A - 一元二次方程的解数

难度	考点
1	浮点数误差

题目分析

浮点数存在精度误差，比较两浮点数 a, b 的时候，不宜用 `a == b`，而应当用 `|a - b| < eps` 形式，其中 `eps` 为一个合适且接近零的数。

对于本题，三个小数位数均不超过 6，故判断 a 是否为零的时候，eps 不宜超过 10^{-6}；判断 Δ 是否为零的时候，eps 在 $[10^{-12}, 10^{-6}]$ 区间较为合适。

易错点提示

有的同学虽然判断了绝对值，但是 if 语句的顺序却是这么写的：

```
double delta; // 判别式
if (delta > 0)
    printf("2");
else if (fabs(delta) < 1e-10) // WARNING
    printf("1");
else
    printf("0");
```

当 `delta` 是一个很小的正数（比如 10^{-12}）时，即使它满足 `WARNING` 处的分支，也会优先进入第一个分支，进而导致判断失败。编写代码时需要注意这种情况。

示例代码

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

int main()
{
    double a, b, c;
    double delta;

    scanf("%lf%lf%lf", &a, &b, &c);
    if (fabs(a) < 1e-10)
    {
        printf("No");
        return 0;
    }

    delta = b * b - 4 * a * c;
    if (fabs(delta) < 1e-10)
        printf("1");
    else if (delta > 0)
        printf("2");
```

```
    else
        printf("0");

    return 0;
}
```

B - 字节清零

难度	考点
1	位运算

题目分析

解法1

为了消去输入数据从低位往高位的第二个连续 8 位，可以将它和 `11111111 11111111 00000000 11111111` 求位与。在 C 语言中，你不必将这个数字算出来，只要在程序里写 `0xFFFF00FF`（这会是一个 `unsigned int`）就可以了。

解法2

我们可以采用左移与按位或来得到 `0xFFFF00FF`。

易错点提示

1. 对位运算不太熟悉，不知道如何利用位运算构造指定位为 1 的数、取出某数的特定位等基础操作。
2. 不理解整数在内存中的真实形式，额外地写了循环进行手工二进制转换。

示例代码

解法1

```c
#include <stdio.h>

int main()
{
    int a, b;

    scanf("%d", &a);
    b = a & 0xFFFF00FF;
    printf("%d", b);

    return 0;
}
```

解法2代码片段

```c
int n, a = ((1 << 8) | (1 << 9) | (1 << 10) | (1 << 11) | (1 << 12) | (1 << 13) | (1 << 14) | (1 << 15));
scanf("%d", &n);
printf("%d", n & (~a));
```

```c
int n, a = 0, i;
for (i = 8; i <= 15; i++)
    a |= (1 << i);
scanf("%d", &n);
printf("%d", n & (~a));
```

C - Overflow

难度	考点
2	溢出

题目分析

本题考察同学们对于“溢出”的理解，以及如何识别处理“溢出”。

根据提示，可以将数据转化为 `unsigned long long` 求和后进行比较，因为 `unsigned long long` 的范围大约是 `long long` 的两倍，所以它们的和在 `unsigned long long` 中不会溢出（除了下面的一种特殊情况）。

由于 `unsigned long long` 不能存负数，所以我们先对输入数字的符号进行判断：

- 如果是两个正数，那么将其转换为 `unsigned long long` 并相加和判断是否溢出
- 如果是两个负数，那么将其取反后，再转换为 `unsigned long long` 并相加和判断是否溢出
- 如果是一正一负，那么它们相加一定不会溢出（想一想，为什么？）

注意，由于 $(-9223372036854775808)+(-9223372036854775808)=18446744073709551616$ 恰好超出了 `unsigned long long` 的范围，因此这组数据需要进行特判。

示例程序

```c
#include <stdio.h>
#define INF 92233720368547758O7ull // long long 的最大值（类型为 unsigned long long）
#define NINF -9223372036854775808  // long long 的最小值
#define ULL unsigned long long     // 合理运用宏定义可以简化程序

int main()
{
    long long a, b;

    scanf("%lld%lld", &a, &b);

    if (a == NINF && b == NINF)
        printf("NO!\n"); // 特判一下特殊情况
    else if (a > 0 && b > 0 && (ULL)a + b > INF)
        printf("PO!\n"); // 正溢出
    else if (a < 0 && b < 0 && (ULL)(-a) + (-b) > INF + 1)
        printf("NO!\n"); // 负溢出
    else
        printf("%lld\n", a + b);

    return 0;
}
```

示例程序2

```c
#include <stdio.h>
#define MAX 9223372036854775807L
#define MIN -9223372036854775808L

int main()
{
    long long a, b;

    scanf("%lld%lld", &a, &b);

    if (a > 0 && b > 0)
    {
        if (a > MAX - b)
        {
            printf("PO!");
            return 0;
        }
    }
    else if (a < 0 && b < 0)
    {
        if (a < MIN - b)
        {
            printf("NO!");
            return 0;
        }
    }

    printf("%lld", a + b);

    return 0;
}
```

科普：什么是未定义行为

在本题的 HINT 中，我提到了不建议使用“正数相加溢出变成负数”这个规律去优化程序，因为它是一个 UB。那么什么是 UB?

未定义行为（Undefined Behavior，UB）是在 C 语言标准中没有做出规定的行为，如有符号数的溢出、数组越界等。对于这些行为，C 语言标准编译器对其自行处理（既可以报错，也可以忽略，甚至可以利用其进行代码优化）。如臭名昭著的 `i = i++ + ++i` 就是一个 UB，在不同的平台和编译器上编译运行时可能产生不一样的结果（因为 C 语言标准没有限定这些行为）。

在本题中，有符号数字的溢出也是一种 UB，编译器可以随意处理有符号数溢出的情况，所以一般不建议使用有符号数溢出的规律来判断是否溢出。

需要注意的是，无符号整数溢出不是未定义行为。在标准中，无符号数（如 `unsigned int`）的溢出是有明确规定的，其结果与自然溢出的结果相同（如对于 32 位无符号数类型 `unsigned int` 而言 `4294967295 + 1 == 0`）。

关于 UB，你可以看看这篇文章（注意，这篇文章使用的是 C++，不过其中出现的 UB 原理与 C 语言大致相同）。

UB 与优化

在 C/C++ 系语言的编译器中，UB 也是编译器进行优化的一种重要方式（但是有时候也会导致意想不到的错误，如本题）。

下面这段程序利用有“符号数的溢出是 UB”来“优化代码”：

```
// 以下默认 int 为 32 位
int i = 0x7fffffff;
if (i + 1 < 0) {
    printf("Overflow!");
}
```

平时我们知道如果 `int` 溢出了，那么结果一般是一个负数。由于 `i + 1` 的结果是 `2147483648`，即 `-2147483648`（还记得正数溢出的规律吗？），这段程序应该要输出 `Overflow!`。编译这段程序时，如果我们不开任何优化，那么确实会输出 `Overflow`，但是如果我们开了 `-O2` 优化（测试环境为 `macOS 10.15.7, GNU/gcc v10.2.0`，那么这段程序什么也不会输出。

为什么开优化前后程序表现不一样？因为开启了优化后，编译器“十分聪明”地发现了 `i` 是一个正数，那么在它就开始自己的推断：一个正数加一后，结果一定还是一个正数（还记得吗，有符号溢出是 UB，所以编译器不会考虑溢出成负数的情况），因此 `i + 1 < 0` 是恒不成立的，那不如干脆不管这个 `if` 了！于是我们的 `if` 语句就这样被可怜地忽视了……然而，如果你用的编译器不是 `GNU/gcc`，而是 `Clang + LLVM`，那么即使开了 `-O2` 优化后也不会将这个 `if` 优化掉，这就说明不同编译器对于 UB 的处理方法是不一样的。

同理，在本题中由于评测机也开了 `-O2` 优化，所以最好不要用溢出的规律来判断溢出。

为什么会有 UB 存在呢？学过其他类型语言的同学可能会发现，似乎只有 C/C++ 中广泛存在着 UB 的现象，这基本可以归结到历史因素。因为当初委员会进行标准化的时候，也没想到有人会写出 `i++++++i` 这种奇葩的程序……当然，如果对其进行进一步探讨，还会发现这和 C/C++ 的设计理念（如 `zero-overhead`，即零开销理念）相关，在这里就不展开了。作为 C/C++ 程序员的一个重要任务能快速分析出哪里产生了 UB，这也是这些语言（尤其是 C++）如此复杂的原因之一。

有兴趣的同学还可以阅读王垠的这篇文章，看看编译器是如何利用 UB 进行激进优化的：C 编译器优化的 Bug。以及知乎上的这个问题：C 和 C++ 中有哪些容易被坑的 undefined behaviour?

这里还有个关于 UB 的彩蛋：在 `1.17` 版本的 `GNU/gcc` 中，如果你触发了特定类型的 UB，那么编译器会直接运行系统自带的游戏 :)。（来源：GCC Easter Egg）

D - 补码赛高！

难度	考点
2	一维数组，补码的定义，简单位运算

题目分析

- 关于基础知识请仔细阅读课件和题面，下面介绍一下使用位运算的写法。
- 定义一个变量 a ，使其二进制表示有且只有一位为 1，通过不断位移 a 或者我们的输入 n，来直接取出对应位并输出，便可简洁地解决这道题。

易错点提示

有的同学采用 `int a = (1 << 31)`，并通过不断进行 `a = a >> 1` 与 `(x & a) > 0` 这样的判断来取出 x 的每一位。这样的写法看似正确，但实际上是会出问题的；

具体来说，是因为 `a = (1 << 31)` 在 `int` 下并不是一个正数——a 的第 31 位表示符号位，故 a 是一个负数。同时，C 语言中 `int` 的左移是逻辑左移，而右移是算数位移（参考本题的提示部分），故不断右移得到的数会自动补齐符号位，使得 a 一直是一个负数。此时如果 x 也是一个负数，那么表达式 `x & a` 的结果将会是一个负数，导致判断大于 0 的条件失效。

因此为了避免这种情况，我们推荐用 `((x >> a) & 1)` 取出 x 的第 a 位（从 0 标号），而不是用 `(x & (1 << a)) > 0` 的方法。如果你需要让右移变成算数右移，需要先将 `int` 转换为 `unsigned int`，再进行右移。

示例程序1（不使用任何位运算的基础解法，仅供参考）

```c
#include <stdio.h>

#define min_int -2147483648

int main()
{
    int n, i, a[35], flag;

    flag = 0;
    scanf("%d", &n);
    //由于int的范围是-2147483648~2147483647
    //如果输入 -2147483648，取负数的时候必定溢出
    //于是加入特判 -2147483648

    if (n == min_int)
    {
        printf("1");
        for (i = 1; i <= 31; i++)
            printf("0");
        return 0;
    }

    if (n < 0) //看看这个数是不是负数
```

```
    {
        n = -n;
        flag = 1;
    }

    for (i = 1; i <= 32; i++) //取出每一位 ， 存进数组
    {
        a[i] = n % 2;
        n /= 2;
    }

    if (flag == 1) //是个负数
    {
        for (i = 1; i <= 32; i++)
            a[i] = 1 - a[i]; //按位取反
        a[1]++;              //再+1
    }

    for (i = 1; i <= 32; i++) //处理可能发生的进位
    {
        if (a[i] > 1)
        {
            a[i] = 0;
            a[i + 1]++;
        }
    }

    for (i = 32; i >= 1; i--)
        printf("%d", a[i]);

    return 0;
}
```

示例程序2 (通过位移最低位来实现)

```
#include <stdio.h>

int main()
{
    int n, i, t, a[35];

    scanf("%d", &n);
    t = 1;
    for (i = 1; i <= 32; i++) //正序存入
    {
        a[i] = (n & t) / t;
        t = t << 1;
    }
    for (i = 32; i >= 1; i--) //倒序输出
        printf("%d", a[i]);

    return 0;
}
```

示例程序3（通过位移最低位来实现）

```c
#include <stdio.h>

int main()
{
    int n, i, t, a[35];

    scanf("%d", &n);
    t = 1;
    for (i = 1; i <= 32; i++) //正序存入
    {
        a[i] = n & t;
        n = n >> 1;
    }
    for (i = 32; i >= 1; i--) //倒序输出
        printf("%d", a[i]);

    return 0;
}
```

示例程序4（通过位移最高位来实现）

```c
#include <stdio.h>

#define min_int -2147483648

int main()
{
    int n, i, t;

    scanf("%d", &n);
    t = min_int; // 获得 10000000 00000000 00000000 00000000
    for (i = 1; i <= 32; i++)
    {
        printf("%d", (n & t) / t); // 每次将暴力取出对应位的值
        t = (unsigned)t >> 1;       // 注意这里是逻辑右移，算术右移会直接在高位补1导致出错
    }

    return 0;
}
```

E - 单词统计

难度	考点
3	一维数组，字符读入

题目分析

- 因为字符序列中可能有空格，所以用 `while( (c=getchar()) != EOF ){ }` 循环读入单个字符。
- `islower(x)` 当 `x` 是小写字母时返回 `1` ，否则返回 `0` ；类似的， `isupper(x)` 函数当 `x` 是大写字母时返回 `1` ，否则返回 `0` 。要使用这两个函数，需要引用头文件 `"ctype.h"` 。
- 宏定义 N 为数组大小 26。
- `letter[x]` 记录字母表中的第 x 个字母在字符序列中出现的次数。 字母 'a' 或 'A' 出现 `letter[0]` 次，字母 'b' 或 'B' 出现 `letter[1]` 次， 字母 x 或其相应的大写字母出现 `letter[x - 'a']` 次。
- 因为输入数据中每两个单词之间未必只有一个空格，所以计算单词数量不能单纯地统计空格个数（或非字母字符的个数）。本题中，判断一串字符是否是一个“单词”有两个判断时机：在这一个单词开始输入时判断，或者在这个单词输入结束后判断。实例代码所用的方法是在一个单词开始输入第一个字母时判断：如果当前正在输入的字符是一个字母而且前一个输入的字符不是一个字母，那么说明当前输入的字符是一个单词的起始字母，那么统计单词数量的计数器就可以增 1 。实例代码中，用变量 `pre` 存储前一个输入的字符，用变量 `c` 存储当前输入的字符，注意要给变量 `pre` 指定一个初始值，否则 `pre` 将被随机赋值，可能造成结果错误。
- `max = letter[i] > max ? letter[i] : max;` 这句代码使用了三目运算符，效果等价于： `if (letter[i] > max) max = letter[i];` 。当然，要实现取较大值还有很多种方法，如定义宏或者内联函数。
- 注意到实例代码中，把存储 `char` 类型数据的变量 `c` 和 `pre` 定义成了 `int` 类型，程序也是正确的。这是因为， `int` 类型存储的范围比 `char` 类型更大，所以把 `char` 类型数据放在 `int` 类型变量中不会发生溢出等错误。但是反之不正确，即不能用 `char` 类型变量存储 `int` 类型数据。

示例程序

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

#define N 26

int main()
{
    int i, c, pre = 0, max = 0;
    int letter[N] = {0}, total = 0;

    while ((c = getchar()) != EOF)
    {
        if (islower(c))
        {
            letter[c - 'a']++;
            if (!islower(pre) && !isupper(pre))
                total++;
        }
```

```
        else if (isupper(c))
        {
            letter[c - 'A']++;
            if (!islower(pre) && !isupper(pre))
                total++;
        }
        pre = c;
    }
    for (i = 0; i < N; i++)
        max = letter[i] > max ? letter[i] : max;
    for (i = 0; i < N; i++)
        if (letter[i] == max)
            printf("%c", i + 'a');
    printf("\n%d", total);

    return 0;
}
```

F - 扫描条形码

难度	考点
3	位运算

题目分析

题目实际上是结合了“统计二进制中 1 的个数”与“将二进制数前后翻转”两个部分。对于前一个部分，我们可以通过 `(x >> a) & 1` 的方法检查 x 的第 i 位（从 0 标号）是否为 1；对于后一个部分，可以依次将其第 $0, 1, \ldots, 13$ 位取出，并将其分别左移 $13, 12, \ldots, 0$ 位的结果用按位或拼接在一起。

示例代码

```c
#include <stdio.h>

int main()
{
    int t, n;
    int left, right;
    int i, ans;

    scanf("%d", &t);

    while (t--)
    {
        scanf("%d", &n);
        left = 0, right = 0;

        // 统计高 13...7 位中 1 的个数
        for (i = 7; i < 14; i++)
            if ((n >> i) & 1) left++;

        if (left & 1)
            printf("%d\n", n);
        else
        {
            ans = 0;
            // 将每一位 i 依次拿出来，并左移 13 - i 位
            for (i = 13; i >= 0; i--)
                ans |= ((n >> i) & 1) << (13 - i);
            printf("%d\n", ans);
        }
    }

    return 0;
}
```

G - SUBSET or SUPERSET?

难度	考点
3	位运算，循环

题目分析

不难发现，如果用题目中的方式对 n 个元素的集合进行表示，则 $0, 1, 2, \ldots, 2^n - 1$ 依次表示了其所有子集。证明：每个数的第 i 位（从 0 开始）依次表示元素 i 是否被选中，而 n 个元素只有选或不选两种选择，故其恰好与所有子集一一对应。因此，我们可以从 0 到 $2^n - 1$ 分别枚举，判断哪些数字是所给集合 S 的子集与超集即可。

同时，二进制整数的与、或、取反运算，正好对应了集合的交、并、补三种运算，我们可以用 `(s & p) == p` 和 `(q & s) == s` 代表 $S \cap P = P$ 与 $Q \cap S = S$，依次用于判断 $P \subseteq S$ 与 $S \subseteq Q$ 两种情况，即 P, Q 分别为 S 的子集与 S 的超集。

示例代码

```c
#include <stdio.h>

int main()
{
    int n, x;
    int upp, i;

    scanf("%d%d", &n, &x);

    upp = 1 << n;   // upp 为 2 的 n 次幂
    for (i = 0; i < upp; i++)
        if ((i & x) == i)
            printf("%d ", i);
    printf("\n");
    for (i = 0; i < upp; i++)
        if ((i & x) == x)
            printf("%d ", i);

    return 0;
}
```

H - 純粋な進数変換

难度	考点
4	进制转换

题目分析

在这道题中，输入的 m 进制数是一个字符串形式，为了方便后续操作，我们首先要考虑将其转换为 int 整数类型（题目明确规定了 m 进制数在 int 范围内）。

对于一个 i 位的 m 进制非负数 d，假设 d 的每一位从高到低分别是 $L_iL_{I-1}\dots L_1$，则有：

$$
\begin{aligned}
d &= L_i \cdot m^{i-1} + L_{i-1} \cdot m^{i-2} + \dots + L_1 \cdot m^0 \\
&= (L_i \cdot m + L_{i-1}) \cdot m^{i-2} + L_{i-2} \cdot m^{i-3} + \dots + L_1 \cdot m^0 \\
&= (\dots((L_i \cdot m + L_{i-1}) \cdot m + L_{i-1}) \cdot m + \dots + L_2) \cdot m + L_1 \\
&= (\dots(((0 \cdot m + L_i) \cdot m + L_{i-1}) \cdot m + L_{i-1}) \cdot m + \dots + L_2) \cdot m + L_1
\end{aligned}
$$

也就是说，可以从最高位出发，循环执行以下操作直至最低位：$d = d * m + L_j\ (1 \leqslant j \leqslant i)$，其中 d 的初始值为 0，这样最终就可以将 d 从字符串转换为一个 int 类型的数。

> - 这个做法对有前置 0 的数一样适用；
> - 如果求得 $d = 0$，最好在这里直接输出 0 并退出程序，不再进行后续操作，原因后面会说。

接下来考虑将 d 转换为 n 进制表示。对于 d 有：

$$
\begin{aligned}
d &= L_i \cdot m^{i-1} + L_{i-1} \cdot m^{i-2} + \dots + L_1 \cdot m^0 \\
&= (L_i \cdot m^{i-2} + L_{i-2} \cdot m^{i-3} + \dots + L_2) \cdot m + L_1
\end{aligned}
$$

故 $d \bmod m = L_1$ 成立。又 $L_1 < m$ 成立（否则会进位到 L_2），故 $\lfloor \frac{d}{m} \rfloor = L_i \cdot m^{i-2} + L_{i-2} \cdot m^{i-3} + \dots + L_2$ （$\lfloor x \rfloor$ 意为向下取整，在这里整个式子 $\lfloor \frac{d}{m} \rfloor$ 代表 int 类型的除法），这样 L_2 就取代 L_1 成为了最低位。

我们可以重复这个动作直至 $d = 0$，这样可以依次得到 $L_1, L_2 \dots L_i$，也就是说，我们是先得到低位再得到高位的。而题目要求按位倒置输出，故可以直接按这个顺序依次输出。

当然，考虑到输出需要忽略前置 0，所以我们需要从第一位不是 0 的数开始输出。

> - 这就是之前特判 0 的原因，如果不对 0 进行特判，程序将不会进行任何输出。

易错点提示

- 对于字符的读入，因为在字符串首先需要输入一行两个整数，故如果直接按字符依次读入，会读入 `\r` `\n` 等格式符。这里列出几个解决方法：
  - 在读入整数时在格式化字符串的末尾加入 `' '` 空格，空格会匹配之后所有的空白符，之后字符读入就会直接从第一个不是空白符的字符开始；
  - 在读入字符后手动判断，在读到第一个大写字母或数字之后再进入程序主体；
  - 使用 `scanf()` 函数中的修饰符 `%s` 或其他方法直接读入字符串。
- 对于判断字符串的结束，可以在读入不是大写字母、也不是数字的字符后结束；
- 注意区分字符中的数字和数字的区别，如 `1` 和 `'1'`；
- 若要使用字符数组，数组大小至少要在 20000 以上。

示例程序

```c
#include <stdio.h>

int main()
{
    int n, m, num = 0;
    int i, j;
    char c;

    scanf("%d%d ", &m, &n);
    c = getchar();
    while (c >= '0' && c <= '9' || c >= 'A' && c <= 'Z')
    {
        if (c >= '0' && c <= '9')
        {
            num = num * m + c - '0';
        }
        else if (c >= 'A' && c <= 'Z')
        {
            num = num * m + c - 'A' + 10;
        }
        c = getchar();
    }
    if (num == 0)
    {
        printf("0");
        return 0;
    }
    while (num % n == 0)
    {
        num = num / n;
    }
    while (num > 0)
    {
        if (num % n >= 10)
        {
            printf("%c", num % n + 'A' - 10);
        }
        else
        {
            printf("%c", num % n + '0');
        }
        num = num / n;
    }

    return 0;
}
```

I - NAF

难度	考点
5	二进制，进制转换

题目分析

通过Hint不难发现二进制表示转化为 NAF 表示时，要将连续多位的 "1" 向高位推进。

如 $78 = 2^6 + 2^3 + 2^2 + 2^1 = 1001110_{(2)}$，有 $2^1 + 2^2 + 2^3 = (2^1 + 2^1 + 2^2 + 2^3) - 2^1 = 2^4 - 2^1$。

即 $78 = 2^6 + 2^4 - 2^1 = 10100\bar{1}0_{(\text{NAF})}$， 可见二进制串 1110 会转化为 $100\bar{1}0$，即连续 "1" 的最低位变为 -1，最高位的下一位变为 1，中间全部变为 0。

解法1

二进制转换生成 01 串，再从低位至高位遍历，遇到连续 "1" 则进行上述操作。注意每组数据在运算之前要将储存二进制串的数组清零。

解法2

直接进行特殊的二进制转换，设 a 为 NAF 表示的储存数组，i 从最低位 0 至最高位，转换过程中只需判断当前 K 的奇偶：

- 若 K 为奇数，则 `a[i] = 2 - (K \% 4), K = K - a[i]`。
  - $K \bmod 4$ 若为 1，则代表 K 当前二进制下末两位为 01，此位 1 与高位无连续，`a[i] = 1`，且 `K = K - a[i]` 不会影响 `K >> 1`（右移）之后的结果。
- $K \bmod 4$ 若为 3，则代表 K 当前二进制下末两位为 11，此位 1 与高位有连续，置 `a[i] = -1`，此时无需考虑更高位，只需 `K = K - a[i]`（即 `K = K + 1`）让 K 自行产生进位，虽然 `K >> 1` 之后的结果改变，但其 NAF 表示的正确性不变。
  - 例：$K = 39 = 2^5 + 2^3 + 2^1 + 2^0 = 100111_{(2)} = 10100\bar{1}_{(\text{NAF})}$，末两位为连续 1，则 `a[0] = -1` 并使 `K = K + 1`，进位后
    $K = 39 + 1 = 40 = 2^5 + 2^3 = 101000_{(2)} = 101000_{(\text{NAF})}$，
  - $K =$ `K >> 1` $= 20 = 2^4 + 2^2 = 10100_{(2)} = 10100_{(\text{NAF})}$，求 39 的 NAF 转化为求 `(39 + 1) >> 1` 的 NAF。
- 否则 `a[i] = 0`。
- `i++, K = K >> 1`。

示例代码1

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

int main()
{
    int t, k;
    int a[35];
    int wei, pre, i, j;
    int tag;
```

```
    scanf("%d", &t);
    while (t--)
    {
        scanf("%d", &k);
        if (k == 0)
        { //特判k = 0
            printf("0\n");
            continue;
        }

        memset(a, 0, sizeof(a)); //初始化数组
        wei = 0;
        while (k)
        {                   //等价于while(k > 0)
            a[wei] = k & 1; //按位与(二进制下最低位)，等价于k % 2
            k = k >> 1;     //右移，等价于k / 2
            wei++;
        }

        pre = -1, i, j;
        for (i = 0; i < 32; i++)
        {
            if (a[i] == 1)
            {
                if (pre == -1) pre = i; //记录连续1的起始位置
            }
            else
            { //遇到0则处理先前的连续1串
                if (pre != -1)
                {
                    if (i - pre == 1)
                        pre = -1; //单个1不连续则重置pre
                    else
                    { //多个1连续
                        a[pre] = -1;
                        for (j = pre + 1; j < i; j++) a[j] = 0;
                        a[i] = 1;
                        pre = -1; //记得重置
                        i--;      //由于此位0变成1，所以要从这里开始继续向后遍历
                    }
                }
            }
        }

        tag = 0;
        for (i = 31; i >= 0; i--)
        {
            if (a[i] != 0) tag = 1; //遇到第一个非0数，则已跳过前导零，打上标记
            if (tag)
            { //等价于if(tag == 1)
                if (a[i] == -1)
                    putchar('!');
                else
                    printf("%d", a[i]);
            }
        }
        puts(""); //记得换行，也可以用 printf("\n"); 或 putchar('\n');
```

```
    }

    return 0;
}
```

示例代码2

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

int main()
{
    int t;
    unsigned int k; //无符号整形，范围[0, 2^32 - 1];
    //注意int最大值2147483647的二进制表示为31个连续1，进位会溢出，此处使用long long亦可
    int a[35];
    int i, tag;

    scanf("%d", &t);
    while (t--)
    {
        scanf("%d", &k);
        memset(a, 0, sizeof(a));
        i = 0;
        while (k)
        {
            if (k & 1)
            {
                a[i] = 2 - (k % 4);
                k -= a[i];
            }
            else
                a[i] = 0;
            k /= 2;
            i++;
        }

        tag = 0;
        for (i = 31; i >= 0; i--)
        {
            if (a[i] != 0) tag = 1;
            if (tag)
            {
                if (a[i] == -1)
                    putchar('!');
                else
                    printf("%d", a[i]);
            }
        }
        if (!tag) putchar('0'); //特判k = 0
        puts("");
    }

    return 0;
}
```

相关论文：some proofs of the properties of NAF

J - 声之刑

难度	考点
5	二进制，位运算

题目分析

对于一个长度为 m 的二进制串 S，和 m 维空间中一个点 $P(x_1, x_2, \cdots, x_m)$，定义

$$f(P, S) = \sum_{i=1}^{m} (-1)^{S_i} x_i$$

其中 S_i 表示 S 从高到低第 i 位。例如，对于 $m = 4,\ S = \texttt{1110},\ P(2, 3, 3, 3)$，有

$$f(P, S) = (-2) + (-3) + (-3) + 3 = -5$$

对于 m 维空间中两点 $A(x_1, x_2, \cdots, x_m)$ 与 $B(y_1, y_2, \cdots, y_m)$ 的曼哈顿距离，其一定可以写成如下形式

$$\begin{aligned}
&\sum_{i=1}^{m} |x_i - y_i| \\
=&\sum_{i=1}^{m} \max(x_i - y_i, -x_i + y_i) \\
=&\sum_{i=1}^{m} \max_{S_i \in \{0,1\}} (-1)^{S_i} (x_i - y_i) \\
=&\max_{S=0}^{2^m-1} \sum_{i=1}^{m} (-1)^{S_i} (x_i - y_i) \\
=&\max_{S=0}^{2^m-1} \left[\sum_{i=1}^{m} (-1)^{S_i} x_i - \sum_{i=1}^{m} (-1)^{S_i} y_i \right] \\
=&\max_{S=0}^{2^m-1} \left[\sum_{i=1}^{m} (-1)^{S_i} x_i + \sum_{i=1}^{m} (-1)^{\overline{S_i}} y_i \right] \\
=&\max_{S=0}^{2^m-1} \ f(A, S) + f(B, \overline{S})
\end{aligned}$$

其中，$\overline{S}$ 表示将二进制按位取反，例如 $m = 4,\ S = \texttt{1110},\ \overline{S} = \texttt{0001}$。这里记 $\oplus$ 为按位异或，则还有 $\overline{S} = (2^m - 1) \oplus S$。

一个对上式的简单证明如下：考虑每一个 $|x_i - y_i|\ (1 \le i \le m)$，必有 x_i 和 y_i 前面的系数中一个为 1 另一个为 -1，因此曼哈顿距离一定是 $f(A, S) + f(B, (2^m - 1) \oplus S)$ 的形式。

同样可以证明，一定存在一个 S，使得对固定的 A 点和 B 点，它们之间的曼哈顿距离恰为 $f(A, S) + f(B, (2^m - 1) \oplus S)$，并且不存在其他 S 的取值使得上式更优（如果有更优的 S，说明一开始 $\max$ 里的二选一就没选中较大的那个）。

因此，只要枚举每种 S 的取值并进行考虑，维护 $f(A, S)$ 的最大值和 $f(B, (2^m - 1) \oplus S)$ 的最大值，并设

$$g_S = \max_{i=1}^{n} f(P_i, S)$$

那么最终答案即为

$$\max_{S=0}^{2^m-1}(g_S + g_{(2^m-1)\oplus S})$$

即所有 S 中求出的答案最大值一定对应了一种最大的曼哈顿距离方案。

我们可以直接通过 g_S 的定义通过简单位运算求出答案，时间复杂度 $O(n2^m)$。

示例程序

```c
#include <stdio.h>
#define max(a, b) ((a) > (b) ? (a) : (b))

int main()
{
    int t, n, m, a[5], i, j, k;
    long long val[32], x, ans;

    scanf("%d", &t);
    while (t--)
    {
        scanf("%d%d", &n, &m);
        for (i = 0; i < (1 << m); i++)
            val[i] = -1e18;
        for (i = 1; i <= n; i++)
        {
            for (j = 0; j < m; j++)
                scanf("%d", &a[j]);
            for (j = 0; j < (1 << m); j++)
            {
                x = 0;
                for (k = 0; k < m; k++)
                {
                    if (j & (1 << k))
                        x += a[k];
                    else
                        x -= a[k];
                }
                val[j] = max(val[j], x);
            }
        }
        ans = -1e18;
        for (i = 0; i < (1 << m); i++)
            ans = max(ans, val[i] + val[i ^ ((1 << m) - 1)]);
        printf("%lld\n", ans);
    }

    return 0;
}
```