回归本源——位运算及其应用

镇海中学 沈洋

摘 要

在电子计算机中，位运算可以说是比四则运算更基本的运算，但由于它们不是数学的基本运算且不常在生活中应用，容易被部分选手，特别是初学信息学的选手忽视。本文希望通过对位运算及其应用的介绍，来使这一情况得到一定的改善。

本文大致可分为三个部分。第一部分（第1节）对位运算基本概念进行了介绍，第二部分（第2至4节）对位运算的几个应用进行了介绍，第三部分（第5节）展示了一些例题。

几点说明

- 约定二进制最高位为最左边，最低位为最右边，最低位记为第0位。
- 本文中的所有程序均以C++语言描述，所涉及的语言细节、内建函数等均以GNU C++（GCC编译器）为标准。
- 若无特别说明，本文中的位运算使用C++运算符表示。
- 本文中汇编指x86汇编。

1 基本运算

1.1 与、或、非和异或

对于逻辑运算的“与”、“或”、“非”和“异或”，想必大家都耳熟能详了。其中“与”（$\wedge$）、“或”（$\vee$）、“异或”（$\oplus$）的真值表如下：

p	q	$p \wedge q$	$p \vee q$	$p \oplus q$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

“非”（¬）的真值表如下：

p	$\neg p$
0	1
1	0

位运算的“与”、“或”、“非”和“异或”可以看作逻辑运算在整数上的扩展。它们的运算法则如下：

与 运算结果的每个二进制位的值等于两个运算数的对应二进制位进行逻辑“与”运算的结果。

或 运算结果的每个二进制位的值等于两个运算数的对应二进制位进行逻辑“或”运算的结果。

非 运算结果的每个二进制位的值等于运算数的对应二进制位进行逻辑“非”运算的结果。

异或 运算结果的每个二进制位的值等于两个运算数的对应二进制位进行逻辑“异或”运算的结果。

这四种位运算对应的汇编指令如下：

与	或	非	异或
and	or	not	xor

由此可以看出，对于这四种运算，带符号整数和无符号整数并没有对应不同的汇编指令，它们的行为是相同的。也就是说，对于带符号整数，符号位也会与其它位按照同样的方式处理。

1.2 移位运算

移位运算分为逻辑移位、算数移位、循环移位、带进位循环移位四种。其中带进位循环移位涉及到CF标志位，因此不在我们的讨论范围内。为了准确区分各种移位运算，在这一部分中我们使用汇编指令来表示它们。

当我们说将x左移或右移y位时，默认y非负且小于x的位宽[1]。例如对于32位整数，y只能在0到31之间取值。

逻辑移位

逻辑左移对应的汇编指令为shl x y，它的功能是把x的每个二进制位向左移动y位，移动造成的最右边的空位由0补足，最左边的数溢出[2]。

逻辑右移与逻辑左移完全相反，它对应的汇编指令为shr x y，它的功能是把x 的每个二进制位向右移动y位，移动造成的最左边的空位由0 补足，最右边的数溢出。

算术移位

算术左移的汇编指令为sal x y，它与逻辑左移完全相同。

算术右移的汇编指令为sar x y，它与逻辑右移大体相同，唯一的区别在于移动造成的最左边的空位由符号位（最高位）补足而不是由0 补足。

循环移位

循环左移对应的汇编指令为rol x y，它的功能是把x的每个二进制位向左移动y位，移动造成的最右边的空位由最左边溢出的位补足。

循环右移与循环左移完全相反，它对应的汇编指令为ror x y，它的功能是把x 的每个二进制位向右移动y位，移动造成的最左边的空位由最右边溢出的位补足。

移位运算的数学意义

在逻辑移位、算术移位这两个看似复杂的定义背后实际上是有其数学含义的。逻辑移位被设计用于处理无符号整数，它的左移和右移分别与无符号整数的$x \times 2^y$ 和$x \div 2^y$ 具有相同的效果；算术移位被设计用于处理带符号整数，它的

[1]若超出这个范围，汇编指令只考虑y的后几位，C语言将产生未定义行为。

[2]实际上，最后一个溢出的位会被放入标志位CF中，其他标志位也会相应发生变化，但这不在我们的讨论范围内，故在本文中略去对这些部分的描述。

左移和右移分别与带符号整数的$x \times 2^y$和$x \div 2^y$具有相同的效果（舍入方式为向下舍入）。这一点根据补码的定义比较容易证明，在此就不赘述了。

C++中的移位运算

在C++中，无符号整数的移位使用逻辑移位，有符号整数的移位使用算术移位。这样一来，无论是无符号整数还是带符号整数，我们都可以放心的使用左移和右移来代替乘以二的幂或除以二的幂的操作。

C++中没有专门的对带符号整数进行逻辑右移的运算符，不过我们可以通过强制类型转换将它转换成无符号整数后再进行运算。

C++中没有提供循环移位操作符，但我们可以通过其他运算的组合来实现它。例如对于32位无符号整数x，表达式`(x << y) | (x >> (32 - y))`可以实现循环左移的功能，而`(x >> y) | (x << (32 - y))`则可以实现循环右移的功能。

2 二进制位的修改和查询

借由汇编指令、内建函数以及基本位运算的组合，我们可以在二进制位的层面上对整数进行一些修改和查询。本节将介绍一些常用操作并给出一些易于理解和扩展的实现[3]。由于带符号整数的性质较为复杂，在这里我们只讨论无符号整数的相关操作，并以常用的32位无符号整数为例。

2.1 读某些位

读取x的第pos个二进制位是一个很常用的操作，它的实现方式是先将x右移pos位，使要读取的位移到最低位，再通过 `& 1` 将其取出。代码如下[4]：

```
bool readBit(u32 x, int pos) {
    return (x >> pos) & 1;
}
```

[3]一些“黑技术”如利用64位指令处理32位操作数、利用浮点数进行计算等并未涉及。对此感兴趣的读者可以参考http://graphics.stanford.edu/%7Eseander/bithacks.html

[4]为简化代码，我们使用u32来表示32位无符号整数(unsigned int)。

有时候我们会将多个整数“打包”在一个整数中，一个典型的应用是将四个字节打包为一个32位整型，那么读取的时候就需要形如“读x的第pos 位开始的cnt位”这样的操作。它的实现方式与上面别无二致：首先将x右移pos位，再通过`& mask` 来取出最后cnt位，其中`mask`的后cnt个位为1，其余位为0，可以由`(1 << cnt) - 1`得到。代码如下：

```
u32 readBits(u32 x, int pos, int cnt) {
    return (x >> pos) & ((1u << cnt) - 1);
}
```

通过上面两个例子，不难发现读取某个或某些二进制位的关键实际上是与运算。通过将原数和一个遮罩进行与运算，可以达到保留指定一些位（将遮罩的对应位设为1），清零其它位（遮罩的对应位设为0）的目的。

2.2 改某些位

对二进制位的修改实际上是“与”、“或”和“异或”运算的实际应用。

将某些位置为1

要实现这个功能，我们的办法是将原数与一个遮罩进行或运算。对于要改为1 的位，我们将遮罩的对应位设为1，否则将对应位设为0，这样，原数与遮罩进行或运算的结果就是答案。举例来说，将x的第pos 位置为1的代码如下：

```
u32 setBit(u32 x, int pos) {
    return x | (1u << pos);
}
```

将某些位置为0

这个操作与上一个操作正好相反，这一次我们要利用的是与运算。我们构造一个遮罩，对于要修改的位，我们将遮罩的对应位设为0，否则将其设为1（注意，这与上一个操作中的遮罩完全相反），随后将原数与这个遮罩进行与运算即可得到答案。将x的第pos位置为0 的代码如下：

```
u32 clearBit(u32 x, int pos) {
    return x & ~(1u << pos);
}
```

取反某些位

这一次是异或运算的应用。同样构造一个遮罩，如果我们要取反某位，则将遮罩的对应位设为1，否则将其设为0，随后将原数与这个遮罩进行异或运算即可得到答案。将x的第pos位取反的代码如下：

```
u32 flipBit(u32 x, int pos) {
    return x ^ (1u << pos);
}
```

2.3 求1的个数

分治法

求二进制位中1的个数与求各个二进制位的和是等价的，因此我们转而思考如何求各个二进制位的和。我们可以使用分治来解决这个问题：每次将整个数分成两个部分，分别求出每个部分的和，再将它们相加。利用位运算，我们可以并行地完成每一层的工作，像下面这样：

```
int bitCount_1(u32 x) {
    x = ((x & 0xAAAAAAAAu) >>  1) + (x & 0x55555555u);
    x = ((x & 0xCCCCCCCCu) >>  2) + (x & 0x33333333u);
    x = ((x & 0xF0F0F0F0u) >>  4) + (x & 0x0F0F0F0Fu);
    x = ((x & 0xFF00FF00u) >>  8) + (x & 0x00FF00FFu);
    x = ((x & 0xFFFF0000u) >> 16) + (x & 0x0000FFFFu);
    return x;
}
```

它是这样工作的：

- 第一步，有32个项需要相加，每一项占1 bit。我们将其中的奇数项和偶数项分别取出来（利用2.1 小节中提到的方法），并将奇数项右移1位和偶数项“对齐”，然后将他们相加。这一步过后我们实际上将16对1 bit的项分别相加，并将结果存放在了原来这两项所在的2 bit空间上。

- 第二步，有16个项需要相加，每一项占2 bit。我们将奇偶项分别相加，形成8个4 bit的项。

  ……

- 第五步，有两项需要相加，每一项占16 bit。将这两项相加我们便得到了答案。

那么，还能再优化么？答案是肯定的。看下面这段代码：

```
int bitCount_2(u32 x) {
    x-= ((x & 0xAAAAAAAAu) >> 1);
    x = ((x & 0xCCCCCCCCu) >> 2) + (x & 0x33333333u);
    x = ((x >>  4) + x) & 0x0F0F0F0Fu;
    x = ((x >>  8) + x) & 0x00FF00FFu;
    x = ((x >> 16) + x) & 0x0000FFFFu;
    return x;
}
```

这段代码较上一段的第一个差别是第一步的实现。我们知道第一步的作用是将奇数位和偶数位相加，根据第一个程序它的结果应为：

```
((x & 0xAAAAAAAAu) >> 1) + (x & 0x55555555u)
```

而x的值等于：

```
((x & 0xAAAAAAAAu) + (x & 0x55555555u)
```

将两式作差得到：

```
(x & 0xAAAAAAAAu) >> 1
```

因而将x减去`(x & 0xAAAAAAAAu) >> 1`便可以得到我们所求的结果。

另一个差别是第三步之后的实现，我们以第三步为例来说明。第三步的作用是将4对4 bit整数相加，但实际上此时每个整数最大只可能是4，这就表示，即使是两个数相加的结果也能在4 bit的空间存下。因此`(x >> 4) + x`可以正

确地依次求出第0个数加第1个数，第1个数加第2个数，第2个数加第3个数……我们从中取出我们需要的结果即可。

还能继续优化么？答案依然是肯定的：

```
int bitCount_3(u32 x) {
    x-= ((x & 0xAAAAAAAAu) >> 1);
    x = ((x & 0xCCCCCCCCu) >> 2) + (x & 0x33333333u);
    x = ((x >> 4) + x) & 0x0F0F0F0Fu;
    x = (x * 0x01010101u) >> 24;
    return x;
}
```

这段代码与上一段的不同点在于，上一段代码的最后两步操作被语句：

```
x = (x * 0x01010101u) >> 24
```

代替了。我们知道上一段代码中第四、第五步的作用是将x中的4个8 bit数相加，那么，如果我们令：

```
x = (a<<24)+(b<<16)+(c<<8)+(d<<0)
```

则我们所求的答案便是：

```
a + b + c + d
```

我们来看看计算x * 0x01010101u时发生了什么：

```
  x * 0x01010101
=(x<<24)+(x<<16)+(x<<8)+(x<<0)
=(a<<48)+(b<<40)+(c<<32)+(d<<24)+
         (a<<40)+(b<<32)+(c<<24)+(d<<16)+
                 (a<<32)+(b<<24)+(c<<16)+(d<< 8)+
                         (a<<24)+(b<<16)+(c<< 8)+(d<< 0)
=((a)       << 48) +
 ((a+b)     << 40) +
 ((a+b+c)   << 32) +
 ((a+b+c+d)<< 24) +
 ((b+c+d)   << 16) +
 ((c+d)     << 8 ) +
 ((d)       << 0 )
```

由于x是32位整数，因此上式的前三项应当溢出，于是x * 0x01010101的

结果为((a+b+c+d)<<24) + ((b+c+d)<<16) + ((c+d)<<8) + (d)考虑到a,b,c的实际意义，它们的值都不会超过8，因此上式的后三项的值必定小于1 << 24，也就是说它们不会对结果的最高8位造成任何影响。因此，x * 0x01010101的结果的高8位便是我们所求的a+b+c+d，我们将它右移24位即可得到答案。

查表法

如果我们要求0到n中每一个数的答案的话，那么可以使用如下递推：

```
f(0) = 0,
f(i) = f(i >> 1) + (i & 1)
```

显然对于所有32位整数保存答案（至少在目前看来）是一件不太现实的事情，不过好在这个问题是可以分割的，我们可以预处理所有16位整数的答案，在询问时将被询问的整数拆成高16位和低16位分别计算答案。于是预处理的代码如下：

```
int cnt_tbl[65537];
void bitCountPre() {
    cnt_tbl[0] = 0;
    for (int i=1; i<65536; ++i)
        cnt_tbl[i] = cnt_tbl[i >> 1] + (i & 1);
}
```

回答询问的代码如下：

```
int bitCount_4(u32 x) {
    return cnt_tbl[x >> 16] + cnt_tbl[x & 65535u];
}
```

内建函数

GCC中提供了如下内建函数[5]来实现这个功能：

```
int __builtin_popcount (unsigned int x);
```

对于支持SSE4.2的机器，如果在编译时开启相应开关，则该函数会被翻译成汇编指令popcnt，否则该函数使用类似上面“查表法”的方法进行计算。

2.4 翻转位序

对于一个32位整数来说，翻转位序是指将它的第0位与第31位交换，第1位与第30位交换，……，第i位与第31 – i位交换，……，第15 位与第16 位交换。例如对于32 位整数5，对它进行翻转位序将得到2684354560。

分治法

我们可以采用分治法来解决这个问题：首先将整个数分割成两个部分，分别翻转这两个部分，再将这两个部分对调。同样地，借由位运算，我们可以把每一层的工作并行完成，实现如下：

```
u32 bitRev_1(u32 x) {
    x = ((x & 0xAAAAAAAAu) >>  1) | ((x & 0x55555555u) <<  1);
    x = ((x & 0xCCCCCCCCu) >>  2) | ((x & 0x33333333u) <<  2);
    x = ((x & 0xF0F0F0F0u) >>  4) | ((x & 0x0F0F0F0Fu) <<  4);
    x = ((x & 0xFF00FF00u) >>  8) | ((x & 0x00FF00FFu) <<  8);
    x = ((x & 0xFFFF0000u) >> 16) | ((x & 0x0000FFFFu) << 16);
    return x;
}
```

这段代码是这样工作的：

- 第一步，交换相邻两位。这样就形成了16个长度为2的已经翻转的组。
- 第二步，每两位为一组，交换相邻两组。这就将16个长度为2的组变成了8个长度为4的已翻转的组。

[5]关于本文中提到的GCC内建函数的具体说明及其他相关内建函数，请参阅GCC手册：http://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

……

- 第五步，每16位为一组，交换相邻两组。整个数翻转完成。

查表法

我们首先预处理出所有16位整数翻转后的结果，这可以由如下递推得到：

```
f(0) = 0,
f(i) = (f(i >> 1) >> 1) | ((i & 1) << 15)
```

接下来，我们将待翻转的32位整数拆成两个16位整数，通过查表来得到这两个16位整数的答案，再将它们对调便可以完成整个32位整数的翻转。预处理的代码如下：

```
u32 rev_tbl[65537];
void bitRevPre() {
    rev_tbl[0] = 0;
    for (int i=1; i<65536; ++i)
        rev_tbl[i] = (rev_tbl[i>>1]>>1) | ((i&1)<<15);
}
```

查询的代码如下：

```
u32 bitRev_2(u32 x) {
    return (rev_tbl[x & 65535] << 16) | rev_tbl[x >> 16];
}
```

2.5 求前缀/后缀0的个数

二分法

求前缀0与求后缀0是类似的，我们不妨以求前缀0为例。首先给出代码：

```
int countLeadingZeros_1(u32 x) {
    int ans = 0;
    if (x >> 16) x >>= 16; else ans |= 16;
    if (x >>  8) x >>=  8; else ans |=  8;
    if (x >>  4) x >>=  4; else ans |=  4;
    if (x >>  2) x >>=  2; else ans |=  2;
    if (x >>  1) x >>=  1; else ans |=  1;
    return ans + !x;;
}
```

这段代码的实质是二分查找。它是这样工作的：

- 第一步，判断高16位是否为空。若是，则高16位必然均为前缀0，我们给答案加上16，并在第0至15位上继续二分；若不是，则前缀0只出现在则高16位上，我们将它们右移到低16位上以便对它继续二分。
- 第二步，判断此时的高8位是否均为前缀0，并选择一边继续二分下去。

  ……

- 第六步，判断仅剩的那一位是否为0，若是，则给答案加1。

求后缀0个数的方式与此别无二致，因此根据上一段代码炮制一段求后缀0个数的代码并不是件难事：

```
int countTrailingZeros_1(u32 x) {
    int ans = 0;
    if (!(x & 65535u)) x >>= 16, ans |= 16;
    if (!(x & 255u  )) x >>=  8, ans |=  8;
    if (!(x & 15u   )) x >>=  4, ans |=  4;
    if (!(x & 3u    )) x >>=  2, ans |=  2;
    if (!(x & 1u    )) x >>=  1, ans |=  1;
    return ans + !x;
}
```

这段代码的工作原理与上一段基本相同，就不再赘述了。

查表法

我们依然以前缀0个数为例。这个问题与同样是可以分割的。我们可以先确定第一个1出现在高16位中还是低16位中，再通过查表来得到这个16 位整数的答案。这张表可以通过如下递推得到：

```
    f(0) = 16,
    f(i) = f(i >> 1) - 1
```

于是我们就不难写出预处理的代码：

```
int clz_tbl[65537];
void countLeadingZerosPre() {
    clz_tbl[0] = 16;
    for (int i=1; i<65536; ++i)
        clz_tbl[i] = clz_tbl[i >> 1] - 1;
}
```

以及查询的代码：

```
int countLeadingZeros_2(u32 x) {
    if (x >> 16)
        return clz_tbl[x >> 16]; else
        return clz_tbl[x & 65535u] + 16;
}
```

求后缀0个数也可以通过几乎相同的方法实现。

内建函数

对于这两个问题，GCC也提供相应的内建函数。

求前缀0的内建函数为：

```
    int __builtin_clz (unsigned int x);
```

它对应的汇编指令为bsr (Bit Scan Reverse)外加一个异或。

求后缀0的内建函数为：

```
    int __builtin_ctz (unsigned int x);
```

它对应的汇编指令为bsf (Bit Scan Forward)。

需要注意的是，当参数x为0时，这两个函数的行为是未定义的。

2.6 求第k个1的位置

这个问题可以转化为求最大的w，使得第0位到第$w-1$中1的个数小于k。这显然是可以二分的。于是，借助2.3小节中的cnt_tbl，我们便能实现一个简单的二分：

```
int kthBit_1(u32 x, int k) {
    int ans = 0;
    if (cnt_tbl[x & 65535u] < k)
        k -= cnt_tbl[x & 65535u], ans |=16, x >>=16;
    if (cnt_tbl[x & 255u  ] < k)
        k -= cnt_tbl[x & 255u  ], ans |= 8, x >>= 8;
    if (cnt_tbl[x & 15u   ] < k)
        k -= cnt_tbl[x & 15u   ], ans |= 4, x >>= 4;
    if (cnt_tbl[x & 3u    ] < k)
        k -= cnt_tbl[x & 3u    ], ans |= 2, x >>= 2;
    if (cnt_tbl[x & 1u    ] < k)
        k -= cnt_tbl[x & 1u    ], ans |= 1, x >>= 1;
    return ans;
}
```

不借助cnt_tbl自然也是可以的。我们可以发现，我们在二分的过程中所需要的区间和都是2.3小节中分治法计算过的区间，因此我们可以将分治的中间值存下来供二分时使用，像下面这样：

```
int kthBit_2(u32 x, int k) {
    int s[5], ans = 0, t;
    s[0] = x;
    s[1] = x - ((x & 0xAAAAAAAAu) >> 1);
    s[2] = ((s[1] & 0xCCCCCCCCu) >> 2) + (s[1] & 0x33333333u);
    s[3] = ((s[2] >>  4) + s[2]) & 0x0F0F0F0Fu;
    s[4] = ((s[3] >>  8) + s[3]) & 0x00FF00FFu;
    t = s[4] & 65535u;
    if (t < k) k -= t, ans |=16, x >>=16;
    t = (s[3] >> ans) & 255u;
    if (t < k) k -= t, ans |= 8, x >>= 8;
    t = (s[2] >> ans) & 15u;
    if (t < k) k -= t, ans |= 4, x >>= 4;
    t = (s[1] >> ans) & 3u;
    if (t < k) k -= t, ans |= 2, x >>= 2;
    t = (s[0] >> ans) & 1u;
    if (t < k) k -= t, ans |= 1, x >>= 1;
    return ans;
}
```

2.7 提取末尾连续的1

我们知道对x加1之后，最右边连续的1会变为0，最右边的0则变为1，而其它位不变。利用这一点我们可以通过下面这个公式来提取x末尾连续的1：

```
    x & (x ^ (x + 1))
```

2.8 提取lowbit

正整数x的lowbit是指x在二进制下最右边一个1开始至最低位的那部分，记为lowbit(x)。例如28 (11100B)[6]的lowbit 为4 (100B)，16 (10000B) 的lowbit则为16 (10000B)。lowbit的典型应用是树状数组和遍历集合。

有如下三个公式能帮我们求解lowbit：

```
lowbit(x) = x & (x ^ (x - 1))
lowbit(x) = x ^ (x & (x - 1))
lowbit(x) = x & -x
```

前两个公式利用了“将x减去1后，x中最右侧的1变为0，其之后的0变为1”这一点，第三个公式则主要利用了补码的特性。读者可以自行完成这三个公式的证明。

2.9 遍历所有1

一般情况下，我们可以通过不断求lowbit并将它从原数中删去（通过异或）来遍历每一个1。当需要用到这个1所在的位置时，我们可以通过2.5小节中介绍的求后缀0个数的方法来获知。

3 将整数用作集合

集合的种类有很多，但将它们的元素标号之后，都能映射到整数集合上。因此在这里我们只讨论全集为$[0,n)$的整数集合。

二进制的每个位有0和1两种状态，这正好可以对应集合中某个元素是否存在，因此我们可以用二进制数来表示集合——如果某位为1就表示对应元素存在于这个集合中，否则不存在。然而计算机中单个二进制数的位数是有限的，不妨设其能处理的单个二进制数的最大位宽为w，那么，表示上述全集为$[0,n)$的整数集合就需要至少$\lceil n/w \rceil$个二进制数。我们把每个二进制数叫做一“块”，并规定第i块记录第wi至$w(i+1)-1$这个w 个数是否存在（$0 \le i < \lceil n/w \rceil$，最后一块略有不同）。这样，我们便得到了一个维护集合的数据结构——bitset。我们将某个bitset能表示的全集的大小称为该bitset的大小。

[6]我们用后缀B来表示一个二进制数。

从本质上来说，bitset实际上是压w位的二进制高精度整数，因此也可以进行与、或、非、异或、左移、右移这些位运算，甚至可以进行加减乘除。

3.1 加入某个元素

首先计算出被加入元素所属的块，再使用2.2小节所述的方法将对应位置为1。

时间复杂度：O(1)

3.2 删除某个元素

首先计算出被删除元素所属的块，再使用2.2小节所述的方法将对应位置为0。

时间复杂度：O(1)

3.3 查询某个元素是否存在

首先计算出待查询元素所属的块，再利用2.1小节所述的方法检查对应位是否为1。

时间复杂度：O(1)

3.4 交集

集合的交和bitset的与运算相对应。`x`与`y`的交为`x & y`。

时间复杂度：$O(n/w)$

3.5 并集

集合的交和bitset的或运算相对应。`x`与`y`的并为`x | y`。

时间复杂度：$O(n/w)$

3.6 差集

记A、B是两个集合，则所有属于A且不属于B的元素构成的集合叫做A与B的差。集合的差可以通过bitset的与运算和异或运算实现。`x`与`y`的差为`x ^ (x & y)`。

时间复杂度：O(n/w)

3.7 补集

集合的补与bitset的非运算相对应。x的补为~x。

时间复杂度：O(n/w)

3.8 统计元素个数

对于每个块分别使用2.3小节中介绍的方法计算1的个数，再将所有块的答案相加。

时间复杂度：O(n/w)

3.9 遍历集合元素

遍历所有块，对每个块使用2.9小节中介绍的方法遍历元素。

时间复杂度：O($n/w + cnt$)，其中cnt表示集合的元素个数。

3.10 求集合中第k小元素

朴素做法

首先从小到大依次遍历每个块，找出第k小元素所在的块，再利用2.6 小节中介绍的方法确定具体是哪个元素。

时间复杂度：O($n/w + \log w$)

更加高效的做法

在某些应用中，求第k小元素可能比其它集合操作的使用频率更大，此时我们就会需要一个复杂度更优的查询第k小元素操作，同时要尽可能小地影响其它集合操作的复杂度。

一种改进方式是利用线段树。我们使用一颗辅助线段树来维护bitset中每个块的元素个数。这样一来，我们就可以通过在线段树上二分的方式快速确定第k小元素所属的块。这是一个经典的线段树操作，在这里就不再赘述。找出

第k小元素所在的块之后，我们就可以利用2.6小节中介绍的方法确定具体是哪个元素了。这样改进之后，求第k小元素的时间复杂度降为了$O(\log n)$。

添加了辅助线段树之后，其他的集合操作也要做出相应修改。在加入、删除集合元素时，该元素所属的块的元素个数会发生改变，我们需要同时维护辅助线段树的信息，因此这两个操作的时间复杂度变为了$O(\log n)$。在集合求交、并、差、补的时候，我们可以直接暴力重建辅助线段树，时间复杂度维持不变为$O(n/w)$。

3.11 求集合中大于x的最小元素

朴素做法

首先判断x所在块中是否存在大于x的元素，若是则直接返回这个元素，否则从x所在的块开始向后遍历，找到x之后第一个包含元素的块，然后返回该块的第一个元素。其中，某块的第一个元素以及某块中比x大的第一个元素均可以利用2.5小节中介绍的统计后缀0个数的方法求出。

时间复杂度：$O(n/w)$

更加高效的做法

类似于求第k小元素，我们也可以利用辅助数据结构来加速大于x的最小元素的查询。

这次我们利用的是bitset本身。我们使用一个大小为n/w的辅助bitset[7]来保存每个块是否有元素，这样一来，我们可以通过如下流程来完成该查询：

1. 若x所在块中存在大于x的元素，则直接返回这个元素，否则转2。
2. 利用辅助线段树查询x所在块之后第一个含有元素的块，转3。
3. 在该块中找第一个元素并返回结果。

改进之后，该查询的时间复杂度降为了$O(\log_w n)$。

同样地，其他的集合操作也要做出相应修改。在加入、删除集合元素时，我们需要同时维护辅助bitset中该元素所在块的信息，因此这两个操作的时间复

[7]这里的bitset是递归定义的，因此这个辅助bitset也支持快速求集合中大于x的最小元素。

杂度变为了$O(\log_w n)$。在集合求交、并、差、补的时候，我们可以直接暴力重建辅助bitset，时间复杂度维持不变为$O(n/w)$。

3.12 将集合的全部或部分元素加上/减去x

将集合的全部元素加上/减去x

对集合a的每个元素都加上x只需要将a左移x位即可。同样的，对集合a的每个元素都减去x 只需要将a右移x位即可。

时间复杂度：$O(n/w)$

将集合的部分元素加上/减去x

设原集合为a，要求改动的那部分元素组成的集合为b。在这次修改中，a与b的差这部分元素是不会改变的，记为集合p；a与b的交这部分元素是需要加上/减去x的，记为集合q。之后我们利用移位操作对集合q作相应修改，再与集合p求并即可。

时间复杂度：$O(n/w)$

3.13 枚举子集

枚举子集的关键在于，对于某个集合的某个子集，我们需要求出字典序排在它前一位的集合或后一位的集合。对于集合x和它的一个子集y，我们可以通过(y - 1) & x来求出字典序排在它前一位的子集。因此，若要枚举x的子集，我们只需从x本身开始，不断求前一个子集，直到枚举到空集为止即可。

3.14 枚举含有k个元素的集合

与枚举子集类似，枚举含有k个元素的集合的关键同样是对于某个含有k个元素的集合求出字典序排在它前一位或后一位的集合。这一次我们选择的是构造字典序后一位的集合。

朴素的做法如下：设当前集合中的数从小到大依次为$a_0, a_2, \ldots, a_{k-1}$，其中最小的可以增大的数为$a_j$，那么我们把$a_j$ 加上1，并将a_0到a_{j-1}重置为0到$j-1$即可得到下一个包含k 个元素的集合。

这个做法反映到bitset上是这样的：设最右边连续的1是第i位到第j位，那么我们要做的就是将第j位上的1向左移一位，同时将第i位到第$j-1$位上的1向右移到最右边。这可以由下面这段代码完成：

```
u32 nextComb(u32 x) {
    u32 l = x & -x, y = x + l;
    return y | (((x ^ y) / l) >> 2);
}
```

它是这样工作的：

- 第一步，使用`l = x & -x`求出`x`的lowbit。
- 第二步，`y = x + l`将第j位上的1向左移动一位，同时将第i位到第$j-1$位置为0（i和j的含义见上文）。
- 第三步，`((x ^ y) / l) >> 2`将第i位到第$j-1$位上的1提取出来并移到最右边。
- 第四步，将这些1与`y`合并得到答案。

4 其他应用

除上两节所介绍的应用外，我们还可以利用位运算做一些其他事情。

4.1 判断奇偶性

根据奇偶性的定义，二进制最低位为0的数为偶数，最低位为1的数为奇数，因此我们可以直接使用`x & 1`来判断`x`的奇偶性。若`x & 1`等于`1`则表明`x`为奇数，否则`x`为偶数。

4.2 乘以或除以二的幂

根据移位运算的实际意义，我们可以直接利用左移和右移实现，参见1.2 小节中的相关介绍。

4.3 对二的幂取模

$x \bmod 2^y$ 相当于取出x的后y位，因此可以用`x & ((1 << y) - 1)`来实现相同的效果。

4.4 求$\log_2 x$的整数部分

设x的位宽为w，使用2.5小节中介绍的方法计算出的前缀0个数为l，那么$\lfloor \log_2 x \rfloor = w - 1 - l$。

4.5 交换两个数

这是异或的经典应用。我们可以通过如下代码来交换`a,b`：

```
a ^= b, b ^= a, a ^= b
```

4.6 比较两个数是否相等

我们可以计算两个数的异或值，若等于零则说明这两个数相等，否则说明这两个数不相等。

4.7 取绝对值

对于一个w位带符号整数x，它的符号位`sign = x >> (w - 1)`，我们可以通过表达式`(x ^ -sign) + sign`来得到x的绝对值。这一点根据补码的定义比较容易证明，在此就不赘述了。

4.8 选择

我们可以利用表达式`y ^ ((x ^ y) & -cond)`来实现选择的功能：

- 若$cond = 1$，其结果为x
- 若$cond = 0$，其结果为y

5 例题

5.1 筷子

试题来源

经典问题

试题大意

有$2n+1$个整数，其中某个数出现了奇数次，其他数出现了偶数次。求出现了奇数次的那个数。

算法介绍

由于异或运算满足交换律，并且满足$x \oplus x = 0$，因此将所有数异或起来即为答案。

5.2 Robot in Basement

试题来源

Codeforces 97D[8]

Yandex.Algorithm 2011 Finals

试题大意

在一个$n \times m$ $(n, m \leq 150)$的网格上，有一些格子是障碍，并且网格边界上的格子均是障碍，另有一个非障碍的格子是出口。有一个机器人可以根据程序在该网格上行走。一段程序是一个由UDLR 四种指令组成的字符串，机器人会依次执行每个指令，一个指令会使机器人向指定的方向移动一格，如果对应格子为障碍则不动。

现在给定一个长度为l $(l \leq 10^5)$的程序，求它的一个最短前缀p，使得对于一开始在网格图上任意非障碍位置的机器人，在执行完程序p之后都停在出口上。

[8]http://codeforces.com/problemset/problem/97/D

算法介绍

首先不难想到一个O(nml)的朴素算法。我们直接顺序执行给定的程序，并按照题目指定的移动方式暴力维护网格上哪些格子有机器人，直到执行完某一步后发现网格上只有出口位置有机器人时结束。

实际上我们可以利用bitset来维护网格上哪些格子有机器人。一种表示方式是利用bitset的第$i \cdot m + j$个位置表示格子(i, j) $(0 \le i < n, 0 \le j < m)$上是否有机器人。这样一来一个向左或向右移动的指令就对应了该bitset的左移或右移1位，一个向上或向下的指令就对应了该bitset的左移或右移m位。障碍的问题可以这样解决：预处理cL, cR, cU, cD四个bitset，分别表示向左、向右、向上及向下走一步会碰到障碍的位置。以向左走为例，设原bitset为`x`，那么执行一个指令L之后会撞到障碍的位置为`x & cL`，能正常向左走的位置为`x ^ (x & cL)`，我们知道能正常向左走的那些机器人应该被左移一位，而那些会撞到障碍的机器人则应留在原地，因此最终的bitset应变为`(x & cL) | (x ^ (x & cL))`。改进后的算法的时间复杂度为$O\left(\frac{nml}{w}\right)$，可以通过本题。

5.3 Quick Tortoise

试题来源

Codeforces 232E[9]

试题大意

在一个$n \times m$ $(n, m \le 500)$的网格上，有一些格子是障碍。共有q $(q \le 6 \cdot 10^5)$个询问，每次询问是否能只通过向下走和向右走从格子(x_1, y_1)走到格子(x_2, y_2)。

算法介绍

首先考虑所有询问满足$x_1 \le p \le x_2$的情况。对于所有在第p行上方的格子(x, y)，求它能走到第p行的哪些格子，记这个点集为$f_{x,y}$，它可以通过递推式$f_{x,y} = f_{x+1,y} \cup f_{x,y+1}$求出。类似地，对于所有在第$p$行下方的格子$(x, y)$，求第$p$行的哪些格子能走到它，记这个点集为$g_{x,y}$，它可以通过$g_{x,y} = g_{x-1,y} \cup g_{x,y-1}$求

[9]http://codeforces.com/problemset/problem/232/E

出。求出$f_{x,y}$和$g_{x,y}$之后，我们就可以来处理询问了。对于询问$(x_1, y_1), (x_2, y_2)$，若$f_{x_1,y_1} \cap g_{x_2,y_2} \neq \varnothing$，则表示可以从$(x_1, y_1)$走到$(x_2, y_2)$，反之则不能。

对于一般情况，我们可以使用分治转化成上面的情况进行求解。例如我们正在处理所有满足$l \le x_1 \le x_2 \le r$的询问，令$mid = \left\lfloor \frac{l+r}{2} \right\rfloor$，我们可以先按照上一段所述的方法处理所有满足$l \le x_1 \le mid \le x_2 \le r$的询问，再递归地处理剩下的满足$l \le x_1 \le x_2 < mid$的询问和满足$mid < x_1 \le x_2 \le r$的询问。如果我们只对$x$轴进行分治，那么这个算法时间复杂度为$\mathrm{O}\left(\frac{n \cdot m^2}{w} \log n + q \log n\right)$。如果我们交替对$x$轴和$y$轴进行分治，它的时间复杂度可以进一步降为$\mathrm{O}\left(\frac{(nm)^{1.5}}{w} + q \log nm\right)$

5.4 Bags and Coins

试题来源

Codeforces 356D[10]

试题大意

有n ($n \le 70000$)个包，每个包可以直接放在地上，也可以放在其他包里面，并且允许多层嵌套。有s ($s \le 70000$)个硬币分布在这n个包里。如果拿出某个硬币必须要打开第i个包，我们就称该硬币在第i个包里。现在已知每个包里的硬币数，第i个包里有a_i ($a_i \le 70000$)个硬币。求一种满足条件的包的嵌套方式以及硬币分布方式，或确定问题无解。

算法介绍

不妨设$a_1 \ge a_2 \ge \ldots \ge a_n$。

我们首先考虑$s = a_1$的情况。在这种情况下，我们只需要将第1个包放在地上，并在第i ($1 \le i < n$)个包中放入$a_i - a_{i+1}$ 个硬币以及第$i+1$个包，最后在第n个包中放入a_n个硬币即可。

于是对于一般情况，即$s \ge a_1$的情况，问题就转化成了选一些包放在地上(第1个包是必选的)，让它们的硬币个数之和等于s。这是一个经典的背包问题。我们可以用一个bitset来表示当前背包（通过当前添加的物品能够达到哪

[10]http://codeforces.com/problemset/problem/356/D

些总体积），设当前bitset为S，那么添加一个体积为x的物品后的bitset就会变为S | (S << x)。当添加完所有物品之后，我们检查最终的bitset中是否包含s这个元素就能确定是否有解。

如何记录方案呢？我们引入一个$from$数组，其中$from[i]$表示在某种体积达到i的方案中最后加入的物品。这样一来，如果问题有解，我们就可以通过$from$数组不断追溯到上一个加进来的物品，也就可以恢复方案了。$from$ 数组的维护也很容易，在添加第i个物品的时候，设添加之前的bitset为S，之后为Sn，记Sn与S的差为D，我们枚举D的每个元素j，并将$from[j]$ 设为i即可。由于D的实际意义是加入第i个物品时新产生的可行体积的集合，因此$from$数组的每个元素至多被设置一次，也就保证了复杂度。

该算法的时间复杂度为$O\left(\frac{ns}{w}\right)$。

参考文献

[1] Sean Eron Anderson, “Bit Twiddling Hacks”.

[2] Christer Ericson, “Advanced bit manipulation-fu”.

[3] 刘汝佳, 黄亮, 《算法艺术与信息学竞赛》, 清华大学出版社。