注

1.排序使用sort包

```
//基本类型排序
func Ints(x []int)
func Float64s(x []float64)
func Strings(x []string)
//自定义比较器 使用sort.Slice()函数排序任意类型的切片
arr := []int{0, 3, 5, 6, 7}
sort.Slice(arr, func(i, j int) bool {
    return arr[i] > arr[j]
})
```

2.rand获取随机数

math/rand

```
//随机int值
a := rand.Int()
//0-n-1之间的随机值
a := rand.Intn(n)
```

数据结构

稀疏数组

当一个数组中大部分元素为 0，或者为同一个值的数组时，可以使用稀疏数组来保存该数组。

```
type node struct {
    r, c, v int
}

var arr []node

func main() {
    var g [111][111]int
    g[23][32] = 1
    for i := 0; i < 111; i++ {
        for j := 0; j < 111; j++ {
```

```
            if g[i][j] == 1 {
                arr = append(arr, node{i, j, g[i][j]})
            }
        }
    }
}
```

排序

1.选择排序

```
func selectSort(arr []int) {
    for i := 0; i < len(arr); i++ {
        min := i
        for j := i; j < len(arr); j++ {
            if arr[j] < arr[i] {
                min = j
            }
        }
        arr[i], arr[min] = arr[min], arr[i]
    }
}
```

2.冒泡排序

```
//n次迭代，每次迭代确定该位置的最小值
func bubbleSort(arr []int) {
    for i := 0; i < len(arr); i++ {
        for j := 1; j < len(arr); j++ {
            if arr[j] < arr[j-1] {
                arr[j], arr[j-1] = arr[j-1], arr[j]
            }
        }
    }
}
```

3.快排序

```
var a = []int{3, 2, 1, 0}

func quickSort(l, r int) {
    if l ⩾ r {
        return
    }
    var i, j, x = l, r, a[l]
    for i < j {
        for a[j] ⩾ x && i < j {
            j--
        }
        for a[i] ⩽ x && i < j {
            i++
        }
        t := a[i]
        a[i] = a[j]
        a[j] = t
    }
    //递归排左右序列
    t := a[i]
    a[i] = a[l]
    a[l] = t
    quickSort(l, i-1)
    quickSort(i+1, r)
}
```

二分

原理

将问题分为两种情况，要么满足左边，要么满足右边。问题答案就是找两个边界，注意找左边界时，找中点要加1，防止l=mid时无限循环。

```
int l = 0, r = n - 1;
//找左边界
for l < r {
    var mid = (l+r+1)/2;
    //是否满足左边
    if check(a[mid]) l = mid;
    else r = mid-1;
```

```
}
//找右边界
for l < r {
    int mid = (l+r)/2;
    //是否满足右边
    if check(a[mid]) r = mid;
    else l = mid + 1;
}
```

例题

1.790数的三次方根

给定一个浮点数 n，求它的三次方根。

```
func main() {
    var a float64
    fmt.Scan(&a)
    var l, r float64 = -10000, 10000
    for r-l > 1e-8 {
        mid := (l + r) / 2
        if mid*mid*mid ⩾ a {
            r = mid
        } else {
            l = mid
        }
    }
    fmt.Println(l)
}
```

2.789数的范围

给定一个按升序排列的数组和一个数，求这个数的起始坐标和终点坐标

输入样例:

6

1 2 2 3 3 4

3

输出样例:

3 4

```
const int N = 100;
int a[N], n;
```

```cpp
//找左边界（终点）
int getleft(int x) {
    int l = 0, r = n-1;
    while(l < r) {
        int mid = (l+r+1)/2;
        if(a[mid] ⩽ x) l = mid;
        else r = mid - 1;
    }
    return l;
}
//找右边界（起始）
int getright(int x) {
    int l = 0, r = n-1;
    while(l < r) {
        int mid = (l+r)/2;
        if(a[mid] ⩾ x) r = mid;
        else l = mid + 1;
    }
    return l;
}
int main() {
    int x;
    cin >> n;
    for(int i = 0; i < n; i++) cin >> a[i];
    cin >> x;
    cout << getright(x) << ' ';
    cout << getleft(x) << endl;
}
```

前缀和与差分

原理

一维前缀和：在O(1)时间复杂度求[l,r]之和

- 前缀和：$s_i = a_i + s_{i-1}$
- 区间和：$\mathbf{s[l,r] = S_r - S_{l-1}}$

二维前缀和：在O(1)时间复杂度求(x1,y1)到(x2,y2)的子矩阵和

- 前缀和：$s_{i,j} = a_{i,j} + s_{i-1,j} + s_{i,j-1} - s_{i-1,j-1}$
- $\mathbf{s = S_{x2,y2} - S_{x2,y1-1} - S_{x1-1,y2} + S_{x1-1,y1-1}}$

一维差分：前缀和数组s对应差分数组a，在O(1)时间复杂度将s数组[l,r]元素全加c，输出之后的数组。
a[l]+c,a[r+1]-c

二维差分：O(1)时间将某子矩阵(x1,y1)到(x2,y2)所有元素加c
a[x1][y1]+c, a[x1][y2+1]-c, a[x2+1][y1]-c, a[x2+1][y2+1]+c

例题

1.acwing795 前缀和

输入一个长度为n的整数序列。

接下来再输入m个询问，每个询问输入一对l，r。

对于每个询问，输出原序列中从第l个数到第r个数的和。

输入格式
第一行包含两个整数n和m。

第二行包含n个整数，表示整数数列。

接下来m行，每行包含两个整数l和r，表示一个询问的区间范围。

输出格式
共m行，每行输出一个询问的结果。

数据范围
1≤l≤r≤n,
1≤n,m≤100000,
-1000≤数列中元素的值≤1000
输入样例：
```
5 3
2 1 3 6 4
1 2
1 3
2 4
```
输出样例：
```
3
6
10
```

```
const int N = 1e5+10;
int a[N], s[N], n, m;

int main() {
    int l, r;
    cin >> n >> m;
```

```
    s[0] = 0;
    for(int i = 1; i ⩽ n; i++) {
        cin >> a[i];
        s[i] = s[i-1] + a[i];
    }
    while(m--) {
        cin >> l >> r;
        int ans = 0;
        ans = s[r] - s[l-1];
        cout << ans << endl;
    }
}
```

2.acwing796 子矩阵的和

输入一个n行m列的整数矩阵，再输入q个询问，每个询问包含四个整数x1，y1，x2，y2，表示一个子矩阵的左上角坐标和右下角坐标。

对于每个询问输出子矩阵中所有数的和。

输入格式
第一行包含三个整数n，m，q。
接下来n行，每行包含m个整数，表示整数矩阵。
接下来q行，每行包含四个整数x1，y1，x2，y2，表示一组询问。

输出格式
共q行，每行输出一个询问的结果。

数据范围
1 ≤ n ， m ≤ 1000 ， 1≤n，m≤1000，1≤n，m≤1000，
1 ≤ q ≤ 200000 ， 1≤q≤200000，1≤q≤200000，
1 ≤ x 1 ≤ x 2 ≤ n ， 1≤x1≤x2≤n，1≤x1≤x2≤n，
1 ≤ y 1 ≤ y 2 ≤ m ， 1≤y1≤y2≤m，1≤y1≤y2≤m，

输入

```
3 4 3

1 7 2 4
3 6 2 8
2 1 2 3

1 1 2 2
2 1 3 4
1 3 3 4
```

输出
17
27

```
const int N = 1e3+10;
int a[N][N], s[N][N], n, m, q;

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

    for(int i = 0; i <= n; i++) {
        for(int j = 0; j <= m; j++) {
            if(i == 0 || j == 0) {
                s[i][j] = 0;
                continue;
            }
            cin >> a[i][j];
            s[i][j] = a[i][j] + s[i-1][j] + s[i][j-1] - s[i-1]
[j-1];
        }
    }

    while(q--) {
        int x1, x2, y1, y2;
        cin >> x1 >> y1 >> x2 >> y2;
        int ans = 0;
        ans = s[x2][y2] - s[x2][y1-1] - s[x1-1][y2] + s[x1-1]
[y1-1];
        cout << ans << endl;
    }
}
```

3.acwing797 差分

输入一个长度为n的整数序列。

接下来输入m个操作，每个操作包含三个整数l,r,c 表示将序列中[l,r]之间的每个数加上c

请你输出进行完所有操作后的序列。

输入
第一行包含两个整数n和m。

第二行包含n个整数，表示整数序列。

接下来 m 行，每行包含三个整数 l，r，c，表示一个操作。

1≤n,m≤100000,
1≤l≤r≤n,
-1000≤c≤1000,
-1000≤整数序列中元素的值≤1000

输出
共一行，包含n个整数，表示最终序列。

样例输入

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

样例输出

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

```cpp
const int N = 1e5+10;
int n, m, a[N], s[N];

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

    s[0] = 0;
    for(int i = 1; i <= n; i++) {
        cin >> s[i];
        //求差分数组
        a[i] = s[i] - s[i-1];
    }
    //m次操作O(1)
    while(m--) {
        int l, r, c;
        cin >> l >> r >> c;
        a[l] += c;
        a[r+1] -= c;
    }
    for(int i = 1; i <= n; i++) {
        s[i] = a[i] + s[i-1];
        cout << s[i] << ' ';
    }
}
```

4.acwing798 差分矩阵

输入一个 n 行 m列的整数矩阵，再输入 q 个操作，每个操作包含五个整数 x1,y1,x2,y2,c，其中（x1,y1）和（x2,y2）表示一个子矩阵的左上角坐标和右下角坐标。

每个操作都要将选中的子矩阵中的每个元素的值加上 c。

请你将进行完所有操作后的矩阵输出。

输入格式

第一行包含整数 n,m,q。

接下来 n 行，每行包含 m 个整数，表示整数矩阵。

接下来 q 行，每行包含 5 个整数 x1,y1,x2,y2,c，表示一个操作。

输出格式

共 n 行，每行 m 个整数，表示所有操作进行完毕后的最终矩阵。

数据范围

1≤n,m≤1000,
1≤q≤100000,
1≤x1≤x2≤n,
1≤y1≤y2≤m,
-1000≤c≤1000,
-1000≤矩阵内元素的值≤1000

输入样例：

```
3 4 3
1 2 2 1
3 2 2 1
1 1 1 1
1 1 2 2 1
1 3 2 3 2
3 1 3 4 1
```

输出样例：

```
2 3 4 1
4 3 4 1
2 2 2 2
```

```
const int N = 1e3+10;
int a[N][N], s[N][N], n, m, q;
```

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

    for(int i = 0; i ⩽ n; i++) {
        for(int j = 0; j ⩽ m; j++) {
            if(i == 0 || j == 0) {
                s[i][j] = 0;
                continue;
            }
            cin >> s[i][j];
            a[i][j] = s[i][j] - s[i-1][j] - s[i][j-1] + s[i-1]
[j-1];
        }
    }
    //q次操作O(1)
    while(q--) {
        int x1, x2, y1, y2, c;
        cin >> x1 >> y1 >> x2 >> y2 >> c;
        a[x1][y1] += c;
        a[x1][y2+1] -= c;
        a[x2+1][y1] -= c;
        a[x2+1][y2+1] += c;
    }
    //重新计算前缀和
    for(int i = 1; i ⩽ n; i++) {
        for(int j = 1; j ⩽ m; j++) {
            s[i][j] = a[i][j] + s[i-1][j] + s[i][j-1] - s[i-1]
[j-1];
            cout << s[i][j] << ' ';
        }
        cout << endl;
    }
}
```

双指针

```
for(int i = 0, j = 0; i < n; i++) {
    if(i < j && check(i,j)) {
        j++
    }
}
```

1.acwing799 最长连续不重复子序列

给定一个长度为 n 的整数序列，请找出最长的不包含重复的数的连续区间，输出它的长度。

输入格式

第一行包含整数 n。

第二行包含 n 个整数（均在 0~105 范围内），表示整数序列。

输出格式

共一行，包含一个整数，表示最长的不包含重复的数的连续区间的长度。

数据范围

1≤n≤105

输入样例：

```
5
1 2 2 3 5
```

输出样例：

```
3
```

```
const int N = 200;
int n, a[N], s[N];

int main() {
    cin >> n;
    for(int i = 0; i < n; i++) {
        cin >> a[i];
    }
    int res = 0;
    for(int i = 0, j = 0; i < n; i++) {
        s[a[i]]++;
        while(s[a[i]] > 1) {
            s[a[j]]--;
            j++;
```

```
        }
        res=max(res,i-j+1);
    }
    cout<<res<<endl;
}
```

2.判断子序列

判断s是否为t的子序列

```
func isSubsequence(s string, t string) bool {
    n, m := len(t), len(s)
    i, j := 0, 0
    for i < m && j < n{
        if s[i] != t[j] {
            j++
        } else {
            i, j = i+1, j+1
        }
    }
    return i==m
}
```

2.两数之和2

给一个数组arr，和一个数target。找到两个数满足arr[i]+arr[j]=target（i≠j）

```
func twoSum(numbers []int, target int) []int {
    n := len(numbers)
    for i, j := 0, n-1; i < j;  {
        if numbers[i] + numbers[j] > target {
            j--
        } else if numbers[i] + numbers[j] < target {
            i++
        } else {
            return []int{i+1, j+1}
        }
    }
    return []int{-1,-1}
}
```

3.盛最多水的容器

经典贪心

给定一个长度为 `n` 的整数数组 `height` ，返回容器可以储存的最大水量。

使用双指针分别指向数组首尾，计算容量，如果height[i]<height[j]，左指针右移，否则右指针左移。

```
func maxArea(height []int) int {
    n , res := len(height), 0
    for i, j := 0, n-1; i < j; {
        if height[i] < height[j] {
            res = max(res, (j-i)*height[i])
            i++
        } else {
            res = max(res, (j-i)*height[j])
            j--
        }
    }
    return res
}
```

4.三数之和

给你一个整数数组 `nums` ，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足 `i ≠ j`、`i ≠ k` 且 `j ≠ k` ，同时还满足 `nums[i] + nums[j] + nums[k] == 0` 。请你返回所有和为 `0` **且不重复**的三元组。

O(n^2)，首先将数组从小到大排序O(nlogn)，然后从小到大枚举第一个数，用双指针寻找另外两个数O(n)。

本题难点在于去重。枚举k时，遇重复的就continue。i,j变化时也不能变化到和前一个数一样的数。

```
func threeSum(nums []int) [][]int {
    sort.Ints(nums)
    res := [][]int{}
    n := len(nums)
    for k := 0; k < n-2; k++ {
        if nums[k] > 0 {
            break
        }
        if k ≠ 0 && nums[k] == nums[k-1] {
            continue
```

```
        }
        for i, j := k+1, n-1; i < j; {
            if nums[i]+nums[j] == -nums[k] {
                res = append(res, []int{nums[k], nums[i],
nums[j]})
                i++
                for i < j && nums[i] == nums[i-1] {
                    i++
                }
                j--
                for i < j && nums[j] == nums[j+1] {
                    j--
                }
            } else if nums[i]+nums[j] < -nums[k] {
                i++
                for i < j && nums[i] == nums[i-1] {
                    i++
                }
            } else {
                j--
                for i < j && nums[j] == nums[j+1] {
                    j--
                }
            }
        }
    }
    return res
}
```

单调队列与单调栈

1.acwing830 单调栈

给定一个长度为 N 的整数数列，输出每个数左边第一个比它小的数，如果不存在则输出 - 1 -1

输入样例：

```
5
3 4 2 7 5
```

输出样例：

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

使用单调递增栈，栈底元素永远是最小值，且所有元素是递增的，如果新来的元素比栈顶小，就要出栈。

```
const int N = 200;
int n, a[N];

int main() {
    stack<int> st;
    cin >> n;
    for(int i = 0; i < n; i++) cin >> a[i];

    for(int i = 0; i < n; i++) {
        //如果a[i]比栈顶元素小，就出栈
        while(!st.empty() && st.top() > a[i]) st.pop();
        if(st.empty()) cout  << -1 << ' ';
        else cout << st.top() << ' ';
        //a[i]压栈
        st.push(a[i]);
    }
}
```

2.acwing154 滑动窗口

给定一个大小为 n≤1e6 的数组。有一个大小为 k 的滑动窗口，它从数组的最左边移动到最右边。你只能在窗口中看到 k 个数字。每次滑动窗口向右移动一个位置。求每个窗口的最小值。

输入样例：

n和窗口长度k

```
8 3
1 3 -1 -3 5 3 6 7
```

输出样例：

输出每个窗口的最小值

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

解：

使用单调递增队列，队头永远是队列中最小值，**队列存数组的下标**

队列中的下标要在一个窗口内，遍历时先判断，若a[i]不在这个窗口内就出队头

如果如果a[i]比队尾元素的值小，**就出队，维护单调递增**

最后i入队即可

```
const int N = 200;
int n, a[N], k;

int main() {
    //q存每个点的下标
    deque<int> q;
    cin >> n >> k;
    for(int i = 0; i < n; i++) cin >> a[i];

    for(int i = 0; i < n; i++) {
        //判断长度是否大于k
        if(!q.empty() && i-q.front()+1>k) q.pop_front();
        //如果a[i]比队尾元素的值小，就出队
        while(!q.empty() && a[i] < a[q.back()]) q.pop_back();
        //a[i]的下标i入队
        q.push_back(i);
        if(i+1 ⩾ k) cout << a[q.front()] << ' ';
    }
}
```

并查集

每个集合用一棵树来表示，树根编号就是整个集合的编号，数组p存每个节点的父节点。

可以用来合并两个集合O(1)，判断两个点是否在一个集合O(1)。

朴素并查集

```
int p[N]; //存储每个点的祖宗节点

// 返回x的祖宗节点，优化后是O(1)
int find(int x)
{
    if (p[x] ≠ x) p[x] = find(p[x]);
    return p[x];
}
```

```
// 初始化，节点编号是1~n
for (int i = 1; i ≤ n; i ++ ) p[i] = i;

// 合并a和b所在的两个集合：
p[find(a)] = find(b);
```

维护size的并查集

```
int p[N], size[N];
//p[]存储每个点的祖宗节点，size[]只有祖宗节点的有意义，表示祖宗节点所在集合中的点的数量

// 返回x的祖宗节点
int find(int x) {
    if (p[x] ≠ x) p[x] = find(p[x]);
    return p[x];
}

// 初始化，假定节点编号是1~n
for (int i = 1; i ≤ n; i ++) {
    p[i] = i;
    size[i] = 1;
}

// 合并a和b所在的两个集合：
size[find(b)] += size[find(a)];
p[find(a)] = find(b);
```

维护到祖宗距离的并查集

```
int p[N], d[N];
//p[]存储每个点的祖宗节点，d[x]存储x到p[x]的距离

// 返回x的祖宗节点
int find(int x)
{
    if (p[x] ≠ x)
    {
        int u = find(p[x]);
        d[x] += d[p[x]];
        p[x] = u;
    }
    return p[x];
```

```
}

// 初始化，假定节点编号是1~n
for (int i = 1; i ≤ n; i ++ )
{
    p[i] = i;
    d[i] = 0;
}

// 合并a和b所在的两个集合：
p[find(a)] = find(b);
d[find(a)] = distance; // 根据具体问题，初始化find(a)的偏移量
```

1.acwing836 合并集合(朴素并查集)

一共有 n 个数，编号是 1~n，最开始每个数各自在一个集合中。

现在要进行 m 个操作，操作共有两种：

M a b，将编号为 a 和 b 的两个数所在的集合合并，如果两个数已经在同一个集合中，则忽略这个操作；
Q a b，询问编号为 a 和 b 的两个数是否在同一个集合中；
输入格式
第一行输入整数 n 和 m。

接下来 m 行，每行包含一个操作指令，指令为 M a b 或 Q a b 中的一种。

输出格式
对于每个询问指令 Q a b，都要输出一个结果，如果 a 和 b 在同一集合内，则输出 Yes，否则输出 No。

每个结果占一行。

数据范围
1≤n,m≤105
输入样例：

```
4 5
M 1 2
M 3 4
Q 1 2
Q 1 3
Q 3 4
```

输出样例：

```
Yes
```

No
Yes

```
const int N = 200;
int n, p[N];

int find(int x) {
    if (x ≠ p[x]) p[x] = find(p[x]);
    return p[x];
}

int main() {
    int m;
    cin >> n >> m;
    // 初始化，节点编号是1~n
    for (int i = 1; i ≤ n; i ++ ) p[i] = i;

    while(m--) {
        char q;
        int a, b;
        cin >> q >> a >> b;
        if(q=='M') p[find(a)]=find(b);
        else {
            if(find(a)=find(b)) cout<<"Yes\n";
            else cout<<"No\n";
        }
    }
}
```

2.acwing837

给定一个包含 n个点（编号为 1~n）的无向图，初始时图中没有边。

现在要进行 m 个操作，操作共有三种：

C a b，在点 a 和点 b 之间连一条边，a 和 b 可能相等；
Q1 a b，询问点 a 和点 b 是否在同一个连通块中，a 和 b 可能相等；
Q2 a，询问点 a 所在连通块中点的数量；
输入格式：

第一行输入整数 n 和 m。

接下来 m 行，每行包含一个操作指令，指令为 C a b，Q1 a b 或 Q2 a 中的一种。

输出格式：

对于每个询问指令 Q1 a b，如果 aa 和 bb 在同一个连通块中，则输出 Yes，否则输出 No。

对于每个询问指令 Q2 a，输出一个整数表示点 a 所在连通块中点的数量

每个结果占一行。

数据范围：

1≤n,m≤10^5

输入样例：

```
5 5
C 1 2
Q1 1 2
Q2 1
C 2 5
Q2 5
```

输出样例：

```
Yes
2
3
```

```
朴素并查集再维护size数组即可
```

种类并查集

朋友的朋友就是朋友（普通并查集），但**敌人的敌人也是朋友**（维护这种关系就是种类并查集了）。

例如：有两队，1与2敌对，1与3敌对，那2与3是朋友。那就开两倍数组，1与2+n合并表示1与2敌对，1与3+n合并表示1与3敌对，2与3合并表示2与3是朋友。

1.蓝桥侦探

小明是蓝桥王国的侦探。

这天，他接收到一个任务，任务的名字叫分辨是非，具体如下：

蓝桥皇宫的国宝被人偷了，犯罪嫌疑人锁定在 N 个大臣之中，他们的编号分别为 1\sim N1~N。

在案发时这 N 个大臣要么在大厅1，要么在大厅2，但具体在哪个大厅他们也不记得了。

审讯完他们之后，小明把他们的提供的信息按顺序记了下来，一共 m 条，形式如下：

x y，表示大臣 xx 提供的信息，信息内容为：案发时他和大臣 yy 不在一个大厅。
小明喜欢按顺序读信息，他会根据信息内容尽可能对案发时大臣的位置进行编排。

他推理得出第一个与先前信息产生矛盾的信息提出者就是偷窃者，但推理的过程已经耗费了他全部的脑力，他筋疲力尽的睡了过去。作为他的侦探助手，请你帮助他找出偷窃者！

输入描述

第1行包含两个正整数N,M，分别表示大臣的数量和口供的数量。
之后的第2～ M＋1行每行输入两个整数c，g，表示口供的信息。
1<N ,M ≤ 5x10^5, 1 ≤ x, y ≤N。
输入输出样例
示例 1

输入
4 5
1 2
1 3
2 3
3 4
1 4
1
2
3
4
5
6
输出
2

```
const int N = 5e5 + 10;
int p[2*N], n, m;

int find(int x) {
    if(p[x] != x) p[x] = find(p[x]);
    return p[x];
}

int main() {
    cin >> n >> m;
    for(int i = 1; i <= 2*n; i++) p[i] = i;
    while(m--) {
        int a, b;
        //输入a, b要求a, b敌对
        cin >> a >> b;
```

```
            //如果查询到a，b是朋友，发生矛盾
            if(find(a) == find(b)) {
                cout << a;
                break;
            } else {
                //建立a与b的敌对关系
                p[find(a)] = find(b+n);
                p[find(b)] = find(a+n);
            }
        }
    }
```

2.食物链(洛谷2024)

动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B，B吃 C，C 吃 A。

现有 N 个动物，以 1- N 编号。每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

第一种说法是 1 X Y，表示 X 和 Y是同类。
第二种说法是 2 X Y，表示 X 吃 Y。
此人对 NN 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

当前的话与前面的某些真的话冲突，就是假话；
当前的话中 X 或 Y 比 N大，就是假话；
当前的话表示 X 吃 X，就是假话。
你的任务是根据给定的 N和 K 句话，输出假话的总数。

输入格式
第一行两个整数，N,K，表示有 N 个动物，K 句话。

第二行开始每行一句话（按照题目要求，见样例）

输出格式
一行，一个整数，表示假话的总数。

输入输出样例
输入 #1

```
100 7
1 101 1
2 1 2
2 2 3
2 3 3
```

1 1 3
2 3 1
1 5 5
输出 #1

3

解

p数组开3倍，a与b+n、a+n与b+2*n、a+2*n与b合并表示b吃a。a与b合并表示a,b同类

```
const int N = 1e5 + 10;
int p[3*N], n, m, ans;

int find(int x) {
    if(p[x] ≠ x) p[x] = find(p[x]);
    return p[x];
}
//合并
void unite(int a, int b) {
    p[find(a)] = p[find(b)];
}

int main() {
    cin >> n >> m;
    for(int i = 1; i ≤ 3*n; i++) p[i] = i;
    while(m--) {
        int q, a, b;
        cin >>q >> a >> b;
        if(a>n || b>n) {
            ans++;
            continue;
        }
        if(q == 1) {    //a,b同类
            //如果发现a吃b或者b吃a，就是假话
            if(p[find(a)] == p[find(b+n)] || p[find(a)] ==
p[find(b+2*n)]) {
                ans++;
                continue;
            } else {    //是真话，合并a,b
                unite(a, b);
                unite(a+n, b+n);
                unite(a+2*n, b+2*n);
            }
        } else {    //a吃b
```

```
            //如果发现a吃a或b吃a或者a,b同类，就是假话
            if(a==b || p[find(b)] == p[find(a+n)] || p[find(b)]
== p[find(a)]) {
                ans++;
                continue;
            } else {    //是真话
                unite(a, b+n);
                unite(a+n, b+2*n);
                unite(a+2*n, b);
            }
        }
    }
    cout << ans;
}
```

图论

树与图的存储

树是一种特殊的图，与图的存储方式相同。
对于无向图中的边ab，存储两条有向边a→b，b→a。
因此我们可以只考虑有向图的存储。

（1）邻接矩阵：g[a][b] 存储边a→b

（2）邻接表：

```
// 对于每个点k，开一个单链表，存储k所有可以走到的点。h[k]存储这个单链表的
头结点
int h[N], e[N], ne[N], idx;
// 添加一条边a→b
void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}
// 初始化
idx = 0;
memset(h, -1, sizeof h);

//最简单的方法
vector<int> g[N];
```

树与图的遍历

时间复杂度O(n+m),n表示点，m表示边

```
//递归实现深度遍历
void dfs(int u) {
    st[u] = true;
    for(int i = h[u]; i =≠ -1; i = ne[i]) {
        int k = e[i];
        if(!st[k]) dfs(k);
    }
}

const int N = 111;
vector<int> g[N];
int n, m;
void dfs(int u) {
    for(int i = 0; i < g.size(); i++) {
        if(!st[i]) dfs(g[u][i]);
    }
}

//使用queue进行层次遍历
q = append(q, root)
    for len(q) > 0 {
        size := len(q)
        for i := 0; i < size; i++ {
            t := q[0]
            q = q[1:]
            if t.Left =≠ nil {
                q = append(q, t.Left)
            }
            if t.Right =≠ nil {
                q = append(q, t.Right)
            }
        }
        d++
    }
```

单源最短路径

dijkstra

输入

```
7 12
1 2 6
1 3 3
2 3 2
2 4 1
3 4 5
2 5 4
3 6 7
4 5 3
4 6 6
5 6 2
5 7 2
6 7 3
```

输出

```
0
6
3
7
10
10
12
```

```cpp
#include<iostream>
#include<vector>
#include<algorithm>
#include<cstring>
using namespace std;
#define endl '\n'

const int N = 1000;
int h[N], ne[N], e[N], w[N], idx, n, m;
int dist[N];
bool st[N];

void add(int a, int b, int x) {
    e[idx] = b, w[idx] = x;
    ne[idx] = h[a], h[a] = idx++;
```

```
}

void dijkstra() {
    for(int k = 0; k < n; k++) {
        int t = -1;
        for(int i = 1; i <= n; i++) {
            if(st[i]) continue;
            if(t==-1 || dist[i] < dist[t]) t = i;
        }
        st[t] = true;
        //更新节点到源点的距离
        for(int i = h[t]; i != -1; i = ne[i]) {
            int k = e[i];
            dist[k] = min(dist[k], dist[t]+w[i]);
        }
    }
}

int main() {
    memset(dist, 0x3f, sizeof dist);
    memset(h, -1, sizeof h);
    dist[1] = 0;
    cin >> n >> m;
    for(int i = 0; i < m; i++) {
        int a, b, x;
        cin >> a >> b >> x;
        add(a, b, x);
    }
    dijkstra();
    for(int i = 1; i <= n; i++) {
        cout << dist[i] << endl;
    }
}
```

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<cstring>
using namespace std;
#define endl '\n'
typedef pair<int, int> pii;

const int N = 1005;
```

```cpp
vector<pii> g[N];
int n, m, dist[N];
bool st[N];

void dijkstra() {
    for(int k = 0; k < n; k++) {
        int t = -1;
        for(int i = 1; i <= n; i++) {
            if(st[i]) continue;
            if(t==-1 || dist[i]<dist[t]) t = i;
        }
        st[t] = true;

        for(int i = 0; i < g[t].size(); i++) {
            int k = g[t][i].first, d = g[t][i].second;
            dist[k] = min(dist[k], dist[t]+d);
        }
    }
}

int main() {
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    cin >> n >> m;
    for(int i = 0; i < m; i++) {
        int a, b, c;
        cin >> a >> b >> c;
        g[a].push_back({b, c});
    }
    dijkstra();
    for(int i = 1; i <= n; i++) cout << dist[i] << endl;
}
```

最小生成树

```
5 8
1 4 1
1 2 2
1 3 3
2 4 1
2 3 4
```

```
5 4 5
5 2 3
5 3 6f
```

输出(点+到集合长度)

```
1 0
4 1
2 1
3 3
5 3
```

prim（加点法）

```cpp
#include<iostream>
#include<vector>
#include<algorithm>
#include<cstring>
using namespace std;
#define endl '\n'
typedef pair<int, int> pii;

const int N = 1005;
vector<pii> g[N];
vector<pii> ans;
int n, m, dist[N];
bool st[N];

void dijkstra() {
    for(int k = 0; k < n; k++) {
        int t = -1;
        for(int i = 1; i <= n; i++) {
            if(st[i]) continue;
            if(t==-1 || dist[i]<dist[t]) t = i;
        }
        st[t] = true;
        ans.push_back({t,dist[t]});
        for(int i = 0; i < g[t].size(); i++) {
            int k = g[t][i].first, d = g[t][i].second;
            //此处与dijkstra算法不同
            dist[k] = min(dist[k], d);
        }
    }
}
```

```
int main() {
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    cin >> n >> m;
    for(int i = 0; i < m; i++) {
        int a, b, c;
        cin >> a >> b >> c;
        g[a].push_back({b, c});
        g[b].push_back({a, c});
    }
    dijkstra();
    for(int i = 0; i < ans.size(); i++) cout << ans[i].first << ' ' << ans[i].second << endl;
}
```

kruskal（加边法）

首先记录各个点的入度,然后将入度为 0 的点放入队列

然后类似bfs，遍历节点，更新后面节点入度-1

如果所有点都进过队列，则可以拓扑排序，输出所有顶点。否则输出-1，代表不可以进行拓扑排序，有回路不能进行拓扑排序，必须是有向无环图。

拓扑排序

4 4
1 2
1 3
2 4
3 4
输出
1 2 3 4

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<cstring>
#include<queue>
using namespace std;
#define endl '\n'
typedef pair<int, int> pii;
```

```cpp
const int N = 1005;
vector<int> g[N], ans;
int n, m, d[N];
bool st[N];

void topusort() {
    queue<int> q;
    //入度为0的点入队
    for(int i = 1; i ⩽ n; i++) {
        if(d[i]==0) {
            q.push(i);
        }
    }
    while(!q.empty()) {
        int t = q.front();
        ans.push_back(t);
        q.pop();
        for(int i = 0; i < g[t].size(); i++) {
            int k = g[t][i];
            d[k]--;
            if(d[k]==0) {
                q.push(k);
            }
        }
    }
}

int main() {
    cin >> n >> m;
    for(int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        g[a].push_back(b);
        d[b]++;
    }
    topusort();
    //有环路，不能拓扑排序
    if(ans.size() ≠ n) cout << -1;
    else {
        for(int i = 0; i < ans.size(); i++) cout << ans[i] << '
';
    }
```

```
}
```

二分图

二分图通常针对 无向图 问题（有些题目虽然是有向图，但一样有二分图性质）

在一张图中，如果能够把全部的点分到 两个集合 中，保证两个集合内部没有任何边 ，图中的边只存在于两个集合之间，这张图就是二分图

染色法（判断一个图是否为二分图）

算法原理就是，用 **黑 与 白** 这两种颜色对图中点染色（相当于给点归属一个集合），**一个点显然不能同时具有两种颜色**，若有，此图就不是二分图

```
4 4
1 2
1 3
2 4
3 4
1

3 3
1 2
1 3
2 3
0
```

O(n+m)

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<cstring>
#include<queue>
using namespace std;
#define endl '\n'
typedef pair<int, int> pii;

const int N = 1005;
vector<int> g[N];
int n, m, color[N];
bool st[N];

//染色法判断二分图
bool dfs(int u, int c) {
    color[u] = c;
```

```
    for(int i = 0; i < g[u].size(); i++) {
        int k = g[u][i];
        if(!color[k]) { //未染色
            if(!dfs(k, 3-c)) return false;
        } else { //已染色
            if(color[k] == color[u]) return false;
        }
    }
    return true;
}

int main() {
    cin >> n >> m;
    for(int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        g[a].push_back(b);
        g[b].push_back(a);
    }

    bool flag = true;
    //遍历所有点，防止图不连通
    for(int i = 1; i <= n; i++) {
        if(!color[i]) {
            if(!dfs(i, 0)) {
                flag = false;
                break;
            }
        }
    }
    cout << flag;
}
```

1.关押罪犯

S城现有两座监狱，一共关押着 N 名罪犯，编号分别为 1~N。

他们之间的关系自然也极不和谐。

很多罪犯之间甚至积怨已久，如果客观条件具备则随时可能爆发冲突。

我们用“怨气值”（一个正整数值）来表示某两名罪犯之间的仇恨程度，怨气值越大，则这两名罪犯之间的积怨越多。

如果两名怨气值为 c的罪犯被关押在同一监狱，他们俩之间会发生摩擦，并造成影响力为 c 的冲突事件。

每年年末，警察局会将本年内监狱中的所有冲突事件按影响力从大到小排成一个列表，然后上报到 S 城 Z 市长那里。

公务繁忙的 Z市长只会去看列表中的第一个事件的影响力，如果影响很坏，他就会考虑撤换警察局长。

在详细考察了 N 名罪犯间的矛盾关系后，警察局长觉得压力巨大。

他准备将罪犯们在两座监狱内重新分配，以求产生的冲突事件影响力都较小，从而保住自己的乌纱帽。

假设只要处于同一监狱内的某两个罪犯间有仇恨，那么他们一定会在每年的某个时候发生摩擦。

那么，应如何分配罪犯，才能使 Z 市长看到的那个冲突事件的影响力最小？这个最小值是多少？

输入格式

第一行为两个正整数 N 和 M，分别表示罪犯的数目以及存在仇恨的罪犯对数。

接下来的 M 行每行为三个正整数 aj，bj，cj表示 aj号和 bj号罪犯之间存在仇恨，其怨气值为 cj。

数据保证 $1\le aj<bj<N,0<cj\le 10^9$且每对罪犯组合只出现一次。

输出格式

输出共 1 行，为 Z市长看到的那个冲突事件的影响力。

如果本年内监狱中未发生任何冲突事件，请输出 0。

数据范围

$N\le 20000,M\le 100000$

输入样例：

```
4 6
1 4 2534
2 3 3512
1 2 28351
1 3 6618
2 4 1805
3 4 12884
```

输出样例：

```
3512
```

将所有点分成两组，使得各组内边的权重的最大值尽可能小，组间边权重尽可能大。

二分枚举limit，使大于limit的边放组间，判断能否构成二分图。找到的

```cpp
#include<iostream>
#include<vector>
#include<algorithm>
#include<cstring>
#include<queue>
using namespace std;
#define endl '\n'
typedef pair<int, int> pii;

const int N = 2e4+10;
vector<pii> g[N];
int n, m, color[N];

bool dfs(int u, int c, int x) {
    color[u] = c;
    for(int i = 0; i < g[u].size(); i++) {
        if(g[u][i].second <= x) continue;
        int k = g[u][i].first;
        if(!color[k]) {
            if(!dfs(k, 3-c, x)) return false;
        } else {
            if(color[k] == color[u]) {
                return false;
            }
        }
    }
    return true;
}

bool check(int x) {
    //cout << x << endl;
    //判断是否满足二分图,做到大于x的边在组间
    memset(color, 0, sizeof color);
    for(int i = 1; i <= n; i++) {
        if(!color[i]) {
            if(!dfs(i, 1, x)){
```

```
                return false;
            }
        }
    }
    return true;
}

int main() {
    cin >> n >> m;
    for(int i = 0; i < m; i++) {
        int a, b, c;
        cin >> a >> b >> c;
        g[a].push_back({b, c});
        g[b].push_back({a, c});
    }
    //二分枚举limit, 找到最小的limit
    //使其可以满足二分图,做到大于x的边一定在组间
    int l = 0, r = 1e9;
    while(l < r) {
        int mid = (l+r)/2;
        //cout << l << ' ' << r << endl;
        if(check(mid)) r = mid;
        else l = mid+1;
    }

    cout << l;
}
```

匈牙利算法（求出二分图的最大匹配数）

满足 是二分图 这个**前提**，才能使用匈牙利算法

最坏情况会每个点遍历全部边一次，所以时间复杂度是O(nm)

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<cstring>
#include<queue>
using namespace std;
#define endl '\n'
```

```cpp
typedef pair<int, int> pii;

const int N = 1005;
vector<int> g[N];
int n, m, color[N], match[N];
bool st[N];

//染色法判断二分图
bool dfs(int u, int c) {
    color[u] = c;
    for(int i = 0; i < g[u].size(); i++) {
        int k = g[u][i];
        if(!color[k]) { //未染色
            if(!dfs(k, 3-c)) return false;
        } else { //已染色
            if(color[k] == color[u]) return false;
        }
    }
    return true;
}


bool find(int x) {
    for(int i = 0; i < g[x].size(); i++) {
        int k = g[x][i];
        if(!st[k]) {
            st[k] = true;
            //如果k点还没有匹配 或 k点匹配的点可以空出来连下一个点
            if(match[k]==0 || find(match[k])) {
                match[k] = x;
                return true;
            }
        }
    }
    return false;
}

int main() {
    cin >> n >> m;
    for(int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        g[a].push_back(b);
```

```
        g[b].push_back(a);
    }

    int ans = 0;
    for(int i = 1; i <= 4; i++) {
        if(find(i)) ans++;
    }
    for(int i = 1; i <= n; i++) {
        cout << match[i] << endl;
    }
}
```

dfs序

树

根据前序遍历创建二叉搜索树，输出每个节点的左右节点

输入
8
6 3 1 2 5 4 8 7
输出
0 2
0 0
1 5
0 0
4 0
3 8
0 0
7 0

```
const int N = 1e4+10;
int pre[N], l[N], r[N], n, m;
int root;

void add(int u, int k) {
    if(k < u) {
        if(l[u] == 0) l[u] = k;
        else add(l[u], k);
    } else {
        if(r[u] == 0) r[u] = k;
        else add(r[u], k);
```

```
    }
}

int main() {
    cin >> n;
    for(int i = 1; i <= n; i++) {
        cin >> pre[i];
        if(i==1) root = pre[i];
        else add(root, pre[i]);
    }
    for(int i = 1; i <= n; i++) cout << l[i] << ' ' << r[i] <<
endl;
}
```

1.二叉树的深度

```
func dfs(root *TreeNode) {
    if root == nil {
        return 0
    }
    return max(dfs(root.Left), dfs(root.Right))+1
}
```

最近公共祖先(lca)

第一行包含三个正整数 N，M，S，分别表示树的结点个数、询问的个数和树根结点的序号。

接下来 N-1 行每行包含两个正整数 *x*，*y*，表示 x 结点和 y 结点之间有一条直接连接的边（数据保证可以构成树）。

接下来 *M* 行每行包含两个正整数 a，b，表示询问 a 结点和 b 结点的最近公共祖先。

输入

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

输出

```
4
4
1
4
4
```

时间复杂度O(n*m) m为询问次数。会超时，可以用倍增优化

```cpp
#include<iostream>
#include<vector>
#include<algorithm>
#include<cstring>
using namespace std;
#define endl '\n'
typedef pair<int, int> pii;

const int N = 5e5+10;
vector<int> g[N];
int f[N], n, m, root;
bool st[N];

int lca(int u, int v) {
    st[u] = true;
    while(f[u]≠u) {
        u = f[u];
        st[u] = true;
    }

    //如果v是u的祖先直接返回
    if(st[v]) return v;
```

```
    while(f[v]≠v) {
        v = f[v];
        if(st[v]) return v;
    }
}

//记录每个节点的父节点
void dfs(int u, int p) {
    f[u] = p;
    for(int i = 0; i < g[u].size(); i++) {
        int k = g[u][i];
        if(k == p) continue;
        dfs(k, u);
    }
}

int main() {
    cin >> n >> m >> root;
    for(int i = 1; i < n; i++) {
        int a, b;
        cin >> a >> b;
        g[a].push_back(b);
        g[b].push_back(a);
    }
    f[root] = root;
    dfs(root, root);
    while(m--) {
        memset(st, 0, sizeof st);
        int a, b;
        cin >> a >> b;
        int t = lca(a, b);
        cout << t << endl;
    }
}
```

DFS与BFS

dfs模板

```
void dfs(int k) { //递归的参数
    if(k){}      //搜索到了终点，然后回溯

    for(k) {
        st[k] = true;
        dfs(k);          //深搜
        st[k] = false;   //回溯
    }
}
```

bfs模板

```
//使用queue进行层次遍历
void bfs(int u) {
    queue<int> q;
    st[u] = true;
    q.push(u);
    while(!q.empty()) {
        int t = q.front();
        q.pop();
        for(k) {
            if(!st[k]) {
                st[k] = true;
                q.push(k);
            }
        }
    }
}
```

类走迷宫的搜索

1.求最短路径

给定一个n*m的迷宫，0表示路，1表示墙，求到终点的最短路径

输入

输入一个n*m的迷宫，终点坐标

```
5 5
4 4
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
```

输出

8

使用dfs暴力搜索时间复杂度很高O(2^n)

```
const int N = 30;
int g[N][N], ans = 0x3f3f3f3f, n, m;
int x, y; //出口坐标
int d[4][2] = {-1,0, 1,0, 0,-1, 0,1};
bool st[N][N];

void dfs(int i, int j, int len) {
    //坐标不合法 或 遇到障碍物 或 被访问过
    if(i<0 || i≥n || j<0||j≥m || g[i][j] == 1 || st[i][j])
return;
    //cout << i <<' ' << j << endl;

    if(i == x && j == y) {
        ans = min(ans, len);
        //cout << "ans=" << ans << endl;
        return;
    }

    for(int k = 0; k < 4; k++) {
        int x = i+d[k][0], y = j+d[k][1];
        st[i][j] = true;
        dfs(x, y, len+1);
        //回溯
        st[i][j] = false;
    }
}

int main() {
    cin >> n >> m;
    cin >> x >> y;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            cin >> g[i][j];
        }
    }
```

```
    dfs(0, 0, 0);
    cout << ans;
}
```

使用bfs，时间复杂度O(n)

```
const int N = 1000;
//dist存入口到该点的距离
int g[N][N], dist[N][N], n, m;
int x, y; //出口坐标
int d[4][2] = {-1,0, 1,0, 0,-1, 0,1};
bool st[N][N];

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

    while(!q.empty()) {
        pii t = q.front();
        q.pop();
        //遍历4个方向
        for (int k = 0; k < 4; k++) {
            int x1 = t.first + d[k][0], y1 = t.second + d[k]
[1];
            if (x1≥0 && x1<n && y1≥0 && y1<m && !st[x1][y1])
{
                dist[x1][y1] = dist[t.first][t.second] + 1;
                st[x1][y1] = true;
                q.push({x1, y1});
            }
        }
    }
}

int main() {
    cin >> n >> m;
    cin >> x >> y;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            cin >> g[i][j];
```

```
        }
    }
    bfs(0, 0);
    cout << dist[x][y];
}
```

2.求到出口的路径数

给定一个n*m的迷宫，计算出从左上到终点有多少种走法

输入

输入一个n*m的迷宫，终点坐标

```
5 5
4 4
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
```

输出

2

dfs暴力搜索即可

```
const int N = 1e4 + 10;
int g[N][N], ans, n, m;
int x, y; //出口坐标
int d[4][2] = {-1,0, 1,0, 0,-1, 0,1};
bool st[N][N];

void dfs(int i, int j) {
    //cout << i <<' ' << j << endl;
    if(i == x && j == y) {
        ans++;
        //cout << "ans=" << ans << endl;
        //注意这里不要return，要执行st[i][j] = false 回溯
    }

    for(int k = 0; k < 4; k++) {
        int x = i+d[k][0], y = j+d[k][1];
        //坐标不合法 或 遇到障碍物 或 被访问过
```

```
        if(x<0 || x≥n || y<0||y≥m || g[x][y] == 1 || st[x]
[y]) continue;
        st[x][y] = true;
        dfs(x, y);
        st[x][y] = false; //回溯
    }

}

int main() {
    cin >> n >> m;
    cin >> x >> y;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            cin >> g[i][j];
        }
    }
    st[0][0] = true;
    dfs(0, 0);
    cout << ans;
}
```

3.求站在某个点最多有多少个连通块

有一间长方形的房子，地上铺了红色、黑色两种颜色的正方形瓷砖。你站在其中一块黑色的瓷砖上，只能向相邻的（上下左右四个方向）黑色瓷砖移动。请写一个程序，计算你总共能够到达多少块黑色的瓷砖。

1）‘.’：黑色的瓷砖；
2）‘#’：红色的瓷砖；
3）‘@’：黑色的瓷砖，并且你站在这块瓷砖上。该字符在每个数据集合中唯一出现一次。

输入

```
9 6
....#.
.....#
......
......
......
......
......

@...#
```

.#..#.

输出

45

遍历即可，不需要回溯

```
const int N = 1e4 + 10;
char g[N][N];
int ans, n, m;
int d[4][2] = {-1,0, 1,0, 0,-1, 0,1};
bool st[N][N];

void dfs(int i, int j) {
    //不合法 或 红瓷砖 或 已经访问过
    if(i<0 || i≥n || j<0 || j≥m || g[i][j]=='#' || st[i][j])
return;

    //cout << i << ' ' << j << endl;
    ans++;
    //遍历4个方向
    for(int k = 0; k < 4; k++) {
        int x = i+d[k][0], y = j+d[k][1];
        st[i][j] = true;
        dfs(x, y);
        //注意这里不需要回溯
    }

}

int main() {
    cin >> n >> m;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            cin >> g[i][j];
        }
    }
    int x, y;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            if(g[i][j] == '@') {
                x = i;
                y = j;
                break;
```

```
            }
        }
    }
    st[x][y] = true;
    dfs(x, y);
    cout << ans;
}
```

dfs

1.n皇后

在n×n的棋盘上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上

解

每一行都有一个皇后，枚举每一行的皇后的位置即可。

```
const int N = 111;
int n, m, ans;
bool st[N][N];

bool check(int k, int j) {
    for (int i = 0; i < k; i++) {
        if (st[i][j] ||
            ((i - k + j >= 0) && st[i][i - k + j]) ||
            ((k - i + j < n) && st[i][k - i + j])) return
false;
    }
    return true;
}

void dfs(int k) {   //k为当前已经确定了多少行
    if (k == n) {
        ans++;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                cout << st[i][j];
            }
            cout << endl;
        }
        cout << endl;
```

```
            return;
        }

        for (int j = 0; j < n; j++) {    //搜索该行所有点
            if (check(k, j)) {  //判断当前点是否可以放
                st[k][j] = true;
                dfs(k + 1);
                st[k][j] = false;
            }
        }
    }

    int main() {
        cin >> n;
        dfs(0);
        cout << ans;
    }
```

2.危险系数

抗日战争时期，冀中平原的地道战曾发挥重要作用。
地道的多个站点间有通道连接，形成了庞大的网络。但也有隐患，当敌人发现了某个站点后，其它站点间可能因此会失去联系。
我们来定义一个危险系数DF(x,y)：
对于两个站点x和y （x ≠ y），如果能找到一个站点z，当z被敌人破坏后，x和y不连通，那么我们称z为关于x,y的关键点。相应的，对于任意一对站点x和y，危险系数DF(x,y)就表示为这两点之间的关键点个数。
本题的任务是：已知网络结构，求两站点之间的危险系数。

输入
输入数据第一行包含2个整数n(2 ≤ n ≤ 1000)，m(0 ≤ m ≤ 2000),分别代表站点数，通道数；
接下来m行，每行两个整数 u,v (1 ≤ u，v ≤ n；u ≠ v)代表一条通道；
最后1行，两个数u,v，代表询问两点之间的危险系数DF(u，v)。

输出
一个整数，如果询问的两点不连通则输出-1.

样例输入
```
7 6
1 3
2 3
3 4
```

3 5
4 5
5 6
1 6

样例输出
2

解

回溯搜索图从起点到终点求出路径总数，每个路径都访问到的点就是关键点。用数组cnt2记录所有路径中每个点的累计访问次数即可。

```
const int N = 1111;
int h[N], ne[2*N], e[2*N], idx, x, y, n, m, ans;
bool st[N];
int cnt[N], cnt2[N];    //各点访问次数
int t;  //路径数

void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a], h[a] = idx++;
}

void dfs(int u) {
    //cout << u << endl;
    if(u == y) {
        t++;
        //累加各点访问次数
        for(int i = 1; i <= n; i++) {
            cnt2[i] += cnt[i];
        }
    }

    for(int i = h[u]; i != -1; i = ne[i]) {
        int k = e[i];
        if(st[k]) continue;
        st[k] = true;
        cnt[k]++;
        dfs(k);
        cnt[k]--;
        st[k] = false;
    }
}
```

```
int main() {
    memset(h, -1, sizeof h);

    cin >> n >> m;
    while(m--) {
        int a, b;
        cin >> a >> b;
        add(a, b);
        add(b, a);
    }
    cin >> x >> y;
    st[x] = true;
    dfs(x);
    //cout << endl;
    for(int i = 1; i <= n; i++) {
        if(cnt2[i] == t && i != y) {
            //cout << i << endl;
            ans++;
        }
    }
    cout << ans;
}
```

bfs

```
//层次遍历
void bfs() {
    queue<int> q;
    q.push(1);
    while(!q.empty()) {
        int size = q.size();
        while(size--) { //一层一层的遍历
            int t = q.front();
            q.pop();
            for(k){
                q.push(k);
            }
        }
    }
}
```

1.八数码

在一个3 × 3的网格中，1~8 这 8 个数字和一个 x 恰好不重不漏地分布在这3 × 3的网格中。
在游戏过程中，可以把 x 与其上、下、左、右四个方向之一的数字交换（如果存在）。

我们的目的是通过交换，使得网格变为如下排列（称为正确排列）：

123
456
78x

输入格式
输入占一行，将 3 × 3 的初始网格描绘出来。

输入样例:

2 3 4 1 5 x 7 6 8

输出样例

19

```
/*
map中不存在相同的元素，count只能返回 0 或 1
*/
#include<iostream>
#include<algorithm>
#include<unordered_map>
#include<queue>

using namespace std;

int bfs(string state)
{
    queue<string> q;
    //用字符串来标记变换次数，字符串做下标
    unordered_map<string, int> d;

    q.push(state);
    //初始化移动次数
    d[state] = 0;

    //四个方向
    int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};

    //目标状态
```

```
    string end = "12345678x";

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

        if(t == end) return d[t];

        int distance = d[t];
    //find是string中的一个寻找下标的函数
        int k = t.find('x');
    //模拟二维数组
        int x = k / 3, y = k % 3;
    //枚举四个方向
        for(int i = 0; i < 4; i++)
        {
            //枚举最近的四个点
            int a = x + dx[i], b = y + dy[i];
            if(a >= 0 && a < 3 && b >= 0 && b < 3)
            {
                swap(t[a*3 + b], t[k]);
    //只有未枚举过的状态才可以进入队列
    //所以第一次枚举到最终状态一定是最优解
                if(!d.count(t))//count
                {
                    d[t] = distance + 1;
                    q.push(t);
                }
                swap(t[a*3 + b], t[k]);
            }
        }
    }
    return -1;
}

int main()
{
    char s[2];
    string state;
    //以单个字符的形式输入，再连接成字符串
    //避免直接以字符串形式输入时将空格输入
    for(int i = 0; i < 9; i++)
```

```
    {
        cin >> s;
        state += *s;
    }
    cout << bfs(state) << endl;
    return 0;
}
```

2.长草

小明有一块空地，他将这块空地划分为 nn 行 mm 列的小块，每行和每列的长度都为 1。

小明选了其中的一些小块空地，种上了草，其他小块仍然保持是空地。

这些草长得很快，每个月，草都会向外长出一些，如果一个小块种了草，则它将向自己的上、下、左、右四小块空地扩展，

这四小块空地都将变为有草的小块。请告诉小明，kk 个月后空地上哪些地方有草。

输入格式

输入的第一行包含两个整数 n，m。

接下来 n 行，每行包含 m 个字母，表示初始的空地状态，字母之间没有空格。

如果为小数点，表示为空地，如果字母为 g，表示种了草。

接下来包含一个整数 k。 其中2≤n,m≤1000，0≤k≤1000。

输出格式

输出 n 行，每行包含 m 个字母，表示 k 个月后空地的状态。如果为小数点，表示为空地，如果字母为 g，表示长了草。

输出样例

```
4 5
.g...
.....
..g..
.....
2
```

输出样例

```
gggg.
gggg.
ggggg
.ggg.
```

解

初始化queue时，将有草的格子push，直接bfs即可。bfs使用层次访问。

```
#define endl '\n'
typedef pair<int, int> pii;

const int N = 1111;
char g[N][N];
int n, m, k, d[4][2] = {-1,0, 1,0, 0,-1, 0,1};

void bfs() {
    queue<pii> q;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            if(g[i][j] == 'g') {
                q.push({i, j});
            }
        }
    }
    int l = k;
    while(!q.empty()) {
        if(l == 0) break;
        int size = q.size();    //层次遍历
        while(size--) {
            pii t = q.front();
            q.pop();
            for(int k = 0; k < 4; k++) {
                int x=t.first+d[k][0], y=t.second+d[k][1];
                if(x<0 || x≥n || y<0 || y≥m || g[x][y]=='g')
continue;
                q.push({x,y});
                g[x][y] = 'g';
                //cout << x << ' ' << y << endl;
            }
        }
        l--;
        //cout << endl;
    }
}

int main() {
    cin >> n >> m;
    for(int i = 0; i < n; i++) {
```

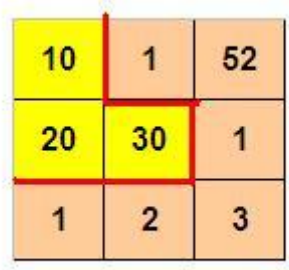
```
        for(int j = 0; j < m; j++) {
            cin >> g[i][j];
        }
    }
    cin >> k;
    bfs();
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            cout << g[i][j];
        }
        cout << endl;
    }
}
```

dfs剪枝搜索

就是根据题目实际需求，在dfs暴力搜索中将不符合要求的子树剪去

1.剪格子

题目描述
如下图所示，3 x 3 的格子中填写了一些整数。



我们沿着图中的红色线剪开，得到两个部分，每个部分的数字和都是60。

本题的要求就是请你编程判定：对给定的 mxn 的格子中的整数，是否可以分割为两个部分，使得这两个区域的数字和相等。
如果存在多种解答，请输出包含左上角格子的那个区域包含的格子的最小数目。

如果无法分割，则输出0。

输入描述
程序先读入两个整数m，n用空格分割（m，n<10），表示表格的宽度和高度。

接下来是n行，每行m个正整数，用空格分开。每个整数不大于104。

输出描述

在所有解中，包含左上角的分割区可能包含的最小的格子数目。

输入：

```
3 3
10 1 52
20 30 1
1 2 3
```

输出：

```
3
```

```
const int N = 20;
int g[N][N], n, m, d[4][2] = {-1,0, 1,0, 0,-1, 0,1};
int ans = 0x3f3f3f3f, sum;
bool st[N][N];

void dfs(int i, int j, int s, int num) {
    if(2*s > sum) return;

    //cout << i << ' ' << j << ' ' << s << endl;
    if(2*s == sum) {
        ans = min(ans, num);
    }

    st[i][j] = true;
    for(int k = 0; k < 4; k++) {
        int x=i+d[k][0], y=j+d[k][1];
        //如果坐标不合法 或 被访问过 或 总和大于sum2
        if(x<0 || x≥n || y<0 || y≥m || st[x][y]) continue;

        dfs(x, y, s+g[x][y], num+1);
    }
    st[i][j] = false;
}

int main() {
    cin >> n >> m;
    //计算出数字总和
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            cin >> g[i][j];
            sum += g[i][j];
        }
```

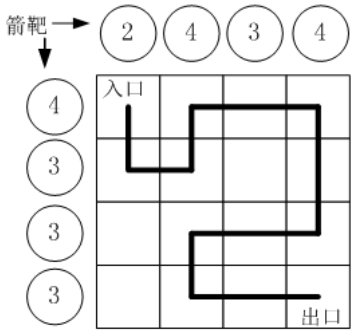
```
    }
    if(sum%2 == 1) {

    } els
    dfs(0, 0, g[0][0], 1);
    if(ans == 0x3f3f3f3f) cout << 0;
    else cout << ans;
}
```

2.路径之谜

题目描述】

小明冒充骑士进入了一个城堡。城堡里边方形石头铺成的地面。假设城堡地面是 n×n 个方格。按习俗，骑士要从西北角走到东南角。可以横向或纵向移动，但不能斜着走，也不能跳跃。每走到一个新方格，就要向正北方和正西方各射一箭。（城堡的西墙和北墙内各有 n 个靶子）同一个方格只允许经过一次。但不必走完所有的方格。如果只给出靶子上箭的数目，你能推断出骑士的行走路线吗?

本题的要求就是已知箭靶数字，求骑士的行走路径（测试数据保证路径唯一）



【输入格式】

第一行一个整数 N（0<N<20），表示地面有 N×N 个方格；第二行 N 个整数，空格分开，表示北边的箭靶上的数字（自西向东）；第三行 N 个整数，空格分开，表示西边的箭靶上的数字（自北向南）。

【输出格式】

一行若干个整数，表示骑士路径。为了方便表示，我们约定每个小格子用一个数字代表，从西北角（左上角）开始编号：0,1,2,3 ....

【输入示例】

4

2 4 3 4

4 3 3 3

【输出示例】

0 4 5 1 2 3 7 11 10 9 13 14 15

```
const int N = 30;
int g[N][N], n, m, d[4][2] = {-1,0, 1,0, 0,-1, 0,1};
int north[N], west[N];
bool st[N][N], flag;
vector<int> v;

void dfs(int i, int j) {
    //剪枝
    //不符合就返回，找到答案就退出
    if(north[j] < 0 || west[i] < 0 || flag) return;

    //cout << i << ' ' << j << endl;
    //到达右下角
    if(i==n-1 && j==n-1) {
        for(int i = 0; i < n; i++) {
            if(north[i] != 0) return;
            if(west[i] != 0) return;
        }
        flag = true;
        for(int i = 0; i < v.size(); i++) {
            cout << v[i] << ' ';
        }
    }

    for(int k = 0; k < 4; k++) {
        int x=i+d[k][0], y=j+d[k][1];
        //如果坐标不合法 或 被访问过
        if(x<0 || x>=n || y<0 || y>=n || st[x][y]) continue;

        v.push_back(x*n+y);
        north[y]--, west[x]--;
        st[x][y] = true;
        dfs(x, y);
        north[y]++, west[x]++;
        v.pop_back();
        st[x][y] = false;
    }
```

```
}

int main() {
    cin >> n;
    for(int i = 0; i < n; i++) cin >> north[i];
    for(int i = 0; i < n; i++) cin >> west[i];

    v.push_back(0);
    st[0][0] = true;
    north[0]--, west[0]--;
    dfs(0, 0);
}
```

3.四阶幻方

把 1~16 的数字填入 4×4 的方格中，使得行、列以及两个对角线的和都相等，满足这样的特征时称为：四阶幻方。

四阶幻方可能有很多方案。如果固定左上角为 1，请计算一共有多少种方案。

解：

此题为填空题，全排列，手动剪枝每行或每列的值一定为34

```
const int N = 20;
int a[N], cnt;
bool st[N];

void dfs(int k) {
    if(k≥4 && a[0]+a[1]+a[2]+a[3]≠34) return;
    if(k≥8 && a[4]+a[5]+a[6]+a[7]≠34) return;
    if(k≥12 && a[8]+a[9]+a[10]+a[11]≠34) return;
    if(k≥13 && (a[0]+a[4]+a[8]+a[12]≠34 ||
a[3]+a[6]+a[9]+a[12]≠34)) return;
    if(k≥14 && a[1]+a[5]+a[9]+a[13]≠34) return;
    if(k≥15 && a[2]+a[6]+a[10]+a[14]≠34) return;
    if(k≥16 && (a[3]+a[7]+a[11]+a[15]≠34 ||
a[12]+a[13]+a[14]+a[15]≠34 || a[0]+a[5]+a[10]+a[15]≠34))
return;

    if(k == 16) {
        cnt++;
    }
```

```
    for(int i = 2; i ⩽ 16; i++) {
        if(st[i]) continue;
        a[k] = i;
        st[i] = true;
        dfs(k+1);
        st[i] = false;
    }
}

int main() {
    a[0] = 1;
    dfs(1);
    cout << cnt;
}
```

记忆化搜索

nxm的滑雪场，g[i,j]为每点的高度，从任一点只能向低处滑，问最多能滑多远

输入

```
4 5
1 4 6 3 1
11 8 7 3 1
9 4 5 2 1
1 3 2 2 1
```

输出

```
7
```

$O(nm*4^{nm})\rightarrow O(nm)$

```
int dfs(int i, int j) {
    //cout << i << ' ' << j << endl;
    if(dp[i][j] ≠ 0) return dp[i][j];
    dp[i][j] = 1;
    for(int k = 0; k < 4; k++) {
        int x=i+d[k][0], y=j+d[k][1];
        if(x<0 || x⩾n || y<0||y⩾m || g[x][y]⩾g[i][j])
continue;
        dp[i][j] = max(dp[i][j], dfs(x, y)+1);
    }
    return dp[i][j];
}
```

```
int main() {
    cin >> n >> m;
    for(int i = 0; i < n ; i++) {
        for(int j = 0; j < m; j++) {
            cin >> g[i][j];
        }
    }
    int ans = 0;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            ans = max(ans, dfs(i, j));
        }
    }
    cout << ans;
}
```

全排列

1.从n个数中选m个数

```
int n, m, a[N];
bool st[N];

void dfs(int k, int start) {
    if(k == m) {
        for(int i = 0; i < m; i++) {
            cout << a[i] << ' ';
        }
        cout << endl;
    }

    for(int i = start; i <= n; i++) {
        a[k] = i;
        dfs(k+1, i+1);
    }
}

int main() {
    n = 5;
    m = 3;
    dfs(0, 1);
}
```

```cpp
const int N = 5e5+10;
int n, m, a[N];
bool st[N];

void dfs(int k, int start) {
void dfs(int k, int num) {
    if(num == m) {
        for(int i = 1; i <= n; i++) {
            if(st[i]) cout << i << ' ';
        }
        cout << endl;
        return;
    }
    if(k > n) return;
    st[k] = 1;
    dfs(k+1, num+1);
    st[k] = 0;
    dfs(k+1, num);
}

int main() {
    n = 5;
    m = 3;
    dfs(1, 0);
}
}

int main() {
    n = 5;
    m = 3;
    dfs(0, 1);
}
```

DP

dp使用了空间换时间的方法。面对一个dp问题，有两个步骤：

- 确定dp数组的含义
- 写出状态转移方程

背包问题

01背包

有n个物品，每个物品有体积w和价值v两个属性，且每种物品都只有一个，将这n个物品放入一个容量为v的背包，并使得此背包的价值最大。

```
//dp数组含义
dp[i][j]: 只有前i个物品参与下，背包容量为j的最大价值
//状态转移方程
dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]]+v[i])
//优化为一维，因为要根据前一行数据计算，故要从后往前遍历，不然会被覆盖
dp[j] = max(dp[j], dp[j-w[i]]+v[i])
```

```
for(int i = 1; i ≤ n; i++) {
    for(int j = w[i]; j ≤ m; j++) {
        dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]]+v[i]);
    }
}
//优化
for(int i = 1; i ≤ n; i++) {
    for(int j = m; j ≥ w[i]; j--) {
        dp[j] = max(dp[j], dp[j-w[i]]+v[i]);
    }
}
```

完全背包

每个物品有无限个，可以使用无限次，使背包价值最大

```
//状态转移方程
dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]]+v[i])
//优化为一维，因为是根据本行已改变数据计算，故要从前往后遍历，
dp[j] = max(dp[j], dp[j-w[i]]+v[i])
```

```
for(int i = 1; i ⩽ n; i++) {
    for(int j = w[i]; j ⩽ m; j++) {
        dp[i][j] = max(dp[i-1][j], dp[i][j-w[i]]+v[i]);
    }
}
//优化
for(int i = 1; i ⩽ n; i++) {
    for(int j = w[i]; j ⩽ m; j++) {
        dp[j] = max(dp[j], dp[j-w[i]]+v[i]);
    }
}
```

1.零钱兑换

求能凑出零钱的组合数和最少硬币数

cnt=5, coin = {1,2,5}

```
//dp[i][j]: 使用前i个硬币，凑出零钱j的组合数
//完全背包
dp[i][j] = dp[i-1][j] + dp[i][j-coin[i]];

//dp[i][j]: 使用前i个硬币，凑出零钱j的最少硬币数
//完全背包
dp[i][j] = min(dp[i-1][j], dp[i][j-coin[i]]+1);
```

线性dp

1.最长递增子序列lis

5 6 7 4 2 8 3

dp[i]：以i结尾的最长递增子序列的长度

```
dp[i] = max(dp[k]+1); (k<i && a[i]>a[k])
        1           (i == 0)
```

```
dp[0] = 1;
for(int i = 0; i < n; i++) {
    for(int k = 0; k < i; k++) {
        if(a[i] > a[k]) {
            dp[i] = max(dp[i], dp[k]+1);
            ans = max(ans, dp[i]);
        }
    }
}
cout << ans;
```

2.最长连续递增子序列

dp[i]：以i结尾的最长连续递增子序列的长度

$$dp[i] = \begin{cases} dp[i-1]+1 & (a[i] > a[i-1]) \\ 1 & (a[i] \leqslant a[i-1] \ || \ i == 0) \end{cases}$$

```
dp[0] = 1;
for(int i = 1; i < n; i++) {
    if(a[i] > a[i-1]) dp[i] = dp[i-1]+1;
    else dp[i] = 1;
    ans = max(ans, dp[i]);
}
cout << ans;
```

3.最长公共子序列lcs

dp[i, j]：分别以i，j结尾的子序列的最长公共子序列的长度

$$dp(i,j) = \begin{cases} max(dp[i-1,j],dp[i,j-1]); & (s1[i] \neq s2[j]) \\ dp[i-1,j-1]+1; & (s1[i] == s2[j]) \\ 0 & (i==0 \ || \ j==0) \end{cases}$$

```
s1 = "abcfbc"
s2 = "abfcab"
ans = dp[n][m];
```

4.最长公共连续子序列(最长公共子串)

dp[i, j]：分别以i，j结尾的子序列的最长公共子串的长度

```
dp(i,j)= 0                          (s1[i] ≠ s2[j])
         dp[i-1,j-1]+1;             (s1[i] == s2[j])
         0                      (i==0 || j==0)
```

```
3 2 1 4 7
1 2 3 2 1
ans = max(dp[i][j], ans);
```

5.最长公共上升子序列

dp[i, j]:所有以a[1,i]和b[1,j]构成的且以b[j]为结尾的公共上升子序列

```
dp[i, j] = dp[i-1, j] (a[i] ≠ b[j])
           max(dp[i-1, k])+1  (k⩾0&&k<j && b[j]>b[k])
```

树形dp

利用dfs在树上进行dp

令f [u] = f[u]=~f[u]= 与树上顶点u uu有关的某些数据，并按照拓扑序（从叶子节点向上到根节点的顺序）进行DP \text{DP}DP，确保在更新一个顶点时其子节点的dp值已经被更新好，以更新当前节点的DP \text{DP}DP值。为方便计算，一般写成dfs的形式，如下：

```
void dfs(int u) { // 遍历节点u
    dp[v] = ...; // 初始化
    for(int v: G[u]) { // 遍历u的所有子节点
        dfs(v);
        update(u, v); // 用子节点的dp值对当前节点的dp值进行更新
    }
}
```

1.子树大小

给定一棵有N个结点的树，根结点为结点1。对于i=1,2,…,N，求以结点i为根的子树大小（即子树上结点的个数，包括根结点）。

dp[u] = 所有dp[v]之和 + 1

```
const int N = 111;
int n, m, dp[N];

vector<int> g[N];
void dfs(int u) { // 遍历节点u
    dp[v] = 1; // 初始化
    for(int v: G[u]) { // 遍历u的所有子节点
        dfs(v);
        dp[u] += dp[v]; // 用子节点的dp值对当前节点的dp值进行更新
    }
}

int main() {
    cin >> n >> m;
    while(m--) {
        int a, b;
        cin >> a >> b;
        g[a].push_back(b);
    }
    dfs(1);
}
```

2.没有上司的舞会

有N NN名职员，编号为1 … N他们的关系就像一棵以老板为根的树，父节点就是子节点的直接上司。每个职员有一个快乐指数r_i，现在要召开一场舞会，使得没有职员和直接上司一起参会。主办方希望邀请一部分职员参会，使得所有参会职员的快乐指数总和最大，求这个最大值。

保证给出的关系一定是一棵树。

输入

```
7
1
1
1
1
```

```
1
1
1
1 3
2 3
6 4
7 4
4 5
3 5
```

输出

```
5
```

每个节点都有选和未选两种情况，而当前dp[u]也要考虑子节点选或未选的情况，故使用二维dp数组

dp[u, 0] 根节点未选的情况，最大快乐指数，dp[u, 1]选了根节点的情况，最大快乐指数

dp[u, 1] += dp[k, 0] (k为u的子节点)
dp[u, 0] += max(dp[k,0], dp[k, 1]) (k为u的子节点)

```
const int N = 6600;
vector<int> g[N];
int n, m, v[N], dp[N][2];
bool st[N];

void dfs(int u) {
    dp[u][0] = 0;
    dp[u][1] = v[u];
    for(int i = 0; i < g[u].size(); i++) {
        int k = g[u][i];
        dfs(k);
        //选择u的情况
        dp[u][1] += dp[k][0];
        //不选u的情况
        dp[u][0] += max(dp[k][0], dp[k][1]);
    }
}

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

```
    for(int i = 1; i < n; i++) {
        int a, b;
        cin >> a >> b;
        st[a] = true;
        g[b].push_back(a);
    }
    //寻找根节点
    int u;
    for(int i = 1; i ≤ n; i++) {
        if(!st[i]) u = i;
    }
    dfs(u);
    cout << max(dp[u][0], dp[u][1]);
}
```

3.生命之树

在X森林里，上帝创建了生命之树。

他给每棵树的每个节点（叶子也称为一个节点）上，都标了一个整数，代表这个点的和谐值。

上帝要在这棵树内选出一个非空节点集 S，使得对于 S 中的任意两个点 a,b都存在一个点列 {a,v1,v2,…,vk,b}使得这个点列中的每个点都是 S里面的元素，且序列中相邻两个点间有一条边相连。

在这个前提下，上帝要使得 S中的点所对应的整数的和尽量大。

这个最大的和就是上帝给生命之树的评分。

经过 atm 的努力，他已经知道了上帝给每棵树上每个节点上的整数。

但是由于 atm 不擅长计算，他不知道怎样有效的求评分。

他需要你为他写一个程序来计算一棵树的分数。

输入格式

第一行一个整数 n表示这棵树有 n个节点。

第二行 n个整数，依次表示每个节点的评分。

接下来 n-1行，每行 2 个整数 u,v表示存在一条 u到 v的边。

由于这是一棵树，所以是不存在环的。

树的节点编号从 1 到 n

输出格式

输出一行一个数，表示上帝给这棵树的分数。

数据范围

$1 \le n \le 10^5$

每个节点的评分的绝对值均不超过 10^6

输入样例：

```
5
1 -2 -3 4 5
4 2
3 1
1 2
2 5
```

输出样例：

```
8
```

dp[u]：根节点为u且包括u的最大连通字块大小

dp[u] += max(dp[k], 0)

```
void dfs(int u) {
    for(int i = 0; i < g[u].size(); i++) {
        int v = g[u][i];
        dfs(v);
        dp[u] += max(dp[k], 0);
    }
}
```

4.树上背包

状态压缩dp

1.小国王

在n×n 的棋盘上放 k 个国王，国王可攻击相邻的 8 个格子，求使它们无法互相攻击的方案总数。

输入

3 2

输出

16

$1 \leqslant n \leqslant 10$

$0 \leqslant k \leqslant n^2$

dp[i,j,s]表示在前i行中放置国王，共放置j个国王的情况下，且第i行的状态是s的时候，所有的方案总数。

$f[i,j,s] = \sum f[i-1,j-count(s),ss]$

```
const int N = 22, K = 111, S=1<<10;
int n, k;
long long f[N][K][S];
vector<int> state;

bool check(int x) {
    for(int i = 0; i < n-1; i++) {
        if((x>>i&1) && (x>>(i+1)&1)) return false;
    }
    return true;
}
int getnums(int x) {
    int res = 0;
    for(int i = 0; i < n; i++) {
        if(x>>i&1) res++;
    }
    return res;
}

int main(){

    cin >> n >> k;
    for(int i = 0; i < 1<<n; i++) {
        if(check(i)) state.push_back(i);
    }
    f[0][0][0] = 1;
    for(int i = 1; i <= n; i++) {
        for(int j = 0; j <= k; j++) {
            for(int s = 0; s < state.size(); s++) {
                int count = getnums(s)
                for(int ss = 0; s < state.size(); s++) {
                    if(((state[ss]&state[s])==0) &&
(check(state[ss]|state[s]))) {
                        int count = getnums(state[s]);
```

```
                        if(j ≥ count) {
                            f[i][j][state[s]] += f[i-1][j-
count][state[ss]];
                            //cout << i << ' ' << j << ' ' <<
state[ss] << ' ' <<f[i-1][j][state[ss]] << ' ' << f[i][j]
[state[s]] << endl;
                        }
                    }
                }
            }
        }

    cout << f[n+1][k][0];
}
```

数位dp

1.windy数

定义不含前导零且相邻数的差值至少为2的正整数被称为windy数，求[l,r]之间有多少个windy数

dfs求解，从高到枚举每位的值。

```
const int N = 22;
int n, bound[N], max_num;

int dfs(int k, int pre, bool flag, bool zero) {
    if(k < 0) return 0;

    if(flag) max_num=bound[k];
    else max_num=9;

    int ret = 0;
    for(int i = 0; i ≤ max_num; i++) {
        if(abs(i-pre)≥2 || k==n) {
            ret += dfs(k-1, i, flag&&(i==bound[k]));
        }
    }
```

```
    return ret;
}

int main(){
    int a, b;
    cin >> a >> b;
    int t = a, i = 1;
    while(a≠0) {
        bound[i] = a%10;
        a⁄=10;
        i++;
    }
    n = i-1;
    cout << dfs(n, 0, true) << endl;
    t = b, i = 1;
    while(a≠0) {
        bound[i] = a%10;
        a⁄=10;
        i++;
    }
    n = i-1;
    cout << dfs(n, 0, true) << endl;
}
```

数学

质数

试除法判断x是否是质数

O(sqrt(n))

```
void isPrime() {
    for(int i = 2; i ⩽ x/i; i++) {
        if(x%i == 0) return false
    }
    return true;
}
```

分解质因数(一个数可以分解为最小质因数相乘)

O(sqrt(n))

```
vector<pii> v;
void divide(int x) {
    for(int i = 2; i ⩽ x/i; i++) {
        if(x%i == 0) { //i一定是质数
            int t = 0;
            while(x%i==0) {
                t++;
                x /= i;
            }
            v.push_back({x, t});
        }
    }
    if(x > 1) v.push_back({x, 1});
}
```

筛质数

约数

求约数

```
vector<int> v;
for(int i = 1; i ⩽ x/i; i++) {
    if(x%i == 0) {
        v.push_back(i);
        if(i≠x/i) v.push_back(x/i);
    }
}
```

求约数个数和约数之和

```

```

欧几里得算法(求最大公约数)

```
int gcd(int a, int b) {
    return b?gcd(b, a%b):a;
}
```

贪心

1.跳跃游戏

给你一个非负整数数组 `nums` ，你最初位于数组的 **第一个下标** 。数组中的每个元素代表你在该位置可以跳跃的最大长度。判断你是否能够到达最后一个下标，如果可以，返回 `true` ；否则，返回 `false` 。

```
输入: nums = [2,3,1,1,4]
输出: true
```

```go
func canJump(nums []int) bool {
    maxLen := 0 //当前能到达的最远的点
    n := len(nums)
    for i := 0; i ≤ maxLen; i++ {
        maxLen = max(maxLen, nums[i]+i)
        if maxLen ≥ n-1 {  //可以到达
            return true
        }
    }
    return false
}
```

2.跳跃游戏2

返回到达 `nums[n - 1]` 的最小跳跃次数。测试用例都可以到达 `nums[n - 1]`。

```
输入: nums = [2,3,1,1,4]
输出: 2
```

```go
func jump(nums []int) int {
    maxLen := 0
    n := len(nums)
    end, step := 0, 0
    for i := 0; i < n-1; i++ {
        maxLen = max(maxLen, nums[i]+i)
        if i == end {
            step++
            end = maxLen
        }
    }
    return step
}
```