭合圈不一定是环形的，可以是任意形状，但保证**闭合圈内**的 0 是连通的（两两之间可以相互到达）。

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

```
0 0 0 0 0 0
0 0 0 1 1 1
0 1 1 2 2 1
1 1 2 2 2 1
1 2 2 1 2 1
1 1 1 1 1 1
```

【输入描述】

> 每组测试数据第一行一个整数 $n(1 \leq n \leq 30)$。
>
> 接下来 n 行，由 0 和 1 组成的 $n \times n$ 的方阵。
>
> 方阵内只有一个闭合圈，圈内至少有一个 0 。

【输出描述】

> 已经填好数字 2 的完整方阵。

【示例一】

> 输入：

```
6
000000
001111
011001
110001
100001
111111
```

输出：

```
000000
001111
011221
112221
122221
111111
```

【解法】

正难则反：直接找出被 1 包围的 0 是很困难的，因为很难确定当前搜索到的这个 0 是否是被包围的。但是，我们如果从边缘的 0 开始搜索，搜索到的 0 一定是没有被包围的。

因此，我们可以从边缘的 0 开始搜索，标记所有与边缘 0 相连的联通块。那么，没有被标记过的 0 就是被包围的。

小技巧：

- 我们可以把整个矩阵的外围包上一层 0 ，这样只用从 $[0,0]$ 位置开始搜即可。不用遍历第一行第一列，最后一行以及最后一列

【参考代码】

```cpp
#include <iostream>
#include <cstring>

using namespace std;

const int N = 35;

int n;
```

```cpp
int a[N][N];
bool st[N][N]; // 只会标记边缘的 0

int dx[] = {0, 0, -1, 1};
int dy[] = {1, -1, 0, 0};

void dfs(int i, int j)
{
    st[i][j] = true;

    for(int k = 0; k < 4; k++)
    {
        int x = i + dx[k], y = j + dy[k];
        // 注意此时的边界
        if(x >= 0 && x <= n + 1 && y >= 0 && y <= n + 1 && a[x][y] == 0 && st[x][y] == 0)
        {
            dfs(x, y);
        }
    }
}

int main()
{
    while(cin >> n)
    {
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= n; j++)
                cin >> a[i][j];

        dfs(0, 0);

        for(int i = 1; i <= n; i++)
        {
            for(int j = 1; j <= n; j++)
            {
                if(a[i][j]) cout << a[i][j] << " ";
                else if(st[i][j]) cout << 0 << " ";
                else cout << 2 << " ";
            }
            cout << endl;
        }
    }


    return 0;
}
```