位运算

——加减乘除都弱爆了

十进制与二进制

- 十进制
  - $4396_{10} = 4\times1000 + 3\times100 + 9\times10 + 6\times1$
  - $= 4\times10^3 + 3\times10^2 + 9\times10^1 + 6\times10^0$
- 二进制
  - $1101_2 = 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$
  - $= 1\times8 + 1\times4 + 0\times2 + 1\times1$
  - $= 13_{10}$

位、位运算

- unsigned int类型，32位
  - 十进制：13　　　　　00000000 00000000 00000000 00001101
  - 十进制：65535　　　00000000 00000000 11111111 11111111
- char类型，8位
  - 字符：'A'(=65)　　　01000001
  - 字符：'{'(=123)　　01111011
- 位运算：按位进行计算
- 不像加减乘除那样涉及进位退位等操作

C++的位运算：按位取反

- ~a
- NOT
- 例子(unsigned int):
  - a = 13   00000000 00000000 00000000 00001101
  - ~a   11111111 11111111 11111111 11110010
  - = 4294967282

C++的位运算：左移

- a << b
- SHL
- 例子：
  - a = 13　　　00000000 00000000 00000000 00001101
  - b = 2
  - a << b　　　00000000 00000000 00000000 00110100
  - = 52

C++的位运算：右移

- a >> b
- SHR
- 例子：
  - a = 13　　00000000 00000000 00000000 00001101
  - b = 2
  - a >> b　　00000000 00000000 00000000 00000011
  - = 3

C++的位运算：按位与

- a & b
- AND
- 例子：
  - a = 13 00000000 00000000 00000000 00001101
  - b = 11 00000000 00000000 00000000 00001011
  - a & b 00000000 00000000 00000000 00001001
  - = 9

C++的位运算：按位异或

- a ^ b
- XOR
- 例子：
  - a = 13 00000000 00000000 00000000 00001101
  - b = 11 00000000 00000000 00000000 00001011
  - a ^ b 00000000 00000000 00000000 00000110
  - = 6

C++的位运算：按位或

- a | b
- OR
- 例子：
  - a = 13 00000000 00000000 00000000 00001101
  - b = 11 00000000 00000000 00000000 00001011
  - a | b 00000000 00000000 00000000 00001111
  - = 15

注意区别

- 位运算和逻辑运算
  - 写一段程序：
    - `int a = 13 & 11, b = 13 && 11, c = 13 | 11, d = 13 || 11;`
    - `cout << a << " " << b << " " << c << " " << d << endl;`
  - 结果是“9 1 15 1”
  - a=9，c=15是位运算的结果，计算时13和11都是二进制的整数
  - b=1，d=1是逻辑运算的结果，会把13和11当成true来计算

简单的总结一下

- C++中有6种位运算
  - 按位取反　NOT　~
  - 左移　SHL　<<
  - 右移　SHR　>>
  - 按位与　AND　&
  - 按位异或　XOR　^
  - 按位或　OR　|
- &和&&，|和||的区别
- 整数类型才有位运算

整数的存储

- 计算机中8位整数的存储：

unsigned char	char	存储
0	0	00000000
1	1	00000001
……	……	……
127	127	01111111
128	-128	10000000
129	-127	10000001
……	……	……
255	-1	11111111

整数的存储

- 计算机中32位整数的存储：

unsigned int	int	存储
0	0	00000000 00000000 00000000 00000001
1	1	00000000 00000000 00000000 00000001
……	……	……
2147483647	2147483647	01111111 11111111 11111111 11111111
2147483648	-2147483648	10000000 00000000 00000000 00000000
2147483649	-2147483647	10000000 00000000 00000000 00000001
……	……	……
4294967295	-1	11111111 11111111 11111111 11111111

整数的存储

- 原码：数值在二进制中的写法
    - x = 5　　　　101
    - x = 13　　　　1101
    - x = 127　　　　1111111
    - x = 128　　　　10000000
    - x = -5　　　　-101
    - x = -13　　　　-1101
    - x = -127　　　　-1111111
    - x = -128　　　　-10000000
- 原码存储数据需要单独存储正负号，导致计算加减法需要先判断符号再进行计算，不便于计算机使用

整数的存储

- 补码：用一个“互补”的二进制数表示负数
  - x = 5 00000101
  - x = -5 11111011

  - x = 12 00001100
  - x = -12 11110100

  - x = 127 01111111
  - x = -127 10000001

  - x = 0 00000000
  - x = -128 10000000
- x和-x的二进制码相加，进位后刚好完全溢出，得到0。
- 计算负数的补码：先取反，再加1。
- 负数的补码 = 反码 + 1
  - -x = ~x + 1

整数的存储

- 补码表示法
  - 正数和0用原码
  - 负数用反码再1
- 那么问题来了：
  - 把“10000000”看做正数的原码，则表示128
  - 把“10000000”看做负数的补码，则表示-128
  - “10000000”是到底该表示128呢？还是-128呢？

整数的存储

- 观察除了“10000000”以外的数
  - 0到127，用原码表示，都是“0XXXXXXX”，即最左边的位都是0
  - -1到-127，用补码表示，都是“1XXXXXXX”，即最左边的位都是1
  - 于是规定，最左边的位称为“符号位”
    - 符号位为0，说明这个数是正数或0的原码；
    - 符号位为1，说明这个数是负数的补码；
- 所以，“10000000”是-128的补码。

整数的存储

- 从取模角度来看
- 00000000到11111111，都是x模256之后的数
  - 无符号类型，用0到255来表示x模256后的结果
  - 有符号类型，用-128到127来表示x模256后的结果
- 有符号类型的负数-x，在无符号类型中是256-x
  - -1和255是相同的，它们都是11111111
  - -2和254是相同的，它们都是11111110
  - ……
  - -127和129是相同的，它们都是10000001
  - -128和128是相同的，它们都是10000000

简单的总结一下

- 有符号类型使用补码表示法存储整数
  - 正数和0用原码
  - 负数用补码，补码=反码+1
- 有符号类型的最左边的二进制位是符号位
- n位有符号类型的表示范围是$[-2^{n-1}, 2^{n-1}-1]$，可以看做是模2^n意义下的数。

一个大坑：右移运算+有符号类型

- 无符号类型：
    - unsigned char, unsigned short, unsigned int, unsigned long long...
- 最左边补0
- 例子(unsigned char)：
    - a = 135　　10000111
    - a >> 2　　<u>00</u>100001
    - = 33

一个大坑：右移运算+有符号类型

- 有符号类型：
    - char, short, int, long long...
- 最左边补的数等于原来的符号位
- 例子(char)：
    - a = -121　　10000111
    - a >> 2　　**11**100001
    - = -31
    - b = 63　　00111111
    - b >> 2　　**00**001111
    - = 16

简单的总结一下

- 左移和右移
  - 负数右移时，最左边补1；其他时候都补0。

位运算的优先级

- ~:
  - 比乘(*)、除(/)、取模(%)的优先级高。
- <<、>>:
  - 比比较运算(<,>,<=,>=,==,!=)优先级高;
  - 比加(+)、减(-)优先级低。
- &,^,|:
  - 这三个是优先级由高到低,
  - 比比较运算(<,>,<=,>=,==,!=)优先级低,
  - 比逻辑运算(&&,||)优先级高。

位运算的优先级

- if(5 >> 1 & 1 == 0)
  - 先计算5 >> 1和1 == 0，得到2和false;
  - 把false强制类型转换0;
  - 计算2 & 0，得到0;
  - 判断结果是false。
- if((5 >> 1 & 1) == 0)
  - 先算5 >> 1，得到2;
  - 再算2 & 1，得到0;
  - 再算0 == 0，得到true;
  - 判断结果是true。

优先级

- C++有四大类运算符：
  - 单目运算符
  - 双目运算符
  - 三目运算符
  - 赋值运算符
- 四个大类的优先级由高到低。
  - ~a * b
    - ~属于单目运算符，所以优先级比所有双目运算符都高，所以先算~，再算*。
  - a <<= b ? c : d
    - <<=属于赋值运算符，运算符比三目运算符?:低，所以先算?:，再算<<=。

位运算的常用操作

- 针对x二进制的第i位
  - 获取，得到0或1　　`x >> i & 1`
  - 获取，得到2^i　　`x & 1 << i`
  - i=0时判断奇偶　　`x & 1`
  - 取反　　`x ^ 1 << i`
  - 置为1　　`x | 1 << i`
  - 置为0　　`x & ~(1 << i)`
- 针对x二进制的前i位
  - 获取　　`x & (1 << i) - 1`
  - 取反　　`x ^ (1 << i) - 1`
  - 置为1　　`x | (1 << i) - 1`
  - 置为0　　`x & ~((1 << i) - 1)`

位运算的常用操作

- 把x二进制低位的连续的0都变成1
    - `x | x - 1`
- 把x二进制低位的连续的1都变成0
    - `x & x + 1`
- 枚举子集
    - `for (T = S; T != 0; T = S & T - 1) ...`
    - 这样写可以枚举除了空集0以外的S的所有子集T
    - 如果既要跳过空集0，又要跳过全集S，可以这样写
    - `for (T = S & S - 1; T != 0; T = S & T - 1) ...`

位运算的常用操作

- 获取x二进制的最低位（树状数组的lowbit）
  - 得到1,2,4,8,16等，不是得到0,1,2,3,4等
  - x & -x
- lowbit(13) = 13 & -13 = 1
  - 13 00001101
  - -13 11110011
- lowbit(12) = 12 & -12 = 4
  - 12 00001100
  - -12 11110100
- lowbit(8) = 8 & -8 = 8
  - 8 00001000
  - -8 11111000

位运算的常用操作

- 构造常数
  - unsigned int:
    - 上限 ~0u　　-1u　　0xffffffff
  - int:
    - 上限 ~0u >> 1　　-1u >> 1　　0x7fffffff
    - 下限 1 << 31　　0x80000000
  - long long:
    - 上限 ~0ull >> 1　　-1ull >> 1　　0x7fffffffffffffff
    - 下限 1ll << 63　　0x8000000000000000
- 交换两个整数的值（仅供娱乐）
    - a = a ^ b; b = a ^ b; a = a ^ b;
    - a = a + b; b = a - b; a = a - b;

位运算的常用操作

- 代替乘、除、模
  - ×2、×4、×8等：
    - x << 1　　　x << 2　　　x << 3　　　……　　　x << i
  - ÷2、÷4、÷8等：
    - x >> 1　　　x >> 2　　　x >> 3　　　……　　　x >> i
    - C++中，除法是向0取整，负数时就是向上取整，例如-13 / 4答案是-3；
    - 使用>>无论正负都是向下取整，例如-13 >> 2答案是-4。
  - %2、%4、%8等：
    - x & 1　　　x & 3　　　x & 7　　　……　　　x & (1 << i) - 1
    - C++中，负数取模答案在0到-x+1范围内，例如-13 % 4答案是-1；
    - 使用位运算代替取模会得到非负的结果，例如-13 & 3答案是3。
- 配套使用/和%、>>和&，“被除数=除数×商+余数”都成立。

简单的总结一下

- ~的优先级很高，<<和>>的优先级比较低，&,^,|的优先级很低
- <<=,>>=,&=,^=,|=的优先级和普通赋值是相同的，都非常低
- 常用操作需要背下来，滚瓜烂熟
  - 操作从低到高第i位
  - 操作最低的i个位
  - 操作低位连续的0或1
  - 枚举子集
  - lowbit
  - 构造常数
  - 代替乘、除、模

一个简单的问题

- 给出一个小于2^{32}的正整数，这个数写成32位二进制数后，前16位称为“高位”，后16位称为“低位”。将他的高位和低位交换，可以得到一个新的数，这个新的数是多少？
  - 例如1314520用二进制表示是00000000 00010100 00001110 11011000，高位是00000000 00010100，低位是00001110 11011000，交换后得到00001110 11011000 00000000 00010100，是十进制的249036820。

一个简单的问题

- 低位要移动到高位的位置上，需要左移16位；
- 高位要移动到低位的位置上，需要右移16位；
- 把两个移动的结果合并起来，用|运算。
- 程序非常简单
    - ```
      unsigned int n;
      cin >> n;
      cout << (n << 16 | n >> 16);
      ```
      ```

又一个简单的问题

- 有n（n是个很大的奇数）个整数，其中某个数字出现了奇数次，其他数字都出现了偶数次。找出这个出现了奇数次的数字，空间限制$O(1)$，时间限制$O(n)$。
  - 例：n=9
    6 2 5 6 5 2 6 3 6

- n很大，显然不能把这些数存下来，只能在读入的同时进行计算。
- 如何计算？

又一个简单的问题

- 6 2 5 6 5 2 6 3 6
- 6^2^5^6^5^2^6^3^6 = 3
- 为什么全部异或起来，恰好就是出现奇数次的那个数？
- x ^ x = 0，偶数次的数会被全部消掉，奇数次的数只保留1次。
- 程序非常简单

```
ans = 0;
for (i = 1; i <= n; i ++)
    scanf(“%d”, &k), ans ^= k;
printf(“%d\n”, ans);
```

又一个简单的问题

- 问题变化一下：有1个数出现了3k+1次或者3k+2次，其他的数出现了3k次，找出出现3k+1次或者3k+2次的这个数，以及它出现的次数。
- n很大，空间限制$O(1)$，时间限制$O(n)$。

又一个简单的问题

- 考虑每个二进制位，扫一遍所有数之后
  - 如果某个二进制位上，1出现了3k+0次，则那个数的二进制的这一位是0；
  - 如果某个二进制位上，1出现了3k+1次，则那个数出现了1次，且那个数的二进制的这一位是1；
  - 如果某个二进制位上，1出现了3k+2次，则那个数出现了2次，且那个数的二进制上这一位是1。
  - 如果既有位上1出现3k+1次，又有位上1出现3k+2次，说明程序写错了；
  - n可能是3k+1或3k+2，可以帮助判断那个数的次数。
- 开两个变量a1,a2，分别记录哪些位上1出现了3k+1和3k+2次，每次读进来一个x后做相应的更新操作。
  - 新a1 = (a1 ^ (a1 & x)) | (x ^ (x & (a1 | a2)))
  - 新a2 = (a2 ^ (a2 & x)) | (a1 & x)

再一个简单的问题

- 统计x的二进制中“1”出现的次数。
  - 例：x=13，二进制1101，“1”出现了3次。
- 如何让这个问题算的飞快？

再一个简单的问题

- 方法1：枚举所有位
  - ```
    ans = 0;
    while (x > 0) {
        ans += x & 1;
        x >>= 1;
    }
    ```
- 对于64位整数，最坏需要64次循环，平均需要63次循环。
- 很慢。
    ```

再一个简单的问题

- 方法2：枚举“1”出现的位
  - ```
    ans = 0;
    while (x > 0) {
        ans ++;
        x -= x & -x;
    }
    ```
- 对于64位整数，最坏需要64次循环，平均需要32次循环。
- 还是很慢。
    ```

再一个简单的问题

- 方法3：打表
  - 预处理：
    ```
    int bit_cnt[65536];
    for (int i = 1; i < 65536; i ++)
        bit_cnt[i] = bit_cnt[i >> 1] + (i & 1)
    ```
  - 每次查询：
    ```
    bit_cnt[x & 65535] + bit_cnt[x >> 16 & 65535]
    + bit_cnt[x >> 32 & 65535] + bit_cnt[x >> 48 & 65535];
    ```
- 预处理需要一定的时间和空间，但每次查询非常快。
- 可以根据需求调整打表的大小，表越小预处理越快但查询越慢。

一个有趣的问题

- 假设现在需要一种位运算，假设符号是$，满足：
  - 0 $ 0 = a   0 $ 1 = b
  - 1 $ 0 = c   1 $ 1 = d
- 其中a,b,c,d是0或者1。
- 任意一组a,b,c,d的值，能不能将表达式“x $ y”用x,y以及位运算组成的表达式来代替？
  - 例如a = 1, b = 0, c = 0, d = 1;
  - 可以用“~x ^ y”来代替“x $ y”。
- 对a,b,c,d的16种情况分别给出表达式。

一个无聊的答案

- 一种方法
  - x$y当且仅当x=0且y=0时得到1：~x&~y
  - x$y当且仅当x=0且y=1时得到1：~x&y
  - x$y当且仅当x=1且y=0时得到1：x&~y
  - x$y当且仅当x=1且y=1时得到1：x&y
- 假如现在需要满足
  - 0$0=a=1，0$1=b=0，1$0=c=0，1$1=d=1
  - 令x$y = (~x&~y)|(x&y)，满足题意
- 用a,b,c,d来写表达式
  - x$y = (~x&~y&a)|(~x&y&b)|(x&~y&c)|(x&y&d)
- 使用了~,&,|三种位运算

一个同样无聊的答案

- 一种方法
  - x$y当且仅当x=0且y=0时得到0：x|y
  - x$y当且仅当x=0且y=1时得到0：x|~y
  - x$y当且仅当x=1且y=0时得到0：~x|y
  - x$y当且仅当x=1且y=1时得到0：~x|~y
- 假如现在需要满足
  - 0$0=a=1，0$1=b=0，1$0=c=0，1$1=d=1
  - 令x$y = (x|~y)&(~x|y)，满足题意
- 用a,b,c,d来写表达式
  - x$y = (x|y|a)&(x|~y|b)&(~x|y|c)&(~x|~y|d)
- 还是使用了~,&,|三种位运算

更有趣的问题，更无聊的答案

- $x_0, x_1, \ldots, x_{n-1}$都是01变量，$f(x_0, x_1, \ldots, x_{n-1})$是一个函数，函数的输出也是0或1，求$f$的表达式。
- 例如$n = 3$，函数f的输出如下

```
f(0,0,0)=0  f(0,0,1)=1  f(0,1,0)=1  f(0,1,1)=0
f(1,0,0)=0  f(1,0,1)=0  f(1,1,0)=0  f(1,1,1)=1
```

- 仿照前面的方法，可以写出表达式：

```
f(x1,x2,x3)=(~x1 & ~x2 & x3)|(~x1 & x2 & ~x3)|(x1 & x2 & x3)
f(x1,x2,x3)=(x1 | x2 | x3) & (x1 | ~x2 | ~x3) & (~x1 | x2 | x3)
            & (~x1 | x2 | ~x3) & (~x1 | ~x2 | x3)
```

- 类似的，任意的n和任意的函数f都可以写出表达式。

完备性

- 定义：对于任意的n和任意的函数f，都可以写出位运算表达式，表达式中只包含某些位运算，那么称这些位运算具有完备性。
  - 刚才表达式中只包含~, &, | 三种运算，这三种运算具有完备性。
- 那么问题来了：只有两种运算能不能具有完备性？一种呢？

完备性

- 已知：~，&，|三种运算具有完备性。
- 已知：x|y = ~(~x&~y)在任何时候都成立。
- 在刚才的方法中，所有|运算可以全部替换为~和&的组合，表达式中只包含~和&。
- 结论：~和&两种运算具有完备性。

- 已知：x&y = ~(~x|~y)在任何时候都成立。
- 在刚才的方法中，所有&运算可以全部替换为~和|的组合，表达式中只包含~和|。
- 结论：~和|两种运算具有完备性。

完备性

- 单独一种位运算，能否具有完备性？
- 与非　　NAND　　↑
  - x↑y = ~(x&y)
    - ~x = ~(x&x) = x↑x
    - x&y = ~(x↑y) = (x↑y)↑(x↑y)
  - 与非运算具有完备性
- 或非　　NOR　　↓
  - x↓y = ~(x|y)
    - ~x = ~(x|x) = x↓x
    - x|y = ~(x↓y) = (x↓y)↓(x↓y)
  - 或非运算也具有完备性

简单总结一下

- 已知函数f，如何利用~, &, | 三种运算构造表达式
- 如何用~, &两种运算代替 |
- 如何用~, | 两种运算代替&
- 如何用↑一种运算符代替~和&
- 如何用↓一种运算符代替~和 |

彻底总结一下

- C++的六种位运算，它们的优先级
- 整数的存储，负数右移的大坑
- 位运算的常用操作
- 表达式的构造方法，完备性

习题

- 3671, 3672, 3673, 2134, 3099, 3679, 3680