2019 年程序设计与算法练习题解析

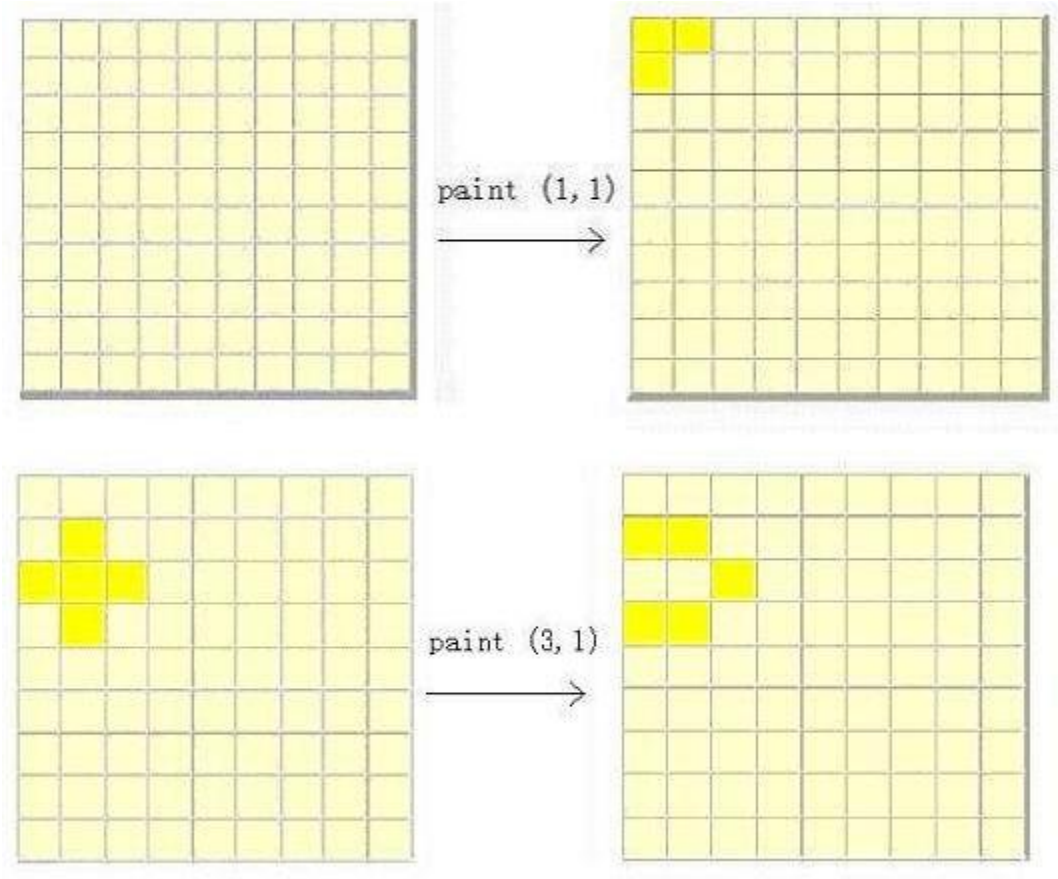
1. 枚举、分治

1.1. 画家问题

总时间限制： 1000ms 内存限制： 65536kB

描述

有一个正方形的墙，由 N*N 个正方形的砖组成，其中一些砖是白色的，另外一些砖是黄色的。Bob 是个画家，想把全部的砖都涂成黄色。但他的画笔不好使。当他用画笔涂画第(i, j)个位置的砖时， 位置(i-1, j)、 (i+1, j)、 (i, j-1)、 (i, j+1)上的砖都会改变颜色。请你帮助 Bob 计算出最少需要涂画多少块砖，才能使所有砖的颜色都变成黄色。



输入

第一行是一个整数 n (1≤n ≤15)，表示墙的大小。接下来的 n 行表示墙的初始状态。每一行包含 n 个字符。第 i 行的第 j 个字符表示位于位置(i,j)上的砖的颜色。“w”表示白砖，“y”表示黄砖。

输出

一行，如果 Bob 能够将所有的砖都涂成黄色，则输出最少需要涂画的砖数，否则输出“inf”。

样例输入

```
5
wwwww
wwwww
wwwww
wwwww
wwwww
```

样例输出

```
15
```

思路分析

题目的关键在于，如果第一行的状态集合确定了，那么以后各行的状态集合也就确定了。

我们按照从上到下、从左到右的顺序确定需要粉刷的砖块。考虑相邻两行间的相互影响，下一行的每一格只能影响上一行正对的一格。因此，当上一行的粉刷方案确定后，下一行的粉刷方案也随之确定。

枚举第一行的粉刷方案，然后用上述方法依次确定下面各行的粉刷方案。第 k 行的粉刷方案保证第 k－1 行为黄色，对于第 k 行的每一个点，需要判断它上边的点是否为黄色，不是则涂它自己。n 行全部完成后，只需检查第 n 行是否全为黄色，就可以判断这种方案是否合法。

代码技巧

- 考虑到墙的四边和墙的中间不太一样，我们可以对数组从 index 为 1 作为开始。
- 因为题目中已经限制墙的大小小于 15，所以我们那 20*20 的二维数组保存就可以了。
- 对于第一行的枚举，不管原来是什么状态，我们可以有 2^n 种枚举状态。可以拿一个 line 数组保存这种枚举状态下，第一行要涂那几个点。
- 数组可以不存字符 w 和 y，而是存 0 和 1 表示涂与没涂两种状态。

参考代码

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

int N, count, result;
char c;
char board[20][20];
```

```
8.  char map[20][20];
9.  char line[20];
10.
11. void draw(int x, int y) {
12.     map[x][y] = !map[x][y];
13.     map[x-1][y] = !map[x-1][y];
14.     map[x+1][y] = !map[x+1][y];
15.     map[x][y-1] = !map[x][y-1];
16.     map[x][y+1] = !map[x][y+1];
17.     count++;
18. }
19.
20. void enumerate(int k) {
21.     // enumerate all the cases to paint of the first line
22.     int j = 1;
23.     while (j <= N) {
24.         line[j] = k % 2;
25.         k /= 2;
26.         j++;
27.     }
28. }
29.
30. bool guess() {
31.     // judge from the second line, if the point above it is
32.     // not yellow(i.e. white), draw the point
33.     for (int i = 2; i <= N; i++) {
34.         for (int j = 1; j <= N; j++) {
35.             if (map[i-1][j] == 0) draw(i, j);
36.         }
37.     }
38.
39.     // judge if all the point of the last line is yellow
40.     // if it is, return the draw-count, else return false
41.     for (int j = 1; j <= N; j++)
42.         if (map[N][j] != 1)
43.             return false;
44.
45.     // result keeps the minimum draw-count number
46.     if (count < result) result = count;
47.     return true;
48. }
49.
50. int main() {
51.     cin >> N;
52.     result = N * N + 1;
53.     for (int i = 1; i <= N; i++) {
54.         for (int j = 1; j <= N; j++) {
55.             cin >> c;
56.             if (c == 'w') board[i][j] = 0;
57.             else board[i][j] = 1;
58.         }
59.     }
60.
61.     for (int k = 0; k < (1 << N); k++) {
62.         count = 0;
63.         memcpy(map, board, sizeof(board));
64.         enumerate(k);
65.         for (int j = 1; j <= N; j++) {
66.             if (line[j] == 1)
67.                 draw(1, j);
68.         }
```

```
        guess();
    }

    if (result != N * N + 1) cout << result << endl;
    else cout << "inf" << endl;

    return 0;
}
```

1.2. Sophie 开关灯

总时间限制： 5000ms 单个测试点时间限制： 1000ms 内存限制： 65536kB

描述

Sophie 手里有一盏灯和一个时间序列 t[1...N]（保证 t[1] < t[2] < ... < t[N])<t[2]<...<t[n]）

在 T=0 时刻灯被打开，然后 T=t[1]时刻关闭，T=t[2]时刻打开，T=t[3]时刻关闭...直到 T=M 时刻。那么在 T=[0,M]这段时间内，亮灯时间就能被计算出来。

Sophie 现在有一次修改序列 t[1...N]的机会：她可以在序列中插入一个值，并且仍然保持序列中相邻元素的‘<’关系。

请问修改之后，在[0,M]时段中亮灯时间的最大值，特别地：Sophie 也可以选择不使用这次机会。例如 M = 10，时间序列[4,6,7]，则[0,4]、[6,7]这两个时段为亮灯时间；若将序列改为[3,4,6,7]，则亮灯时段为[0,3]+[4,6]+[7,10] = 8，为最长亮灯时间。

输入

第一行 2 个整数 N，M
第二行 N 个正整数代表时间序列，保证严格升序

输出

输出一个正整数，代表最长亮灯时间

样例输入

```
#1:
3 10
4 6 7

#2:
2 12
1 10

#3:
2 7
3 4
```

样例输出

```
#1:
8

#2:
9

#3:
6
```

提示

* 1<=N<=1e5, 2 <= M <= 1e9, 0 < t[1] < t[2] < ... < t[N] < M

参考代码

```cpp
#include <iostream>
using namespace std;
int main() {
    int n, m, ca;
    cin >> n >> m;
    int a[n + 3] = {0}, b[n + 3] = {0}, t[n + 3];
    int sumj = 0, sumo = 0, num = 1;
    t[0] = 0;
    t[n + 1] = m;
    for(int i = 1; i <= n; i++) cin >> t[i];
    for(int i = 1; i <= n + 1; i++) {
        if(i % 2 == 1) sumj += t[i] - t[i - 1];
        else sumo += t[i] - t[i - 1];
    }
    b[0] = sumo;
    for(int i = 1; i <= n + 1; i++) {
        if(i % 2 == 1) {
            a[i] = a[i - 1] + t[i] - t[i - 1];
            b[i] = b[i - 1];
        }
        else {
            a[i] = a[i - 1];
            b[i] = b[i - 1] - t[i] + t[i - 1];
        }
    }
    int maxn = sumj;
    for(int i = 1; i <= n + 1; i++) {
        maxn = max(maxn, a[i] + b[i] - 1);
    }
    cout << maxn << endl;
    return 0;
}
```

1.3. Sophie 吃甘蔗

总时间限制： 5000ms 单个测试点时间限制： 1000ms 内存限制： 65536kB

描述

1 根甘蔗共有 N 段，美味度为 a[1...N]，Sophie 想要吃掉其中 L 段，并且甘蔗只能从 2 头开始吃。求 Sophie 能获得的最大总美味度。

输入

第一行 2 个正整数 N，L
第二行 N 个整数 a[i]

输出

输出 1 个正整数，代表能获得的最大总美味度

样例输入

```
#1:
4 2
1 2 3 4

#2:
4 2
2 3 -1 4
```

样例输出

```
#1:
7

#2:
6
```

提示

样例 1 中，吃掉末端的[3,4]可获得最多美味度=7
样例 2 中，吃掉首端的[2]和末端的[4]可获得最多美味度=6
1 <= L <= N <= 1e5, a[i]为 int 范围

思路分析

- 本来第一思路是用双指针来做，类似于第四题，但是发现和题意不符，可以看看第二个样例输入
- 因为要吃 L 段甘蔗，而且只能从两头吃，可以认为我们需要从这 N 段里面取出连续的 L 段甘蔗（将 N 段的头和尾连起来），所以我们只需要枚举出 L 个连续的 L 段即可。
- 可以理解为枚举从最后的连续 L 段，到前连续的 L 段。

代码技巧

- 题目中说明 a[i]是 int 范围，并不代表最多美味度是 int 范围，需要用 long long 表示
- 对于连续的 L 段，我们可以用另外一个求和数组表示

参考代码

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

long long a[1000001];
long long s[1000001];
int main() {
    int N, L;
    cin >> N >> L;

    for (int i = 0; i < N; i++) {
        cin >> a[i];
        s[i+1] = s[i] + a[i];
    }

    long long result = s[N] - s[N-L];
    for (int i = 1; i <= L; i++) {
        long long sweet = s[i] + s[N] - s[N-L+i];
        if (sweet > result) result = sweet;
    }
    cout << result << endl;
    return 0;
}
```

1.4. 数据清洗

总时间限制： 1000ms 内存限制： 65536kB

描述

在数据分析中，数据清洗是非常关键的一环。数据清洗的目的是过滤掉一些噪音和异常数据，例如去掉样本数据中过大或过小的数。
给定一个正整数 m 和一个含有 n 个元素的列表（m<n/2），列表的每一个元素都是一个整数，求去掉列表中前 m 大和前 m 小的数后剩余数据的平均值（大小相等的数按照输入顺序定排名，先输入的“较小”），结果保留两位小数。

输入

第一行，m 和 n，使用空格分隔，0 ⩽ m < n/2 < n ⩽ 30。
第二行，一个 n 个元素的列表，以空格分隔。

输出
一行，去掉前 m 大和前 m 小的数后的平均值，结果保留两位小数。

样例输入

```
#Example 1:
2 6
2 3 1 6 5 4

#Example 2:
2 7
1 2 2 1 4 3 3
```

样例输出

```
#Example 1:
3.50

#Example 2:
2.33
```

思路分析

这道题很简单。输入之后排序，加和取平均即可。

参考代码

```cpp
#include<iostream>
#include<algorithm>
#include<iomanip>
using namespace std;
int main()
{
    int m, n;
    int sum = 0;
    int a[31] = {0};

    cin >> m >> n;
    for(int i = 0; i < n; i++) cin>>a[i];
    sort(a, a+n);

    for(int i = m; i <= n-m-1; i++) sum += a[i];

    double ave = (double)sum / (double)(n-2*m);
```

```
    cout << fixed << setprecision(2) << ave << endl;
    return 0;
}
```

1.5. 数组合并

总时间限制： 1000ms 内存限制： 35536kB

描述

输入两个有序数组，把它们合并成一个有序数组(相同数据只保留一个)

输入

三行 第一行输入两个数组个数 下面一行一个数组

输出

一行 合并完的数组

样例输入

```
11 10
1 2 3 4 5 6 7 8 10 11 15
6 9 10 11 15 16 19 20 21 26
```

样例输出

```
1 2 3 4 5 6 7 8 9 10 11 15 16 19 20 21 26
```

提示

每个数组的长度<=10000

思路分析

典型的双指针问题。因为输入的两个数组是有序的，两个指针每次比较，将小的输出，指针后移。如果两个数相同，则两个指针都往后移。第一个循环结束之后，可能还有一个数组的数没有遍历到，再循环一次即可。

参考代码

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

int main() {
    int m, n;
```

```
    int i, j;
    cin >> m >> n;
    vector<int> a(m, 0), b(n, 0);
    for (i = 0; i < m; i++) cin >> a[i];
    for (i = 0; i < n; i++) cin >> b[i];

    for (i = 0, j = 0; i < m && j < n;) {
        if (a[i] < b[j]) cout << a[i++] << ' ';
        else if (a[i] > b[j]) cout << b[j++] << ' ';
        else {
            cout << a[i++] << ' ';
            j++;
        }
    }

    while (i < m) cout << a[i++] << ' ';
    while (j < n) cout << b[j++] << ' ';

    return 0;
}
```

2. 回溯

2.1. 迷宫问题

总时间限制:

1000ms

内存限制:

65536kB

描述

定义一个二维数组:

```
int maze[5][5] = {

0, 1, 0, 0, 0,

0, 1, 0, 1, 0,

0, 0, 0, 0, 0,

0, 1, 1, 1, 0,

0, 0, 0, 1, 0,
```

```
};
```

它表示一个迷宫，其中的 1 表示墙壁，0 表示可以走的路，只能横着走或竖着走，不能斜着走，要求编程序找出从左上角到右下角的最短路线。

输入

一个 5 × 5 的二维数组，表示一个迷宫。数据保证有唯一解。

输出

左上角到右下角的最短路径，格式如样例所示。

样例输入

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

样例输出

```
(0, 0)
(1, 0)
(2, 0)
(2, 1)
(2, 2)
(2, 3)
(2, 4)
(3, 4)
(4, 4)
```

思路分析：

和课上的内容很相似。解决这道题有两种思路：

一种是利用回溯，枚举得到所有可能到达终点的路线，从中选择路径长度最小的路线。具体来说，假设当前路径中有（x，y）点，那么将该点设置为不可访问，因为最短路径中必定不会走回头路。再枚举路径中如果有上、下、左、右点四种情况。最后假设当前路径中不需要这个点，回溯，将该点从当前路径中删除。如果到达终点，比较当前路径的长度和当前最小路径的长度，如果当前路径的长度比最小路径长度小，将用当前路径替换当前最小路径。枚举所有可能的路径后，输出最小的路径。

另一种思维更加直接，可以使用广度优先搜索。因为题目中要求最短路径，使用广度优先搜索得到的一定是最短路径。每次将当前可访问的还没有访问过的节点作为下一批次访问的节点，同时记录到达每点的路径。一旦到达终点则停止访问。根据每访问一个节点就记录访问该节点的路径，可以得到最短的访问路径。

参考代码：

回溯

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

using namespace std;
struct Point
{
    int x;
    int y;
    Point(int x_ = 0, int y_ = 0) : x(x_), y(y_) {}
};
int maze[5][5];//迷宫地图
unsigned int minStep = 30;//最小路径的步数
vector<Point> minRoute;//存储最小路径
vector<Point> route;//当前路径

int dx[4] = { 0, 1, 0, -1 };//4 个方向
int dy[4] = { 1, 0, -1, 0 };

void dfs(int x, int y)
{
    route.push_back(Point{ x, y });
    if (x == 4 && y == 4)//到达终点
    {
        if (route.size() < minStep)//记录当前最小路径
        {
            minRoute = route;
            minStep = route.size();
        }
        route.pop_back();
        return;
    }
    maze[x][y] = 1;//当前点不可再次访问
    for (int forward = 0; forward < 4; ++forward)//枚举所有能够访问到的点
    {
        int nextX = x + dx[forward];
        int nextY = y + dy[forward];
        if (nextX < 0 || nextY < 0 || nextX >= 5 || nextY >= 5 || maze[nextX][nextY] ==
    1)
            continue;
```

```
        dfs(nextX, nextY);//递归访问
    }
    route.pop_back();
    maze[x][y] = 0;//回溯
}

int main()
{
    minRoute.clear();
    route.clear();
    for (int y = 0; y < 5; ++y)
    {
        for (int x = 0; x < 5; ++x)
        {
            cin >> maze[x][y];
        }
    }
    dfs(0, 0);
    for (auto &p : minRoute)
    {
        printf("(%d, %d)\n", p.y, p.x);
    }
    return 0;
}
```

广度优先搜索：

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>

using namespace std;
struct Point
{
    int x;
    int y;
    Point(int x_ = 0, int y_ = 0) : x(x_), y(y_) {}
};

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

int main()
{
    int maze[5][5];
    int preStep[5][5];
    for (int y = 0; y < 5; ++y)
    {
        for (int x = 0; x < 5; ++x)
        {
            cin >> maze[x][y];
            preStep[x][y] = -1;
        }
    }
    queue<Point> q;//广搜使用的先入先出队列
    Point p(0, 0);//第一批次访问的节点
    q.push(p);
    while (p.x != 4 || p.y != 4)
```

```
    {
        maze[p.x][p.y] = 1;//已经访问过的节点不可再次访问
        p = q.front();
        q.pop();
        for (int forward = 0; forward < 4; ++forward)
        {
            Point next(p.x + dx[forward], p.y + dy[forward]);//访问所有可以访问并且还没有访问的节点
            if (next.x < 0 || next.y < 0 || next.x >= 5 || next.y >= 5 || maze[next.x][next.y] == 1)
            {
                continue;
            }
            preStep[next.x][next.y] = forward;//记录当前点的路径
            q.push(next);
        }
    }
    /*
    可以通过终点的路径一步一步地找到起点，
    但是这一过程的输出为倒序，可以借助栈调整为顺序
    */
    stack<Point> ans;
    p = Point{ 4, 4 };
    while (p.x != 0 || p.y != 0)
    {
        ans.push(p);
        Point tmp;
        tmp.x = p.x - dx[preStep[p.x][p.y]];
        tmp.y = p.y - dy[preStep[p.x][p.y]];
        p = tmp;
    }
    printf("(0, 0)\n");
    while (!ans.empty())
    {
        p = ans.top();
        ans.pop();
        printf("(%d, %d)\n", p.y, p.x);
    }
    return 0;
}
```

2.2. 棋盘问题

总时间限制:

1000ms

内存限制:

65536kB

描述

在一个给定形状的棋盘（形状可能是不规则的）上面摆放棋子，棋子没有区别。要求摆放时任意的两个棋子不能放在棋盘中的同一行或者同一列，请编程求解对于给定形状和大小的棋盘，摆放 k 个棋子的所有可行的摆放方案 C。

输入

输入含有多组测试数据。
每组数据的第一行是两个正整数，n k，用一个空格隔开，表示了将在一个 n*n 的矩阵内描述棋盘，以及摆放棋子的数目。 n <= 8 , k <= n
当为-1 -1 时表示输入结束。
随后的 n 行描述了棋盘的形状：每行有 n 个字符，其中 # 表示棋盘区域， . 表示空白区域（数据保证不出现多余的空白行或者空白列）。

输出

对于每一组数据，给出一行输出，输出摆放的方案数目 C （数据保证 C<2^31）。

样例输入

```
2 1
#.
.#
4 4
...#
..#.
.#..
#...
-1 -1
```

样例输出

```
2
1
```

思路分析：

类似八皇后问题。记录每一列是否被占用，对每一行，枚举每一种占用可能，尝试在可以下棋子的点下棋，再回溯。如果所有棋子都已经有位置，表明得到一组可行解。注意：不是每一行都必须放棋子。

参考代码：

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

using namespace std;

```

```cpp
int board[8][8];//棋盘
bool row[8];//每一列的占用情况
int C;//可行解的数目
int n, k;

void Find(int y, int remain) {//y 行号，remain：需要放的棋子
    if (remain == 0) {//没有需要放的棋子，得到一组可行解
        C++;
        return;
    }
    if (y == n)//超出棋盘范围
        return;
    if (n - y < remain)//剩余棋子数目>行数，不可能得到解
        return;
    for (int x = 0; x < n; ++x) {//枚举所有可能情况
        if (board[x][y] || row[x])
            continue;
        row[x] = true;
        Find(y + 1, remain - 1);//递归放置棋子
        row[x] = false;
    }
    Find(y + 1, remain);//考虑这一行不放置棋子的情况
}


int main() {
    while (true) {
        cin >> n >> k;
        if (n == -1 && k == -1)
            break;
        C = 0;
        for (int y = 0; y < n; ++y) {
            for (int x = 0; x < n; ++x) {
                char c;
                cin >> c;
                if (c == '.') {
                    board[x][y] = 1;
                }
                else {
                    board[x][y] = 0;
                }
            }
        }
        memset(row, 0, sizeof(row));
        Find(0, k);
        printf("%d\n", C);
    }
    return 0;
}
```

2.3. 神奇的口袋

总时间限制:

1000ms

内存限制:

65536kB

描述

有一个神奇的口袋，总的容积是 400，用这个口袋可以变出一些物品，这些物品的总体积必须是 400。John 现在有 n 个想要得到的物品，每个物品的体积分别是 a_1，a_2……a_n。John 可以从这些物品中选择一些，如果选出的物体的总体积是 400，那么利用这个神奇的口袋，John 就可以得到这些物品。现在的问题是，John 有多少种不同的选择物品的方式。

输入

输入的第一行是正整数 n (1 <= n <= 200)，表示不同的物品的数目。接下来的 n 行，每行有一个 1 到 400 之间的正整数，分别给出 a_1，a_2……a_n 的值。

输出

输出不同的选择物品的方式的数目**对 10000 取模的结果**（因为结果可能很大，为了避免高精度计算，只要求对 10000 取模的结果）。

样例输入

```
3
200
200
200
```

样例输出

```
3
```

思路分析：

按照顺序枚举是否生成某一样物品，从而得到所有的生成情况。每一种物品可以选择生成或者不生成。注意应当建立一个数组存储枚举到某一个物品时还剩余 x 体积时的可行解数目，如果下次遇到这种情况则不用重复计算，否则会超时。

参考代码：

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

using namespace std;
int total[201][401];//[object][remain volume]
int volume[200];
int n;
```

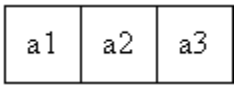
```
9.
10. int df(int object, int remain) {//object 枚举到的物品编号，remain 剩余的体积
11.     if (remain == 0) {//体积为0，得到一组可行解
12.         return 1;
13.     }
14.     if (object >= n)//所有的物品已经枚举完了，还没有找到，说明这一次尝试不可行
15.         return 0;
16.     if (total[object][remain] != -1)//如果之前计算过这种情况，则不需要计算
17.         return total[object][remain];
18.     if (remain < volume[object])//剩余体积<当前物品体积，不使用当前物品
19.         total[object][remain] = df(object + 1, remain);
20.     else
21.         total[object][remain] = (df(object + 1, remain - volume[object]) + df(object +
    1, remain)) % 10000;
22.     //注意记录当前结果
23.     return total[object][remain];
24. }
25.
26. int main() {
27.     cin >> n;
28.     for (int i = 0; i < n; ++i) {
29.         cin >> volume[i];
30.     }
31.     for (int i = 0; i < n; ++i) {
32.         for (int w = 0; w <= 400; ++w) {
33.             total[i][w] = -1;
34.         }
35.     }
36.     cout << df(0, 400);
37.     return 0;
38. }
```

2.4. 数字方格

总时间限制:
1000ms

内存限制:
65536kB

描述



如上图，有 3 个方格，每个方格里面都有一个整数 a1，a2，a3。已知 0 <= a1, a2, a3 <= n，而且 a1 + a2 是 2 的倍数，a2 + a3 是 3 的倍数， a1 + a2 + a3 是 5 的倍数。你的任务是找到一组 a1，a2，a3，使得 a1 + a2 + a3 最大。

输入
一行，包含一个整数 n (0 <= n <= 100)。

输出
一个整数，即 a1 + a2 + a3 的最大值。

样例输入

```
3
```

样例输出

```
5
```

思路分析：

暴力枚举即可。不加任何优化即可通过此题。枚举方式有多种，包括枚举 a1,a2,a3（最直接的思路），从大到小枚举 a1+a2+a3 和 a1,a2（得到的满足条件的第一个结果即最优解）

提示：题目中 0 <= a1, a2, a3 <= n, n<=100。如果直接暴力枚举需要10^6数量级的操作，时间限制为 1000ms，最多能支持10^9数量级操作

参考代码：

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

using namespace std;

int main() {
    int n;
    cin >> n;
    int maxSum = -1;
    for (int a1 = 0; a1 <= n; a1++) {
        for (int a2 =0; a2 <= n; a2 ++) {
            if ((a1 + a2) % 2 != 0)
                continue;
            for (int a3 = 0; a3 <= n; a3 ++) {
                int result = a1 + a2 + a3;
                if ( result % 5 == 0 && (a2+a3)%3 == 0) {
                    maxSum = max(result, maxSum);
                }
            }
        }
    }
    cout << maxSum << endl;
    return 0;
}
```

3. 分支界限

3.1. 抓住那头牛

总时间限制： 2000ms 内存限制： 65536kB

描述

农夫知道一头牛的位置，想要抓住它。农夫和牛都位于数轴上，农夫起始位于点 N(0<=N<=100000)，牛位于点 K(0<=K<=100000)。农夫有两种移动方式：

1、从 X 移动到 X-1 或 X+1，每次移动花费一分钟

2、从 X 移动到 2*X，每次移动花费一分钟

假设牛没有意识到农夫的行动，站在原地不动。农夫最少要花多少时间才能抓住牛？

输入

两个整数，N 和 K

输出

一个整数，农夫抓到牛所要花费的最小分钟数

样例输入

```
5 17
```

样例输出

```
4
```

思路分析

题目待求农夫抓牛的最短时长，农夫下一时刻可能到达的位置有 3 个（当前位置 X）：X-1，X+1，2*X，考虑利用广度优先搜索的策略解题。

这里需要维护两个状态：1.农夫从初始位置出发，到达某个特定位置的最短时长；2.农夫从特定位置出发，接下来可能到达的位置。前者可以用数组实现，后者可以用队列实现。

接下来将所有可能到达的位置加入队列，然后逐个从队列中弹出，判断是否到达终点，并继续将之后可能到达的位置入队，循环直至到达终点或队列为空。

参考代码

```
1. #include <iostream>
2. #include <algorithm>
3. #include <queue>
```

```cpp
using namespace std;

int N, K;
const int MAX_N = 100000;

//steps 数组存储农夫到达该位置需要的时长
//-1 表示该位置还未被访问
//大于-1 的值表示该位置已被访问，且数值大小即为到达该位置所需的时长
int steps[MAX_N + 1];

//posQueue 队列存储待访问的位置
queue<int> posQueue;

int main() {
    cin >> N >> K;
    fill(steps, steps + MAX_N + 1, -1);

    //农夫起始位置为 N
    posQueue.push(N);
    steps[N] = 0;

    while(!posQueue.empty()) {
        int pos = posQueue.front();

        //抓到牛
        if (pos == K) {
            cout << steps[pos] << endl;
            return 0;
        }

        //农夫每次可能的移动
        else {
            if (pos - 1 >= 0 && steps[pos - 1] == -1) {
                posQueue.push(pos - 1);
                steps[pos - 1] = steps[pos] + 1;
            }
            if (pos + 1 <= MAX_N && steps[pos + 1] == -1) {
                posQueue.push(pos + 1);
                steps[pos + 1] = steps[pos] + 1;
            }
            if (pos * 2 <= MAX_N && steps[pos * 2] == -1) {
                posQueue.push(pos * 2);
                steps[pos * 2] = steps[pos] + 1;
            }
            posQueue.pop();
        }
    }
    return 0;
}
```

3.2. 象棋中的马的问题

总时间限制： 1000ms 内存限制： 64000kB

描述

现在棋盘的大小不一定，由 p，q 给出，并且在棋盘中将出现障碍物（限制马的行动，与象棋走法相同）

输入

第一行输入 n 表示有 n 组测试数据
每组测试数据第一行输入 2 个整数 p,q 表示棋盘的大小(1<=p,q<=100)
每组测试数据第二行输入 4 个整数，表示马的起点位置与终点位置
第三行输入 m 表示图中有多少障碍
接着跟着 m 行，表示障碍的坐标

输出

马从起点走到终点所需的最小步数
如果马走不到终点输入“can not reach!”

样例输入

```
2
9 10
1 1 2 3
0
9 10
1 1 2 3
8
1 2
2 2
3 3
3 4
1 4
3 2
2 4
1 3
```

样例输出

```
1
can not reach!
```

思路分析

题目待求马移动的最小步数，马每次移动下一时刻可能到达的位置有 8 个，同时受到障碍物限制。考虑利用广度优先搜索的策略解题。

考虑到这里是二维棋盘，为方便记录坐标点和到达对应坐标点的最小步数，这里定义了结构体 Pos，包含位置坐标（x,y）和到达该位置的最小步数 steps。

这里需要维护 3 个状态：1.特定位置是否访问过；2.障碍物的位置；3.马可能移动到的位置。前 2 个状态可以分别用数组 visited 和 barriers 实现，第三个状态可以用队列 pos_queue 实现。

接下来将所有可能到达的位置加入队列，然后逐个从队列中弹出，判断是否到达终点，并继续将之后可能到达的位置入队，循环直至到达终点或队列为空。过程中重点考虑是否超出棋盘、是否访问过和是否受到障碍物限制。

向 8 个方向移动可以用数组 dir 记录相对距离，这样就可以利用一个循环来依次判断 8 个方向的情况。对障碍物的判断也是同理。

参考代码

```
#include <iostream>
#include <algorithm>
#include <queue>
using namespace std;

//定义结构体 Pos，包含位置坐标 x，y，和到达该位置的最小步数 steps
struct Pos {
    int x;
    int y;
    int steps;
    Pos(int xx, int yy, int ss):x(xx), y(yy), steps(ss) {}
};

//马可以向 8 个方向移动，同时会受到四个方向障碍物的限制
//dir 和 bar_dir 分别对应 8 个方向移动的相对位置和障碍物的相对位置
int dir[8][2] = {-1, 2, 1, 2, 2, 1, 2, -1, 1, -2, -1, -2, -2, -1, -2, 1};
int bar_dir[8][2] = {0, 1, 0, 1, 1, 0, 1, 0, 0, -1, 0, -1, -1, 0, -1, 0};

//visited 数组标志某特定位置是否访问过（障碍物位置可以当做已访问过处理）
//barriers 数组标志障碍物位置
//pos_queue 队列保存待搜索的位置
bool visited[101][101];
bool barriers[101][101];
queue<Pos> pos_queue;

int main() {
    int n, p, q;
    int x_start, y_start, x_end, y_end;
    int num_barriers;
    int x_tmp, y_tmp;

    cin >> n;
    while(n--) {
        fill(*visited, *visited + 101 * 101, false);
        cin >> p >> q;
        cin >> x_start >> y_start >> x_end >> y_end;
        //起点入队
        pos_queue.push(Pos(x_start, y_start, 0));
        visited[x_start][y_start] = true;
        //输入障碍物坐标
        cin >> num_barriers;
```

```
        for (int i = 0; i < num_barriers; i++) {
            cin >> x_tmp >> y_tmp;
            visited[x_tmp][y_tmp] = true;
            barriers[x_tmp][y_tmp] = true;
        }

        while(!pos_queue.empty()) {
            Pos pos = pos_queue.front();
            //结束条件之一：到达终点
            if (pos.x == x_end && pos.y == y_end) {
                cout << pos.steps << endl;
                break;
            }
            else {
                pos_queue.pop();
                //向 8 个方向试探移动
                for (int i = 0; i < 8; i++) {
                    x_tmp = pos.x + dir[i][0];
                    y_tmp = pos.y + dir[i][1];
                    if (x_tmp >= 0 && x_tmp < p && y_tmp >=0 && y_tmp < q && !visited[x_tmp][y_tmp] && !barriers[pos.x + bar_dir[i][0]][pos.y + bar_dir[i][1]]) {
                        pos_queue.push(Pos(x_tmp, y_tmp, pos.steps + 1));
                        visited[x_tmp][y_tmp] = true;
                    }
                }
            }
        }
        //结束条件之二：队列变空，即终点不可达
        if (pos_queue.empty())
            cout << "can not reach!" << endl;
        //清空队列
        while(!pos_queue.empty()) {pos_queue.pop();}
    }
    return 0;
}
```

3.3. 拯救公主

总时间限制： 5000ms 内存限制： 65536kB

描述

公主被魔王抓起来关在了迷宫的某处，骑士想要拯救公主，也进入了迷宫。骑士现在身处迷宫的某处，但他不知道他能否走到公主的所在地，请帮忙找出骑士能否走到公主的所在地。

迷宫由 m*n 个方块组成，每个方块有墙或者路，骑士在其中一个方块上，他每个时间单位可以四个方向（上、下、左、右）走到相邻方格。请判断骑士能否走到公主被囚禁的方格，如果能，请计算骑士走到公主囚禁方格所需花费的最少时间。

输入

第一行为两个整数 m 和 n(2=<m,n<=20)，以空格分隔。
第 2 至 m+1 行描述了迷宫，迷宫以 m 行 n 列的方格组成，若方格为"0"则表示骑士可以通过。若方格为"1"则表示墙，骑士不能通过。"*"表示了骑士当前所在的位置，"+"表示公主被囚禁的位置。
</m,n<=20)，以空格分隔。

输出

若骑士能走到公主囚禁的位置，则输出骑士走到公主所囚禁方格所需花费的最少时间，否则输出0。

样例输入

```
5 6
00*000
010111
010001
011001
01+001
```

样例输出

```
6
```

思路分析

与上一题思路类似，考虑广度优先搜索策略解题。同样定义了结构体 Pos，来方便存储位置信息与最少时间信息。

这里需要维护的状态有 2 个：1.迷宫地形（考虑到不可重复访问的位置可以用墙代替，所以用 maze 同时记录访问状态和障碍物信息）；2.骑士可能移动的位置，用队列 q 实现。

接下来将所有可能到达的位置加入队列，然后逐个从队列中弹出，判断是否到达终点，并继续将之后可能到达的位置入队，循环直至到达终点或队列为空。过程中重点考虑是否超出迷宫边界、是否遇到墙。

向 4 个方向移动可以用数组 dir 记录相对距离，这样就可以利用一个循环来依次判断 4 个方向的情况。

参考代码

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

//定义结构体 Pos，包含位置坐标 x，y，和到达该位置的最小步数 steps
struct Pos {
    int x;
    int y;
    int steps;
    Pos(int xx, int yy, int ss):x(xx), y(yy), steps(ss) {}
};
```

```
12. 
13. //maze 记录迷宫地形
14. //队列 q 保存待搜索点坐标
15. char maze[1001][1001];
16. queue<Pos> q;
17. //4 个方向移动的坐标变化
18. int dir[4][2] = {1, 0, 0, -1, -1, 0, 0, 1};
19. 
20. int main() {
21.     int m, n;
22.     int x_start, y_start;
23.     int x_end, y_end;
24.     int x_tmp, y_tmp;
25.     cin >> m >> n;
26.     for (int i = 0; i < m; i++)
27.         for (int j = 0; j < n; j++) {
28.             cin >> maze[i][j];
29.             switch(maze[i][j]) {
30.                 case '*': x_start = i; y_start = j; break;
31.                 case '+': x_end = i; y_end = j; break;
32.                 default: break;
33.             }
34.         }
35.     //起点入队，走过的位置标记为"1"，与墙的效果相同，避免重复访问
36.     q.push(Pos(x_start, y_start, 0));
37.     maze[x_start][y_start] = '1';
38.     while(!q.empty()) {
39.         Pos pos = q.front();
40.         //结束条件之一：到达终点
41.         if (pos.x == x_end && pos.y == y_end) {
42.             cout << pos.steps << endl;
43.             break;
44.         }
45.         else {
46.             q.pop();
47.             //向 4 个方向试探移动
48.             for (int i = 0; i < 4; i++) {
49.                 x_tmp = pos.x + dir[i][0];
50.                 y_tmp = pos.y + dir[i][1];
51.                 if (x_tmp >= 0 && x_tmp < m && y_tmp >=0 && y_tmp < n && maze[x_tmp][y_
    tmp] != '1') {
52.                     q.push(Pos(x_tmp, y_tmp, pos.steps + 1));
53.                     maze[x_tmp][y_tmp] = '1';
54.                 }
55.             }
56. 
57.         }
58.     }
59.     //结束条件之二：队列变空，即终点不可达
60.     if(q.empty()) cout << 0 << endl;
61.     return 0;
62. }
```

3.4. 踩方格

总时间限制： 1000ms 内存限制： 65536kB

描述

有一个方格矩阵，矩阵边界在无穷远处。我们做如下假设：
a.每走一步时，只能从当前方格移动一格，走到某个相邻的方格上；
b.走过的格子立即塌陷无法再走第二次；
c.只能向北、东、西三个方向走；
请问：如果允许在方格矩阵上走 n 步，共有多少种不同的方案。2 种走法只要有一步不一样，即被认为是不同的方案。

输入

允许在方格上行走的步数 n(n <= 20)

输出

计算出的方案数量

样例输入

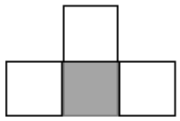
```
2
```

样例输出

```
7
```

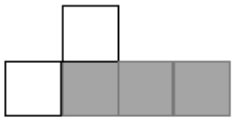
思路分析

在如题假设下，可以发现移动中的状态可以分为这 3 类：



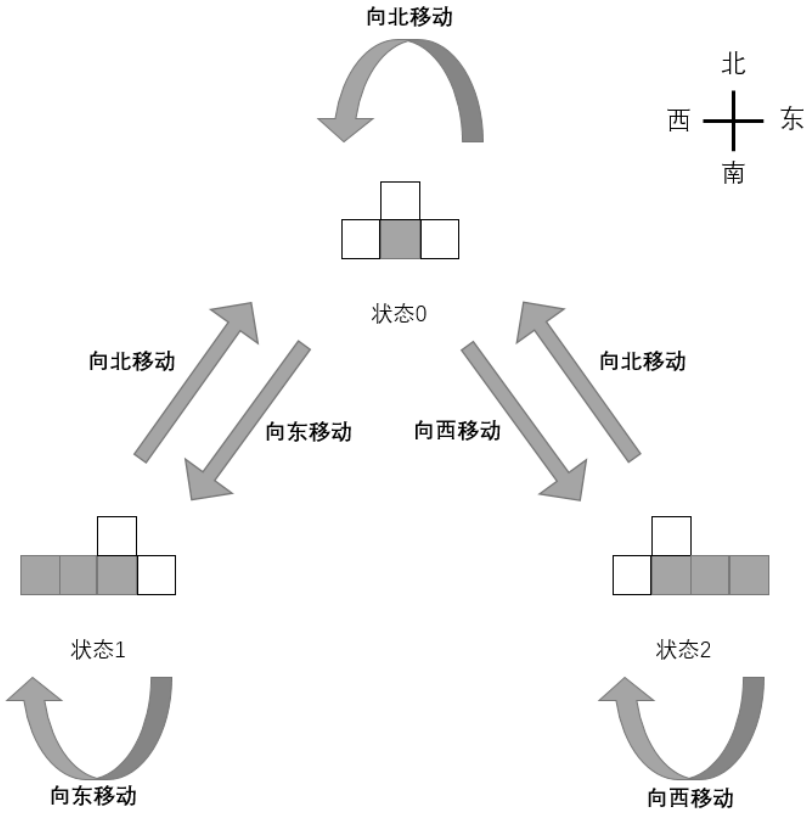
状态0:
连续向北移动若干次，可以向北、东、西三个方向移动

状态1:
连续向东移动n次(n≥1)，只可以向北、东两个方向移动



状态2:
连续向西移动n次(n≥1)，只可以向北、西两个方向移动

这三类的相互转化过程如下：



如果用 $Num_{(0,k)}$ 表示在上面描述的状态 0 下移动 k 步能够得到的方案总数量，那么根据如上的相互转化关系，可以得到如下公式：

$$Num_{(0,k)} = Num_{(0,k-1)} + Num_{(1,k-1)} + Num_{(2,k-1)}$$

$$Num_{(1,k)} = Num_{(0,k-1)} + Num_{(1,k-1)}$$

$$Num_{(2,k)} = Num_{(0,k-1)} + Num_{(2,k-1)}$$

而初始条件易知应为：

$$Num_{(0,1)} = 3$$

$$Num_{(1,1)} = 2$$

$$Num_{(2,1)} = 2$$

依照递推公式即可求得最终结果。

同时，如上递推公式还可以进一步简化：

$$Num_{(0,k)} = 2 * Num_{(0,k-1)} + Num_{(0,k-2)}$$

题目待求结果即是 $Num_{(0,n)}$，按递推公式求解即可。

参考代码

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

int num[3][21];

int main() {
    int N;
    cin >> N;
    num[0][1] = 3;
    num[1][1] = 2;
    num[2][1] = 2;
    for (int i = 2; i <= N; i++) {
        num[0][i] = num[0][i - 1] + num[1][i - 1] + num[2][i - 1];
        num[1][i] = num[0][i - 1] + num[1][i - 1];
        num[2][i] = num[0][i - 1] + num[2][i - 1];
    }
    cout << num[0][N] << endl;
    return 0;
```

```
}
```

简化后：

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

int main() {
    int N;
    cin >> N;
    int num[21] = {1, 3};
    for (int i = 2; i <= N; i++)
        num[i] = 2 * num[i - 1] + num[i - 2];
    cout << num[N] << endl;
    return 0;
}
```

4. 贪心

贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的是在某种意义上的局部最优解。

贪心算法不是对所有问题都能得到整体最优解，关键是贪心策略的选择，选择的贪心策略必须具备无后效性，即某个状态以前的过程不会影响以后的状态，只与当前状态有关。

4.1. 硬币找零

总时间限制:

10000ms

单个测试点时间限制:

1000ms

内存限制:

131072kB

描述

在现实生活中，我们经常遇到硬币找零的问题，例如，在发工资时，财务人员就需要计算最少的找零硬币数，以便他们能从银行拿回最少的硬币数，并保证能用这些硬币发工资。我们应该注意到，人民币的硬币系统是 100，50，20，10，5，2，1，0.5，0.2，0.1，0.05，0.02，0.01 元，采用这些硬币我们可以对任何一个工资数用贪心算法求出其最少硬币数。但不幸的是：我们可能没有这样一种好的硬币系统，因此用贪心算法不能求出最少的硬币数，甚至有些金钱总数还不能用这些硬币找零。例如，如果硬币系统是 40，30，25 元，那么 37 元就不能用这些硬币找零；95 元的最少找零硬币数是 3。又如，硬币系统是 10，7，5，1 元，那么 12 元用贪心法得到的硬币数为 3，而最少硬币数是 2。

你的任务就是：对于任意的硬币系统和一个金钱数，请你编程求出最少的找零硬币数；如果不能用这些硬币找零，请给出一种找零方法，使剩下的钱最少。

输入

第 1 行，为 N 和 T，其中 1≤N≤50 为硬币系统中不同硬币数；1≤T≤100000 为需要用硬币找零的总数。
第 2 行为 N 个数值不大于 65535 的正整数，它们是硬币系统中各硬币的面值。

输出

如 T 能被硬币系统中的硬币找零，请输出最少的找零硬币数。
如 T 不能被硬币系统中的硬币找零，请输出剩下钱数最少的找零方案中的最少硬币数。

样例输入

```
4 12
10 7 5 1
```

样例输出

```
2
```

思路分析

- 本题看似是贪心法，实际上却不能用贪心法求解。比如说 12，那么首先由贪心法先找 10，然后从 10 的基础上找两个 1，这样的话最终的结果是 3，而不是答案的 2.
- 所以实际上这道题应该用动态规划来求解，设置 dp[i]表示的是 i 这个钱最少可以用多少硬币找零，那么应该满足如下的递推关系：

  dp[j] = min(dp[j] , dp[j - a[i]] + 1);

  这是解决问题的核心。

代码技巧

- 注意初始化的时候，所有的 dp[j]都需要初始化成一个极大值，因为是要求最小。但是 dp[0]需要初始化成 0，这是一个已知条件，0 块钱不需要任何找零。
- 注意题里面的要求：“如 T 不能被硬币系统中的硬币找零，请输出剩下钱数最少的找零方案中的最少硬币数”。

参考代码

```
1. #include <iostream>
2. #include <string.h>
```

```cpp
#include <algorithm>
using namespace std;
int a[105], d[100005];
int main()
{

    int n, t;
    int coin[51];//存放硬币系统
    int dp[100001];//动态规划，dp[i]表示 i 块钱需要找零所用的最少的硬币的数量
    cin >> n >> t;

    for (int i = 0; i < n; i++){
        cin >> coin[i]; //输入硬币系统的每个硬币的数值
    }

    memset(dp, 0x3f, sizeof(dp));//初始化，将 dp 中除了 0 以外的元素都初始化成最大值
    dp[0] = 0;//这是因为 0 不需要找零，作为初始化条件

    for (int i = 0; i<n; i++)
        for (int j = coin[i]; j <= t; j++)
            dp[j] = min(dp[j], dp[j - coin[i]] + 1);  //求最小硬币个数

    //从 i=t 向 i=0 寻找的原因，是考虑到如果不能被硬币系统中的硬币找零，请输出剩下钱数最少的找零方案中的最少硬币数这种情况
    //如果能完全找开，那么 i=t 的时候就直接符合条件；否则应该从 i=t-1,i=t-2 往下找到距离 i=t 最近的
    for (int i = t; i >= 0; i--)
    {
        if (dp[i] <= 100000)
        {
            printf("%d\n", dp[i]);
            break;
        }
    }
    system("pause");
    return 0;
}
```

4.2. Radar Installation

总时间限制:

1000ms

内存限制:

65536kB

描述

Assume the coasting is an infinite straight line. Land is in one side of coasting, sea in the other. Each small island is a point locating in the sea side. And any radar installation, locating on the coasting, can only cover d distance, so an island in the sea can be covered by a radius installation, if the distance between them is at most d.

We use Cartesian coordinate system, defining the coasting is the x-axis. The sea side is above x-axis, and the land side below. Given the position of each island in the sea, and given the distance of the coverage of the radar installation, your task is to write a program to find the minimal number of radar installations to cover all the islands. Note that the position of an island is represented by its x-y coordinates.

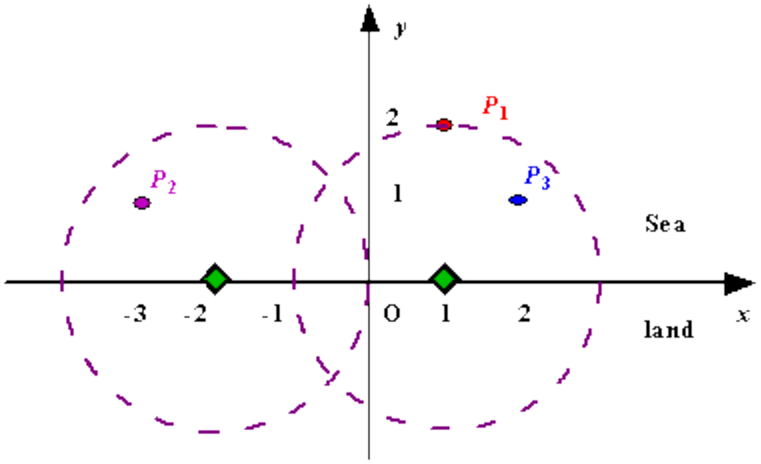


Figure A Sample Input of Radar Installations

输入

The input consists of several test cases. The first line of each case contains two integers n (1<=n<=1000) and d, where n is the number of islands in the sea and d is the distance of coverage of the radar installation. This is followed by n lines each containing two integers representing the coordinate of the position of each island. Then a blank line follows to separate the cases.

The input is terminated by a line containing pair of zeros

输出

For each test case output one line consisting of the test case number followed by the minimal number of radar installations needed. "-1" installation means no solution for that case.

样例输入

```
3 2
1 2
-3 1
2 1

1 2
```

```
0 2

0 0
```

样例输出

```
Case 1: 2
Case 2: 1
```

思路分析

以每一座岛屿为圆心，雷达范围为半径作圆，记录下与 x 轴的左右交点。如果与 x 轴没交点，则直接退出输出“-1”。以左交点为关键字进行排序，从左到右进行贪心。容易知道，离每一个雷达最远的那一座岛与雷达相距恰巧为半径的时候，可以得到最优解。假设上一个雷达与第 before 座岛相距为半径大小，对于当前的岛屿 i：

如果 before 岛的右交点在 i 岛左交点的左侧，此时必然需要一个新的雷达，将当前 before 暂定为 i，雷达数加一；

如果 before 岛的右交点在 i 岛右交点的右侧，为了让雷达举例尽可能地广，将雷达移至当前岛屿与 x 轴的交点上，将当前 before 暂定为 i，雷达数不变。

参考代码

```
#include<iostream>
#include<cstdio>
#include<cmath>
#include<algorithm>
using namespace std;
double sqrt(double n);
struct rec
{
    double left,right;
    bool operator < (const rec& x) const
    {
        return left<x.left;
    }
};

const int MAXN=1000+5;
int n,d;
rec inter[MAXN];

int calculate()
{
    sort(inter,inter+n);
    int ans=1;
    int before=0;
    for (int i=1;i<n;i++)
    {
```

```
        if (inter[before].right<inter[i].left)
        {
            ans++;
            before=i;
        }
        else if (inter[before].right>=inter[i].right)
        {
            before=i;
        }
    }
    return ans;
}

int main()
{
    int t=0;
    while (scanf("%d%d",&n,&d))
    {
        if (n==d && d==0) break;
        t++;
        int f=1;
        for (int i=0;i<n;i++)
        {
            double x,y;
            cin>>x>>y;
            if (d<y)
            {
                f=0;
            }
            else
            {
                inter[i].left=x*1.0-sqrt(d*d-y*y);
                inter[i].right=x*1.0+sqrt(d*d-y*y);
            }
        }
        cout<<"Case "<<t<<": ";
        if (f) cout<<calculate()<<endl;else cout<<-1<<endl;

    }
    return 0;
}
```

5. 动态规划

要使用动态规划求解问题，需要问题具有两个最基本的属性，即“最优子结构”与“边界”。最优子结构指的是当前问题可由与当前问题结构相同的子问题计算得出，边界指的是问题最终会分解为确定的最小子问题，而不是子子孙孙无穷尽也。

类比高中时学习的数学归纳法，动态规划问题需要有确定的起始状态 F(0)，也需要有确定的推导方程，即 F(n)=G(F(n-1))。综上，解决动态规划是一个通过分解问题，确定问题的初始状态，并通过状态之间的递推关系得到递推方程，最后依据递推方程得到当前问题的解的过程。

5.1. 爬楼梯

总时间限制： 1000ms
内存限制： 65535kB

描述

这里共有 20 阶楼梯，rj 每次只能走一阶或两阶，请问 rj 共有多少种方法走完此楼梯?

输入

无

输出

rj 共有多少种方法走完此楼梯

样例输入

```
无
```

样例输出

```
无
```

思路分析

依据动态规划问题的定义，我们先考虑当前问题是否能被相同结构的子问题表出：

使用函数 F(i)表示 rj 到第 i 级台阶的方法数量，又考虑到 rj 一次可以跨一级或两级台阶，则 F(i)=F(i-1)+F(i-2)，可以通过相同结构的子问题表出。

接下来考虑当前问题是否有边界：

考虑到台阶数只有非负数，0 级台阶表示 rj 站在地上，所以显然有边界 F(0)=1。

综上，可以得到如下的递推方程：

$$\mathrm{F(i)} = \begin{cases} 1 & i = 0 \\ F(i-1) & i = 1 \\ F(i-1) + F(i-2) & i \geq 2 \end{cases}$$

并在 i=20 时取得题目所需要的结果。

代码技巧

1. 使用原生的 C++数组很难实现动态伸缩的能力，推荐使用 vector

2. N=20，数组中有 21 个元素，注意不要越界

参考代码

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

int main() {
  vector<int> result(21,0);
  result[0] = 1;
  for(int i = 0;i < 21;i ++) {
    if(i + 1 < 21)
      result[i + 1] += result[i];
    if(i + 2 < 21)
      result[i + 2] += result[i];
  }
  cout<<result[20];
  return 0;
}
```

5.2. 最长上升子序列

总时间限制： 2000ms
内存限制： 65536kB

描述

一个数的序列 b_i，当 $b_1 < b_2 < ... < b_S$ 的时候，我们称这个序列是上升的。对于给定的一个序列(a_1, a_2, ..., a_N)，我们可以得到一些上升的子序列(a_{i1}, a_{i2}, ..., a_{iK})，这里 1 <= i_1 < i_2 < ... < i_K <= N。比如，对于序列(1, 7, 3, 5, 9, 4, 8)，有它的一些上升子序列，如(1, 7), (3, 4, 8)等等。这些子序列中最长的长度是 4，比如子序列(1, 3, 5, 8).

你的任务，就是对于给定的序列，求出最长上升子序列的长度。

输入

输入的第一行是序列的长度 N (1 <= N <= 1000)。第二行给出序列中的 N 个整数，这些整数的取值范围都在 0 到 10000。

输出

最长上升子序列的长度。

样例输入

```
7
1 7 3 5 9 4 8
```

样例输出

```
4
```

思路分析

首先是问题分解，要求最长的上升子序列长度，可以先分别求出以每一个数结尾的上升子序列的长度，最后取所有结果的最大值。要求以第 i 个数结尾的最长上升子序列，可以先找出前 i-1 个数中比第 i 个数小的数 a0,a1,a2…，并求出以 a0,a1,a2…结尾的所有子序列中最长的上升子序列长度 L，则以 i 结尾的最长上升子序列长度即为 L+1。综上，该问题可以被分解为同等结构的子问题求解。

然后寻找问题边界，如果在第 i 个数之前找不到比它小的数，则说明这个数是上升子序列的第一个数，以该数结尾的最长上升子序列长度为 1，该问题有确定的边界。

接下来得到该问题的转移方程，令 F(i)为以第 i 个数为结尾的最长上升子序列长度，则

$$F(i)=\begin{cases}1 & \text{没有比第 i 个数小的数}\\ \max\{F(a_0,a_1,\dots)\}+1 & \text{能找到比第 i 个数小的数}\end{cases}$$

最后，再从所有的 F(i)中求出最大值即可。

代码技巧

由于该题的输入是无序的，所以每次得到 F(i)的值都需要遍历之前所有的结果，因此，要输入 i 个数，每个输入都需要遍历前 i-1 个结果，计算复杂度可估计为$\sum_{i=1}^{N} i-1=\frac{(N-1)(N-2)}{2}\approx N^2$，当输入数据非常多的时候可能会超时，所以可以考虑一些优化。

由于大部分的时间消耗都在查找最大值上，所以考虑如何减少在查找最大值的复杂度开销。最容易想到的就是，计算出的每一个 F(i)对于后续的计算来说是否都是真的有用的呢？如果部分 F(i)存在与否都不会对后续结果产生影响，那直接将其从记录中删掉就可以节约很多的时间。考虑这样情况，在序列[1,4,2,6,9]中，F(4)和 F(2)均为 2，且 4 比 2 大。删除 F(4)后，对于以比 4 大的值 X 结尾的序列来说，[1,4,X]和[1,2,X]在长度上并没有区别，因此，将 F(4)删除后并不会对后续的计算产生影响。

能得到这样一个结论，在相同长度的最长上升子序列中，我们只需要留下结尾的数最小的结果即可，删除其他结果并不会对后续的计算结果产生影响。因此，在表示记录时，可以使用子序列的长度 L 作为数组下标，用数组的值记录所有长度为 L 的最长上升子序列中最小的结尾数字。这样，每种长度的最长上升子序列只会被保存一份，且只保存结尾数字，极大地减少了查找最大值过程中的时间消耗。

此外，这种以数组长度为下标，结尾数字为值的表示方式还能保证数组的有序性，进而使用二分查找法改善查询性能，感兴趣的同学可以自己尝试一下。

参考代码

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> result(n + 1, -1);
  int maxLen = 0;
  for(int i = 0;i < n;i ++) {
    int val, tLen;
    cin >> val;
    tLen = 0;
    for(int j = maxLen;j > 0 && tLen == 0;j --) {
      if(val > result[j])
        tLen = j;
    }
    if(result[tLen + 1] > val || result[tLen + 1] < 0) {
      result[tLen + 1] = val;
      maxLen = maxLen < tLen + 1 ? tLen + 1: maxLen;
    }
  }
  cout<<maxLen;
  return 0;
}
```

5.3. 神奇的口袋

总时间限制: 10000ms
内存限制: 65536kB

描述

有一个神奇的口袋，总的容积是 40，用这个口袋可以变出一些物品，这些物品的总体积必须是 40。John 现在有 n 个想要得到的物品，每个物品的体积分别是 a_1，a_2……a_n。John 可以从这些物品中选择一些，如果选出的物体的总体积是 40，那么利用这个神奇的口袋，John 就可以得到这些物品。现在的问题是，John 有多少种不同的选择物品的方式。

输入

输入的第一行是正整数 n (1 <= n <= 20)，表示不同的物品的数目。接下来的 n 行，每行有一个 1 到 40 之间的正整数，分别给出 a_1，a_2……a_n 的值。

输出

输出不同的选择物品的方式的数目。

样例输入

```
3
20
20
```

```
20
```

样例输出

```
3
```

思路分析

对于这类给定某种限制，求满足条件的结果种类的问题统称为背包问题，感兴趣的同学可以自行搜索《背包九讲》，学习这一类动态规划问题的精髓。

首先是问题分解，放入第 i 个物品，使得总体积为 Vi 后，对于总体积为 V 的情况，满足条件的结果种类可以用放入第 i-1 个物品，总体积为 V 的结果数量加上放入第 i-1 个物品，总体积为 V-Vi 的结果数量即可。

然后是问题的边界，显然，什么物品都不放，且总体积为 0 的放物品方法只有 1 种，问题有确定的边界。

综上，对于放入第 i 个物品后，总体积为 V 的情况可以得到方程

$$F_i(\mathrm{V}) = \begin{cases} 1 & V = 0 \\ F_{i-1}(V - V_i) + F_{i-1}(V) & V \geq V_i \\ 0 & Other \end{cases}$$

最终，$F_N(40)$即为我们需要的结果

代码技巧

由于涉及到两种下标 i 与 V，所以可以考虑使用二维数组，也可使用一维数组保存，但需要注意更新状态的流程应为从体积大的情况开始，避免计算出错。

参考代码

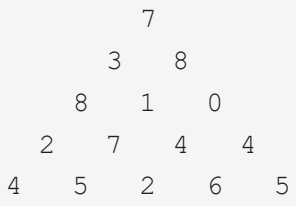
```cpp
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> result(n + 1, -1);
  int maxLen = 0;
  for(int i = 0;i < n;i ++) {
    int val, tLen;
    cin >> val;
    tLen = 0;
    for(int j = maxLen;j > 0 && tLen == 0;j --) {
      if(val > result[j])
        tLen = j;
    }
    if(result[tLen + 1] > val || result[tLen + 1] < 0) {
      result[tLen + 1] = val;
      maxLen = maxLen < tLen + 1 ? tLen + 1: maxLen;
```

```
    }
  }
  cout<<maxLen;
  return 0;
}
```

5.4. 数字三角形

总时间限制: 1000ms
内存限制: 65536kB

描述



(图 1)

图 1 给出了一个数字三角形。从三角形的顶部到底部有很多条不同的路径。对于每条路径，把路径上面的数加起来可以得到一个和，你的任务就是找到最大的和。

注意：路径上的每一步只能从一个数走到下一层上和它最近的左边的那个数或者右边的那个数。

输入

输入的是一行是一个整数 N (1 < N <= 100)，给出三角形的行数。下面的 N 行给出数字三角形。数字三角形上的数的范围都在 0 和 100 之间。

输出

输出最大的和。

样例输入

```
5
7
3 8
8 1 0
2 7 4 4
```

```
4 5 2 6 5
```

样例输出

```
30
```

思路分析

首先为问题分解，定义V(i,j)为第i行第j个数的值，F(i,j)为从顶端到第i行第j个数的最大值，如果这个数在三角形的左右边上，则F(i,j)可以直接通过其右肩或者左肩上的F(i,j)或F(i,j-1)加上V(i,j)计算出来。若这个数不在边上，则F(i,j)可以通过这个数肩上的F(i-1,j-1)和F(i-1,j)加上V(i,j)计算出来。因此，该问题可以被分解为同种结构的子问题。

显然，该问题有边界，即对于顶端的F(0,0)其结果即为V(0,0)。

综上，可得该问题的转移方程

$$\mathrm{F(i,j)}=\begin{cases} F(i-1,j)+V(i,j) & j=0 \\ \max\big(F(i-1,j),F(i-1,j-1)\big)+V(i,j) & j<i-1 \\ F(i-1,j-1)+V(i,j) & j=i-1 \end{cases}$$

最终结果为max(F(N,0),F(N,1)⋯)。

代码技巧

注意不要越界即可

参考代码

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> result(n + 1, 0);
  for(int i = 0;i < n;i ++) {
    vector<int> tResult(n + 1, 0);
    for(int j = 0;j <= i;j ++) {
      int temp;
      cin >> temp;
      tResult[j] = result[j] + temp;
      if(j > 0) {
        temp += result[j - 1];
        tResult[j] = tResult[j] < temp?temp:tResult[j];
      }
    }
    result = tResult;
  }
  for(int i = 1;i < n;i ++)
```

```
    result[0] = result[0] < result[i] ? result[i] : result[0];
  cout<<result[0];
  return 0;
}
```

5.5. 宠物小精灵之收服

总时间限制: 1000ms
内存限制: 65536kB

描述

宠物小精灵是一部讲述小智和他的搭档皮卡丘一起冒险的故事。

一天，小智和皮卡丘来到了小精灵狩猎场，里面有很多珍贵的野生宠物小精灵。小智也想收服其中的一些小精灵。然而，野生的小精灵并不那么容易被收服。对于每一个野生小精灵而言，小智可能需要使用很多个精灵球才能收服它，而在收服过程中，野生小精灵也会对皮卡丘造成一定的伤害（从而减少皮卡丘的体力）。当皮卡丘的体力小于等于 0 时，小智就必须结束狩猎（因为他需要给皮卡丘疗伤），而使得皮卡丘体力小于等于 0 的野生小精灵也不会被小智收服。当小智的精灵球用完时，狩猎也宣告结束。

我们假设小智遇到野生小精灵时有两个选择：收服它，或者离开它。如果小智选择了收服，那么一定会扔出能够收服该小精灵的精灵球，而皮卡丘也一定会受到相应的伤害；如果选择离开它，那么小智不会损失精灵球，皮卡丘也不会损失体力。

小智的目标有两个：主要目标是收服尽可能多的野生小精灵；如果可以收服的小精灵数量一样，小智希望皮卡丘受到的伤害越小（剩余体力越大），因为他们还要继续冒险。

现在已知小智的精灵球数量和皮卡丘的初始体力，已知每一个小精灵需要的用于收服的精灵球数目和它在被收服过程中会对皮卡丘造成的伤害数目。请问，小智该如何选择收服哪些小精灵以达到他的目标呢？

输入

输入数据的第一行包含三个整数：N(0 < N < 1000)，M(0 < M < 500)，K(0 < K < 100)，分别代表小智的精灵球数量、皮卡丘初始的体力值、野生小精灵的数量。
之后的 K 行，每一行代表一个野生小精灵，包括两个整数：收服该小精灵需要的精灵球的数量，以及收服过程中对皮卡丘造成的伤害。

输出

输出为一行，包含两个整数：C，R，分别表示最多收服 C 个小精灵，以及收服 C 个小精灵时皮卡丘的剩余体力值最多为 R。

样例输入

样例输入 1：

```
10 100 5
7 10
2 40
2 50
1 20
4 20
```

样例输入 2：

```
10 100 5
8 110
12 10
20 10
5 200
1 110
```

样例输出

样例输出 1：

```
3 30
```

样例输出 2：

```
0 100
```

思路分析

这是一个二维背包问题，在神奇的口袋的一维背包问题的基础上，加上了另外一个维度的条件限制。

首先是问题分解，定义$F_i(m,n)$为遇到第 i 个小精灵后，智爷消耗 m 个精灵球，皮神消耗 n 的体力后，可以捕捉到的精灵个数。则$F_i(m,n)$可通过$F_{i-1}(m-m_i,n-n_i)+1$求出，但前提是前一状态能捕捉到小精灵，否则不能直接加一。

之后是问题的边界，显然，智爷不使用精灵球，皮神不消耗体力是问题的边界，此时能捕捉到 0 个精灵。

综上，该问题的递推方程为

$$F_i(m,n)=\begin{cases}0 & , \quad m=0,n=0\\ F_{i-1}(m-m_i,n-n_i)+1 & , \quad F_{i-1}(m-m_i,n-n_i)\geq 0\\ -1 & , \quad Other\end{cases}$$

最终，找到所有结果里能抓到最多的口袋妖怪，且皮神体力消耗最小的情况，输出相应结果即可。

代码技巧

如果将 i，m，n 都作为数组下标，会得到一个三维的矩阵，可以将所有结果都存在一个二维矩阵中，按照精灵球使用数目从大到小与皮神体力消耗从大到小的顺序更新矩阵里的记录，即可极大地提高空间理用率。

参考代码

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int N,M,K;
  cin>>N>>M>>K;
  vector<vector<int>> record(N+1, vector<int>(M+1, -1));
  record[0][0] = 0;
  while(K --) {
    int nCost, mCost;
    cin >>nCost>>mCost;
    for(int i = N;i >= nCost;i --) {
      for(int j = M;j >= mCost;j --) {
        if(record[i-nCost][j-mCost] >= 0)
          record[i][j] = record[i][j] < record[i - nCost][j - mCost] + 1 ? record[i - nCost][j - mCost] + 1:record[i][j];
      }
    }
  }
  int catchNum = 0;
  int usedHP = 0;
  for(int i = 0;i <= N;i ++) {
    for(int j = 0;j <= M;j ++) {
      if(record[i][j] > catchNum || (record[i][j] == catchNum && usedHP > j)) {
```

```
        catchNum = record[i][j];
        usedHP = j;
      }
    }
  }
  cout<<catchNum << ' ' << M - usedHP;
  return 0;
}
```