DataLab Report By 2019201408陈志朋

1、bitXor

求x与y按位异或所得的结果，只能使用~，&。

按照异或的运算法则，两个二进制数的对应位置相同，则结果的对应位置为0，否则结果的对应位置为1。由于只能使用~和&运算符，那么可以先找出两个数中都为0的位置，和都为1的位置，这些位置在结果中都是0。在全1串中，将这些位置设置成0即可得到答案。

代码如下（7步）：

```
int bitXor(int x, int y) {
  int ans;
  ans = ~(~x & ~y);
  ans = (ans & ~(x & y));
  return ans;
}
```

2、evenBits

将一个二进制的偶数位置设置为1，只能使用！，~，&，^，|，+，<<，>>。

因为可以使用$[0,255]$范围内的常数，所以容易使最低的8位二进制位符合要求，即0x55。运用倍增的思想，可以通过将之前得到的8位符合要求的二进制串左移8位，在与其本身取或，这样就能得到长度为16位的符合要求的二进制串。同理，可以左移16位后取或，得到最终答案。

代码如下（4步）：

```
int evenBits(void) {
  int ans = 0x55;
  ans = ans | (ans << 8);
  ans = ans | (ans << 16);
  return ans;
}
```

3、fitsShort

判断32位整数x是否能被16位的二的补码表示，只能使用！，~，&，^，|，+，<<，>>。

若一个数能被16位的二的补码所表示，那么这个数在$[-32768,32767]$的范围内。当这个数被32位的二的补码表示时，其15~31位（从0开始标号）都需要与符号位相同，否则这个数就不在所需的范围内。

得到这个性质之后，问题转化为了一个二进制数的15~31位是否相同。设所需要判断的数为x。先通过x ^ (x << 1)，将x中连续且相同的位设置成0（**边界除外**）。那么如果x能被16位的二的补码表示，则处理后其16~31位应该全部为0。将处理后得到的结果右移16位，判断是否为0即可。

代码如下（4步）：

```
int fitsShort(int x) {
  int ans;
  ans = (x ^ (x << 1));
  ans = ans >> 16;
  return !ans;
}
```

4、isTmax

判断一个有符号的32位整数是否为2147483647，只能使用！，~，&，^，|，+。

思路一：

考虑2147483647的二进制表示，除了符号位为0，其他的所有二进制位都是1。所以当这个数加上它自己再加1之后等于0，是这个数为Tmax的一个必要条件。通过计算发现，除了Tmax，有且仅有−1也满足这个条件。所以我们得到了判断一个数是否为Tmax的充要条件：**一个数加上它自己再加1后等于0，且这个数不是−1，那么这个数为Tmax**。

代码如下（6步）：

```
int isTmax(int x) {
  int tmp, ans;
  tmp = x + 1;
  ans = x + tmp;
  return !((!tmp) | (~ans));
}
```

思路二：

思路一的步数较多，考虑如何减少计算步数。

观察思路一的代码可以发现，第4行我们需要的ans的结果应该为一个全1的二进制串，这造成了最终判断需要对当前的ans取反才能进行，造成了步数的冗余。所以，对代码的优化应该使得再最终判断时，我们需要的ans的结果为一个全0的二进制串。

再次观察Tmax的二进制表示，发现当Tmax取反再乘2时（乘2可以用加法来替代），得到的是全0的二进制串。通过计算发现，除了Tmax之外，仍然是有且仅有−1满足该条件。所以，我们得到了一个数是否为Tmax的另一个充要条件：**一个数取反再乘2的结果为0，且这个数不是-1，那么这个数为Tmax**。通过优化，我们再进行最终判断的时候无需在对之前的结果取反，进而减少了一步运算操作。

代码如下（5步）：

```c
int isTmax(int x) {
  int tmp, ans;
  tmp = (~x);
  ans = tmp + tmp;
  ans = !(!tmp | ans);
  return ans;
}
```

5、fitsBits

判断一个有符号的32位整数数是否能够表示成n位的二的补码，只能使用！，~，&，^，|，+，<<，>>。

这道题的思路和第3题fitBits的思路相似，判断一个数的第(n-1)~31位（从0开始标号）是否与符号位相同即可。先将该数算术右移$(n-1)$位，然后将原数算术右移31位得到一个全为符号位的二进制串，比较二者是否相同即可。

代码如下（6步）：

```c
int fitsBits(int x, int n) {
  int ans, tmp;
  tmp = x >> (n + (~0));
  ans = tmp ^ (x >> 31);
  return !ans;
}
```

考虑左移的性质，一个32位整数当左移一个数时，会将这个数对32取模后再进行操作，所以再模32的意义下，$n-1=n+31$。所以代码的第3行可以进行更改，从而减少一次运算操作。

代码如下（5步）：

```c
int fitsBits(int x, int n) {
  int ans, tmp;
  tmp = x >> (n + 31);
  ans = tmp ^ (x >> 31);
  return !ans;
}
```

6、upperBits

将一个32位二进制的最高n位全部设置为1，其他的位全部设置为0，只能使用！，~，&，^，|，+，<<，>>。

按照题目要求，当$n=0$的时候，序列全部为0；其他情况，序列至少有一个1。所以对于n是否为0需要分开判断。当$n=0$时，先构造一个全零的序列；否则构造一个最高位为1的序列。由于已经存在一个1了，所以最后只需要算术右移$(n-1)$为即可。减一的操作同样运用第5小题的方式减少一步运算。

代码如下（5步）：

```c
int upperBits(int n) {
  int ans;
  ans = (!!n) << 31;
  ans = ans >> (n + 31);
  return ans;
}
```

7、allOddBits

要求判断一个32位二进制数的偶数二进制位是否全部为1，只能使用！，~，&，^，|，+，<<，>>。

由于可以使用$[0,255]$范围内的常数，所以我们可以用0xaa进行扩展，扩展为偶数二进制位都是1的序列。然后将原数与构造出来的数取与，即去除奇数位的干扰。然后用异或操作判断处理后的原数与构造的常数是否一致即可。

代码如下（7步）：

```c
int allOddBits(int x) {
  int aob = 0xaa, ans;
  aob = aob | (aob << 8);
  aob = aob | (aob << 16);
  ans = x & aob;
  ans = ans ^ aob;
  return !ans;
}
```

8、byteSwap

将一个32位二进制数的第n个字节和第m个字节的内容交换，只能使用！，~，&，^，|，+，<<，>>。

在64位机器中，一个字节共有8个二进制位，所以第n个字节的起点为$n << 3$（从0开始标号）。通过对原数的右移，可以得到第n个字节和第m个字节的内容（两个8位的二进制数）。将这两个数异或之后，得到一个新的数。根据异或运算的性质，将这个数移动到原数的对应位置再次异或即可交换这两个字节的内容。

代码如下（10步）：

```c
int byteSwap(int x, int n, int m) {
  int t0, tn, tm, tnx, tmx, tmp, ans;
  t0 = 0xff;
  tn = n << 3;
  tm = m << 3;
  tnx = (x >> tn);
  tmx = (x >> tm);
  tmp = (tnx ^ tmx) & t0;
  ans = x ^ (tmp << tn);
  ans = ans ^ (tmp << tm);
  return ans;
}
```

9、absVal

求一个数的绝对值，只能使用！，~，&，^，|，+，<<，>>。

如果传入的数为正数，即符号位为0，则这个数的绝对值等于它本身。如果传入的数为为负数，即符号位为1，根据负数的表示法，这个数的绝对值等于它减1在取反。

由于正数与负数取绝对值时的操作不同，且程序中无法只用if语句，所以考虑在操作过程中，运用符号位来使正负数的操作一致。根据取反运算的性质，当一个数与全1串异或时，相当于对这个数取反；当一个数与全0串异或时，相当于不变。同时，全1串表示十进制的-1，全0串表示十进制的0。所以，比较自然的想到构造一个每一位都与符号位相同的二进制数，用异或来代替取反操作，用加上构造的二进制数来代替减1操作。

代码如下（3步）：

```
int absVal(int x) {
  int tmp, ans;
  tmp = x >> 31;
  ans = x + tmp;
  ans = ans ^ tmp;
  return ans;
}
```

10、divpwr2

将一个数除以2^n并向零取整，只能使用！，~，&，^，|，+，<<，>>。

考虑32位有符号数的性质，所有数除以2^n均向下取整，所以正数无需进行操作，只有负数需要进行操作。为了让负数也能够做到向零取整，需要将负数加上2^n-1即可。为了区分正负数，只需要构造一个全为符号位的二进制串，将2^n-1与其取与即可。

代码如下（6步）：

```
int divpwr2(int x, int n) {
  int tmp, sign, ans;
  tmp = x >> 31;
  sign = tmp & 1;
  ans = x + (sign << n) + tmp;
  ans = ans >> n;
  return ans;
}
```

11、leastBitPos

求一个32位二进制数的最低位的1，只能使用！，~，&，^，|，+，<<，>>。

求最低位的1，相当于将其取之后，求最低位的0。将原数取反后加1，能够将末尾连续的1转化成0，最低位的0转化成1。此时问题再次转化为了求最低位的1。由于处理前后的两个数，有且仅有在最低位均为1，所以将两数相与即可。

当输入的数为0时，经检查，发现原算法仍然适用。

代码如下（3步）：

```
int leastBitPos(int x) {
  int ans;
  ans = ((~x) + 1) & x;
  return ans;
}
```

12、logicalNeg

将一个数进行逻辑上的取反操作，只能使用~，&，^，|，+，<<，>>。

由第11题的性质发现，当一个数与它取反加一的数取或，得到的结果为从原数最低位的1开始，到最高位全为1。那么，如果一个数不是零，操作时候符号位会被设置位1，否则符号位仍然为0。所以可以通过判断处理后的数的符号位来达到取反的效果。将处理后的数算术右移31位，得到一个每一位都和符号位相同的数。容易发现，如果原数为0，那么得到的数为全零序列；否则，得到的数为全一序列。所以，将得到的数加一即可实现逻辑取反。

代码如下（5步）：

```
int logicalNeg(int x) {
  int tmp;
  tmp = x | ((~x) + 1);
  return (tmp >> 31) + 1;
}
```

13、bitMask

把一个32位二进制数从lowbit位开始到highbit位都设置为1，特别的，当highbit<lowbit，则不进行操作。只能使用~，&，^，|，+，<<，>>。

用全1序列左移lowbit位后，得到的序列从lowbit位开始到最高位全为1，其他位全为0的序列。用全1序列左移highbit+1（先左移highbit位，在左移1位，防止出现左移32位，导致结果出错的情况，后文所有的左移highbit+1位均为分开操作）位后，得到的序列从highbit位开始到最低位全为0，其他位全为1的序列。考虑将两个序列取异或得到了一个答案。

代码如下：

```
int bitMask(int highbit, int lowbit) {
  int ans, tmp;
  ans = (~0);
  tmp = ans << lowbit;
  ans = ans << highbit;
  ans = ans << 1;
  ans = ans ^ tmp;
  return ans;
}
```

但是，上述代码是**有问题**的，无法处理highbit小于lowbit的情况。考虑到我们需要的是全1序列左移lowbit位后为1的位置，且全1序列左移highbit+1位后为0的位置。由于有且仅有这种情况是我们需要的，而异或操作或导致与其相反的情况也被选取。根据所需要情况的特性，采取将全1序列左移highbit+1位后取反和全1序列左移lowbit位取与来解决这个问题。

代码如下（6步）：

```
int bitMask(int highbit, int lowbit) {
  int ans, tmp;
  ans = (~0);
  tmp = ans << lowbit;
  ans = ans << highbit;
  ans = ans << 1;
  ans = ~ans;
  ans = ans & tmp;
  return ans;
}
```

14、isLess

判断第一个数是否比第二个小，只能使用！，~，&，^，|，+，<<，>>。

当两个数的符号不同时，可以通过符号快速判断这两个数的大小。如果第一个数是正数，则第一个数大于第二个数。如果第一个数是负数，那么第一个数小于第二个数。先将两个数分别算术左移31位，得到全为符号位的序列。然后将得到的序列异或即可判断两个数的符号位是否相同。

当两个数的符号相同时，可以用第一个数加上第二个数的取反加一（相当于用第一个数减去第二个数）。由于这两个数符号相同，所以保证了两数相减不会溢出导致符号位的改变。那么，只要判断相减之后的符号位是否为1即可。

代码如下（12步）：

```
int isLess(int x, int y) {
  int tmpx, tmpy, sign, ans, diff;
  tmpx = x >> 31;
  tmpy = (~y) + 1;
  diff = tmpx ^ (y >> 31);
  ans = !(diff & (~tmpx));
  sign = (x + tmpy) >> 31;
  ans = ans & (diff | sign);
  return ans;
}
```

15、logicalShift

求一个32位二进制数的逻辑右移的结果，只能使用！，~，&，^，|，+，<<，>>。

由于$0 \leq n \leq 31$（n表示逻辑右移的位数），所以可以得到一个快速计算$31 - n$的方法：**用31与n异或后加一**。

当一个数为正数时，其逻辑右移与算术右移的结果相同。当一个数为负数时，其算术右移的最高若干位（右移的位数）为1，而逻辑右移的为0。由此可见，我们进行操作的方法与数的符号位有关。

首先，可以将原数算术右移31位得到一个全部位符号位的序列，再将得到的序列先左移 $31-n$ 位（n表示逻辑右移的位数），再左移1位（防止左移32位导致结果错误）。处理得到的序列为最高的n位为符号位，剩下的每个位置都为0。根据异或运算的性质，将处理后的序列与原数异或即可。

代码如下（6步）：

```
int logicalShift(int x, int n) {
  int ans, tmp;
  tmp = (x >> 31);
  tmp = tmp << (0x1f ^ n);
  tmp = tmp << 1;
  ans = (x >> n) ^ tmp;
  return ans;
}
```

16、satMul2

将一个数乘二，如果向上溢出就返回最大值，如果向下益处就返回最小值，否则返回原数乘二的值。只能使用！，~，&，^，|，+，<<，>>。

将一个数乘二与将一个数右移一位是等价的，是否溢出只需要判断右移前后其符号位是否发生改变即可。将右移前后的两个数异或后算术右移31位，记这个序列为s1（对应程序中的flag变量），如果溢出得到一个全1的序列，否则得到一个全0的序列。

通过分类发现，溢出时所返回的数符号位与原数相同，其余位于符号位相反。那么只需要将原数乘二的数算术右移31位后最高位取反（即异或1<<31）即可。当原数乘二溢出时，通过该方法得到的值为所需要返回的值；当原数乘二未溢出时，虽无法得到正确的最大值（或最小值），但是这个值对结果没有作用，所以不影响答案的正确性。

得到了原数乘二后的数字和最大值（或最小值），接下来需要分类讨论以返回正确的答案。如果溢出了，那么原数乘二是一个没用的数字。如果没溢出，那么最大值（或最小值）是一个没用的数。发现，当s1是否为全1序列是，原数乘二没用，而最大值（或最小值）有用，否则相反。那么将s1的反与原数乘二取与，将s1与最大值（或最小值）取与，二者再取或即可实现分类讨论的效果。

代码如下（10步）：

```
int satMul2(int x) {
  int s2, tmp, ans, flag, maxn;
  tmp = x << 1;
  s2 = tmp >> 31;
  flag = (x ^ tmp) >> 31;
  ans = tmp & (~flag);
  maxn = s2 ^ (1 << 31);
  ans = ans | (maxn & flag);
  return ans;
}
```

17、subOK

判断第一个参数减去第二个参数是否会造成溢出，只能使用！，~，&，^，|，+，<<，>>。

先将第二个参数取反加一，将原问题转化为两个数相加是否会造成溢出。显然，当两个数的符号不一样时，这两个数相加不会溢出。当两个数的符号相同时，尝试将这两个数相加。如果符号位发生了改变，那么发生了溢出；否则没发生溢出。

根据异或和算术右移的性质，(x>>31) ^ (y>>31) = (x ^ y) >> 31。所以可以运用这个性质将运算操作减少。

代码如下（10步）：

```
int subOK(int x, int y) {
  int tmpx, tmpy, sum, ans;
  tmpy = (~y) + 1;
  ans = ((x ^ y) >> 31) & 1;
  sum = x + tmpy;
  ans = ans & ((x ^ sum) >> 31);
  return !ans;
}
```

18、bitParity

判断一个32位二进制数中1的个数的奇偶性，只能使用！，~，&，^，|，+，<<，>>。

由于偶数个1异或在一起的结果为0，奇数个1异或在一起的结果为1。所以将0 ~ 15位和16 ~ 31位异或，如果对应的两个位置均为1，则互相抵消，否则会生成一个1。然后对0 ~ 15位继续处理即可。最后判断第0位的即、偶性即可。

代码如下（11步）：

```
int bitParity(int x) {
  int ans = x;
  ans = ans ^ (ans >> 16);
  ans = ans ^ (ans >> 8);
  ans = ans ^ (ans >> 4);
  ans = ans ^ (ans >> 2);
  ans = ans ^ (ans >> 1);
  return ans & 1;
}
```

19、isPower2

判断一个数是否为2的幂，只能使用！，~，&，^，|，+，<<，>>。

根据题目要求，如果一个数是负数或零时，那么其一定不是2的幂。

当输入的数为正数时，其二进制表示上有且仅有一个一。可以用第11题（leastBitPos）的方法，求出这个数的最低位的一。然后将求出的数与原数取异或，判断其和原数是否相等即可。

代码如下（11步）：

```
int isPower2(int x) {
  int t1, t2, ans, sign;
  sign = !(x >> 31);
  t1 = ((~x) + 1) & x;
  t2 = !(t1 ^ x);
  ans = sign & t2 & (!!t1);
  return ans;
}
```

附加题1 float_i2f

未做。

附加题2 leftBitCount

求一个32位的二进制数，其从最高位开始的连续的1的个数。只能使用！，~，&，^，|，+，<<，>>。

思路一

将原数右移1位后与原数取与后赋值给原数，然后右移2位后进行相同操作。依次类推，得到的数有且仅有从最高位开始连续的1，剩下的位置全部都是0。然后判断每一位是否为1，累计即可。

代码如下（106步）：

```
int leftBitCount(int x) {
  int tmp, ans = 0;
  tmp = x;
  tmp = tmp & (tmp >> 1);
  tmp = tmp & (tmp >> 2);
  tmp = tmp & (tmp >> 4);
  tmp = tmp & (tmp >> 8);
  tmp = tmp & (tmp >> 16);

  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
```

```
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  ans = ans + (tmp & 1); tmp = tmp >> 1;
  return ans;
}
```

思路二

由于思路一的算法过于暴力，导致运算操作次数过多，所以沿着思路一继续优化算法。先运用与思路一相同的方法，将原序列处理成有且仅有从最高位开始连续的1，剩下的位置全部都是0。将这个序列取反，计算取反后的序列的1的个数。最后的答案等于32减去计算得到的结果。

计算过程采用分治的思想，先处理最低的16位，如果这16位全部为1（用 ^ 操作判断），那么将答案加上16（可以用 & 操作来实现）。如果这16位全部为1，则将其设置为0（通过与前面判断是否全为1得到的序列的取与即可）。然后与接下来的的16取或，得到一个新的数。处理这个新的数，用类似的方法处理最低的8位。依次类推，即可得到答案。

当需要处理的序列长度较短的时候，暴力算法的运算次数比分治更少，故可以采用暴力的算法来计算，以减少运算次数。

暴力计算可以将两个二进制位一起计算。经过枚举发现，将一个2位的二进制数加一（可以临时溢出）在除二下取整，可以得到2位二进制数里1的个数。这个性质能够减少左移次数，以减少运算次数。

代码如下（50步）：

```
int leftBitCount(int x) {
  int tmp, ans = 0;
  int s1, s2, s4, s8, s16;
  s1 = 0x1; s2 = 0x3; s4 = 0xf; s8 = 0xff; s16 = (s8 << 8) +
s8;
  x = x & (x >> 1);
  x = x & (x >> 2);
  x = x & (x >> 4);
  x = x & (x >> 8);
  x = x & (x >> 16);
```

```
  x = ~x;

  tmp = x ^ s16;
  tmp = (tmp << 16) >> 31;;
  ans = ans + ((~tmp) & 16);
  x = x & ((x >> 16) | tmp);

  tmp = x ^ s8;
  tmp = (tmp << 24) >> 31;;
  ans = ans + ((~tmp) & 8);
  x = x & ((x >> 8) | tmp);

  tmp = x ^ s4;
  tmp = (tmp << 28) >> 31;;
  ans = ans + ((~tmp) & 4);
  x = x & ((x >> 4) | tmp);

  ans = ans + (((x & 0x3) + 1) >> 1);
  ans = ans + (((x & 0xc) + 0x4) >> 3);

  return 33 + (~ans);
}
```