# E8 -Solution

## A 种树

| 难度 | 考点 |
| --- | --- |
| 1 | 循环输出字符 |

### 题目分析

逐行逐个字符输出，计算清楚每行每种字符的个数，利用循环输出即可。

### 示例代码

```
#include<stdio.h>
int main()
{
    int h, h1, h2, i, j;
    scanf("%d", &h);
    h1 = h / 3 * 2;            //树冠高度
    h2 = h / 3;                //树干高度
    int w = (h2 / 4) * 2 + 1; //树干宽度
    for(i = 1; i <= h1; i++)  //打印树冠，共h1层
    {
        for(j = 0; j < h1 - i; j++)               //第i层h1-i个空格
            printf(" ");
        for(j = 0; j < 2 * i - 1; j++)            //第i层2*i-1个*
            printf("*");
        for(j = 0; j < h1 - i; j++)               //第i层h1-i个空格
            printf(" ");
        printf("\n");
    }
    for(i = 1; i <= h2; i++)  //打印树干，共h2层
    {
        for(j = 0; j < (2 * h1 - 1 - w) / 2; j++) //(2*h1-1-w)/2=h1-1-h2/4个空格
            printf(" ");
        for(j = 0; j < w; j++)                    //w=h2/4*2+1个|
            printf("|");
        for(j = 0; j < (2 * h1 - 1 - w) / 2; j++) //(2*h1-1-w)/2=h1-1-h2/4个空格
            printf(" ");
        printf("\n");
    }
    return 0;
}
```

# B 湖中舟

| 难度 | 考点 |
| --- | --- |
| 1 | 模拟 |

## 问题分析

直接根据题意模拟即可；或记录各字符的个数，通过坐标差来判断是否能抵达终点。

## 参考代码 #1

```c
#include <stdio.h>
#include <string.h>

int main()
{
    int x1, x2, y1, y2, T, cnt, len, i;
    char s[1005];
    scanf("%d", &T);
    while (T--)
    {
        scanf("%d%d%d%d", &x1, &y1, &x2, &y2);
        scanf("%s", s);
        len = strlen(s);
        cnt = 0;
        for (i = 0; i < len; i++)
        {
            if (s[i] == 'N' && y1 < y2)
            {
                y1++;
                cnt++;
            }
            else if (s[i] == 'S' && y1 > y2)
            {
                y1--;
                cnt++;
            }
            else if (s[i] == 'W' && x1 > x2)
            {
                x1--;
                cnt++;
            }
            else if (s[i] == 'E' && x1 < x2)
            {
                x1++;
                cnt++;
            }
            if (x1 == x2 && y1 == y2)
                break;
        }
        if (x1 == x2 && y1 == y2)
            printf("%d\n", cnt);
        else
```

```
            printf("We want to live in Gensokyo forever...\n");
    }
    return 0;
}
```

## 参考代码 #2

```
#include <stdio.h>
#include <string.h>
#include <math.h>

int main()
{
    int x1, x2, y1, y2, T, a[26], len, i;
    char s[1005];
    scanf("%d", &T);
    while (T--)
    {
        scanf("%d%d%d%d", &x1, &y1, &x2, &y2);
        scanf("%s", s);
        len = strlen(s);
        memset(a, 0, sizeof(a));
        for (i = 0; i < len; i++)
            a[s[i] - 'A']++;
        if (x2 - x1 <= a['E' - 'A'] && x1 - x2 <= a['W' - 'A'] && y2 - y1 <=
a['N' - 'A'] && y1 - y2 <= a['S' - 'A'])
            printf("%d\n", abs(x2 - x1) + abs(y2 - y1));
        else
            printf("We want to live in Gensokyo forever...\n");
    }
    return 0;
}
```

# C 进位统计

| 难度 | 考点 |
| --- | --- |
| 2 | 函数，简单的递归，位运算 |

## 题目分析

按照题目描述，实现函数 $popcount$ 和函数 $f$ 即可。

## 示例代码

```
#include <stdio.h>
int popcount(int a) //计算a的二进制表示中有多少位是1
{
    int ret = 0;
    for(int i = 0; i < 32; ++i)
    {
```

```
        if(a >> i & 1) ret++;
    }
    return ret;
}
int f(int a, int b)
{
    if(b == 0) return 0;
    return popcount(a & b) + f(a ^ b, (a & b) << 1);
}
int main()
{
    int a, b;
    while(~scanf("%d%d", &a, &b))
        printf("%d\n", f(a, b));
    return 0;
}
```

## 补充

$f(a,b)$ 的值恰好是在二进制下计算 $a+b$ 时进位的次数，想想为什么?

*Author:* 哪吒

# D - 最小的K个数

| 难度 | 考点 |
| --- | --- |
| 3 | 排序 |

## 题目分析

• https://visualgo.net/zh/sorting 可以前往这个网站观看各类排序的过程。

在进行从大到小的冒泡排序的第 $1$ 轮，会把最小的数放在位置 $n$，第 $2$ 轮，会把第二小的数放在位置 $n-1$，因此进行 $K$ 轮冒泡排序即可得到前 $K$ 小的数。具体见示例代码-1。

也可使用选择排序的思路，进行从小到大的选择排序，第 $1$ 轮，会把最小的数放在位置 $1$，第 $2$ 轮，会把第二小的数放在位置 $2$，进行 $K$ 轮，同样可以得到前 $K$ 小的数。

本题使用 `qsort` 函数进行排序会超时，因为无需排序整个数组。但是也可以使用快速排序的思想实现本题（但有些大材小用），具体见示例代码-2。

## 示例代码 - 1

```
#include <stdio.h>
#define N 1000005
int a[N];
int main() {
    int n, K;
    scanf("%d%d", &n, &K);
    for(int i = 1; i <= n; i ++)
        scanf("%d", &a[i]);
    for(int i = 1; i <= K; i ++)
```

```
        for(int j = 1; j <= n - i; j ++)
            if(a[j] < a[j + 1]) {
                int t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }
    for(int i = 0; i < k; i ++)
        printf("%d ", a[n - i]);
    return 0;
}
```

## 示例代码 - 2

```
#include <stdio.h>
int a[1000005], n, k;
//仅供参考，快速排序选择轴值的时候没有随机选择，因此在某些极端数据的情况下可能会退化造成超时
void f(int l, int r)
{
    int x = l, y = r;
    while(l < r)
    {
        while(l < r && a[r] >= a[l]) r--;
        int t = a[l];
        a[l] = a[r];
        a[r] = t;
        while(l < r && a[l] < a[r]) l++;
        t = a[l];
        a[l] = a[r];
        a[r] = t;
    }
    if(l > x) f(x, l - 1);
    if(l < k) f(l + 1, y);
}
int main()
{
    scanf("%d%d", &n, &k);
    for(int i = 1; i <= n; ++i)
        scanf("%d", &a[i]);
    f(1, n);
    for(int i = 1; i <= k; ++i)
        printf("%d ", a[i]);
    return 0;
}
```

# E violet求最大和

| 难度 | 考点 |
| --- | --- |
| 3 | 多维数组 |

## 题目分析

此题的逻辑思路较为简单，求解步骤如下：

- 假设矩阵为 $m$ 行 $n$ 列，则矩阵有 $(m-2)\times(n-2)$ 个 $3\times3$ 子矩阵
- 遍历所有子矩阵，对每个子矩阵，遍历 $6$ 个字母，将 $6$ 个字母子矩阵中和最大的字母记录下来 (对应代码中的 $maxi[6]$ 数组，例如 $maxi[0]=1$ 代表字母 $V$ 是当前特殊子矩阵中和最大的 )
- 用变量 $max$ 存储全局最大和，当最大和更新时，需要清空 $maxi$ 数组
- 最终输出全局最大和 $max$ 和满足此最大和的数组 $maxi$ 所记录的字母即可

本题由于新定义了六种子矩阵，可以用三维的 $01$ 数组来存储（本质上就是 $6$ 个二维数组）这样在求特殊子矩阵的和的时候，可以直接与该其中一个二维数组做逐个的矩阵乘法。代码如下

## 示例代码

```
#include <stdio.h>
#include <string.h>
#include <limits.h>

int a[1005][1005];
int b[6][3][3] = {
    {1, 0, 1,
     1, 0, 1,
     0, 1, 0},
    {0, 1, 0,
     0, 1, 0,
     0, 1, 0},
    {1, 1, 1,
     1, 0, 1,
     1, 1, 1},
    {1, 0, 0,
     1, 0, 0,
     1, 1, 1},
    {1, 1, 1,
     1, 1, 1,
     1, 1, 1},
    {1, 1, 1,
     0, 1, 0,
     0, 1, 0}};

int find_sum(int x, int y, int k)
{
    int i, j, ans = 0;
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 3; j++)
        {
            ans += a[x + i][y + j] * b[k][i][j];
        }
    }
    return ans;
}

int main()
```

```
{
    int m, n, i, j, k;
    scanf("%d%d", &m, &n);
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < m; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }

    int max = INT_MIN, maxi[6] = {0};
    for (i = 0; i < m - 2; i++)
    {
        for (j = 0; j < n - 2; j++)
        {
            for (k = 0; k < 6; k++)
            {
                int temp = find_sum(i, j, k);
                if (temp > max)
                {
                    max = temp;
                    memset(maxi, 0, sizeof(maxi));
                    maxi[k] = 1;
                }
                else if (temp == max)
                {
                    maxi[k] = 1;
                }
            }
        }
    }

    printf("%d\n", max);
    char s[] = "VIOLET";
    for (i = 0; i < 6; i++)
    {
        if (maxi[i])
            printf("%c", s[i]);
    }
    return 0;
}
```

## F 某咸鱼与过河卒

| 难度 | 考点 |
| --- | --- |
| 4 | 二维数组、递推、杨辉三角形变种 |

## 题目分析

由地图递推可得。注意到在 $m \times n$ 的范围内，若想抵达 格点 $(x, y)$ ，则可以从格点 $(x-1, y)$ 向右走一步，或者从格点 $(x, y-1)$ 向上走一步。因此，若记录走到点 $(x, y)$ 的方法总数为 $a_{x,y}$ ，则 $a_{x,y} = a_{x-1,y} + a_{x,y-1}$ ，对于在边界的情况，可以采用 $a_{x,0} = a_{x-1,0}$ 与 $a_{0,y} = a_{0,y-1}$ 。

注意本题有部分不可到达的点，在遇到不可到达的点的时候，需要将不可到达的点标记为 $0$ 。

## 示例代码

```
#include<stdio.h>
int a[30][30] = {0};
int p[30][2];
int main() {
    int m, n, t;
    scanf("%d%d%d", &m, &n, &t);
    for (int i = 1; i <= t; i++) {
        scanf("%d%d", &p[i][0], &p[i][1]);
    }
    a[0][0] = 1;
    for (int i = 0; i <= m ; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 && j == 0) continue; //原点
            else if (i == 0) a[i][j] = a[i][j - 1]; //y轴上
            else if (j == 0) a[i][j] = a[i - 1][j]; //x轴上
            else a[i][j] = a[i - 1][j] + a[i][j - 1]; //一般情况
            for (int k = 1; k <= t; k++) { //判断该点是否不可到达
                if (i == p[k][0] && j == p[k][1]) {
                    a[i][j] = 0;
                }
            }
        }
    }
    /*for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }*/
    //调试部分代码，如果出现了WA的情况
    printf("%d", a[m][n]);
    return 0;
}
```

## 示例代码 - 2

```
#include <stdio.h>
int a[12][12] = {0, 1};
int main()
{
    int m, n, t;
    scanf("%d%d%d", &m, &n, &t);
    while(t--)
    {
```

```
        int x, y;
        scanf("%d%d", &x, &y);
        a[++x][++y] = 1;
    }
    for(int i = 1; i <= m + 1; ++i)
        for(int j = 1; j <= n + 1; ++j)
            a[i][j] = a[i][j] ? 0 : a[i - 1][j] + a[i][j - 1];
    printf("%d", a[++m][++n]);
    return 0;
}
```

# 扩展阅读

## 代码调试方法

本题常见的错误有两种，一是行和列讨论不清，二是坐标轴上的元素错误地全部初始化为 $1$ 。注意到本题采用二维数组进行计算，因此可以将二维数组全部输出，观察是否有误。代码如下：

```
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
```

## 知识扩展

注意到本题的递推公式为 $a_{x,y} = a_{x-1,y} + a_{x,y-1}$ 。如果地图中没有障碍物，令 $n = x + y, m = x$ ，则可得到 $c_{n,m} = c_{n-1,m-1} + c_{n-1,m}$ ，即 $C_n^m = C_{n-1}^{m-1} + C_{n-1}^m$ ，这就是组合数的性质公式。

同时地，对于本题的二维数组，如果我们按照右上-左下的斜线分为若干组的话，我们会发现这个二维数组的左上部分构成的三角形是经典的“杨辉三角形”（Pascal's Triangle）。利用这一三角形，我们可以轻易求出组合数，同时规避溢出的风险。

对于组合数 $C_n^m$ ，其对应 $t = 0$ 时，题解中的数组元素 `a[m][n-m]` 。接下来，我们利用组合数的公式 $C_n^m = \frac{n!}{m!(n-m)!}$ 与本题代码，分别计算 $C_{30}^{15}$ 的值，代码如下：

```
#include<stdio.h>
int a[30][30] = {0};
int p[30][2];
int main() {
    int m, n, t;
    scanf("%d%d%d", &m, &n, &t);
    for (int i = 1; i <= t; i++) {
        scanf("%d%d", &p[i][0], &p[i][1]);
    }
    a[0][0] = 1;
    for (int i = 0; i <= m ; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 && j == 0)continue;
            else if (i == 0)a[i][j] = a[i][j - 1];
            else if (j == 0)a[i][j] = a[i - 1][j];
            else a[i][j] = a[i - 1][j] + a[i][j - 1];
            for (int k = 1; k <= t; k++) {
```

```
                if (i == p[k][0] && j == p[k][1]) {
                    a[i][j] = 0;
                }
            }
        }
    }
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            //printf("%d ", a[i][j]);
        }
        //printf("\n");
    }
    printf("%d\n", a[m][n]);
    // 根据公式计算
    unsigned long long up=1,down=1;
    for(int i=1;i<=30;i++){
        up=up*i;
    }
        for(int i=1;i<=15;i++){
            down=down*i*i;
        }
        printf("%llu %llu %llu",up,down,up/down);
    return 0;
}
```

输入为：

```
15 15 0
```

运行上述代码后，我们不难发现上述代码的运行结果：

```
155117520
9682165104862298112 17523835397698748416 0
```

不难发现，直接套用公式会发生溢出。因此我们可以利用这一性质，在C语言中计算组合数。

## G 魔法少女★大类分流

| 难度 | 考点 |
| --- | --- |
| 4~5 | 模拟，循环，排序 |

### 题目分析

根据题目算法步骤描述，首先需要将所有学生的姓名和志愿按照排名进行升序排序，然后使用循环遍历第 $1 \sim m$ 志愿位置，对当前循环到的志愿位置（设为第 $i$ 志愿），按排名从前到后的顺序遍历所有学生的第 $i$ 志愿，判断每个学生是否已被录取，如果发现某个学生尚未被录取，就判断该学生的第 $i$ 志愿学院是否还有剩余名额，如果有就执行录取，为该学生记下录取他的学院号，相应学院剩余名额减 $1$，如果没有就不执行录取。

注意到本题需要使用数组保存每个学生的姓名、志愿、排名信息，然后依据学生排名对所有信息进行排序。然而，以目前课内学过的知识，无法将这些不同类型的学生信息整合到一个数组中，只能分开各自存储到 `char` 或 `int` 数组中。而 `qsort` 函数只能对一个数组进行排序，无法在排序环节直接对存有学生信息的多个数组同时使用 `qsort` 函数。

但是，即使分开存储在不同数组中的内容，也可以靠一个东西建立联系，那就是**下标**！只要在不同数组的相同下标位置保存同一个学生的各项信息，那么**使用相同的下标值去访问这些不同的数组，取出来的内容一定就是同一个学生的各项信息。**也就是说，**学生与下标值之间可以建立一个一一对应关系。**

因此，可以提前多开一个下标数组 `order`，用 `order[i]` 保存第 $i$ 个读入的学生的信息在其他数组中的存放位置下标。当读入结束后需要访问或修改某个学生的信息时，就使用每个 `order[i]` 的值作为下标去访问存储信息的数组。排序时，可以将 `order` 数组中存储的下标值视作学生，以排名为依据对 `order` 数组进行排序。排序结束后，`order` 数组中从前到后的顺序即是按排名升序排序的学生下标顺序。

因为只需要对 `order` 数组进行排序就可以实现依据学生排名对所有信息进行排序，所以就可以使用 `qsort` 函数了。

## 示例代码（对下标数组排序的方法）

```c
#include <stdio.h>
#include <stdlib.h>

int rank[1001] = {};    //学生排名，rank[i]表示第i个读入的学生的排名，开在全局变量是因为需要被cmp函数访问到

int cmp(const void *P1, const void *P2) {
    int num1 = *((int *) P1), num2 = *((int *) P2);
    //num1和num2是传入的两个指针指向的order数组中的元素
    if (rank[num1] < rank[num2]) {  //排序依据是num1和num2对应的学生的排名
        return -1;
    } else if (rank[num1] > rank[num2]) {
        return 1;
    } else {
        return 0;
    }
}

int main() {
    int n, m;
    int headCount[11] = {};     //headCount[i]表示i号学院当前剩余招生名额数
    char name[1001][10] = {};   //学生姓名
    int choice[1001][11] = {};  //学生志愿，choice[i][j]表示第i个读入的学生的第j志愿
    int order[1001] = {};       //记录每个学生的读入顺序，即学生信息在各数组中的存储位置下标，后面将用于排序
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; ++i) {
        scanf("%d", &headCount[i]);
    }
    for (int i = 1; i <= n; ++i) {
        scanf("%s%d", name[i], &rank[i]);
        order[i] = i;   //当前学生是第i个读入的，学生信息在其他数组中存储位置下标值为i
        for (int j = 1; j <= m; ++j) {   //读入第i个学生的志愿顺序
            scanf("%d", &choice[i][j]);
        }
    }
```

```
    //对学生信息的下标（即order数组元素）按学生排名进行排序
    qsort(order + 1, n, sizeof(int), cmp);
    //这里使用order数组的下标范围是1~n，因此需要从下标为1的位置开始，对连续n个数组元素进行排序，所以要传入order+1
    //排序完成后，order[i]为排第i名的学生的信息在数组中的存储位置下标
    int accepted = 0, result[1001] = {};
    //accepted变量保存已被录取的人数
    //result数组记录录取结果，result[i]不为0时表示排第i名的学生录取到的学院，为0时表示排第i名的学生还没有被录取
    for (int i = 1; i <= m && accepted < n; ++i) {   //循环变量i表示当前执行的是第i志愿的录取
        for (int j = 1; j <= n && accepted < n; ++j) {   //从前往后遍历每个学生的第i志愿
            //在两层for循环的条件中加入accepted < n，当所有学生都录取完后该表达式不满足，就可以退出两层循环
            if (result[j] == 0) {    //排第j名的学生还没有被录取时
                if (headCount[choice[order[j]][i]] > 0) {   //该学生的第i志愿学院还有名额时，执行录取
                    result[j] = choice[order[j]][i];
                    headCount[choice[order[j]][i]]--;
                    accepted++;
                }
            }
        }
    }
    for (int i = 1; i <= n; ++i) {
        printf("%s %d\n", name[order[i]], result[i]);
    }

    return 0;
}
```

另外，如果同学们看了 E7-J 有理数2023 的题解，那么也可以使用本课程知识范围外的、自定义结构体变量的方法，自行定义一个包含学生所有信息的变量类型 STUDENT，然后对结构体数组进行读入、赋值、排序，更加方便。代码在下面给出，但本课程中不要求掌握，作为补充拓展知识。

## 示例代码（使用结构体方法，本课程不要求掌握）

```
#include <stdio.h>
#include <stdlib.h>

//自定义STUDENT类型结构体变量，包含4个成员变量
typedef struct {
    char name[10];  //名字
    int rank;       //排名
    int choice[11]; //志愿
    int result;     //录取结果
} STUDENT;

//规定不同的STUDENT类型变量比较逻辑的函数
int cmp(const void *p1, const void *p2) {
    STUDENT stu1 = *((STUDENT *) p1), stu2 = *((STUDENT *) p2);
    if (stu1.rank < stu2.rank) {    //排名靠前的留在前面
        return -1;
    } else if (stu1.rank > stu2.rank) { //排名靠后的换到后面
```

```
        return 1;
    } else {
        return 0;
    }
}

int main() {
    int n, m;
    int headCount[11] = {}; //headCount[i]表示i号学院当前剩余招生名额数
    STUDENT stu[1001];      //声明一个STUDENT类型的结构体数组
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; ++i) {
        scanf("%d", &headCount[i]);
    }
    for (int i = 1; i <= n; ++i) {
        scanf("%s%d", stu[i].name, &stu[i].rank);   //读入第i个学生的名字与排名
        for (int j = 1; j <= m; ++j) {   //读入第i个学生的志愿顺序
            scanf("%d", &stu[i].choice[j]);
        }
        stu[i].result = 0;  //初始化每个学生的录取结果为0（一个不合法的学院号，表示学生还没有被录取）
    }
    //对存储学生信息的结构体数组按学生排名进行排序
    qsort(stu + 1, n, sizeof(STUDENT), cmp);
    //这里使用stu数组的下标范围是1~n，因此需要从下标为1的位置开始，对连续n个数组元素进行排序，所以要传入stu+1
    int accepted = 0;    //accepted变量保存已被录取的人数
    for (int i = 1; i <= m && accepted < n; ++i) {   //循环变量i表示当前执行的是第i志愿的录取
        for (int j = 1; j <= n && accepted < n; ++j) {   //从前往后遍历每个学生的第i志愿
            //在两层for循环的条件中加入accepted < n，当所有学生都录取完后该表达式不满足，就可以退出两层循环
            if (stu[j].result == 0) {    //排第j名的学生还没有被录取时
                if (headCount[stu[j].choice[i]] > 0) {   //该学生的第i志愿学院还有名额时，执行录取
                    stu[j].result = stu[j].choice[i];
                    headCount[stu[j].choice[i]]--;
                    accepted++;
                }
            }
        }
    }
    for (int i = 1; i <= n; ++i) {
        printf("%s %d\n", stu[i].name, stu[i].result);
    }

    return 0;
}
```

## 小彩蛋

可能有同学已经猜到了，这套规则其实源自近几年北航大类大一结束后专业分流的规则。出题人只进行了很小的改编，希望能够帮助大家认识、理解大类分流规则。如果到时候23级分流规则有变动调整，则以最新版规则为准。

# H 失踪的指挥棒

| 难度 | 考点 |
| --- | --- |
| 5~6 | 二分 —— $bsearch$ 的函数指针 |

## 题目分析

题目中描述的分组与长度关系的规则实际上就是为了说明：同一组内指挥棒长度单调递减，不同组间指挥棒长度单调递增，且后一组的最短指挥棒长于前一组的最长指挥棒，那么这道题就成为了一道非常典型的二分查找题目，因为长度一定是单调的。

## 方法一

由于整体递增局部递减，应进行两次二分，第一次锁定是在哪一个递减区间的范围内，第二次查找有无目标元素。

锁定范围的办法是用 `num` 数组存每一个递减区间第一个指挥棒或最后一个指挥棒的编号（示例代码中存的是最后一个），那么 `num` 数组对应的指挥棒长度一定是递增的。判断是否是某个递减区间的最后一个指挥棒只需判断是否有 `q[i]>q[i-1]`。

第一次二分查找的方式与 $HINT$ 中给出的相同，第二次二分查找是在一个递减区间中查找 **有无** 目标元素，两次的不同之处在于，第一次要查找的“目标元素”是数组中最后一个不超过 `*key` 的，`*key` 本身可能不在数组中；而第二次查找的目标元素就是 `*key` 本身，它必须在数组中，否则就应当返回 `NULL`。（这就是 $HINT$ 中说 `*b 为目标元素` 而不说 `*b==*a` 的原因）

这里为了减少一些边界条件和输出的判断（主要是消除了第一次 `bsearch` 返回 `NULL` 的可能），可以假定存在一个编号、长度均为 0 的指挥棒，将其自己归为一组，并存在 `num[0]` 和 `q[0]` 与 `len[0]` 中。

## 示例代码一

```
#include<stdio.h>
#include<stdlib.h>

int cp1(const void *a,const void *b)  //  第一次二分查找，在递增数列中
{
    if(*(int *)a>=*(int *)b && *(int *)a<*((int *)b+1)) return 0;
    //  *b为最后一个不超过*a的元素
    else if(*(int *)a>*(int *)b) return 1;  //  *b太小了，在目标元素左侧，应该右移
    else return -1;  //  *b太大了，在目标元素右侧，应该左移
}
```

```
int cp2(const void *a,const void *b)  //  第二次二分查找，在递减数列中
{
    if(*(int *)a==*(int *)b) return 0;  //  *b就是*a
    else if(*(int *)a<*(int *)b) return 1;  //  *b太大了，在目标元素左侧，应该右移
    else return -1;  //  *b太小了，在目标元素右侧，应该左移
}

int q[100001],num[100001],len[100001];
int main()
{
    int n,m,t,h=0,x,ans1,ans2;

    scanf("%d%d%d",&n,&m,&t);
    for(int i=1;i<=n;i++)
    {
        scanf("%d",&q[i]);
        if(q[i]>q[i-1])
        {
            h++;
            num[h]=i-1;
            len[h]=q[i-1];  //  别忘了二分查找不是在num中，而是在len中进行
        }
    }
    h++;
    num[h]=n;
    len[h]=q[n];  //  别忘了处理一下边界情况
    for(int i=1;i<=t;i++)
    {
        scanf("%d",&x);
        if(x>=len[h]) ans1=h;  //  边界情况
        else ans1=(int*)bsearch(&x,len,h+1,sizeof(int),cp1)-len;  //  第一次二分
        if(ans1==0) ans2=0;  //  边界情况
        else
        {
            int *p=(int*)bsearch(&x,q+num[ans1-1]+1,num[ans1]-num[ans1-
1],sizeof(int),cp2);  //  第二次二分，是从q+num[ans1-1]+1开始的，而不是从q开始
            if(p==NULL) ans2=0;  //  没找到
            else ans2=p-q;
        }
        printf("%d\n",ans2);
    }

    return 0;
}
```

## 方法二

其实只需要将所有长度排序，就可以直接一次二分查出有没有这个元素了，不过因为要输出这个长度对应的指挥棒的编号，就要用一个二维数组来存储数据了，其中 `a[i][0]` 表示第 $i$ 个指挥棒的长度，`a[i][1]` 表示第 $i$ 个指挥棒的编号，排序时要将 `a[i][0]` 与 `a[i][1]` 同时移动。并且由于数据规模较大，应该使用 `qsort` 对二维数组排序。

**但方法二需要对指针有稍多一些的了解才能保证不出错，请思考以下五个问题：**

1. `cpbsearch` 函数第一行注释掉的内容是什么意思？这里的 `*((int *)b+1)` 与方法一中表示的是一个东西嘛？
2. `int q[100000][2]` 改成 `int q[100001][2]` 行不行？改成 `int q[100000][3]` 行不行？
3. `int *p[2]` 和 `int (*p)[2]` 和 `int *p` 有区别嘛？
4. 把 `...p...=...bsearch(...);` 这句话改成如下语句行不行？

```
int *p=(int *)bsearch(&x,q,n,2*sizeof(int),cpbsearch);
```

5. 为什么这次不处理边界条件了，以及为什么方法一处理边界条件只处理右边？

## 示例代码二

```
#include<stdio.h>
#include<stdlib.h>

int cpqsort(const void *a,const void *b)  //  二维数组快速排序
{
    if(*(int *)a==*(int *)b) return 0;  //  *a就是*b
    else if(*(int *)a>*(int *)b) return 1;  //  *a应该位于*b右侧
    else return -1;  //  *a应该位于*b左侧
}

int cpbsearch(const void *a,const void *b)  //  二维数组二分查找
{
    //printf("? a: %d %d   b: %d %d ?\n",*(int *)a,*((int *)a+1),*(int *)b,*((int
*)b+1));
    if(*(int *)a==*(int *)b) return 0;  //  *b就是目标元素
    else if(*(int *)a>*(int *)b) return 1;  //  *b太小了，在目标元素左侧，应该右移
    else return -1;  //  *b太大了，在目标元素右侧，应该左移
}

int q[100000][2];

int main()
{
    int n,m,t,x,ans;

    scanf("%d%d%d",&n,&m,&t);
    for(int i=0;i<n;i++)
    {
        scanf("%d",&q[i][0]);
        q[i][1]=i+1;
    }

    qsort(q,n,2*sizeof(int),cpqsort);
    //  对二维数组按行排序，注意每行大小是2*sizeof(int)

    for(int i=1;i<=t;i++)
    {
```

```
        scanf("%d",&x);
        int (*p)[2]=(int (*)[2])bsearch(&x,q,n,2*sizeof(int),cpbsearch);
        //  对二维数组二分查找，注意每行大小是2*sizeof(int)
        if(p==NULL) printf("0\n");
        else
        {
            ans=q[(int)(p-q)][1];
            printf("%d\n",ans);
        }
    }

    return 0;
}
```

## 答案提示

1. 方法一、二表示的不是一个东西。方法一是一维数组，当 `*(int *)b` 表示 `q[i]` 的时候，`*((int *)b+1)` 表示 `q[i+1]`；方法二是二维数组，当 `*(int *)b` 表示 `q[i][0]` 的时候，`*((int *)b+1)` 表示 `q[i][1]`。
2. 行数（第一问）可以多一点，列数（第二问）不可以（至少不能只改这一句）。因为 `qsort` 和 `bsearch` 的 `size` 参数都是 `2*sizeof(int)`，即将数组一行的两个元素视为一组，如果列数变成 $3$，就会将 `q[0][0]` 与 `q[0][1]` 一组，`q[0][2]` 与 `q[1][0]` 一组，`q[1][1]` 与 `q[1][2]` 一组 ..... 不过也不是就完全不行，如果想改就要连着 `size` 参数以及 `*p` 的类型（详见第 $3$、$4$ 题）一起改。
3. 有区别。第一个 `p` 是指针数组，`p[0]` 和 `p[1]` 都是 `int` 型指针；第二个 `p` 是数组指针，`p` 是一个指针，指向一个长度为 $2$ 的 `int` 型数组，不存在 `p[0]` 或 `p[1]` 这种东西；第三个 `p` 是一个 `int` 型指针。（数组指针（第二个）和变量指针（第三个）的区别详见第 $4$ 题）
4. 不行（至少不能只改这一句）。改成这样 `p` 变成了 `int` 型指针，在 `ans=q[(int)(p-q)][1];` 中不同类型指针不能进行运算，需要将其中一个转换成与另一个同类型（`q` 是 `int (*)[2]` 型指针，即指向“长度为 $2$ 的 `int` 型数组”的数组指针）。需要注意一个问题，`q` 是不能直接进行类型转换的（会 `CE`）！只能对其解引用转化为 `int` 型指针，即改成如下形式：

```
ans=q[(int)(p-q[0])/2][1];
```

- 这里为什么要除以 $2$：因为 $C$ 语言编译时会识别指针的类型（首先保证是同类型指针），然后自动完成数值与地址之间的转化。当指针 `p` 与 `q` 做减法时，会自动将结果除以 `sizeof(*p)`，实为 `(p-q)/sizeof(*p)`；当指针 `p` 与数字做加法时（如 `p+1`），会自动将数字先乘以 `sizeof(*p)`，实为 `p+sizeof(*p)*1`。所以如果本题中将 `q` 解引用，那么就相当于将二维数组每行依次排列展开成一维数组，`q[i][0]` 与 `q[i][1]` 不再视作同一组了。

5. 第一次之所以要处理边界条件是因为在 `cp1` 函数中涉及到对 `q[i+1]` 的访问，有越出右界的风险，但第二次不会再访问 `q[i+1]` 了（如第 $1$ 题所述），因而不需要处理边界。只要不在 `cmp` 中强行访问 `q[i+1]` 或者 `q[i-1]`，那么 `qsort` 和 `bsearch` 就会避免掉越界的问题，毕竟这是两个安全的库函数。

~~或者你其实可以使用结构体、手搓快排、手搓二分……来避免这些麻烦，实现算法自由（）~~

# I 诗乃的出题

| 难度 | 考点 |
| --- | --- |
| 6 | 二分答案 |

## 题目分析

本题主要利用了二分答案的思想：即以答案为二分对象，每次二分后验证答案的可行性。

二分答案的前提仍然是答案具有“单调性”，它的基本思想是在答案可能的范围 ([L,R]) 内二分查找答案，不断检查当前答案是否满足题目的要求，根据检查结果 更新查找的区间，最终取得最符合题目要求的答案进行输出。

如何理解答案的单调性呢？简单来说，其要求答案的可能存在区间中存在一个“临界点”，位于临界点一侧的答案不不满足条件，而另一侧都满足，且越靠近“临界点”越满足条件。这种现象就是答案的单调性。

说回到本题，做法比较简单：先让数组从小到大排序，然后二分答案：

初始 $l=0$，$r=a[n]-a[1]$，每次二分答案为 $mid$，遍历数组，如果当前数比上一个数的差距大于等于 $mid$，计数器自增一次，如果计数器比 $m$ 大，$l=mid+1$，不然 $r=mid-1$。

## 示例代码 - 1

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int a[100002];
int cmp_int(const void* e1, const void* e2){
    return *(int*)e1 - *(int*)e2;
}
int main()
{
    int n,m;
    scanf("%d %d",&n,&m);
    for(int i=1; i<=n; i++){
        scanf("%d",&a[i]);
    }
    qsort(a+1, n, sizeof(int), cmp_int);
    int l=0,r=a[n]-a[1];
    int t,tot;
    int ans;
    while(l<=r){
        int mid=(l+r)>>1;
        t=a[1];tot=1;
        for(int i=1; i<=n; i++){
            if(a[i]-t>=mid){
                t=a[i];
                tot++;
            }
        }
```

```
        if(tot>=m){
            ans=mid;
            l=mid+1;
        }
        else r=mid-1;
    }
    printf("%d\n",ans);
    return 0;
}
```

## 示例代码 - 2

```
#include <stdio.h>
#include <stdlib.h>
int a[100000];
int cmp(const void*p, const void*q)
{
    return *(int*)p - *(int*)q;
}
int main()
{
    int n, m;
    scanf("%d%d", &n, &m);
    for(int i = 0; i < n; ++i)
        scanf("%d", &a[i]);
    qsort(a, n, sizeof(int), cmp);
    int l = 0, r = a[n - 1] - a[0], ans = (l + r + 1) / 2;
    while(l < r)
    {
        int cnt = 1, t = a[0];
        for(int i = 1; i < n; ++i)
        {
            if(a[i] - t >= ans)
            {
                t = a[i];
                cnt++;
            }
        }
        if(cnt >= m) l = ans;
        else r = ans - 1;
        ans = (l + r + 1) / 2; // (l+r)/2向上取整
    }
    printf("%d", ans);
    return 0;
}
```

# J 封闭异或

| 难度 | 考点 |
| --- | --- |
| 7 | 思维题，异或，qsort, bsearch |

## 题意分析

异或运算有很多有趣的性质，如交换律，结合律，以及如下的结论等等：

对任意的自然数 $a$，有 $a \oplus a = 0$；

对任意的自然数 $a, b, c$，若 $a \oplus b = c$，则有 $a \oplus c = b,\ b \oplus c = a$。

因此，如果自然数集 $A = \{a_1, a_2, \cdots, a_n\}$ 对异或运算满足封闭性，则 $0$ 一定在这些数之中，且对任意一个不在该集合中的自然数 $b$，有自然数集 $A' = \{a_1, a_2, \cdots, a_n, a_1 \oplus b, a_2 \oplus b, \cdots, a_n \oplus b\}$ 对异或运算也满足封闭性。

由此可知，如果自然数集 $A = \{a_1, a_2, \cdots, a_n\}$ 对异或运算满足封闭性，则 $n$ 一定为 $2$ 的幂，即存在自然数 $k$ 使得 $n = 2^k$，并且能够在 $A$ 中找到 $k$ 个数 $a_{m_1}, a_{m_2}, \cdots, a_{m_k}$，使得任选其中 $0$ 到 $k$ 个数，将它们全部异或起来的结果能够得到集合 $A$，即 $a_{m_1}, a_{m_2}, \cdots, a_{m_k}$ 这 $k$ 个数能够通过异或运算生成 $a_1, a_2, \cdots, a_n$ 这 $n$ 个数。

借用向量空间的概念，每一个数视作一个向量，异或运算视作向量加法，自然数集 $A = \{a_1, a_2, \cdots, a_n\}$ 对异或运算满足封闭性，等价于它是一个向量空间，能在其中找到 $k$ 个数 $a_{m_1}, a_{m_2}, \cdots, a_{m_k}$ 作为基向量张成向量空间 $A$。

## 思路一：

从小到大遍历集合 $A$ 中的元素。若 $A$ 中最小的元素不为 $0$，说明 $A$ 对异或运算不满足封闭性。

若 $A$ 中最小的元素为 $0$，维护一个对异或运算满足封闭性的集合 $B = \{b_1, b_2, \cdots, b_m\}$，初始时 $B = \{0\}, m = 1$。

在遍历 $A$ 中元素的过程中，每遇到一个不在集合 $B$ 中的数 $a_i$ 时，进行对 $B$ 的扩增：将 $a_i$ 与 $B$ 中的每个数分别进行异或运算，并且在 $A$ 中查找运算结果是否存在，若存在则将其加入到集合 $B$ 中，若不存在说明 $A$ 对异或运算不满足封闭性。每次成功扩增 $B$ 中元素个数 $m$ 变为之前的 $2$ 倍。

由之前分析的性质可知，若 $A$ 对异或运算满足封闭性，则应该能够成功对 $B$ 进行 $\log_2 n$ 次扩增，第 $k$ 次扩增增加了 $2^{k-1}$ 个数，最终 $B$ 应与 $A$ 相同，$m = n$。

时间复杂度分析：每次增加一个数时查找步骤可以用二分查找实现，复杂度为 $O(\log n)$，$k$ 次扩增一共增加了 $n - 1$ 个数，时间复杂度为 $O(n \log n)$，在最初需要对 $A$ 进行排序，时间复杂度为 $O(n \log n)$，因此总的时间复杂度为 $O(n \log n)$。

## 示例代码 - 1

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int n, a[1000005], b[1000005], h[1000005]; //h[i]记录a[i]是否已经增加到B之中
int cmp(const void *p, const void *q)
{
    return *(int*)p - *(int*)q; //由于输入均为自然数，做减法不会溢出int范围
}
int judge()
{
    if(n & (n - 1)) return 0; //若n不是2的幂，则A对异或运算不满足封闭性
    qsort(a, n, sizeof(int), cmp); //按从小到大的顺序排序
    if(a[0]) return 0; //若A中最小数不为0，则A对异或运算不满足封闭性
    memset(h, 0, sizeof(h)); //将h数组重新置0
    b[0] = a[0]; //初始化B={0}
```

```
    h[0] = 1; //记录a[0]=0已经被加入到B之中
    int m = 1; //初始化m=1
    for(int i = 1; i < n; ++i)
    {
        if(h[i]) continue; //若a[i]已经在B之中，跳过。
        //开始扩增
        for(int j = 0; j < m; ++j)
        {
            int key = b[j] ^ a[i]; //计算b[j]与a[i]异或的结果
            int *p = (int*)bsearch(&key, a, n, sizeof(int), cmp); //在A中查找
b[j]^a[i]
            if(p == NULL) return 0; //没有找到，说明A对异或运算不满足封闭性
            //成功找到，此时p为数组a中该元素的地址
            b[m + j] = *p; //将其加入到B中
            h[p - a] = 1; //标记该元素已被加入到B之中
        }
        m <<= 1; //更新m为原来的2倍
    }
    return 1; //如果每次扩增都成功，说明A对异或运算满足封闭性
}
int main()
{
    while(~scanf("%d", &n))
    {
        for(int i = 0; i < n; ++i)
            scanf("%d", &a[i]);
        puts(judge() ? "YaHaHa!" : "DaKiki.");
    }
    return 0;
}
```

## 思路二

维护一个数组 $e$ 记录基向量组，要求 $e[i]$ 对二进制表示中比 $i$ 低的位没有影响，即 $e[i]$ 的二进制表示中低于 $i$ 的位均为 $0$。

每次读入一个 $x$，利用 $x$ 的二进制表示判断 $x$ 能否被当前基向量组 $e$ 表示，如果 $x$ 的第 $i$ 位为 $1$，若 $e[i]$ 有值，则将去掉 $x$ 中 $e[i]$ 的分量，即令 $x$ 与 $e[i]$ 进行按位异或；若 $e[i]$ 没有值，则使 $x$ 作为一个基向量，即令 $e[i]$ 等于 $x$。

统计在上述过程中向数组 $e$（基向量组）中添加的向量个数 $cnt$，读入完所有 $x$ 并更新数组 $e$ 之后，此时能够保证任意一个向量 $x$ 能够被 $e$ 中的向量表出，此时只需要判断给出的向量组（给出的 $n$ 个自然数）是否能够覆盖整个基向量组张成的向量空间，即 $n$ 是否等于 $2^{cnt}$。

每次读入 $x$ 并维护更新数组 $e$ 的时间复杂度为 $O(\log x)$，一共读入 $n$ 个 $x$，因此该方法时间复杂度为 $O(n \log M)$，其中 $M$ 为 $x$ 的最大值。

## 示例代码 - 2

```
#include <stdio.h>
int insert(int e[], int x) //维护更新数组e（基向量组），返回e中基向量个数是否增加
{
    for(int i = 0; x; ++i)
        if(x >> i & 1)
            if(e[i])
```

```
                x ^= e[i];
            else
            {
                e[i] = x;
                return 1;
            }
    return 0;
}
int main()
{
    int n;
    while(~scanf("%d", &n))
    {
        int cnt = 0, e[31] = {0};
        for(int i = 0; i < n; ++i)
        {
            int x;
            scanf("%d", &x);
            cnt += insert(e, x);
        }
        puts(n == 1 << cnt ? "YaHaHa!" : "DaKiki.");
    }
    return 0;
}
```

## 补充

感兴趣的同学可以了解一下[线性基 - OI Wiki (oi-wiki.org)](#)。

*Author:* 哪吒

# - End -