C3 - Solution

A 摩卡与补码

难度	考点
1	位运算

题目分析

整数在计算机中存储的即为二进制补码形式，所以直接输出其二进制位即可。

输出一个数的二进制可以参考 Hint 的代码

示例代码

```c
#include <stdio.h>

int main() {
    int n;
    while(scanf("%d", &n) != EOF) {
        for(int i = 31 ; i >= 0 ;i--) {
            printf("%d", (n >> i) & 1);
        }
        printf("\n");
    }
    return 0;
}
```

扩展延伸

我们使用了 `>>`，即右移操作，但需要注意地是，C 语言标准并没有明确对应对于有符号数应该使用哪种类型的右移（包括算数右移和逻辑右移，即是在前面补符号位还是补 0）。然而实际上，几乎所有的编辑器/机器组合都对有符号数使用算数右移。当然，这里不论机器使用的是算数右移还是逻辑右移对我们进行的操作都不会有任何影响（想一想为什么）。

B Unsigned medium int

难度	考点
2	模拟。二进制

题目分析

一个 `int` 有 32 位，所以直接使用 `int` 模拟即可。

本题中 `Unsigned medium int` 的范围是多少呢，最小是 0, 最大是$2^{24}-1$.

示例代码

```
#include<stdio.h>
int main(){
    int t;
    scanf("%d",&t);
    while(t--){
        int a;
        scanf("%d",&a);
        if(a<0||a>=16777216){
            printf("We need a new cpu\n");
        } else {
            for(int i=23;i>=0;--i){
                printf("%d",(a>>i)&1);
            }
            printf("\n");
        }
    }

    return 0;
}
```

C 摩卡与小水獭密码

难度	考点
3	数组、哈希思想、字符处理

题目分析

参考 `例 3-11`，我们可以知道如何统计每个字母的出现次数。记录了每个字母的出现次数后，我们就能用类似于 `E1 - H - shtog 挑武器` 中的方法，记录下出现次数最多的元素的下标，和它的出现次数。

对于大小写字母的互换，我们可以参考上节课所学的内容，比如将一个大写字母 `ch` 转化为小写字母，那么就可以利用诸如 `ch = ch - 'A' + 'a'` 这样的代码完成大写字母到小写字母的转换，当然使用 `<ctype.h>` 中的 `tolower` 函数也是可以的，这个库专门用来对字母进行检验或者转换。

最后，这道题其实可以避免存储读进来的字符串，直接一个字符一个字符这样读入处理，申请没必要的空间对资源是一种浪费。

示例代码

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

int cnt[128];   // ASCII 的值最大为 128

int main() {
    int n;
    scanf("%d", &n);
    getchar();  // 处理 n 后面的换行符
```

```
    for(int i = 1; i <= n ;i++) {
        char ch = getchar();
        cnt[ch]++;

        if(ch >= 'a' && ch <= 'z')  // 是小写字母，当然也可以用 ctype.h 中封装好的函数，同学们可以查一查这个库提供了哪些函数
            printf("%c", ch + 'A' - 'a');
        else if(ch >= 'A' && ch <= 'Z') // 是大写字母
            printf("%c", ch - 'A' + 'a');
        else    // 其它字母，原样输出
            printf("%c", ch);
    }
    printf("\n");   // 第一行

    // 记录哪个元素出现的最多，出现的次数，初始值都是 0
    char ans = 0;
    int maxCnt = 0;
    for(int i = 32; i <= 126 ;i++) {
        if(cnt[i] > maxCnt) {
            maxCnt = cnt[i];
            ans = i;
        }
    }

    printf("%c\n", ans);
    printf("%d", maxCnt);

    return 0;
}
```

D 置 0 置 1

难度	考点
3	位运算

题目分析

该题主要涉及两个知识点，第一个为 `unsigned int` 的数据范围，第二个知识点是位运算的相关知识。

对于一个二进制的数，将它的第 k 位置 1 的方法是：

先将 1 左移 k 位，这样我们就得到了一个仅有第 k 位为 1 的，其他位均为 0 的二进制数，将输入的数字 n 与它进行或运算。这样，对于第 k 位以外的位上，与 0 进行或运算均为其本身，即仅有第 k 位上与 1 进行或运算，即为 1 。语句为 `x | (1 << k)` 。

同样的，将 n 的第 k 位置 0 的方法是：

先将 1 左移 k 位，这样我们就得到了一个仅有第 k 位为 1 的，其他位均为 0 的二进制数，然后对于该二进制数取反，这样我们就得到了一个仅有第 k 位为 0 的，其他位均为 1 的二进制数。将输入的数字 n与它进行与运算，这样，对于第 k 位以外的位上，与 1 进行与运算均为其本身，即仅有第 k 位上与 0 进行与运算，即为 0 。语句为 `x & ~(1 << k)` 。

课件也给出了以上两个操作对应的语句。

特别要注意的是，本题数据在 `unsigned int` 范围内，因此使用 `1u` 表示 `unsigned int` 类型的 1 更为规范。

示例代码

```c
#include <stdio.h>

int main() {
    int  t, k;

    unsigned int n;
    scanf("%u", &n);
    scanf("%d", &t);

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

        scanf("%d", &k);

        int choice;

        scanf("%d", &choice);

        if (choice == 0) {
            n = n & ~(1u << k);
        }
        else if (choice == 1) {
            n = n | (1u << k);
        }

        printf("%u\n", n);
    }

    printf("%u\n", n);

    return 0;
}
```

E 点分十进制

难度	考点
3	位运算

题目分析

这道题其实就是把32位二进制数转成十进制，只是这32位二进制数从高位到低位是以四段儿十进制数给出的，每段儿十进制数代表了32位中的8位，因此只需要将每个8位提取出来，然后放到对应的位置上即可，比如：第一段儿十进制数应该放在32位的高31位到高24位上（将其左移24位），第二段儿放在高23位到高16位上（将其左移16位）…… 然后四段儿都放在对应位置后将其加起来即可。注意输入中有小数点，要在`scanf`中考虑到小数点。

示例代码

```
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    while (n--) {
        int a, b, c, d;
        scanf("%d.%d.%d.%d", &a, &b, &c, &d);
        unsigned int e = (unsigned int)((a << 24) + (b << 16) + (c << 8) + d);
        printf("%u\n", e);
    }
    return 0;
}
```

F Baymax 的整数变换

难度	考点
3~4	位运算、二进制

题目分析

首先观察题目提供的两种操作：加 1 和乘 2，不难联想到这两种操作应该与二进制有关。

我们可以任意考虑一个整数，如 6，只考虑低 4 位，其二进制表示为 0110。当执行加 1 操作时，相当于在最低位进行加 1，得到 7，其二进制表示为 0111；当执行乘 2 操作时，相当于把二进制的每一位都乘 2，得到 12，其二进制表示为 1100，即实现了左移 1 位的效果。

而对于初始整数 0 和目标整数 n，设 n 的二进制表示为 $s_{31}s_{30}\dots s_1s_0$。若要进行最少次操作，一定是希望尽可能多地进行乘 2 操作，最少需要多少次呢？则需要观察 n 的二进制表示中最高位的 1 所在的位置，假设其位于第 k 位（设第 1 位为最低位）。因为每进行一次乘 2 操作，相当于将一个数的二进制表示左移 1 位，所以想要得到 n，至少应进行 $k-1$ 次左移操作。而对于需要进行多少次加 1 操作，则只需要观察 n 的二进制表示中有多少位为 1，至此可以求出最小的操作次数。

示例代码

```
#include <stdio.h>
int main()
{
    int T;
    scanf("%d", &T);
    for (int i = 0; i < T; i++)
```

```
    {
        int ans = 0, n, flag = 1; //flag用于标记,以只求一次最高位1的位置
        scanf("%d", &n);
        for (int i = 31; i >= 0; i--)
        {
            if ((n >> i) & 1)
            {
                if (flag)
                {
                    ans += i; //加上最高位1的位置，即需要进行的乘2操作的次数
                    flag = 0;
                }
                ans++;  //二进制表示的每一位1都需要进行加1操作
            }
        }
        printf("%d\n", ans);
    }
}
```

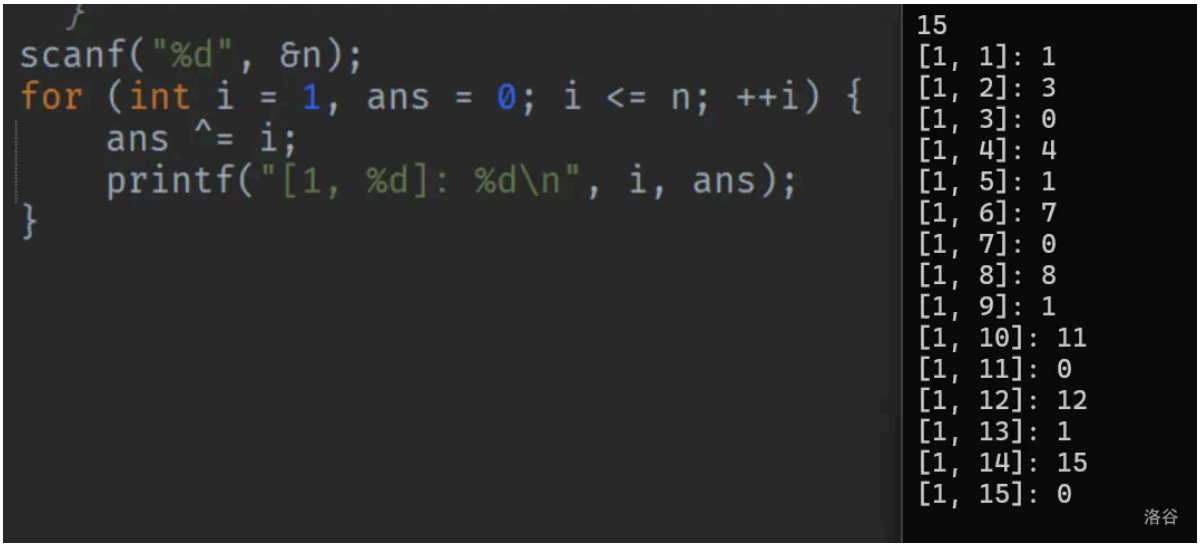
G 小 p 与异或

难度	考点
4	位运算，找规律

题目分析

首先，根据异或运算的性质，我们可以把每次询问的区间 $[l,r]$ 拆成两个区间 $[1,l-1]$ 和 $[1,r]$，那么答案就是 $ans_{1,l-1} \oplus ans_{1,r}$ 。于是我们可以通过预处理所有 $[1,i](1 \le i \le n)$ 的答案，来实现 $O(1)$ 地回答每次询问。

但是这道题的范围很大，所以这么做还是无法通过本题。此时注意到 Hint 的提示，可以尝试手动（或者用程序暴力）模拟一下几个小数据的结果，观察一下是否有什么规律。如下：



显然，存在规律，对于区间 $[1,i]$ ，有 $ans_{1,i} = \begin{cases} 1, & i\%4==1 \\ i+1, & i\%4==2 \\ 0, & i\%4==3 \\ i, & i\%4==0 \end{cases}$

可以多试几组数据验证一下这个规律的正确性~~或者直接让评测机验证一下~~，然后就可以通过本题了。

以下是对上述规律的证明：

由异或运算的性质，我们可以将每个数的各个位分开进行考虑。

设 $f(l,r)$ 表示区间 $[l,r]$ 的异或和，对于任意整数 $k \geq 1$，区间 $[2^k, 2^{k+1}-1]$ 中最高位（也就是第 k 位）的 1 出现了 2^k 次，这一位上的异或和为 0，因此最高位可以直接去掉、对答案没有影响，也就是说 $f(2^k, 2^{k+1}-1) = f(2^k - 2^k, 2^{k+1}-1-2^k) = f(0, 2^k-1)$。

所以，$f(0, 2^{k+1}-1) = f(0, 2^k-1) \oplus f(2^k, 2^{k+1}-1) = 0$。

因此，对于任意整数 $k \geq 2$，有 $f(0, 2^k-1) = 0$。

那么对于 $f(0,n)$，设 n 的最高位是第 k 位，则有 $f(0,n) = f(0, 2^k-1) \oplus f(2^k, n) = f(2^k, n)$。

若 n 为偶数，那么最高位 1 出现了奇数次，故一定有 $f(0,n) = f(2^k, n) = 2^k + f(0, n-2^k)$。

若 n 为奇数，那么最高位 1 出现了偶数次，故一定有 $f(0,n) = f(2^k, n) = f(0, n-2^k)$。

由于 n 与 $n-2^k$ 有相同的奇偶性，于是最终 $f(0,n)$ 的值就取决于最低 2 位的部分，也就是由 $n \pmod 4$ 的值决定。故，

若 $n \equiv 0 \pmod 4$，$f(0,n) = n$

若 $n \equiv 1 \pmod 4$，$f(0,n) = 0 \oplus 1 = 1$

若 $n \equiv 2 \pmod 4$，$f(0,n) = n - 2 + (0 \oplus 1 \oplus 2) = n + 1$

若 $n \equiv 3 \pmod 4$，$f(0,n) = 0 \oplus 1 \oplus 2 \oplus 3 = 0$

示例代码

```
#include <stdio.h>

long long l, r, ansl, ansr;
int n, T;
int main() {
    scanf("%d%d", &n, &T);
    while (T--) {
        scanf("%lld%lld", &l, &r);
        l -= 1;
        if (l % 4 == 0) {
            ansl = l;
        } else if (l % 4 == 1) {
            ansl = 1;
        } else if (l % 4 == 2) {
            ansl = l + 1;
        } else {
            ansl = 0;
        }
        if (r % 4 == 0) {
            ansr = r;
        } else if (r % 4 == 1) {
            ansr = 1;
        } else if (r % 4 == 2) {
            ansr = r + 1;
```

```
        } else {
            ansr = 0;
        }
        printf("%lld\n", ansr ^ ansl);
    }
    return 0;
}
```

H ddz 的 g^c^d

难度	考点
5	位运算、构造

题目分析

本题为构造题，给出一个数 x 需要找到一个 y 满足 x 和 y 的最大公因数等于异或和，构造方法不止一种，下面介绍其中一种构造方法。

由于涉及异或，我们从二进制的角度来看，注意到 $(1011011000)_2 = (1011011)_2 * (1000)_2$，我们记 x 的二进制表示中 1 的最低的位置对应的数为 a，上述例子中 $x = (1011011000)_2$ 则 $a = (1000)_2$，显然 a 是 x 的因数，考虑构造一个 y 满足 $gcd(x, y) = a$，此时有 $x \oplus y = gcd(x, y) = a$（$\oplus$表示按位异或），左右同时异或一个 x，得到 $y = a \oplus x$。

下面说明 $y = a \oplus x$ 在某些情况下是满足题意的。首先两边异或一个 x 得到 $x \oplus y = a$，其次由于 a 的特殊性，不难发现 $a \oplus x = x - a$，所以 $gcd(x, y) = gcd(x, (x - y)) = gcd(x, a) = a$，综上有 $gcd(x, y) = a = x \oplus y$ 且 $y = a \oplus x = x - a < x$，只需满足 $y > 0$ 即可，也就是说当 $a \oplus x > 0$ 时 $y = a \oplus x$ 是满足题意的一个 y。

下面说明 $a \oplus x = 0$ 时不存在满足条件的 y。 $a \oplus x = 0$ 说明 $a = x$，也就是说 x 的二进制表示中只有一个 1，记 x 的第 b 位为 1 其他位全是 0，而 $y < x$，则在 y 的二进制表示中第 b 位一定是 0，所以 $x \oplus y$ 的二进制表示中第 b 位是 1，此时可推出 $x \oplus y > y$，但很显然 $y = min(x, y) \geq gcd(x, y)$，所以有 $x \oplus y > gcd(x, y)$ 不可能满足条件。

综上，一种构造方法是，取 x 的二进制表示中 1 的最低的位置对应的数为 a，令$y = a \oplus x$，如果 $y = 0$ 输出 -1，否则输出 y。

事实上 $a = x\&(-x)$，同学们可以自己思考为什么可以这样算。

示例代码

```
#include <stdio.h>

int main() {
    int n, x;
    scanf("%d", &n);
    for (int i = 0; i < n; ++i) {
        scanf("%d", &x);
        int y = x & (-x) ^ x;
        printf("%d\n", y == 0 ? -1 : y);
    }
    return 0;
}
```

I 丢失的号码牌

难度	考点
6	位运算、异或的性质

题目分析

本题属于经典异或题目，前置知识：异或的交换律，异或的结合律，异或的零一律。

易得，当 n 为奇数时，丢失一张号码牌，即只有一个整数只出现过一次，其他整数均出现两次；当 n 为偶数时，丢失两张号码牌，即有两个整数只出现过一次，其他整数均出现两次。下面依次讨论丢失一张号码牌和丢失两张号码牌的情况。

当 n 为奇数时：

由题，此时将所有整数全部取异或和时，即 $a_1 \oplus a_2 \oplus ..... \oplus a_n$ 由于仅有一个整数只出现一次，剩余整数出现两次。故而不妨设 b_i 是将 a_i 由小至大排序的结果，此时 b_j 是唯一仅出现一次的整数，则：

$$a_1 \oplus a_2 \oplus ..... \oplus a_n = b_1 \oplus b_2 \oplus ..... \oplus b_n = (b_1 \oplus b_2) \oplus ..... \oplus b_j \oplus ..... \oplus b_{n-1} \oplus b_n) = b_j$$

即只剩下那唯一一个出现的整数。

当 n 为偶数时：

由上一种情况可知，当我们求所有数的异或和 $a_1 \oplus a_2 \oplus ..... \oplus a_n$ 后，设出现一次的整数分别为 x 与 y，则必有 $a_1 \oplus a_2 \oplus ..... \oplus a_n = x \oplus y$，且 $x \neq y$。

此时，由于 $x \oplus y$ 必然不等于 0，必然有一个位置 i，使 x 的第 i 位和 y 的第 i 位的异或结果为 1。（等价于找一位置 i，使 $x \oplus y$ 的第 i 位为 1）

此时，由异或运算的性质，x 与 y 的第 i 位必然不同。

对于第 i 位，若将所有数据分为第 i 位为 0 与第 i 位为 1，每一部分的情况与奇数时一样。

此时，可在之前计算异或和时，提前根据每一位的 0、1 情况分为两部分，并分别计算出两部分的异或和。在计算完总异或和后，得出 i 的实际值，取出提前根据第 i 位 0、1 分为两个部分结果即可。

示例代码

std：

```
#include<stdio.h>
int ans1[32];
int ans2[32];
signed main()
{
    int n,ans0,a,i,j;
    scanf("%d",&n);
    ans0=0;
    for(i=0;i<n;i++)
    {
        scanf("%d",&a);
        ans0^=a;
        for(j=0;j<32;j++)
        {
            if((a>>j)&1)
            {
                ans1[j]^=a;
            }
            else
            {
                ans2[j]^=a;
            }
        }
    }
    if(n%2==1)
    {
        printf("%d",ans0);
    }
    else
    {
        for(i=0;i<32;i++)
        {
            if((ans0>>i)&1)
            {
                break;
            }
        }
        if(ans1[i]<ans2[i])printf("%d %d",ans1[i],ans2[i]);
        else printf("%d %d",ans2[i],ans1[i]);
    }
    return 0;
}
```

J 完全数

难度	考点
6	状压，计数

题目分析

一个很朴素且暴力的做法就是，直接遍历每组 (i,j)，然后判断是否满足完全数的条件。可是，这样做的时间复杂度是 $O(n^2)$，显然在 $2 \le n \le 2\times10^5$ 的数据范围下是无法通过的。

为避免产生歧义，在接下来的表述中，「数字」特指 $0 \sim 9$ 的个位数，「数」泛指所有自然数。

注意到，把两个数拼接成一个完全数，只与它们含有哪些不同的数字有关，而与这些数字的数目，排列顺序都没有关系。于是，例如 $12345, 54321, 11223455, 431552$ 这些数都可以被归成一类：仅含有数字 1，2，3，4，5 的数。它们都至少需要和含有数字 0，6，7，8，9 的数拼接才能形成完全数。只要我们把所有「数」都按照它含有的「数字」进行分类，只去关心数所在的类别，而不去关系它的具体的值，就能通过计数和乘法来提高效率。接下来，我们思考如何以一种合适的方式，表示一个数属于哪个分类。

这里就是状压的思想：用二进制的 0 和 1，来代表一个数中，每种数字的出现与否。我们总共有 $0 \sim 9$ 十个数字，因此我们需要十个二进制位。例如，数 54310 含有数字 0，1，3，4，5，它的状态用十位二进制数表示就是 0000111011，或者十进制下的 59。数 1234567890 含有数字 0 到 9，它的状态用十位二进制数表示就是 1111111111，或者十进制下的 1023。分类后，我们只需枚举每两种类别之间的组合能否拼接出完全数。如果状态为 i 的类别可以与状态为 j 的类别拼接出完全数，那么在这两种类别的组合中，方案数为 $cnt_i \times cnt_j$。而怎么判断状态 i 和 j 合起来之后是否含有 $0 \sim 9$ 的全部数字呢？如果 i 和 j 的按位或等于二进制下的 1111111111，或十进制下的 1023，就说明这两个类别是满足条件的。

注意，需要对 $i = j = 1023$ 的情况进行特殊考虑，此时满足条件的方案数为 $C_{cnt_{1023}}^2 = \dfrac{cnt_{1023} \times (cnt_{1023} - 1)}{2}$。

示例代码

```c
#include <stdio.h>
int cnt[1024];
int main(void)
{
    int n;
    scanf("%d", &n);
    long long a;
    for (int i = 0; i < n; i++)
    {
        scanf("%lld", &a);
        int mask = 0;
        while (a)
        {
            mask |= 1 << (a % 10);
            a /= 10;
        }
        cnt[mask]++;
    }
    long long ans = 0; // 答案可能会爆 int
    for (int i = 0; i < 1024; i++)
    {
        for (int j = i; j < 1024; j++)
        {
            if ((i | j) == 1023)
            {
                if (i == 1023 && j == 1023)
```

```
                {
                    ans += 1ll * cnt[1023] * (cnt[1023] - 1) / 2; // 注意这个特殊情
况
                }
                else
                {
                    ans += 1ll * cnt[i] * cnt[j];
                }
            }
        }
    }
    printf("%lld\n", ans);
    return 0;
}
```