# E3 - Solution

## A 补码

| 难度 | 考点 |
| --- | --- |
| 1 | 位运算 |

### 题目分析

输出 $n$ 的补码，语句 `(n >> k) & 1` 的值就是 `n` 的第 `k` 位，从第 $31$ 位依次输出到第 $0$ 位即可。

示例代码中语句 `while(~scanf("%d", &n))` 等价于 `while(scanf("%d", &n) != EOF)` 。已知 `EOF` 的值为 $-1$，想想为什么二者等价?

### 示例代码

```
#include <stdio.h>

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

## B 补码解译

| 难度 | 考点 |
| --- | --- |
| 1 | 二进制， 补码 |

### 题目分析

重述题目：本题给定一个十进制有符号整型数 $x_i$ 对应的二进制表示，求 $x_i$ 。

只需要按顺序读取二进制表示，并通过左移运算符移动到对应位上，最后用或运算放到 $x_i$ 对应的位上即可。

**计算机中的整数一般是按补码存储的。**

## 示例代码

```
#include <stdio.h>
int main()
{
    int y = 0;
    for(int i = 31; i>= 0; i--)
    {
        unsigned int x;
        scanf("%1u", &x);
        y = y | (x << i);
    }
    printf("%d", y);
    return 0;
}
```

## 补充

请注意，现有观点表明，在当前C99标准中，int类型的整数左移31位（比如第零位的1移到符号位）及以上是未定义的，未定义行为是指程序的行为随实现而变动的行为，**遵从标准的实现必须为每个这样的行为的效果提供文档**。如果你的代码存在未定义行为，这可能是导致提交WA的一个可能原因。但是，本题中经过测试，写出的代码1的未定义行为可以得到正确结果（运气好），在实际编写中，建议不要出现未定义行为的书写。

## AC代码 1

```
#include<stdio.h>
#define bits 32
int main()
{
    int x = 0;
    int ops, cntbits;
    for(cntbits = 31;cntbits >= 0; cntbits--)
    {
        scanf("%1d", &ops);
        x += ops << cntbits;
    }
    printf("%d", x);
    return 0;
}
```

## AC代码 2

本代码从补码构成的角度出发，避免书写未定义行为。

```
#include<stdio.h>
#define bits 32
int main()
{
    unsigned int x = 0;
    unsigned int ops, value;
    scanf("%1u", &ops);
    int cntbits;
```

```
    for(cntbits = 30;cntbits >= 0; cntbits--)
    {
        scanf("%1u", &value);
        if(ops == 1u)
        {
            value = 1u - value;
        }
        x += value << cntbits;
    }
    if(ops == 0u)
    {
        printf("%u", x);
    }
    else
    {
        printf("-%u", x + 1u);//1u表示这个1是unsigned int类型的1
    }
    return 0;
}
```

## C 用位运算实现加法！

| 难度 | 考点 |
| --- | --- |
| 1 | 位运算 |

## 题目分析

给出 `a, b` ，设 `c = a & b` ， `d = a ^ b` ， `e = (c << 1) + d` 。

按照题目描述定义变量 `a, b, c, d, e` ，仿照第一题的代码输出每个数的二进制码的每一位，最后输出 `e` 即可。

注意是数据类型为 `unsigned int` ，因此取出 `x` 的第 `k` 位，最好使用语句 `x >> k & 1u` （由于右移运算符 `>>` 优先级高于按位与运算符 `&` ，因此不加括号也不改变表达式计算结果）。

### 示例代码

```
#include <stdio.h>

int main()
{
    unsigned int a, b, c, d, e;
    scanf("%u%u", &a, &b);
    c = a & b;
    d = a ^ b;
    e = (c << 1) + d;
    for(int i = 31; i >= 0; --i)
        printf("%u", a >> i & 1u);
    printf("\n");
    for(int i = 31; i >= 0; --i)
```

```
        printf("%u", b >> i & 1u);
    printf("\n");
    for(int i = 31; i >= 0; --i)
        printf("%u", c >> i & 1u);
    printf("\n");
    for(int i = 31; i >= 0; --i)
        printf("%u", d >> i & 1u);
    printf("\n");
    for(int i = 31; i >= 0; --i)
        printf("%u", e >> i & 1u);
    printf("\n");
    printf("%u", e);
    return 0;
}
```

## 拓展阅读

为什么 `e` 等于 `a + b` 呢?

可以这样理解：`c = a & b` 可以视作进位，当且仅当两个数某一位都为 `1` 的时候该位加法才进位，因此 `c` 的每一位就表示了 `a + b` 的每一位是否需要进位。

而 `d = a ^ b` 的值又是每一位除去进位影响后剩下的数，即
$0+0=0(\text{不进位}), 0+1=1(\text{不进位}), 1+1=0(\text{进位})$，与异或运算结果相同。

因此表示向前进位的 `c` 左移 $1$ 位后，再与 `d` 相加，就等于 `a + b` 。

据此我们可以利用迭代，不使用加号来进行加法运算，代码如下：

```
//计算a+b
while(a != 0)
{
    int c = a & b;
    int d = a ^ b;
    a = c << 1;
    b = d;
}
//此时a为0，b为最初a+b的结果
```

# D 比赛成绩分析

| 难度 | 考点 |
| --- | --- |
| 2 | 统计，数组，循环 |

## 题目分析

每读入一个学生，将其每道题的成绩与数组 `int a[] = {30, 20, 10, 10, 10, 10, 5, 5, 5, 5};` 中的元素进行比较，计算该同学的总成绩和AC题目数量，根据题目要求判断是否属于及格、AK、无AC的情况，特别注意算平均数时，超过 $100$ 应该按照 $100$ 分算。

## 示例代码

```
#include <stdio.h>
int main()
{
    int n;
    scanf("%d", &n);
    int sum = 0, n60 = 0, AK = 0, noAC = 0; //所有同学的总分之和，及格、AK、无AC的人数
    int score[10] = {30, 20, 10, 10, 10, 10, 5, 5, 5, 5};
    for(int i = 0; i < n; ++i)
    {
        int s = 0, num = 0; //该同学得分和AC数量
        for(int j = 0; j < 10; ++j)
        {
            int t;
            scanf("%d", &t);
            s += t;
            if(t == score[j]) num++; //该题AC
        }
        if(s > 100) s = 100;
        if(s >= 60) n60++;
        if(num == 0) noAC++;
        if(num == 10) AK++;
        sum += s;
    }
    printf("%.2f\n", (double)sum / n); //注意计算平均数要转为double类型计算
    printf("%d\n%d\n%d", n60, AK, noAC);
    return 0;
}
```

# E Cirno 的完美位运算教室

| 难度 | 考点 |
| --- | --- |
| 3~4 | 位运算 |

## 问题分析

题意很清晰，我们应当使 $x$ 和 $y$ 的二进制位上至少有一位相同（按位与），有一位不同（按位异或），那么我们可以对 $x$ 的值进行分类讨论：

1. 若 $x = 1$，显然 $y = 3$;
2. 若 $x$ 的二进制位上只有一位为 $1$，由于 $x \neq 1$，仅需令 $y = x + 1$ 即可满足题意;
3. 若 $x$ 的二进制位上有多个 $1$，只需取 $x$ 的最低位的 $1$ 赋给 $y$ 即可。

## 参考代码 #1

```c
#include <stdio.h>

int main()
{
    int T, x, i, y;
    scanf("%d", &T);
    while (T--)
    {
        scanf("%d", &x);
        if (x == 1)
            y = 3;
        else
        {
            for (i = 0; i < 32; i++)
            {
                if ((x >> i) & 1) //x的二进制位为1的最低位
                {
                    y = (1 << i);
                    break;
                }
            }
            if (y == x) //x仅有一位二进制位为1
                y++;
        }
        printf("%d\n", y);
    }
    return 0;
}
```

# 小拓展

## 关于 lowbit 运算

我们定义一个函数 $f(x)=\text{lowbit}(x)$，函数值为 $x$ 的二进制最低位的 $1$ 所对应的值。

例如：$6=(110)_2$，那么 $\text{lowbit}(6)=(10)_2=2$，因为 $(110)_2$ 的最低位的 $1$ 对应的数为 $2^1=2$。

## lowbit 运算的实现

为得到 lowbit 的值，我们只需得到最低位的 $1$ 的位置，将其余位置全部置 $0$ 即可，下面介绍两种方式：

1. `x&(x^(x-1))`
   对 $x$ 的取值进行讨论：
   若 $x$ 为奇数，显然运算结果为 $1$，符合要求；
   若 $x$ 为偶数，那么 $x-1$ 会将 $x$ 从最低位的 $1$ 开始一直到最右位全部取反，即得到一个前面不变，后面为 $011\ldots$ 的串，与 $x$ 进行按位异或，得到一个前面为 $0$，后面为 $111\ldots$ 的串，再与 $x$ 进行按位与，即得到 lowbit。
2. `x&-x`
   我们知道，一个负数的补码是其绝对值的原码取反加一，有了这个前置知识，这个运算的实现原理留给读者思考。

lowbit 运算有很多用途，比如可以用来统计一个数的二进制位为 1 的个数：

```
while (x)
{
    x -= x & -x;
    cnt++;
}
```

有了 lowbit 运算，我们的代码可以得到一些简化。

## 参考代码 #2

```
#include <stdio.h>

int main()
{
    int T, x, y;
    scanf("%d", &T);
    while (T--)
    {
        scanf("%d", &x);
        if (x == 1)
            y = 3;
        else if (x - (x & -x))//x的二进制有多位为1
            y = x & -x;
        else
            y = x + 1;
        printf("%d\n", y);
    }
    return 0;
}
```

# F 失踪的进制

| 难度 | 考点 |
| --- | --- |
| 5 | 进制转换 数据类型 |

## 题意解析

一种比较朴实的做法是：遍历每一种可能的进制，转换成十进制之后进行乘法来判断是否成立。

首先要处理的一件事是：在哪些进制下是允许出现这些数字的。例如 123 不可能出现在三进制下，因为三进制中并不存在数字 3.

因此我们需要找到出现的最大的数字，以此判断 $B$ 最小可能是多少。这一步我们可以依次提取每一位数字并进行比较来完成。注意到题中所给数据位数不超过 6 位，我们可以如下处理。

以寻找 $a = 810975$ 中最大的数字为例

```
int B = 2, base = 1, a = 810975;
for (int i = 0; i < 6; i++) {
    if (a \ base % 10 >= B) {
        B = a \ base % 10 + 1;
    }
    base *= 10;
}
```

在获取了 $B$ 的最小值之后，我们便可遍历 $B$ 的值，计算在 $B$ 进制下该等式是否成立。

一种办法是将数字转化为十进制后再进行乘法运算。类似于上面的做法，由于数据位数不超过 6 位，我们可以依次提取每一位，以此来将其转化为十进制。

以将 $a = 810975_{(12)}$ 转化为十进制为例

```
int base = 1, base0 = 1, a = 810975, a0 = 0, B = 12;
for (int i = 0; i < 6; i++) {
    a0 += a \ base % 10 * base0;
    base *= 10;
    base0 *= B;
}
```

这个计算过程可以用下表来理解

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $base$ | $10^5$ | $10^4$ | $10^3$ | $10^2$ | $10^1$ | $10^0$ | |
| 数字 | $8 = \frac{a}{10^5} \bmod 10$ | $1 = \frac{a}{10^4} \bmod 10$ | $0 = \frac{a}{10^3} \bmod 10$ | $9 = \frac{a}{10^2} \bmod 10$ | $7 = \frac{a}{10^1} \bmod 10$ | $5 = \frac{a}{10^0} \bmod 10$ | |
| $base0$ | $12^5$ | $12^4$ | $12^3$ | $12^2$ | $12^1$ | $12^0$ | |

$$\Rightarrow a_0 = 8 \times 12^5 + 1 \times 12^4 + 0 \times 12^3 + 9 \times 12^2 + 7 \times 12^1 + 5 \times 12^0$$

将 $a$, $b$, $c$ 都转化为十进制的 $a_0$, $b_0$, $c_0$ 之后便可进行乘法运算。

需要注意的是,虽然 $a$, $b$, $c$ 不超过 6 位，其转化为十进制后的最大值为 $16^6 - 1 = 16777215$ 未超过 `int` 类型的储存上限，但是 $a * b$ 的最大值为 $16777215^2 \approx 2.8 \times 10^{14}$ 超过了 `int` 类型的储存上限，因此需要我们改变数据类型为 `long long int` 来进行运算。

综合以上我们就能得到示例代码1

---

如果对字符串处理有所了解，那么可以使用一种更为简洁的做法。

`stdlib.h` 头文件中的库函数 `strtol` 可以将字符串按照给定的进制转化为数字。感兴趣的同学可以自行查阅资料，这里不再详述。用此法可以得到示例代码2

## 示例代码1

```
#include <stdio.h>
int main(void) {
    int a, b, c;      // 原数字
    int B = 2;        // 进制
    int a0, b0, c0;   // 转化成十进制的a,b,c
    int base, base0; // 用于计算进制转换的中间变量
    scanf("%d%d%d", &a, &b, &c);
```

```
    // 寻找最小的可能存在的B
    base = 1;
    for (int i = 0; i < 6; i++) {
        if (a / base % 10 >= B) {
            B = a / base % 10 + 1;
        }
        if (b / base % 10 >= B) {
            B = b / base % 10 + 1;
        }
        if (c / base % 10 >= B) {
            B = c / base % 10 + 1;
        }
        base *= 10;
    }

    // 遍历每一种进制 计算是否成立
    while (B <= 16) {
        a0 = 0;
        b0 = 0;
        c0 = 0;
        base = 1;
        base0 = 1;
        for (int i = 0; i < 6; i++) {
            a0 += a / base % 10 * base0;
            b0 += b / base % 10 * base0;
            c0 += c / base % 10 * base0;
            base *= 10;
            base0 *= B;
        }
        if ((long long int)a0 * b0 == c0) {
            printf("%d", B);
            break;
        }
        B++;
    }

    // 未找到解
    if (B > 16) {
        printf("0");
    }

    return 0;
}
```

## 示例代码2

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    char a[10], b[10], c[10];
    char *aEnd, *bEnd, *cEnd;
    int a0, b0, c0;
    scanf("%s%s%s", a, b, c);
    for (int B = 2; B <= 16; B++) {
```

```
        a0 = strtol(a, &aEnd, B);
        b0 = strtol(b, &bEnd, B);
        c0 = strtol(c, &cEnd, B);
        if (*aEnd || *bEnd || *cEnd) {
            continue; // 数字在该进制下不存在
        } else if ((long long int)a0 * b0 != c0) {
            continue; // 等式不成立
        } else {
            printf("%d", B);
            return 0;
        }
    }
    printf("0"); // 未找到解
    return 0;
}
```

## G 某咸鱼与中秋节

| 难度 | 考点 |
| --- | --- |
| 5 | 位运算 |

### 题目分析

假设我们有 $2k+1$ 个数字 $a_1, a_1, a_2, a_2, \ldots, a_k, a_k, a_{k+1}$，那么这组数据无论通过何种顺序进行异或运算，都可以调整为 $a_1 \oplus a_1 \oplus a_2 \oplus a_2 \oplus \cdots \oplus a_k \oplus a_k \oplus a_{k+1}$，由 Hint 可知任何数异或自己是0，任何数异或0是其本身，可得结果一定是 $a_{k+1}$，得知只有 $a_{k+1}$ 没有出现两次，从而寻找出来。

要注意本题的内存限制，不能开数组去排序。采用 Hint 中的简化判断方法时，要注意判断 $m$ 的奇偶性以后，还要继续读入后面的 $m$ 个数。

### 示例代码

```
#include<stdio.h>
int main() {
    unsigned int n, m;
    scanf("%u", &n);
    for (unsigned int i = 1; i <= n; i++) {
        unsigned int a = 0, b;
        scanf("%u", &m);
        for (unsigned int j = 1; j <= m; j++) {
            scanf("%u", &b);
            a ^= b;
        }
        if (a == 0) {
            printf("Congratulations!\n");
        }
        else {
            printf("Single Dog! %u\n", a);
        }
    }
    return 0;
```

```
}
```

# H 军乐团分组

| 难度 | 考点 |
| --- | --- |
| 5~6 | 区间平均数，前缀和 |

## 题目分析

这道题实际上就是求一个长度为 $n$ 的数列中取长度不小于 $m$ 的区间，求最大的区间平均值。
首先思考朴素的暴力求平均值的办法，三层循环分别枚举区间长度，区间左端点，和区间内每一个数值（用来累加），也就是：

```
int ans=0, ave;
for(int i=m;i<=n;i++)  //枚举区间长度i
{
    for(int j=1;j<=n-i+1;j++)  //枚举区间左端点j，注意边界条件
    {
        ave=0;  //每次枚举注意ave初始化
        for(int k=j;k<=j+i-1;k++)
        {
            ave+=a[k];  //a[k]存储第i个数
        }
        ave=(ave+m/2)/m;  //求均值
        if(ans<ave)
            ans=ave;  //更新最大值
    }
}
```

但是这样做只能得 $0.1$ 分，原因是三层循环每层都是 $O(n)$ 量级的，总的时间复杂度是 $O(n^3)$ 的，计算机要执行 $500000^3$ 运算，而计算机 $1s$ 只能执行 $10^8$~$10^9$ 的运算，这样的时间复杂度必然 `TLE`。
接下来考虑一些优化，我们发现若是不限制区间长度 $m$，就相当于找数列中的最大值，因为任何平均值一定 $\leqslant$ 单一的最大值。
考虑到这一点我们不难发现对于任何长度 $\geqslant m*2$ 的区间 $[a,b]$，一定能拆成左右两个长度 $\geqslant m$ 的子区间 $[a,k]$ 和 $[k+1,b]$，且这两个子区间中一定有一个区间的均值 $\geqslant$ 整个 $[a,b]$ 的均值（当且仅当两个子区间均值相等时等号成立），因此最大均值一定会出现在长度在 $[m,m*2)$ 范围的某个区间上，最外层循环就可以改成：

```
for(int i=m;i<m*2;i++)
{......}
```

此时最外层和最内层循环量级都减小到了 $O(m)$，总时间复杂度减小到了 $O(nm^2)$，但仍然会 `TLE`。
我们发现由于区间具有连续性，每次用下面这种方式计算区间和做了很多重复计算：

```
for(int k=j;k<=j+i-1;k++)
    ave+=a[k];
```

因此可以利用前缀和数组 `pre[k]` 记录前 $k$ 个同学的声调之和，即

$$pre[k]=a[1]+a[2]+\cdots+a[k]$$

这样计算从第 $l$ 个同学到第 $r$ 个同学的声调之和就是

$$a[l]+a[l+1]+\cdots+a[r-1]+a[r]=pre[r]-pre[l-1]$$

我们只需要在读入时预处理所有的前缀和，就能在取消最内层循环，在 $O(1)$ 的时间内求出区间和，现在总时间复杂度缩减到了 $O(nm)$，刚好可以在时限内完成。

**需要注意的是**，`pre[500000]` 的值可能会达到 $20000*500000=10^{10}$，超过了`int`的范围，因此要用 `long long` 定义 `pre` 数组。

## 示例代码

```
#include <stdio.h>

long long pre[500001];

int main()
{
    int n,m,ans=0,ave,a;
    scanf("%d%d",&n,&m);
    for(int i=1;i<=n;i++)
    {
        scanf("%d",&a);
        pre[i]=pre[i-1]+a;  //前缀和
    }
    for(int i=m;i<m*2;i++)  //枚举平均数最值可能出现的区间长度
    {
        for(int j=0;j<=n-i;j++)  //枚举区间左端点，j为(左端点-1)
        {
            int temp=pre[j+i]-pre[j];  //用前缀和快速求区间和
            ave=(temp+i/2)/i;  //四舍五入（避免了用long long整除）
            if(ans<ave)
                ans=ave;  //更新最大值
        }
    }
    printf("%d",ans);
    return 0;
}
```

# *TIPS*

这里虽然全部 $500000$ 个声调总和会超过 `int` 范围，但任何不超过 $m*2$ 个声调之和都 $<4000000$。因此计算 `pre[j+i]-pre[j]` 的结果可以用 `int` 型变量存储。

**因为 `int` 型变量计算整除会比 `long long` 型变量快 $1$ 倍乃至更多。**

其实还有 $O(n)$ 的做法，将本题转化为横坐标是同学编号 `i`，纵坐标是 `pre[i]` 的二维曲线，寻找曲线上水平跨度不小于 $m$ 的两点之间的最大斜率，从而可以采取斜率优化，学有余力的同学可以查询下面这个链接，继续探索这种实现方式。

[题解 P1404 【平均数】 - distantlight 的博客 - 洛谷博客 (luogu.com.cn)]

# 错误分析

## 这道题有些同学采取如下两种动态规划算法：

## 方法 1：

`dp[k]` 表示不考虑区间最小长度限制时，以 `k` 号同学结尾的最优区间长度（最优即指均值最大），`ans[i]` 表示考虑区间最小长度限制时，以 `i` 号同学结尾的最优区间均值，则：

```
int t=m+dp[i-m];
ans[i]=max((pre[i]-pre[i-m]+m/2)/m , (pre[i]-pre[i-t]+t/2)/t);  //pre为前缀和数组
```

这里问题在于 `ans[i]` 与 `dp[i-m]` 的关系错误。
当 `dp[i-m]` 的方案对 `i-m` 号同学最优时，不一定对 `i-m+1` 至 `i` 号同学最优。
设 `i-m` 号同学左边的 $a$ 个同学声调均值为 $p$，`i-m` 号同学声调为 $q$，`i-m+1` 至 `i` 号同学声调为 $r$。

| 区间长度 | a | 1 | m |
|---|---|---|---|
| 平均值 | p | q | r |

则当 $q > p > r$ 时，一定有 `dp[i-m]==1`，但对于 $r$ 来说，$a$ 个 $p$ 的贡献可能比 1 个 $q$ 的贡献更大，即：

$$\frac{a*p+q+m*r}{a+1+m} > \frac{q+m*r}{1+m}$$

化简得：

$$m \times (p-r) > q-p$$

这是很容易成立的。也就是说，当这个式子与 $q > p > r$ 同时成立时，这种动态归划算法就出错了。
实际上，`i-m` 号同学的“最优”方案应该同时取决于其两侧的同学声调，不能单一考虑一侧，因此该动态规划算法必然错误。

## 方法 2：

`dp[i]` 表示考虑区间最小长度限制时以 `i` 号同学结尾的最优区间长度，则：

```
int t=dp[i-1]+1;
if((pre[i]-pre[i-t]+t/2)/t > (pre[i]-pre[i-m]+m/2)/m)  //pre为前缀和数组
    dp[i]=dp[i-1]+1;
else dp[i]=m;
```

这里问题在于 `dp[i]` 的状态转移方程错误。
`dp[i]` 实际上并不能表示以 `i` 号同学结尾的最优区间长度，并且 `dp[i-1]` 的方案也未必适用于第 `i` 号同学。
设第 `i` 号同学声调为 $x$，`i-m+1` 号同学至 `i-1` 号同学平均声调为 $p$，`i-m-b+1` 号同学至 `i-m` 号同学平均声调为 $q$，`i-m-b-a+1` 号同学至 `i-m-b` 号同学平均声调为 $r$。

| 区间长度 | a | b | m-1 | 1 |
|---|---|---|---|---|
| 平均值 | r | q | p | x |

则当满足

$$\frac{a*r+b*q+(m-1)*p}{a+b+m-1}>\frac{b*q+(m-1)*p}{b+m-1}$$

即：

$$b*r+(m-1)*r>b*q+(m-1)*p$$

时，应该会有 `dp[i-1]==a+b+m-1`。（但由于 `dp[i-1]` 有下述错误，实际上也不一定能取到这个值。）
而当 $x>q>r>p$ 时：对于 $p$ 来说，$a$ 个 $r$ 的贡献可能比 $b$ 个 $q$ 的贡献更大；但对于很大的 $x$ 来说，$q$ 已经弥补上了 $p$ 造成的削弱，再继续扩展区间长度反而会再次削弱 $x$ 本身的优势，即：

$$\frac{b*q+(m-1)*p+x}{b+m}>\frac{a*r+b*q+(m-1)*p+x}{a+b+m}$$

化简得：

$$b*q+(m-1)*p+x>b*r+m*r$$

上述两式都很容易满足，且只要 $x>r$，两式就不会产生矛盾。也就是说当上述两式同时成立时，这种动态规划算法就出错了。
实际上，假设该算法算出某个区间 $[\,L\,,\,R\,]$ 为最大均值区间，其均值为 $ans$，则必有 `dp[R-1]==R-L`。记 $[\,L\,,\,R-1\,]$ 和 $[\,R-m+1\,,\,R\,]$ 的均值中较大值为 $S$，则当 $L$ 左侧增加一些声调介于 $(\,S\,,\,ans\,)$ 之间的同学时，正确的结果不应该再向左扩展，而该算法会向左扩展，因此该动态规划算法必然错误。

# I 式神们夜里不睡觉

| 难度 | 考点 |
| --- | --- |
| 5~6 | 进制转换 |

## 简要题意

实现任意进制转换。

## 问题分析

### 关于任意进制的转换

对于一个任意进制转换的问题，我们首先应当把待转换的数码用我们熟悉的进制来表示，即十进制，否则对于一个字符串我们几乎无法获得任何有效信息。$k$ 进制转十进制的方法较为容易，可以按如下实现：

```
    long long n, len, k1, k2, i;
    char s1[100];//转换前的字符串
    scanf("%lld%lld%s", &k1, &k2, s1);
    len = strlen(s1); //获取字符串s1的长度
    for (i = 0; i < len; i++)
    {
        if (isupper(s1[i]))//如果s1[i]是大写字母
            n = n * k1 + s1[i] - 'A' + 10;
        else
            n = n * k1 + s1[i] - '0';
    }
```

注意到上面使用了两个函数，一个是 `strlen()`，用来获取一个字符串的长度，这个函数包含在 `string.h` 头文件中；一个是 `isupper()`，用来判断一个字符是否是大写字母，这个函数包含在 `ctype.h` 头文件中。当然，判断大写字母也可以写成 `'A'<=s1[i]&&s1[i]<='Z'`。

而对于十进制转 $k$ 进制，对于一般正进制的题目来说，基本思路是将待转换的数不停地对 $k$ 取余，然后将余数倒序输出。然而，在负进制的背景下，这个操作有些许不同之处。

## 关于负进制

首先我们应该知道的是，C 语言中 `%` 运算的含义是对某个数**取余**。注意到，在 C99 标准（即 ACOJ 使用的标准）中有规定除法为趋零截尾，即将整除的结果向 $0$ 的方向将小数部分截断。而对于 `r = a % b`，$r$ 的计算方法一般可以表示为 `r = a - a / b * b`。你可以试试运行下面这个代码：

```
#include <stdio.h>

int main()
{
    printf("%d\n", -10 % -3); //输出-1
    printf("%d\n", 10 % -3); //输出1
    return 0;
}
```

因此，如果我们待转换的整数是一个负数，在取余的过程中余数是可能为负的，而我们不能直接将一个负余数作为转换后的结果直接输出。为了得到一个正余数，我们可以从商“借”一位到余数。

对于一个除数是负数的除法：$(\text{商}+1)\times\text{除数}+(\text{余数}-\text{除数})=\text{被除数}$

由于余数的绝对值一定小于除数的绝对值，所以我们新得到的余数一定是一个正数。在负进制的转换中，我们只要每次判断余数的正负，然后根据相应的方法进行处理即可：

```
    char s0[37] = {"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"};//需要用到的数码
    char s2[100];//转换后的字符串
    int i=0;
    while (n)
    {
        long long reNum = n % k2;//余数
        if (reNum < 0)//余数为负数
        {
            reNum -= k2;//余数变为正数
            s2[i] = s0[reNum];
            n = n / k2 + 1;//商加1
        }
```

```
        else
        {
            s2[i] = s0[reNum];
            n /= k2;
        }
        i++;
    }
    while (i--)//逆序输出
        printf("%c", s2[i]);
```

当然，我们不能忘记 `0` 这个特殊数字，它在任意进制下的表示都为 `0`，特判一下就可以了。

## 参考代码

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define LL long long

char s0[37] = {"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"};

int main()
{
    char s1[100], s2[100];
    LL n, k1, k2, T, len, i;
    scanf("%lld", &T);
    while (T--)
    {
        n = 0;
        scanf("%lld%lld%s", &k1, &k2, s1);
        if (s1[0] == '0')
            printf("0\n");
        else
        {
            len = strlen(s1);
            for (i = 0; i < len; i++)
            {
                if (isupper(s1[i]))
                    n = n * k1 + s1[i] - 'A' + 10;
                else
                    n = n * k1 + s1[i] - '0';
            }
            int i = 0;
            while (n)
            {
                LL reNum = n % k2;
                if (reNum < 0)
                {
                    reNum -= k2;
                    s2[i] = s0[reNum];
                    n = n / k2 + 1;
                }
                else
                {
                    s2[i] = s0[reNum];
```

```
                    n /= k2;
                }
                i++;
            }
            while (i--)
                printf("%c", s2[i]);
            printf("\n");
        }
    }
    return 0;
}
```

## J 染色方案

| 难度 | 考点 |
| --- | --- |
| 7 | 状压dp |

## 题目分析

我们先考虑一个esay version，当 $1 \leq n,k \leq 24$ 时，我们可以用一个二进制下 $n$ 位的数该位的 $0/1$ 来表示这个格子染的是黑色还是白色，我们可以枚举出来所有满足这样条件的数。
再回来考虑原题，我们可以发现每 $k$ 位数依旧满足上面的条件。首先，我们先预处理出来所有满足相邻两位不都为 $1$ 的所有的数，接下来考虑用动态规划转移每 $k$ 位的状态，用 $dp_{i,j}$ 表示截至第 $i$ 位时后 $k$ 位的状态为 $j$ 时的答案，那么我们可以得到转移方程如下（需要判断最后一位是否可以放 $1$：

$$\begin{cases} dp_{i+1,j/2} = dp_{i,j}, \text{最后一位为0的转移} \\ dp_{i+1,j/2+2^{k-1}} = w_i dp_{i,j}, \text{最后一位为1的转移} \end{cases}$$

最后我们将所有的 $dp_{n,j}$ 相加就可以得到最后的答案，需要注意的一点是由于空间的限制，我们需要将 $dp$ 数组的第一维滚动掉，也就是将前一次的结果copy到一个数组中从而得到下一次的结果，由于所有的可能状态的个数为 $F_k$ 个（ $F_k$ 是 斐波那契数列的第 $k$ 项），时间复杂度为 $O(1.618^k n)$，空间复杂度 $O(2 \cdot 1.618^k)$ 。
同时注意在运算的过程中要及时取模以及最后的答案应当去掉所有的数都是 $0$ 的情况。

## 示例代码

```
#include<stdio.h>
#define mod 1000000007
int dp[2][150010],w[310];
int vis[17000010],pos[17000010],v[17000010];
int t,n,k;
int main()
{
    scanf("%d%d",&n,&k);
    long long ans = 0;
    for(int i = 0;i < n;i++)
        scanf("%d",&w[i]);
    int now = 0;
    for(int i = 0;i < (1 << k);i++)//预处理符合条件的状态
    {
```

```
        vis[i] = ((i & 3) == 3) | (vis[i >> 1]);//记录哪些状态存在相邻的1
        if(vis[i] == 0)
            pos[i] = now,v[now] = i,now++;
    }
    dp[0][0] = 1;
    for(int i = 0;i < n;i++)
    {
        for(int j = 0;j < now;j++)
        {
            dp[1][pos[v[j] >> 1]] = (dp[1][pos[v[j] >> 1]] + dp[0][j]) % mod;//最
后一位放0的转移
            if((v[j] & ((1 << (k - 1)) + 1)) == 0)//最后一位放0的转移
                dp[1][pos[(v[j] >> 1) | (1 << (k - 1))]] = (dp[1][pos[(v[j] >> 1)
| (1 << (k - 1))]] + 1ll * w[i] * dp[0][j]) % mod;
        }
        for(int j = 0;j < now;j++)
            dp[0][j] = dp[1][j],dp[1][j] = 0;//滚动掉第一维
    }
    for(int i = 0;i < now;i++)
        ans = (ans + dp[0][i]) % mod;//统计答案
    ans = ((ans - 1) % mod + mod) % mod;//除去都不选的情形
    printf("%lld\n",ans);
    return 0;
}
```

## 思考

可以思考当 $1 \leq n, k \leq 300$ 时应该怎么做?

# - End -