

目录

一、 L1212：汉明距离 ☆	1
二、 L1507：值日 ☆	3
三、 L1055：合并水果 ☆☆	5
四、 L1530：二维数组前缀和 ☆☆	7
五、 L1424：IP地址 ☆☆	9
六、 L1426：学生统计 ☆☆	11
七、 L1491：字符串基础2 ☆☆	13
八、 L1532：赌徒 ☆☆☆	15
九、 L1499：宽度搜索1 ☆☆☆	17
十、 L1500：搜索2 ☆☆☆	19
十一、 L1506：数字选取 ☆☆☆	21
十二、 L1509：二叉树 ☆☆☆	23

一、 L1212：汉明距离 ☆

题目描述

汉明距离是指**两个整数对应二进制位不同的数量**。

例如，如果 $x = 1$ (二进制表示为 0001)， $y = 4$ (二进制表示为 0100)，那么它们的汉明距离是 2。

输入输出格式

输入：一行两个数 x y ，表示需要计算的两个整数。其中 ($0 \leq x, y \leq 2^{30}$)

输出： x 和 y 的汉明距离。

输入示例	输出示例
1 4	2

样例解释

$$x = 1_{10} = 0001_2$$

$$y = 4_{10} = 0100_2$$

在二进制下， x 和 y 的表示如下：

位数	x	y	是否相同
第1位	1	0	不同
第2位	0	0	相同
第3位	0	1	不同
第4位	0	0	相同

x 和 y 二进制下有两处不同，所以汉明距离为 2。

背景知识

位运算是对整数在二进制表示下直接对位进行的操作：

- 按位与 ($\&$)：两位都为1时结果为1
- 按位或 ($|$)：两位至少有一个为1时结果为1
- 按位异或 ($^ / \oplus$)：两位不同时结果为1，相同时结果为0
- 左移 ($<<$)：将所有位向左移动，右边补0
- 右移 ($>>$)：将所有位向右移动，左边补0

算法分析：使用位运算计算汉明距离

1. 使用异或运算找出不同位

使用 **异或** (\oplus) 运算的特性：相同为0，不同为1

计算 $x \oplus y$ ，使得 x 和 y 不同的部分标记为1，相同的部分标记为0。

```
int t = x ^ y; // 计算异或结果
```

2. 使用与运算检查最后一位

使用 **&** 运算判断最后一位是否为1：

对于 $(t \& 1)$ ，如果值为1，则说明二进制下的 t 在最后一位是1（因为只有 $1\&1 = 1$ ）。

如果值为0，则说明二进制下的 t 在最后一位是0。

```
cnt += t & 1; // 检查最后一位是否为1
```

3. 使用右移运算逐位处理

使用 **右移** ($>>$) 运算逐位统计1的个数：

通过循环处理最后一位，如果是1（意味着此处 x 和 y 不同）则计数器加1，

然后将数字右移一位，直到数字变为0。

示例：计算 $x = 1$, $y = 4$ 的汉明距离

- 计算 $x \oplus y$:

$$x = 1_{10} = 0001_2$$

$$y = 4_{10} = 0100_2$$

$$x \oplus y = 0101_2$$

- 统计 $x \oplus y$ 中1的个数：

当前目标值	最后一位数值	计数器数值
0101	1 (x 和 y 不同)	1
010	0 (x 和 y 相同)	1
01	1 (x 和 y 不同)	2
0	0 (x 和 y 相同)	2

核心代码总览

```
int t = x ^ y; // 计算异或结果
int cnt = 0;    // 计数器
while (t) {     // 当目标值为0时终止
    cnt += t & 1; // 检查最后一位是否为1
    t >>= 1;      // 右移一位
}
```

二、 L1507：值日 ☆

题目描述

小杨和小红是值日生，负责打扫教室。小杨每 m 天值日一次，小红每 n 天值日一次。今天他们两个一起值日，请问至少多少天后，他们会再次同一天值日？

输入输出格式

输入：第一行，一个正整数 m ，表示小杨的值日周期；

第二行，一个正整数 n ，表示小红的值日周期。

输出：一行，一个整数，表示至少多少天后他们会再次同一天值日。

输入示例	输出示例
2 3	6

样例解释

小杨的值日周期 $m = 2$ ，小红的值日周期 $n = 3$ 。

从第0天开始，他们各自的值日安排如下表所示：

天数	0	1	2	3	4	5	6
小杨	是	否	是	否	是	否	是
小红	是	否	否	是	否	否	是

从表中可以看出，他们下一次同时值日是第6天，输出结果为6。

背景知识

- **最小公倍数（LCM）**：两个或多个整数公有的倍数中最小的一个。例如，2和3的最小公倍数是6 ($\text{LCM}(2, 3) = 6$)，4和8的最小公倍数是8 ($\text{LCM}(4, 8) = 8$)。
- **最大公约数（GCD）**：两个或多个整数共有约数中最大的一个。例如，2和3的最大公约数是1 ($\text{GCD}(2, 3) = 1$)，12和18的最大公约数是6 ($\text{GCD}(12, 18) = 6$)。
- **重要关系**：两个整数的最小公倍数与最大公约数满足：

$$\text{GCD}(m, n) \times \text{LCM}(m, n) = m \times n$$

算法分析

1. 问题转化

小杨值日的日期是 $0, m, 2m, 3m, \dots$ ，小红值日的日期是 $0, n, 2n, 3n, \dots$ 。因此，小杨和小红再次同一天值日的天数必须**既是 m 的倍数，又是 n 的倍数**，从而得出最早的一天就是 m 和 n 的**最小公倍数** ($\text{LCM}(m, n)$)。

2. 计算LCM(m, n)

- (1) 使用 `_gcd()` 计算 $\text{GCD}(m, n)$
- (2) 通过最小公倍数和最大公约数关系 ($\text{GCD} \times \text{LCM} = m \times n$) 计算 $\text{LCM}(m, n)$

```
int GCD = _gcd(n, m);
int LCM = m * n / GCD;
```

3. 欧几里得算法计算GCD(m, n)

证明: $\text{GCD}(m, n) = \text{GCD}(n, m \% n)$

(1) 基本情况

假设 $m > n$, 如果 m 能被 n 整除, 那么 $\text{GCD}(m, n) = n$ 。

(2) 递推关系

如果 m 不能被 n 整除, 我们可以得到关系式: $m = n \times k + r$ ($0 < r < n$)

其中: k 是 $m \div n$ 的商, r 是余数。

例如: $m = 16, n = 6$ 时, $16 = 6 \times 2 + 4$, 所以 $k = 2, r = 4$ 。

(3) 关键证明

设 $d = \text{GCD}(m, n)$, 则: d 能整除 m 和 n , 因此 d 也能整除 $m - n \times k = r$,
所以 d 是 n 和 r 的公约数。

因此我们得到: $d = \text{GCD}(m, n) = \text{GCD}(n, r) = \text{GCD}(n, m \% n)$

算法步骤:

(1) 计算 $\text{GCD}(m, n)$, 当 $n \neq 0$ 时, 重复以下步骤:

- i 计算余数 $r = m \% n$, 令 $m = n$, $n = r$
- ii 当 $n = 0$ 时, m 即为最大公约数

代码实现:

```
while (n != 0) {
    int r = m % n;
    m = n;
    n = r;
}
return m;
```

示例: 计算 $\text{GCD}(48, 18)$

第一步: $48 \div 18 = 2 \cdots\cdots 12$, 所以 $\text{GCD}(48, 18) = \text{GCD}(18, 12)$

第二步: $18 \div 12 = 1 \cdots\cdots 6$, 所以 $\text{GCD}(18, 12) = \text{GCD}(12, 6)$

第三步: $12 \div 6 = 2 \cdots\cdots 0$, 余数出现0, 程序终止, 所以 $\text{GCD}(12, 6) = 6$

最终结果: $\text{GCD}(48, 18) = 6$

拓展思考

- **整数溢出问题:** 如果 n 和 m 很大, 直接计算 $m \times n$ 可能会导致整数溢出。
- **多人情况:** 如果是3个人、4个人, 甚至是10000个人求下次同时值日的天数。

三、 L1055：合并水果 ☆☆

题目描述

在一个果园里，张力恒同学已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。张力恒决定把所有的果子合成一堆。

每一次合并，张力恒可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过 $n - 1$ 次合并之后，就只剩下一堆了。张力恒在合并果子时总共消耗的体力等于每次合并所耗体力之和。

因为还要花大力气把这些果子搬回家，所以张力恒在合并果子时要尽可能地节省体力。假定每个果子重量都为 1，并且已知果子的种类数和每种果子的数目，你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。

输入输出格式

输入：第一行，一个整数 $n(1 \leq n \leq 10000)$ ，表示果子的种类数；第二行，包含 n 个整数，用空格分隔，第 i 个整数 $a_i(1 \leq a_i \leq 20000)$ ，表示第 i 种果子的数目。输出：一个整数，也就是最小的体力耗费值。输入数据保证这个值小于 2^{31} 。

输入示例	输出示例
3	15
1 2 9	

样例解释

步骤	当前果堆状态	合并的果堆	本次消耗体力	累计消耗体力
初始状态	1, 2, 9	-	0	0
第一步	1, 2, 9	$1 + 2 = 3$	3	3
第二步	3, 9	$3 + 9 = 12$	12	15

算法分析

1. 问题分析

要使**总体力消耗最小**，每次需要合并**重量最小**的两堆果子，重量小的果子被合并次数多，重量大的果子被合并次数少，总体力消耗最小。

2. 算法选择

(1) **朴素思想**：每次从果堆中选出**最小**的两堆进行合并，再将合并后的重量插入数组并**重新排序**，重复操作直到只剩一堆。每次排序花费时间为 $O(n \log n)$ ，一共进行 $n - 1$ 次合并，所以总体时间复杂度为 $O(n^2 \log n)$ ，无法在 1s 内执行完。

(2) **使用优先队列 (priority_queue) 优化**：使用**最小堆**维护**最小元素**，每次取出和插入元素均 $O(\log n)$ ，一共进行 $n - 1$ 次取出和插入，因此总体复杂度 $O(n \log n)$ ，可以在 1s 内执行完毕。

(3) 实现思路

- 将所有果堆的重量放入**最小堆**中
- 每次取出两个**最小元素**, 合并累加**体力消耗**
- 将合并后的堆重量放回堆中, 重复直到堆只剩一个元素

背景知识

- **优先队列（堆）**：一种特殊的数据结构，可以快速获取和删除**最小（或最大）元素**。

```
priority_queue<int, vector<int>, greater<int>> pq;
```

- 第一个参数 `int`: 指定队列中存储的元素类型
- 第二个参数 `vector<int>`: 指定底层使用的容器, 通常是vector
- 第三个参数 `greater<int>`: 比较函数, 决定元素的排列顺序
 - * `less<int>`: **最大堆**, 最大的元素在顶部(默认值)
 - * `greater<int>`: **最小堆**, 最小的元素在顶部

常用操作:

操作	功能说明	时间复杂度
<code>pq.push(x)</code>	将元素 <code>x</code> 插入堆	$O(\log n)$
<code>pq.pop()</code>	删除堆顶部(最小或最大)元素	$O(\log n)$
<code>pq.top()</code>	获取堆顶部元素	$O(1)$
<code>pq.empty()</code>	判断堆是否为空	$O(1)$
<code>pq.size()</code>	获取堆中元素个数	$O(1)$

核心代码

```
while (pq.size() >= 2) { // 重复操作直到堆中只剩一个元素
    int u = pq.top();
    pq.pop();
    int v = pq.top();
    pq.pop();
    cnt += u + v; // 取出两个最小的元素, 合并它们, 并累加体力消耗
    pq.push(u + v); // 合并结果放回堆中
}
```

拓展思考

- 如果每次合并三堆、四堆, 甚至 $k(k \leq 10^5)$ 堆果子, 算法该如何修改?
- 如果要求改成求消耗体力的**最大值**, 该如何实现?

四、 L1530：二维数组前缀和 ☆☆

题目描述

给定一个 n 行 m 列的二维数组，你需要计算该数组的前缀和。

所谓前缀和是指：对于每个位置 (i, j) ，求出从左上角 $(0, 0)$ 到当前位置 (i, j) 的所有元素的和。

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \Rightarrow s = \begin{bmatrix} 1 & 3 & 6 \\ 5 & 12 & 21 \\ 12 & 27 & 45 \end{bmatrix}$$

输入输出格式

输入：第一行包含两个整数 n, m ($1 \leq n, m \leq 1000$)，表示二维数组的行数和列数。

接下来的 n 行，每行包含 m 个整数，表示二维数组的元素。

每个元素的值满足 $-10^3 \leq a[i][j] \leq 10^3$ 。

输出：输出 n 行，每行包含 m 个整数，表示前缀和数组对应位置的值。

每行数之间用一个空格分隔，且每行末尾不要有空格。

输入示例	输出示例
3 3 1 2 3 4 5 6 7 8 9	1 3 6 5 12 21 12 27 45

样例解释

以计算 $s[1][2]$ (第二行第三列) 为例，演示二维前缀和的计算过程：

$$\begin{aligned}s[1][2] &= a[0][0] + a[0][1] + a[0][2] + a[1][0] + a[1][1] + a[1][2] \\&= 1 + 2 + 3 + 4 + 5 + 6 \\&= 21\end{aligned}$$

算法分析

1. 直接计算方法

对于每个位置 (i, j) ，直接计算从 $(0, 0)$ 到 (i, j) 的所有元素之和：

```
for(int x = 0; x <= i; x++) {  
    for(int y = 0; y <= j; y++) {  
        s[i][j] += a[x][y];  
    }  
}
```

这种方法的时间复杂度为 $O(n^2 \cdot m^2)$ ，会导致超时。

2. 优化方法：前缀和

利用已计算的前缀和来计算新的前缀和，可以大大降低时间复杂度。以下以 $s[2][2]$ 的计算为例进行说明。

(1) 区域划分

我们用不同颜色表示不同的前缀和区域， $s[2][2] =$ 整个矩形内所有数字的和：

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

- 蓝色部分的数字和表示 $s[1][2]$ ，红色部分的数字和表示 $s[2][1]$ ，绿色部分的数字和表示 $s[1][1]$ （红蓝重叠的部分），橘色部分的数字表示 $a[2][2]$ 。
- 整个矩形的和 = 蓝色区域 + 红色区域 - 绿色区域 + 橘色区域
- 用符号表示就是： $s[2][2] = s[1][2] + s[2][1] - s[1][1] + a[2][2]$
- 得出通用公式： $s[x][y] = s[x-1][y] + s[x][y-1] - s[x-1][y-1] + a[x][y]$
当前位置前缀和 = 左前缀和 + 上前缀和 - 左上角前缀和 + 当前位置的值
- 我们利用这个公式，从左上角到右下角的顺序，计算 $s[i][j]$ ，就可以得到所有 $s[x][y]$ 的值。

代码实现：

处理前缀和

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        s[i][j] += s[i - 1][j] + s[i][j - 1] -
                    s[i - 1][j - 1] + a[i - 1][j - 1];
    }
}
```

输出格式

```
//注意行末不能有多余的空格
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        cout << a[i][j] << " \n"[j == m - 1];
    }
}
```

拓展思考

- 如果换成三维，如何求 $a[i][j][k]$ 的前缀和？
- 如果要求从 (x_1, y_1) 到 (x_2, y_2) 的区域和，该如何计算？

五、 L1424 : IP地址 ☆☆

题目描述

给定一个字符串 s , 请你判断它是否是一个有效的 IPv4 地址。

一个有效的 IPv4 地址由四个整数构成, 这些整数位于 0 到 255 之间, 用 ‘.’ 分隔。

每个整数不能有前导零 (除非该整数本身就是 0), 并且每个部分必须是非空的。

输入输出格式

输入: 输入包含多组测试数据。每组测试数据占一行, 包含一个字符串 s ($1 \leq |s| \leq 15$)。输入以文件结束符 (EOF) 结束。

输出: 对于每组测试数据, 如果输入的字符串 s 是一个有效的 IPv4 地址, 则输出 "YES"; 否则输出 "NO"。每个输出占一行。

输入示例	输出示例
192.168.1.1	YES
192.168.1.01	NO
0.0.0.0	YES
255.255.255.255	YES

样例解释

- **192.168.1.1:** 所有部分都在 0-255 范围内, 没有前导零, 格式正确, 输出 YES
- **192.168.1.01:** 第四部分 "01" 有前导零, 不符合要求, 输出 NO
- **0.0.0.0:** 所有部分都是 0, 允许单个 0, 输出 YES
- **255.255.255.255:** 所有部分都在 0-255 范围内, 没有前导零, 格式正确, 输出 YES

算法分析

1. 验证条件

- 恰好包含 3 个点号 '.', 分成 4 个部分
- 每部分为 0-255 的整数且无前导零

2. 实现思路

- (1) 按 '.' 分割字符串为 4 部分
- (2) 每部分是否都是数字且大小在 0-255 范围内
- (3) 无前导零

代码实现：

判断逻辑：

```
bool check (string s) {
    if (s.size() > 3 || s.size() == 0) return false;
    //超出3位数，或者是空的直接不满足[0,255]条件
    for (int i = 0; i < s.size(); i++) {
        if (s[i] > '9' || s[i] < '0') return false;
        //验证都是数字
    }
    for (int i = 0; i < s.size(); i++) { //验证无前导零
        if (s[i] == '0' && s.size() > 1) {
            return false; //除了"0"之外，字符串开头是0的都不合法
        } else if (s.size() > 1) {
            break;
        }
    }

    int num = stoi(s); //字符串转换为数字
    if (num < 0 || num > 255) return false; //判断是否在[0,255]范围内
    return true;
};
```

字符串处理：

```
while (cin >> s) {
    string t;
    bool ok = true;
    for (int i = 0; i < s.size(); i++) {
        if (s[i] == '.') { // 遇到分割符号，检查当前部分是否满足要求
            ok &= check(t);
            t.clear();
        } else {
            t += s[i];
        }
    }

    ok &= check(t); // 检查最后一部分

    if (ok) {
        cout << "YES\n";
    } else {
        cout << "NO\n";
    }
}
```

拓展思考：

- 给定任意一个字符串和一个分割的字符集，如何实现一个通用的字符串分割函数？

六、 L1426：学生统计 ☆☆

题目描述

快开学了，小徐同学想帮老师统计一下本学期学校的学生信息。

老师给了一张学生信息表，每一条数据包括学生姓名、年级、班级。

老师想知道本学期每个年级有多少个班以及多少个人。

输入输出格式

输入：第一行，一个整数 N ，表示有 N 条数据；接下来 N 行，每行包含三个字符串，分别表示姓名、年级、班级。

输出：按年级从小到大，输出年级的班级数和总人数。（注意，年级、班级总数、年级总数之间有一个空格，每行末尾不要有空格）

输入示例	输出示例
5 Tom G7 A Jerry G7 B Llice G7 A Bob G8 A Apple G8 A	G7 2 3 G8 1 2

样例解释

年级	班级列表	班级数	学生人数
G7	A班： Tom、 Llice B班： Jerry	2	3
G8	A班： Bob、 Apple	1	2

算法分析

1. 问题分析

需要统计**每个年级的班级数**和**学生总数**。同一个年级的不同班级需要去重统计，而同一个年级的学生人数直接累加。

2. 算法选择

- (1) **解法一 - 排序 + 遍历统计**：先将所有学生信息按**年级**和**班级**排序，将年级小的放在前面，年级相同的按照班级排序，然后遍历统计每个年级的信息。
- (2) **解法二 - Map/Set 统计**：使用 `map` 和 `set` 容器自动维护有序性和去重特性，将年级作为键，班级,学生人数作为值

3. 实现思路

• 法一：结构体排序 + 遍历统计

- (1) 定义结构体存储学生信息（姓名、年级、班级）

```
struct Node {  
    string g;  
    string c;  
    string n;  
};  
  
bool cmp(Node a, Node b) {  
    if (a.g == b.g) return a.c < b.c;  
    return a.g < b.g;  
}
```

- (2) 按年级升序、班级升序排序

- (3) 遍历排序后的数据，统计每个年级的班级数和学生数

```
if (g != a[i].g) { // 枚举到了新的年级  
    cout << g << " " << cnt_class << " "  
        << cnt_stu << endl; // 先输出上一个年级的信息  
    g = a[i].g, c = a[i].c;  
    cnt_class = 1, cnt_stu = 0; // 数据清零  
}  
cnt_stu++;  
if (c != a[i].c) { // 遇到新的班级，班级数加1  
    c = a[i].c;  
    cnt_class++;  
}
```

• 法二：map + set 统计

- (1) 使用 `map<string, int>` 统计每个年级的学生总数

```
map<string, int> s; // 年级 -> 学生数  
s[grade]++;
```

- (2) 使用 `map<string, set<string>>` 统计每个年级的班级集合（自动去重）

```
map<string, set<string>> c; // 年级 -> 班级集合  
c[grade].insert(clazz);
```

- (3) 遍历 map 输出结果

```
for (auto i : s) {  
    cout << i.first << " " << c[i.first].size() << " "  
        << i.second << endl;  
}
```

拓展思考

- 如果要求按班级数从大到小输出，该如何修改排序规则？
- 如果数据中存在重复记录（相同姓名、年级、班级），该如何去重？

七、 L1491：字符串基础2 ☆☆

题目描述

一串长度不超过 255 的 PASCAL 语言代码，只有 a, b, c 三个变量，而且只有赋值语句，赋值只能是一个一位的数字或一个变量，每条赋值语句的格式是 [变量]:=[变量或一位整数];。未赋值的变量值为 0，输出 a, b, c 的值。

输入输出格式

输入：一串符合语法的 PASCAL 语言代码，只有 a, b, c 三个变量，而且只有赋值语句，赋值只能是一个一位的数字或一个变量，未赋值的变量值为 0。

输出：输出 a, b, c 最终的值。

输入示例	输出示例
a:=3;b:=4;c:=5;	3 4 5
a:=3;b:=a;c:=a;	3 3 3
a:=3;b:=a;	3 3 0

样例解释

1. a:=3;b:=4;c:=5;

变量 a 被赋值为 3，变量 b 被赋值为 4，变量 c 被赋值为 5，所以输出 3 4 5。

2. a:=3;b:=a;c:=a;

变量 a 被赋值为 3，变量 b 被赋值为 a 的值（3），变量 c 被赋值为 a 的值（3），所以输出 3 3 3。

3. a:=3;b:=a;

变量 a 被赋值为 3，变量 b 被赋值为 a 的值（3），变量 c 没有被赋值（保持初始值 0），所以输出 3 3 0。

算法分析

1. 问题分析

本题需要提取字符串中有效的部分，并跟踪三个变量 a, b, c 的值变化。赋值语句的格式固定，可以按分号分割语句，然后逐个解析。

2. 关键步骤

- (1) 初始化变量：创建三个变量 a, b, c，初始值都为 0。
- (2) 分割语句：将输入字符串按分号分割成多个赋值语句。
- (3) 解析每条语句：对于每条语句，提取左侧变量和右侧值。

- (4) **赋值处理:** 如果右侧是数字, 直接赋值; 如果右侧是变量, 则使用该字母对应的数值。
- (5) **输出结果:** 处理完所有语句后, 输出 a, b, c 的最终值。

3. 字符串处理技巧

- 使用 `map <char, int>` 记录字母对应的数值 (默认初始值为 0)。

```
map<char, int> mp;
mp['a'] = 0; mp['b'] = 0; mp['c'] = 0;
```

- 使用 `map <char, int>` 将 **字符** (char 类型) 和 **数字** (int 类型) 等价。例如 `mp['1'] = 1` (前者是 char 类型, 后者是 int 类型)。

```
for (int i = 0; i < 10; i++) {
    mp['0' + i] = i;
}
```

- 根据一个赋值语句 `【a := 3;】` 占据 5 个字符, 因此把**5 个字符为一个完整的赋值语句**。在每次循环中, `a` 是在0位置, `b` 是在3位置, 所以只需要将 `mp[s[0]]` 赋值为 `mp[s[3]]` 即可。

```
for (int i = 0; i < s.size(); i += 5) {
    mp[s[i]] = mp[s[i + 3]];
}
```

拓展思考

- 如果**变量名不止 a, b, c 三个**, 如何扩展程序?
- 如果**赋值语句不止 3 句**, 如何处理?

八、 L1532：赌徒 ★★★

题目描述

有 n 个赌徒打算赌一局。规则是：每人下一个赌注，赌注为非负整数，且任意两个赌注都不相同。胜者为赌注恰好是其余任意三个人的赌注之和的那个人。如果有多个胜者，我们取赌注最大的那个为最终胜者。

例如， A, B, C, D, E 分别下赌注为 2、3、5、7、12，最终胜者是 E ，因为 $12 = 2 + 3 + 7$ 。

输入输出格式

输入：输入包含多组测试数据。每组首先输入一个整数 n ($1 \leq n \leq 1000$)，表示赌徒的个数。接下来 n 行每行输入一个非负整数 b ($0 \leq b < 32768$)，表示每个赌徒下的赌注。当 $n = 0$ 时，输入结束。

输出：对于每组输入，输出最终胜者的赌注，如果没有胜者，则输出 no solution。

输入示例	输出示例
5	12
2	no solution
3	
5	
7	
12	
5	
2	
16	
64	
256	
1024	
0	

样例解释

1. 第一组数据：赌注为 2、3、5、7、12

$12 = 2 + 3 + 7$ ，满足条件，所以输出 12。

2. 第二组数据：赌注为 2、16、64、256、1024

没有任何一个赌注等于其他三个赌注之和，所以输出 no solution。

算法分析

1. 问题分析

我们需要找最大的 x ，使得存在三个数字 y, z, w ，使得 $x = y + z + w$ 。如果使用暴力做法，分别枚举 x, y, z, w ，时间复杂度会达到 $O(n^4)$ ，超时。

2. 优化思路

- **预处理:** 枚举 y, z 通过二重循环得到所有的 $y + z$ 和，并记录每个和出现的次数为 $cnt[y + z]$ 。
- **枚举优化:** 再枚举 x, w ，通过公式 $x = y + z + w$ ，得到 $y + z = x - w$ ，在预处理的和中查找是否存在 $x - w$ 。
- **避免重复:** 使用判断逻辑来避免重复使用同一个元素：
 - 避免重复元素的逻辑分析：
对于等式 $x - w = y + z$ ，我们需要确保 x, y, z, w 是四个不同的赌注。
当 w 被重复使用时，会出现以下情况：

$$x - w = y + z$$

$$\text{如果 } y = w \Rightarrow x - w = w + z \Rightarrow x - 2w = z$$

- * 如果 $x - 2w > 0$ 且 $cnt[x - 2w] = 1$ ，则必然会出现 w 被重复使用的情况
- * 如果 $x - 2w > 0$ 且 $cnt[x - 2w] > 1$ ，则存在多组 $\{a, b\}$ 满足条件，可以避免重复
- * 如果 $x - 2w \leq 0$ ，则不会出现重复问题

代码实现

预处理 $y + z$

```
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        if (i == j) continue;
        s[a[i] + a[j]]++;
    }
}
```

枚举 x, w

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (i == j) continue;
        int ns = a[i] - a[j]; // x - w
        int t = a[i] - 2 * a[j]; // z = x - 2w
        if (ns <= 0) continue;
        if (s[ns] <= 0) continue;
        if (t > 0 && s[ns] <= cnt[t]) continue;
        res = max(res, a[i]);
    }
}
```

拓展思考

- 如果改成七个人改成寻找满足的 $A = B + C + D + E + F + G$ 该如何实现？

九、 L1499：宽度搜索1 ★★★

题目描述

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

马每次走时，只能根据象棋的规则，走日字形状。

输入输出格式

输入：一行四个整数，分别为 n, m, x, y ($1 \leq x \leq n \leq 400, 1 \leq y \leq m \leq 400$)。

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

输入示例	输出示例
3 3 1 1	0 3 2 3 -1 1 2 1 4

样例分析

以输入 3 3 1 1 为例：

0	3	2
3	-1	1
2	1	4

- (1,1) 是起点，距离为0
- (1,2) 需要3步：(1,1)→(3,2)→(1,3)→(2,1)→(1,2)
- (1,3) 需要2步：(1,1)→(2,3)→(1,3)
- (2,1) 需要3步：(1,1)→(3,2)→(1,3)→(2,1) 或 (1,1)→(2,3)→(3,1)→(2,1)
- (2,2) 无法到达
- (2,3) 需要1步：(1,1)→(2,3)
- (3,1) 需要2步：(1,1)→(2,3)→(3,1) 或 (1,1)→(3,2)→(3,1)
- (3,2) 需要1步：(1,1)→(3,2)
- (3,3) 需要4步：(1,1)→(3,2)→(1,3)→(2,1)→(3,3)

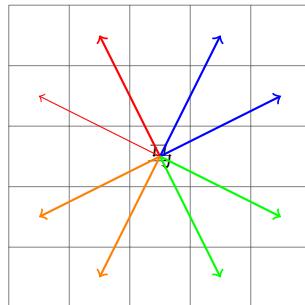
算法分析

1. 问题分析

这是一个典型的**最短路径**问题，由于棋盘上每个位置的移动代价相同（每次移动一步），可以使用**广度优先搜索（BFS）**来解决。

2. 马走日的8个方向

在国际象棋中，马走”日”字，有8个可能的移动方向：



用坐标表示这8个方向：

```
int dx[8] = {-2, -1, 1, 2, 2, 1, -1, -2};  
int dy[8] = {1, 2, 2, 1, -1, -2, -2, -1};
```

3. BFS算法步骤

- (1) 初始化距离数组，所有位置设为-1（表示未访问）
- (2) 将起始位置加入队列，距离设为0
- (3) BFS遍历：

- 从队列中取出当前位置
- 遍历8个方向，计算下一个位置
- 如果新位置在棋盘内且未被访问，更新距离并入队

代码实现

D E C O D I N G T H E F U T U R E

```
while (!q.empty()) {  
    array<int, 2> TMP = q.front();  
    int cx = TMP[0], cy = TMP[1];  
    q.pop();  
    for (int i = 0; i < 8; i++) {  
        int nx = cx + dx[i], ny = cy + dy[i];  
        //判断是否越界  
        if (nx < 0 || nx >= n || ny < 0 || ny >= m) continue;  
        if (res[nx][ny] > res[cx][cy] + 1) {  
            res[nx][ny] = res[cx][cy] + 1;  
            q.push({nx, ny});  
        }  
    }  
}
```

拓展思考

- 如果要求输出具体的行走路径而不仅仅是步数，该如何修改？
- 如果棋盘上有障碍物，且具有拌马腿的限制，该如何处理？

十、 L1500：搜索2 ★★★

题目描述

有一个奇怪的电梯，大楼有 N 层楼，每层楼有一个数字 K_i ($0 \leq K_i \leq N$)。电梯只有四个按钮：开，关，上，下。上下的层数等于当前楼层上的那个数字。如果不能满足要求，相应的按钮就会失灵。

例如：3, 3, 1, 2, 5 代表了 K_i ($K_1 = 3, K_2 = 3, \dots$)，从1楼开始。在1楼，按”上”可以到4楼，按”下”是不起作用的，因为没有-2楼。

那么，从A楼到B楼至少要按几次按钮呢？

输入输出格式

输入：第一行为三个用空格隔开的正整数，

表示 N, A, B ($1 \leq N \leq 200, 1 \leq A, B \leq N$)。

第二行为 N 个用空格隔开的非负整数，表示 K_i 。

输出：一行，即最少按键次数，若无法到达，则输出 -1。

输入示例	输出示例
5 1 5	3
3 3 1 2 5	

样例分析

以输入为例：

D E C O D I N G T H E F U T U R E

楼层	K值
5	5
4	2
3	1
2	3
1	3

最优路径： 1 → 4 → 2 → 5 (3次按键)

1. **第一次按键**：从1楼 ($K_1 = 3$) 按”上”到4楼
2. **第二次按键**：从4楼 ($K_4 = 2$) 按”下”到2楼
3. **第三次按键**：从2楼 ($K_2 = 3$) 按”上”到5楼 (目标)

算法分析

1. 问题分析

这是一个典型的**最短路径**问题，可以使用**广度优先搜索（BFS）**来解决。每个楼层是一个状态，每次按按钮是状态转移。

2. 状态转移

从当前楼层 u , 可以转移到两个可能的楼层:

- 向上: $u + K[u]$ (如果不超过 N)
- 向下: $u - K[u]$ (如果不小于 1)

3. BFS算法步骤

(1) 初始化距离数组, 所有楼层设为-1 (表示未访问)

(2) 将起始楼层加入队列, 距离设为0

(3) BFS遍历:

- 从队列中取出当前楼层
- 如果到达目标楼层, 返回当前距离
- 否则, 尝试向上和向下移动
- 如果新楼层在范围内且未被访问, 更新距离并入队

4. 边界条件处理

- 向上移动: 检查是否不超过 N
- 向下移动: 检查是否不小于 1
- 如果起点和目标相同, 直接返回0

代码实现

```
while (!q.empty()) {  
    int u = q.front();  
    if (u == y) {  
        cout << cost[u];  
        return;  
    }  
    q.pop();  
    for (int i = 0; i < 2; i++) {  
        int v = u + a[u][i];  
        if (v >= n || v < 0) continue;  
        if (cost[v] > cost[u] + 1) {  
            cost[v] = cost[u] + 1;  
            q.push(v);  
        }  
    }  
}
```

拓展思考

- 如果电梯可以有多个按钮, 每个按钮对应不同的移动距离, 该如何解决?
- 如果每次移动的代价不同 (比如上楼比下楼费更多能量), 该如何修改算法?
- 如果要求输出具体的按键序列而不仅仅是最少次数, 该如何实现?

十一、 L1506：数字选取 ☆☆☆

题目描述

给定正整数 n ，现在有 $1, 2, \dots, n$ 共计 n 个整数。你需要从这 n 个整数中选取一些整数，使得所选取的整数中任意两个不同的整数均互质（也就是说，这两个整数的最大公因数为 1）。请你最大化所选取整数的数量。

例如，当 $n = 9$ 时，可以选择 $1, 5, 7, 8, 9$ 共计 5 个整数。可以验证不存在数量更多的选取整数的方案。

输入输出格式

输入：一行，一个正整数 n ，表示给定的正整数。

对于 40% 的测试点，保证 $1 \leq n \leq 1000$ 。对于所有测试点，保证 $1 \leq n \leq 10^5$ 。

输出：一行，一个正整数，表示所选取整数的最大数量。

输入示例	输出示例
6	4

样例解释

当 $n = 6$ 时，可以选择 $1, 2, 3, 5$ 或 $1, 3, 4, 5$ 等方案，都能选取 4 个整数。可以验证无法选取 5 个整数。

算法分析



1. 问题分析

我们需要从 1 到 n 中选取尽可能多的数，使得任意两个数互质。

2. 关键观察

- **数字 1 与任何数都互质：** 1 必须选
- **质数之间必定互质**，因为质数的定义是只有 1 和自身两个正因子
- **一个重要的策略：** 我们可以选择所有的质数和 1，因为它们两两互质
- **合数一定不能选：** 因为合数必然能拆成若干个质数相乘。例如 $6 = 2 \times 3$ ，并且这些质数必然小于该合数，所以这些质数（例如 2, 3）已经选取了，再选取 6 会导致 6 和 3 不互质。

3. 贪心策略

- (1) 选择数字 1（与所有数互质）
- (2) 选择所有不超过 n 的质数（质数之间互质）

4. 试除法判断质数

- **原理:** 检查 2 到 \sqrt{n} 之间是否有 n 的因子
- **时间复杂度:** $O(\sqrt{n})$
- **代码实现:**

```
bool is_prime(int n) {
    if (n < 2) return false;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) return false;
    }
    return true;
}
```

5. 完整算法步骤

- (1) 初始化计数器 $cnt = 1$ (包含数字 1)
- (2) 遍历 2 到 n 的所有整数
- (3) 对每个整数 i , 使用试除法判断是否为质数
- (4) 如果是质数, 计数器 cnt 加 1
- (5) 输出最终的 cnt 值

其他常见的质数筛

方法名称	时间复杂度	空间复杂度	适用场景
试除法	$O(\sqrt{n})$	$O(1)$	单个质数判断
埃拉托斯特尼筛法	$O(n \log \log n)$	$O(n)$	批量质数筛选
欧拉筛法	$O(n)$	$O(n)$	高效批量筛选
米勒-拉宾素性检验	$O(k \log^3 n)$	$O(1)$	大数质数判定
分段筛法	$O(n \log \log n)$	$O(\sqrt{n})$	大范围质数筛选

拓展思考

- 如果数字范围扩大到 10^7 或更大, 算法需要如何优化?
- 还有很多其他的筛选质数的方法, 比如线性筛、欧拉筛等等, 你能解释一下它们的相较于其他方法的优缺点吗?

十二、 L1509 : 二叉树 ★★★

题目描述

小杨有一棵包含 n 个节点的二叉树，且根节点的编号为 1。这棵二叉树任意一个节点要么是白色，要么是黑色。之后小杨会对这棵二叉树进行 q 次操作，每次小杨会选择一个节点，将以这个节点为根的子树内所有节点的颜色反转，即黑色变成白色，白色变成黑色。

小杨想知道 q 次操作全部完成之后每个节点的颜色。

题目描述

小杨有一棵包含 n 个节点的二叉树，且根节点的编号为 1。这棵二叉树任意一个节点要么是白色，要么是黑色。之后小杨会对这棵二叉树进行 q 次操作，每次小杨会选择一个节点，将以这个节点为根的子树内所有节点的颜色反转，即黑色变成白色，白色变成黑色。

小杨想知道 q 次操作全部完成之后每个节点的颜色。

输入输出格式

输入格式：

第一行：正整数 n ，表示节点数量

第二行： $(n - 1)$ 个正整数，表示节点 2 到 n 的父亲节点

第三行：长度为 n 的 01 串，表示初始颜色

第四行：正整数 q ，表示操作次数

接下来 q 行：每行一个正整数 a_i ，表示操作节点

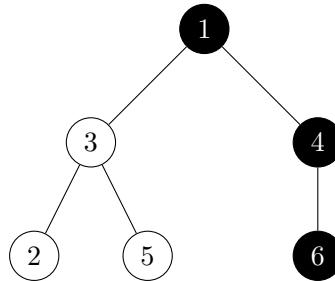
输出格式：

一行：长度为 n 的 01 串，第 i 位为 0 表示节点 i 为白色，第 i 位为 1 表示节点 i 为黑色

输入示例	输出示例
6	010000
3 1 1 3 4	
100101	
3	
1	
3	
2	

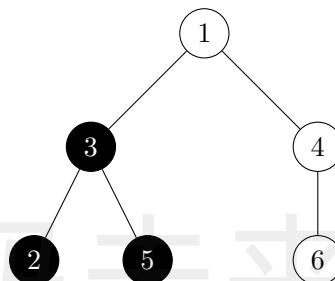
样例解释

1. 初始状态:



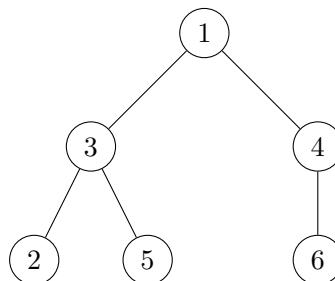
- 节点颜色: 1(1), 2(0), 3(0), 4(1), 5(0), 6(1)
- 颜色序列: 100101

2. 第一次操作 (节点1):



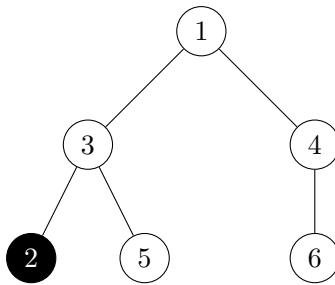
- 反转范围: 整个树 (所有节点)
- 反转后颜色: 1(0), 2(1), 3(1), 4(0), 5(1), 6(0)
- 颜色序列: 011010

3. 第二次操作 (节点3):



- 反转范围: 节点3及其子树 (节点3,2,5)
- 反转后颜色: 1(0), 2(0), 3(0), 4(0), 5(0), 6(0)
- 颜色序列: 000000

4. 第三次操作 (节点2):



- 反转范围: 节点2 (只包含节点2)
- 反转后颜色: 1(0), 2(1), 3(0), 4(0), 5(0), 6(0)
- 颜色序列: 010000

背景知识

在解决树结构相关问题时，图的存储方式至关重要。以下是两种常见的图存储方法：

1. 邻接矩阵

- 使用二维数组 $graph[i][j]$ 表示节点 i 到节点 j 的边

```

vector<vector<int>> graph(n + 1, vector<int>(n + 1, 0));
for (int i = 0; i < m; i++) {
    int u, v, w;
    cin >> u >> v >> w;
    graph[u][v] = w;
    graph[v][u] = w;
}

```

2. 邻接表

- 为每个节点维护一个链表，存储该节点的所有相连的节点

```

vector<vector<int>> adj(n + 1);
for (int i = 0; i < m; i++) {
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}

```

3. 优缺点对比

存储方式	空间复杂度	查询边(u,v)	遍历邻居	优点	缺点
邻接矩阵	$O(n^2)$	$O(1)$	$O(n)$	查询速度快 适合稠密图	空间占用大 不适合稀疏图
邻接表	$O(n + m)$	$O(\deg(u))$	$O(\deg(u))$	空间效率高 适合稀疏图	查询边需遍历 实现稍复杂

算法分析

1. 直接模拟方法

对于每次操作，遍历以该节点为根的子树，反转所有节点的颜色

这种方法的时间复杂度为 $O(n \cdot q)$ ，在 $n, q = 10^5$ 时会超时。

2. 优化方法：标记传递

利用标记数组记录每个节点被反转的次数，最后统一处理。

(1) 核心思想

- 使用一个标记数组 `tag` 记录每个节点被反转的次数
- 对于每次操作，只在操作节点上增加标记
- 通过一次遍历，将父节点的标记传递给子节点
- 如果反转次数是奇数，则反转当前节点，否则不反转

(2) 算法步骤

- 构建二叉树结构（使用邻接表）
- 初始化标记数组 `tag` 为 0
- 对于每个操作，`tag[操作节点]++`
- 使用 BFS 从根节点开始遍历，将父节点的标记累加到子节点

```
queue<pair<int, int>> qu; // 当前节点, 父节点
qu.push({1, 0});
while (!qu.empty()) {
    int u = qu.front().first;
    int fa = qu.front().second;
    qu.pop();

    for (int v : adj[u]) {
        if (v == fa) continue;
        tag[v] += tag[u];
        qu.push({v, u});
    }
}
```

- 计算每个节点的最终颜色

```
for (int i = 1; i <= n; i++) {
    if (tag[i] % 2 == 1) { // 反转颜色
        color[i] = (color[i] == '0') ? '1' : '0';
    }
}
```

拓展思考

- 如果颜色的数目修改为 k 种，且每次操作是将该节点及其子树颜色全部设置为 x ，那么算法该如何修改？
- 如果树不是二叉树，而是多叉树，算法是否仍然适用？