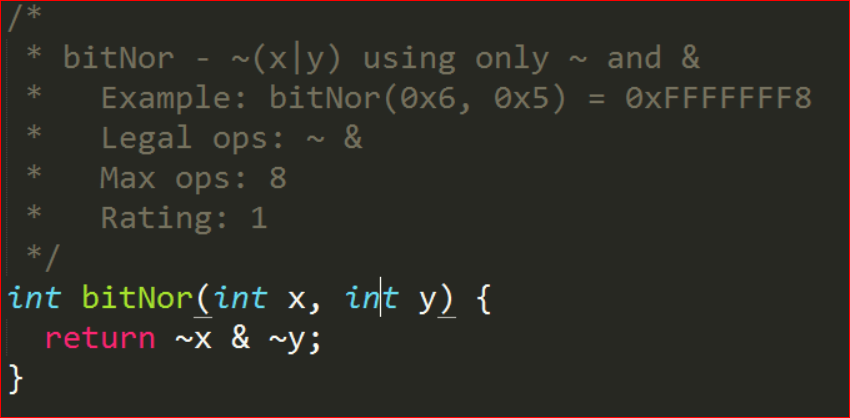
**实验报告**

**PB15111604 计算机学院 金泽文**

**一、DataLab**

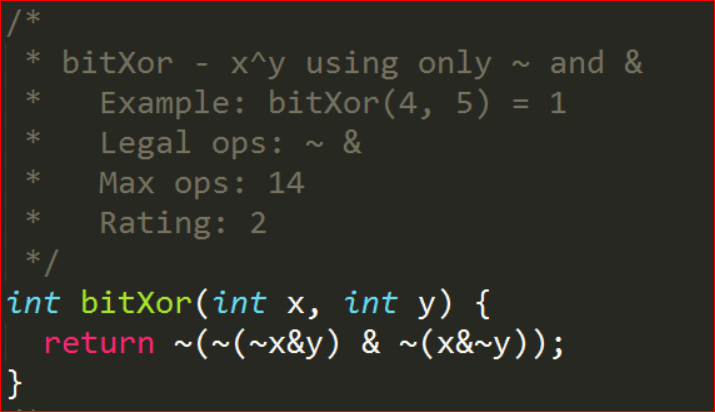
实验目的：这个实验要求实现简单的逻辑和算术运算函数，以了解C语言数据类型的位级表示，以及数据操作的位级行为。

实验过程：

1.bitNor   


算法类似于集合中的 C（X ∪ Y）=CX ∩ CY .

CX , CY代表X,Y的补集.

2.bitXor

算法：

∵x^y = (~x&y) | (x&~y)

//因为 | 不能使用，所以需要替换.

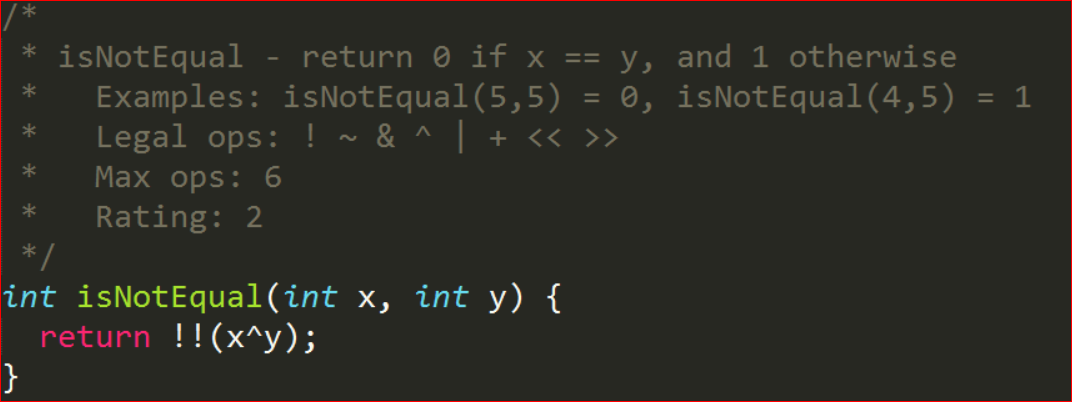
由1.bitNor，~(x|y) = ~x & ~y,

∴x | y = ~(~x & ~y),

∴x^y = (~x&y) | (x&~y)

=~(~(~x&y) & ~(x&~y)).

3.isNotEqual



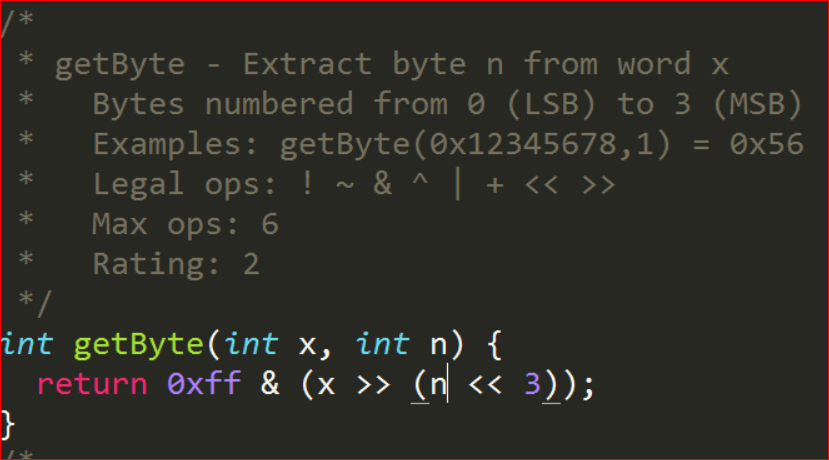
算法：

∵x=y时，x^y=0,

x!=时，x^y!=0

∴return！!(x^y)

4.getByte



算法：

首先，要得到所extract 的字节，

先将x右移n个字节,

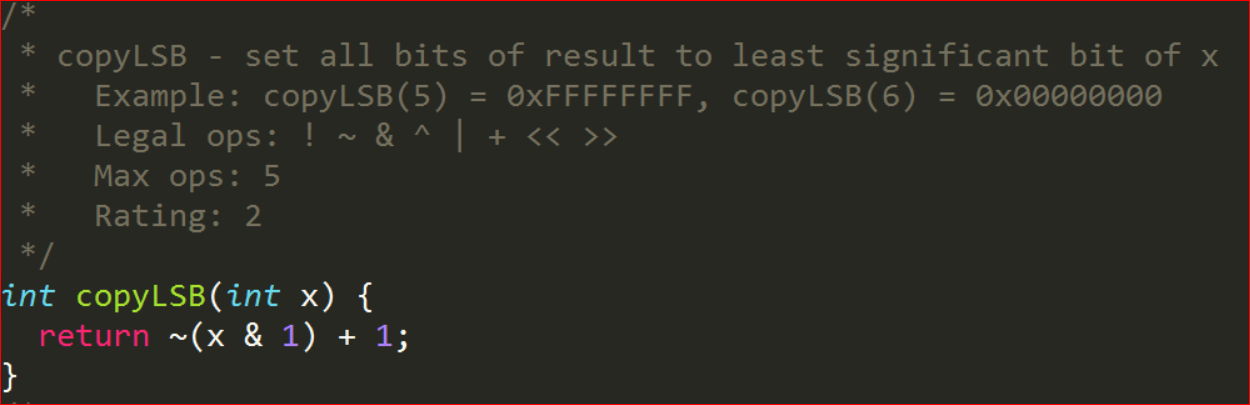
等价于右移n\*8,即n<<3.

之后，要extract的字节就在LSB上，

通过0xff掩码，

0xff & (x>>(n<<3))即可.

5.copyLSB



算法：

此题要求将最低位copy给所有位.

首先，要得到最低位，

通过掩码0x1，

x&1，得到最低位，

要想copy到所有位，

必须将所有位同时操作.

~(x & 1)得到的是

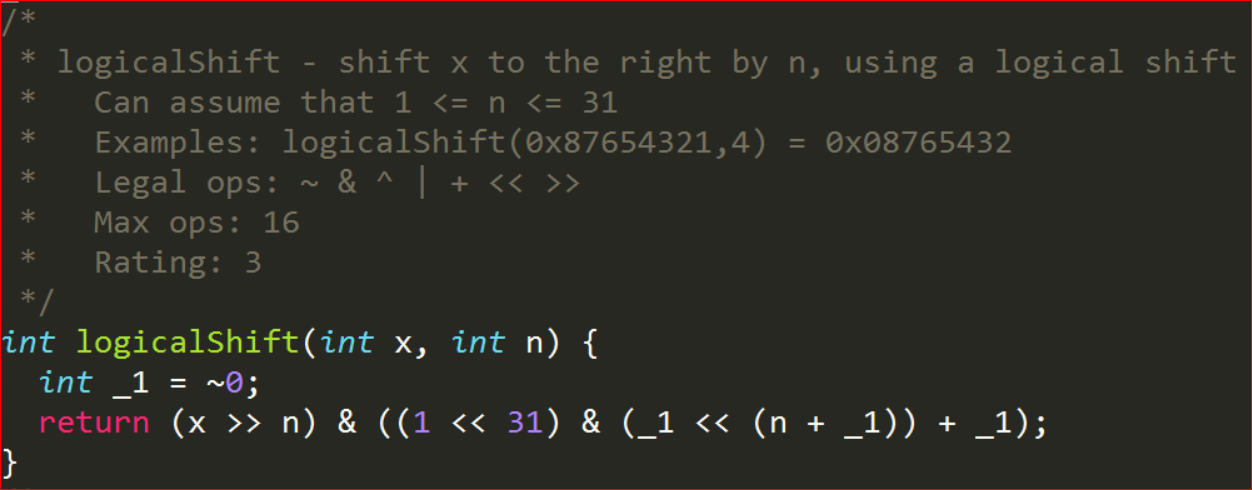
if(LSB==1) 0xfffffffe,

else 0xffffffff.

只需+1,即可得到0xffffffff,

else 0x00000000.

6.logicalShift



算法：

题意：逻辑右移n位.

我的做法是先算术右移n位，

再通过掩码去掉左边多出来的1.

期望的掩码为000...11(2),左边的0的个数为n,

右边的1为32-n个,

通过1<<32-n可以得到000..100(2),

1 << 32 –n 可以通过

（1 <<３１）　＆（－１＜＜（ｎ－１））－１

实现

1右边有32-n个0，再减去1即可，

但是“-”is forbidden,

所以用到一个技巧，

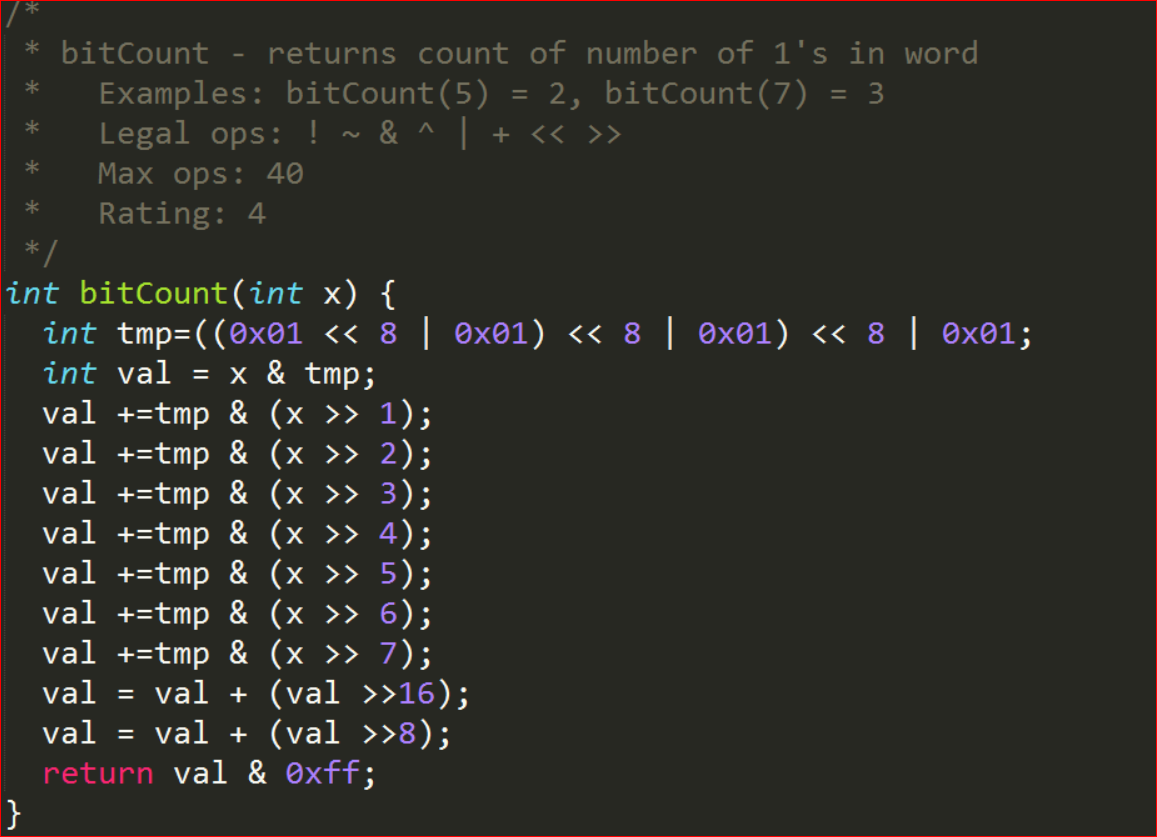
即 -1的位表示，等价于~0：

-1 的位表示为0xffffffff,

而0的位表示刚好为0x00000000.

由此，问题都得到了解决.

7.bitCount

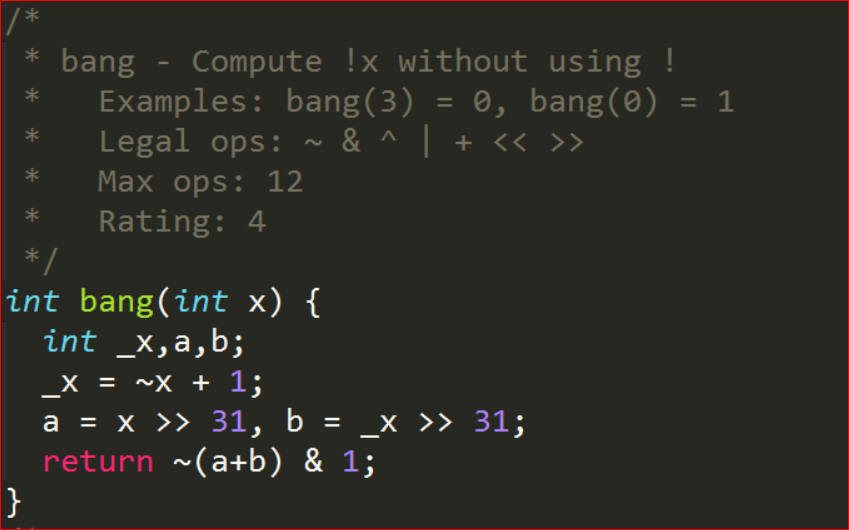


算法：这一题一共用了4-5个小时，毫无头绪，

得到这份答案之后也不甚理解，

应该是整个实验中灰色的一笔了...

8.bang



算法：

由于不能使用’!‘,最后输出时，

想到用0xffffffff + 1 溢出为0的技巧.

0和其他数的不同，可以从1入手，利用下面的leastBitPos，但是操作估计不少，

而不同点还有一个：

0和-0的位级表示都是0x00000000,

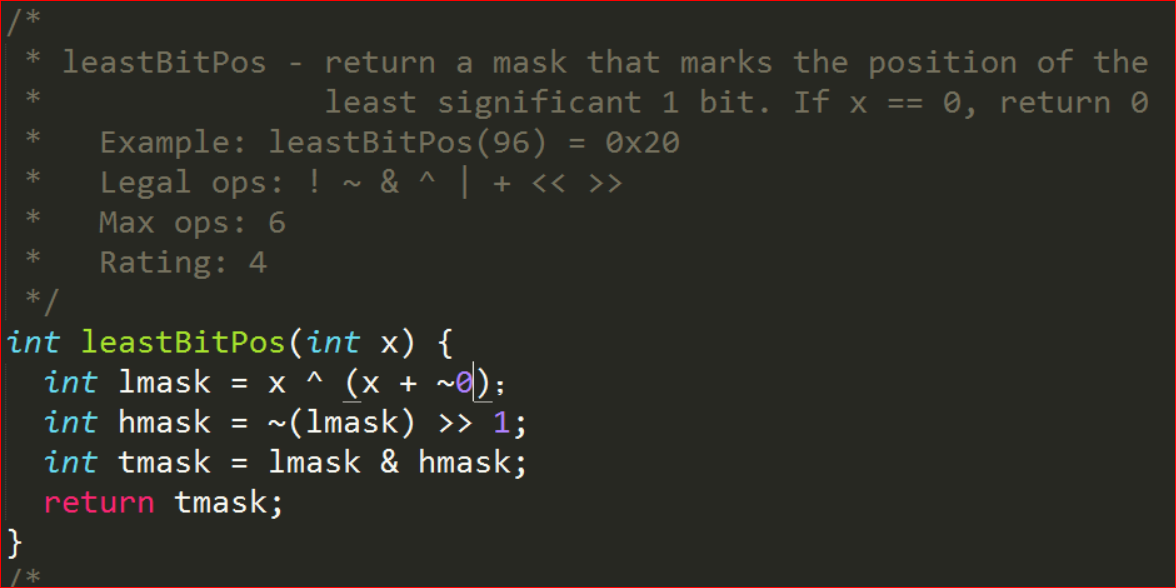
而其他数x和-x中MSB必有一个1，

所以可以通过判断 x>>31 + (-x)>>31是否是0xffffffff，

而 –x = ~x+1，

所以再结合0 输出1，！0输出0的要求，通过取反&1得出结果。

9.leastBitPos

算法（一）：ops：6，

先求掩码lmask = 000...1111,

使得lmask & x 即为 000...1...00,

而x 与 x-1之间不同的最高位，即为所求位.

所以得lmask = x^(x+~0),

要求lmask 最高位的1,

令hmask = ~ lmask >> 1,

再令 tmask = hmask & hmask,

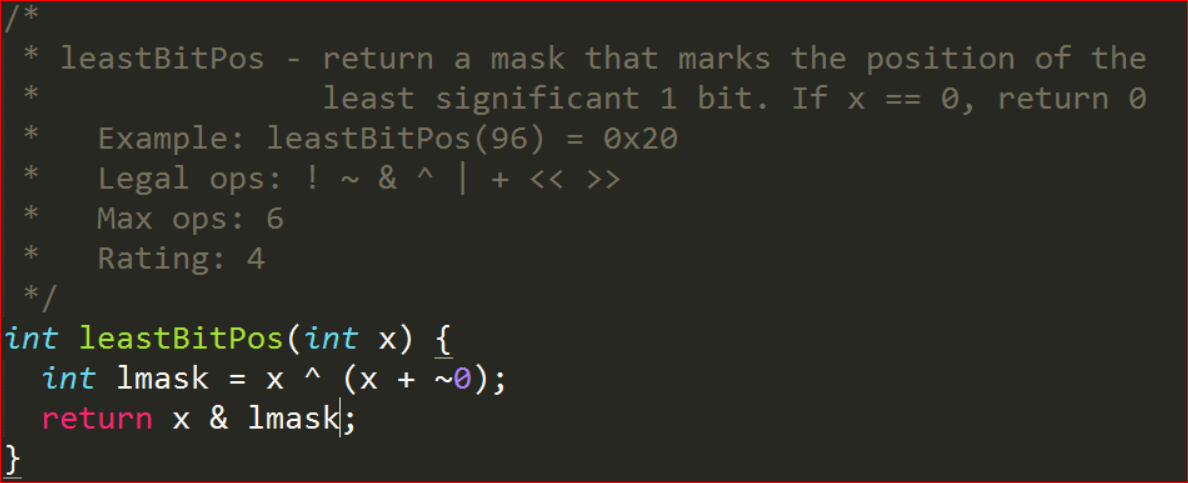
即为所求.

算法（二）：ops：4.

同上， lmask = x ^ (x + ~0).

而x与lmask仅所求位为1，故

x & lmask 即可.

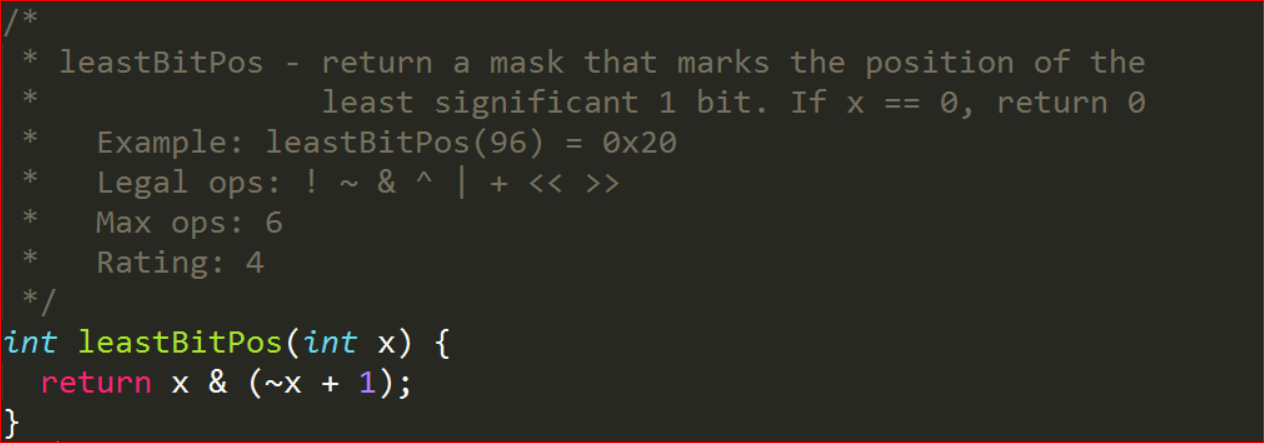


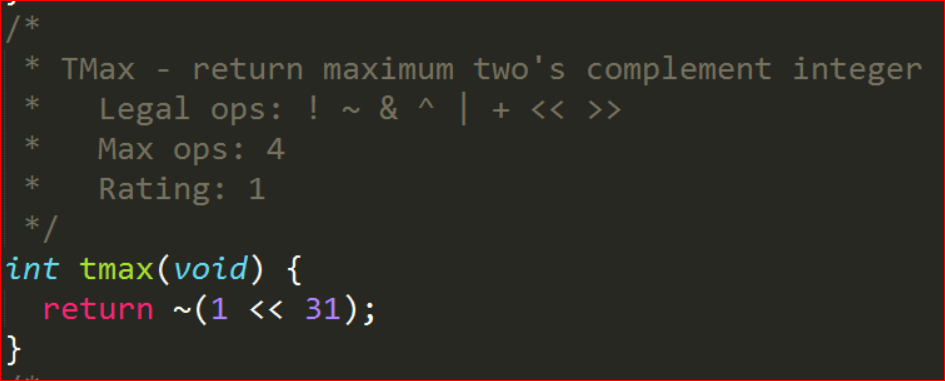
算法（三）：ops：3.

由于 ~x 在所求位后面都是1，

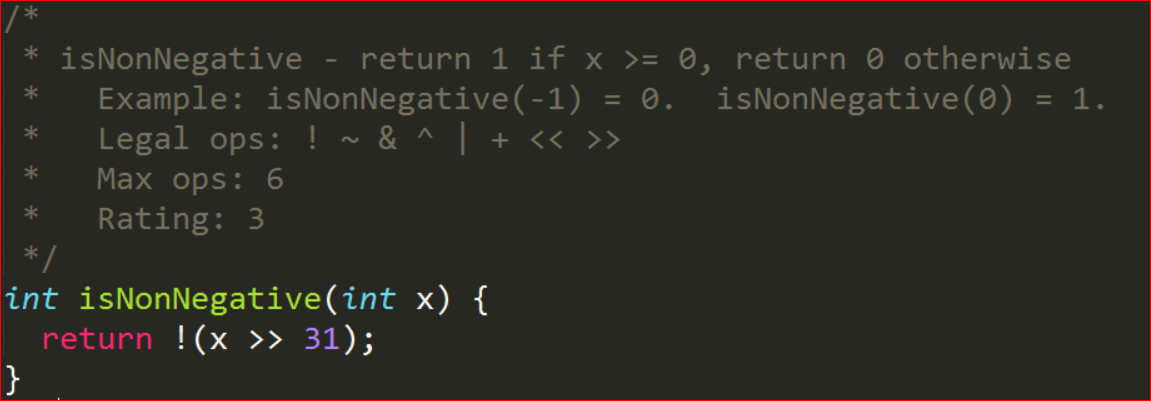
加1之后所求位位1，后面为0，

再取 &，即可.

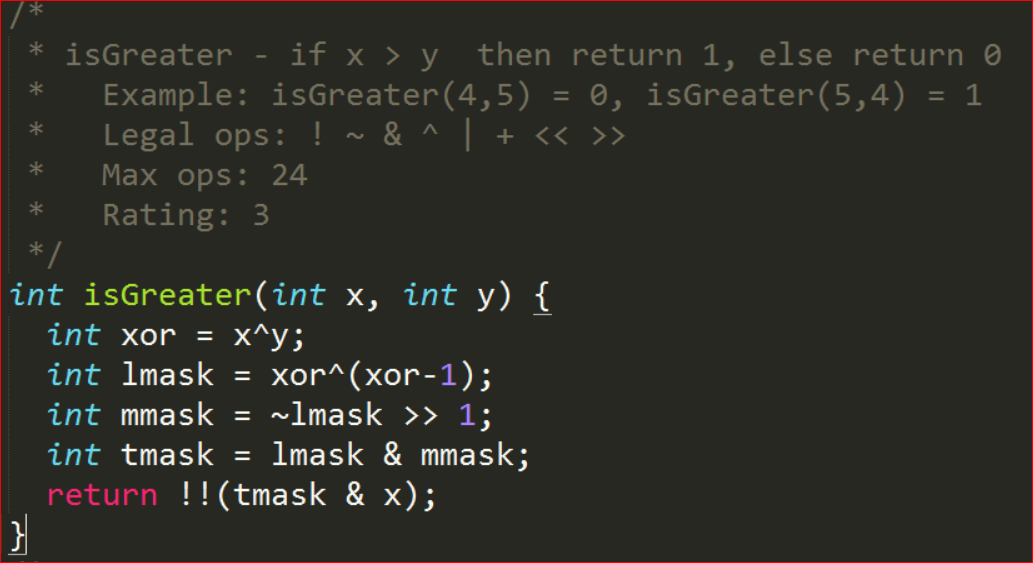


10.TMax 

算法：TMax 一定是01111..11(2),所以只需100..00取反.

11.isNonNegative 

算法：由于非负数和负数的最高位不同，所以此题很显然.

12.isGreater 

算法：

不论正负，x>y时，标记出x与y不同的最高位,

求x^y 的最高有效位1,

令xor = x^y,

利用第9题leastBitPos,

得出低位掩码lmask ：0000...111,

取反 右移1位，得高位掩码mmask: 1111...000.

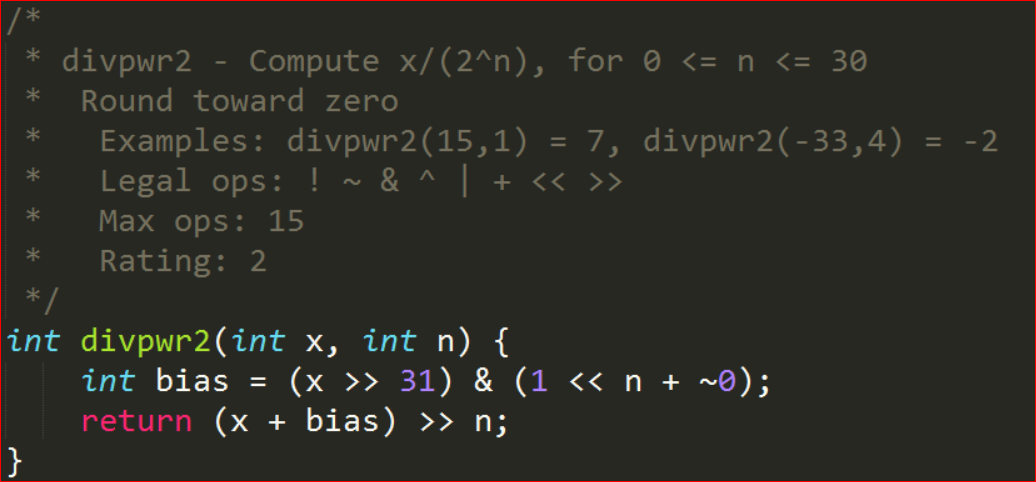
再通过 lmask & mmask, 得到tmask (true mask),

tmask = 0000..010..0...,

1所在的位置即为x与y不同的最高位,

x>y 等价于 x & tmask 不为0,

即可得出结论.

13.divpwr2 

算法：由书上练习题2.42,

div16时，

int bias = (x >> 31) & 0xff；

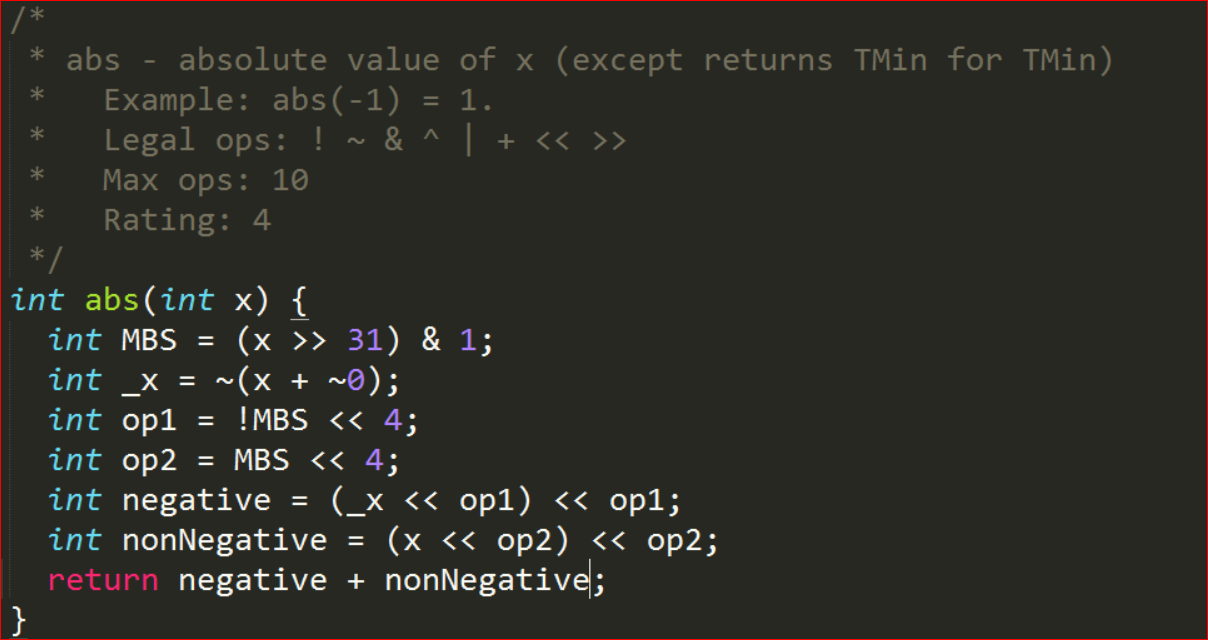
return (x + bias) >> 4;

上面的0xff = (1<<n - 1) = (1<<n + ~0).

即可得.

14.abs(我想,此题应该是这15道题中最有价值的一道题了)

算法：

(一) 在gcc上（ops limit exceeded）

第一次的做法ops : 13,超了3个，

因为在我的gcc上，移位数超过总位数会报错,

所以只能分两次进行.

求绝对值时,x的最高位不同,我用的公式不同,

所以先get MSB=(x >> 31) &1,

根据MSB可能取0,1选择不同的公式,

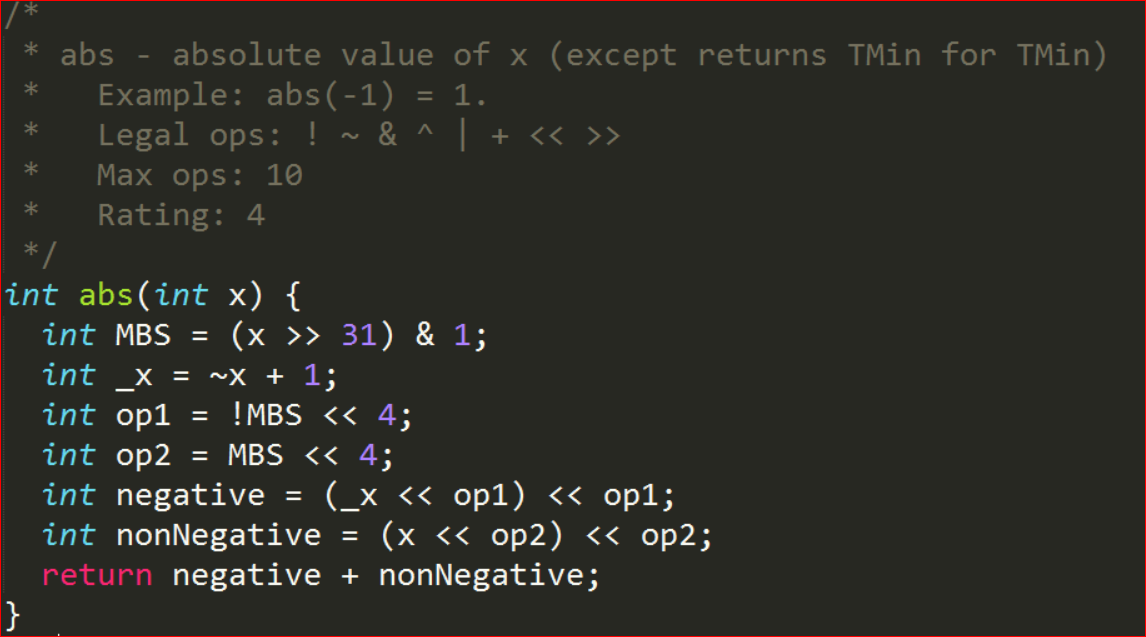
但是总体公式还要相同,

这里就用0,1的不同取值,

通过左移32位,或者0位,做出选择.

即为上图.

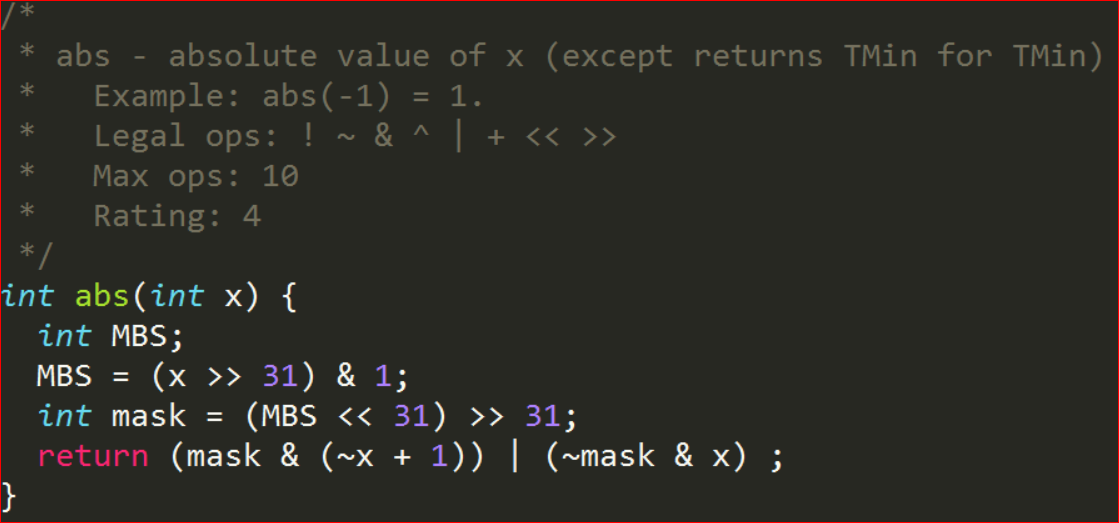
(二)



修改后：ops: 12.

将 ~(x + ~0) 改为 ~x + 1，减少一个op.

(三)



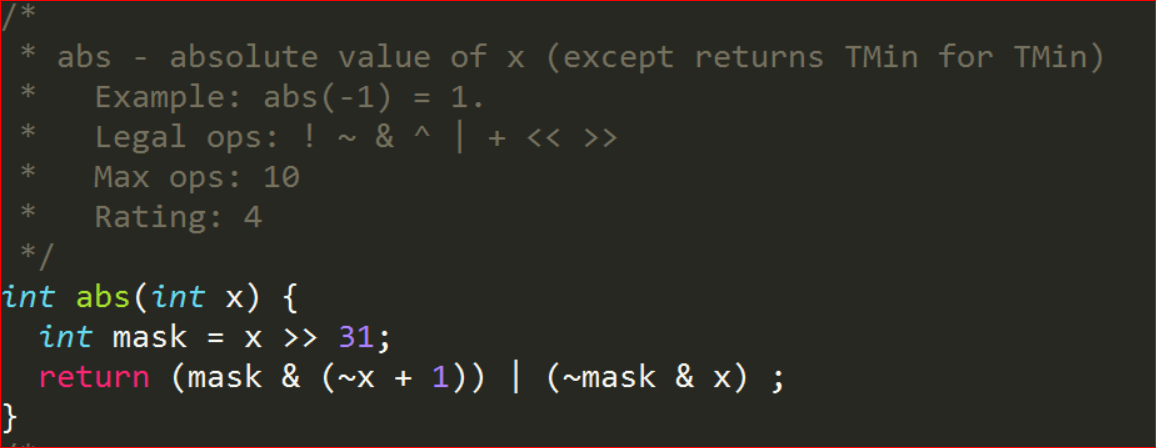
再次修改后：ops：10

鉴于以上做法在移位操作上浪费了大量ops，

改变做法，答题思路仍为根据最高位0,1做选择.

只不过将选择的方式变成了掩码.

(四)



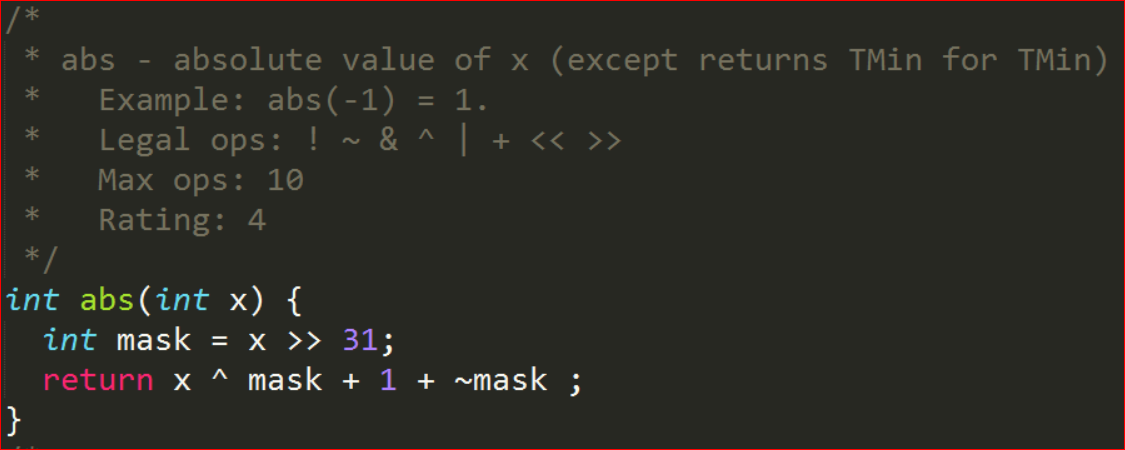
再再次修改之后，ops:7

突然发现，改成掩码之后，取MSB的做法重复,

mask = x >> 31即可.

但是......都在这道题上花了这么久，还有没有更好的做法呢?

(五)



有！（ops：5）

同样使用掩码，

但是并不将MSB为0,1的情况下划分的那么细,

MSB=1时，x^ mask = ~x,

则return ~x+1 即 x^ mask + 1,

MSB=0时，x^mask = x,

应该return x 即x^ mask,1要怎么处理呢？

注意到，~mask 在MSB=1时，为0,

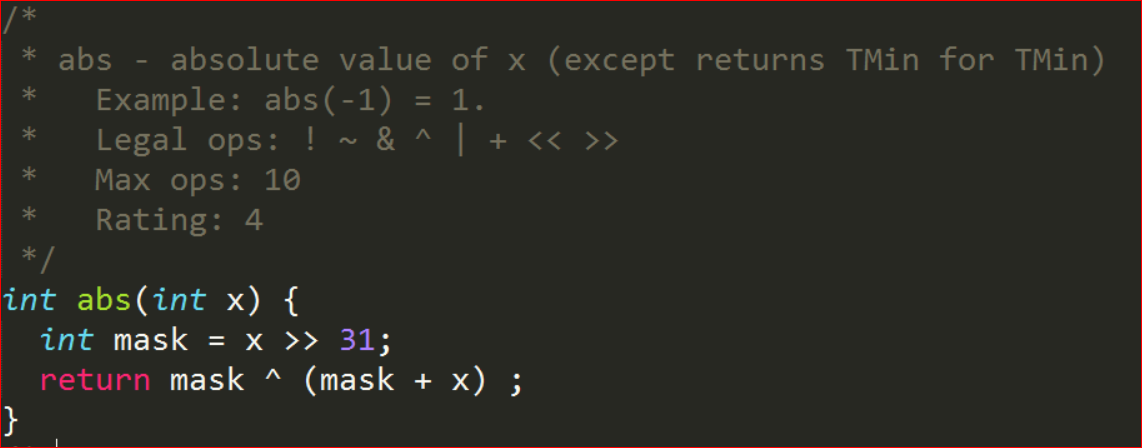
在MSB=0时，为0xffffffff,加1后为0.

所以，最终得到能想到的最优解：

x^mask + 1 + ~mask.

ops只有5。

...最后的最后，找了一下别人的算法...发现竟然有ops=3的算法...



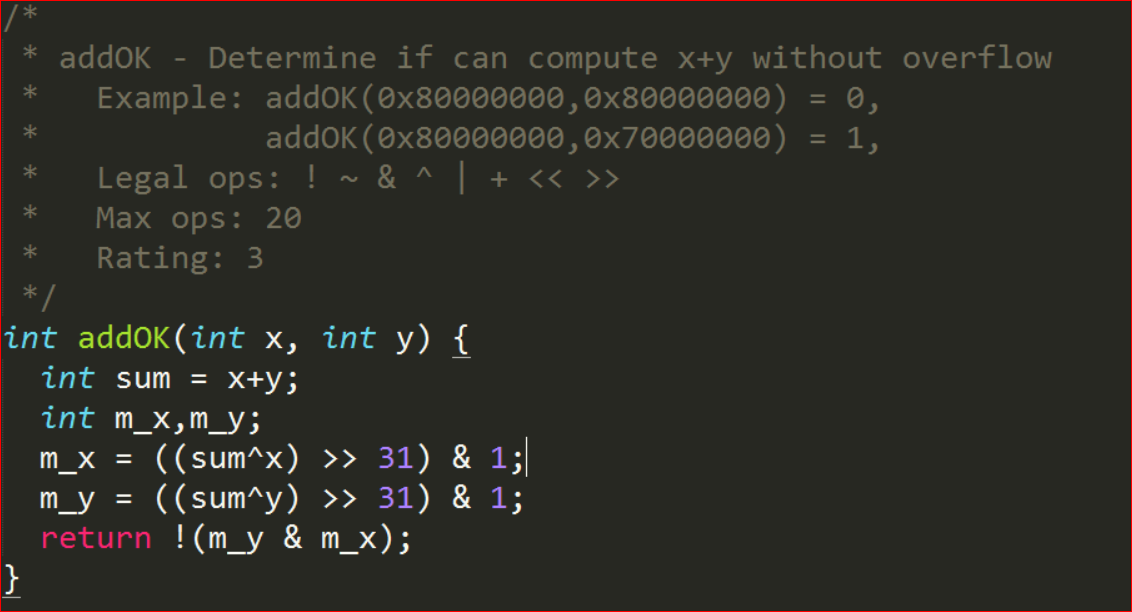
x≥0时，mask^，mask+无影响，输出x

x<0时，mask=0xffffffff,

0xffffffff + x恰好能得到 ~（~x + 1），

这样的话，有了这个技巧，似乎有很多可以改了。。

15.addOK



算法：

判断是否溢出，因为溢出时x，y必然同号，

而x+y与x，y异号，

所以x，y的最高位与x+y相反，

而不溢出时，x,y必有至少一个与x+y同号.

实现过程中，

用m\_x,m\_y来标记 x+y与x，y的最高位关系,

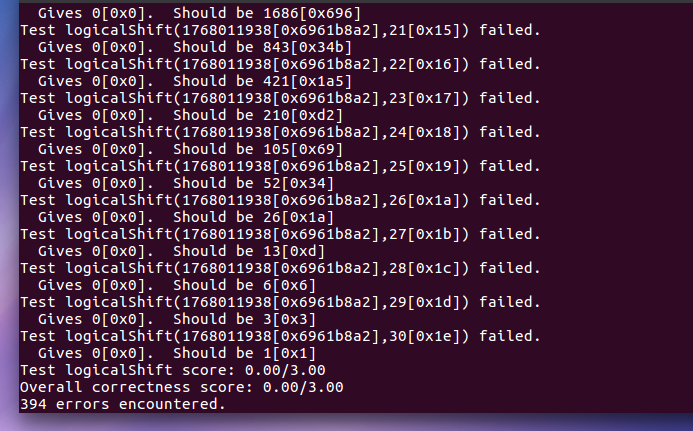
m\_x = ((sum^x)>>31) &1,

m\_y = ((sum^y)>>31) &1,

最后 ！（m\_x & m\_y）即为结果.

检查：

在ubuntu16.04上检查，发现



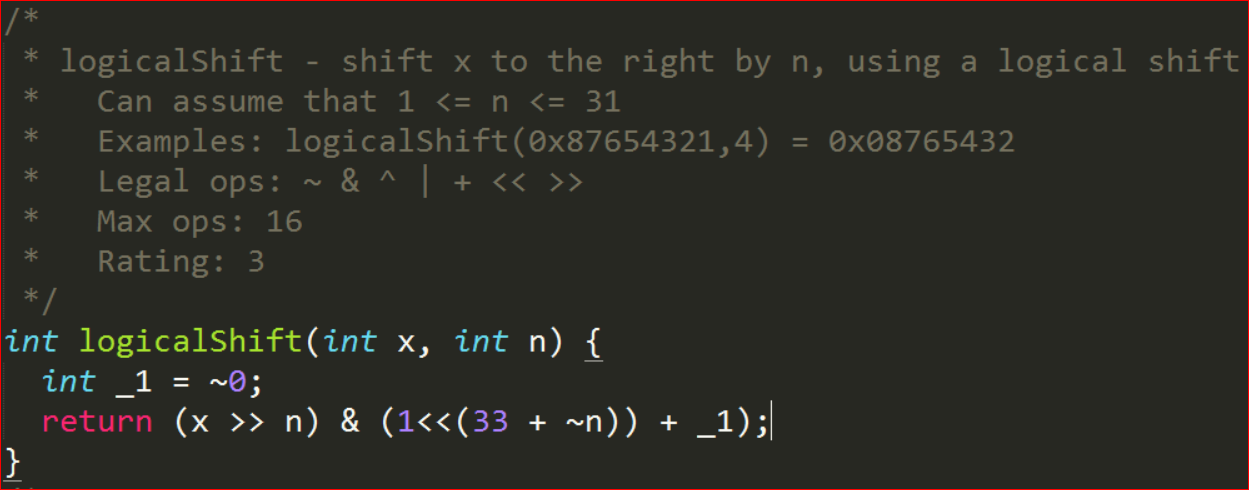
６.logicalShift出错

原来，1 << 32-n不等于

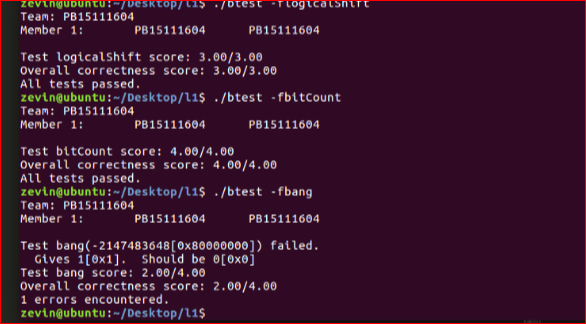
（1 <<３１）　＆（－１＜＜（ｎ－１））－１,

而应该是1 <<(32 +~n +1)

再合并一下，得1<<(33 + ~n).



继续检查

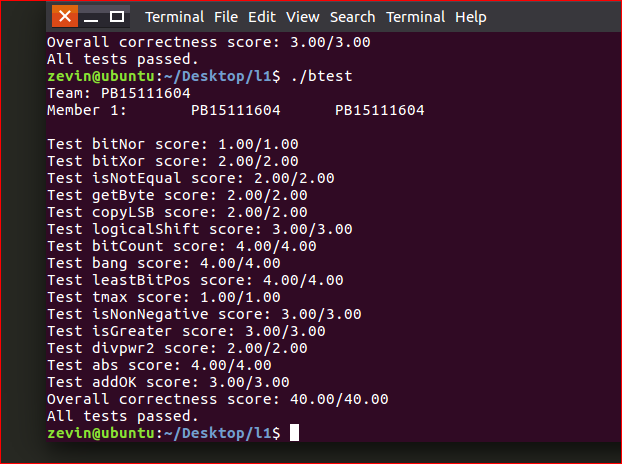


8.bang 出了个错。。

忽略了0x80000000的情况，刚好错一个。。

发现，将 ‘+’改成‘|’，

刚好可以忽略0x80000000的情况！

最后，全部通过！

总结：这个实验中，配置Linux花费了大量的时间和精力，尤其是刚开始装的是64位ubuntu.15个puzzle中，abs给我的印象最深，收获也最大。

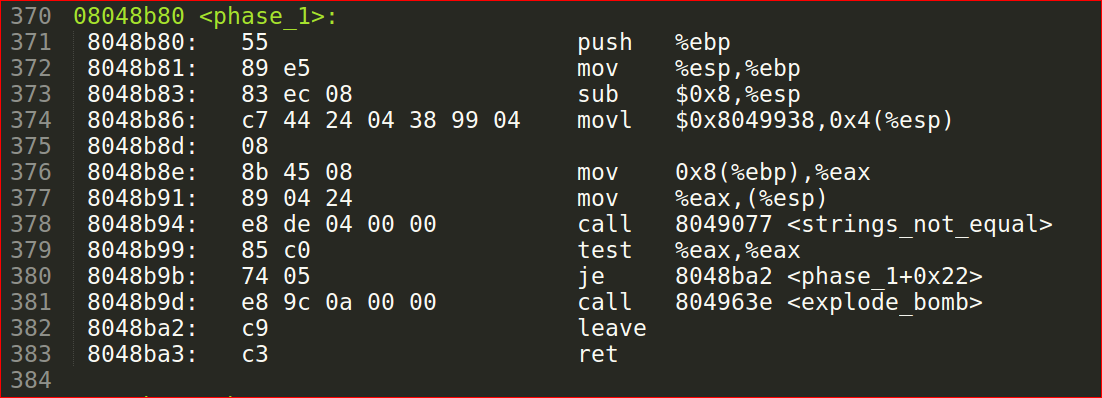
**二、BombLab**

实验目的：为了对程序反汇编和逆向工程，学会阅读并理解汇编语言，学会使用GDB。

实验环境：ubuntu16.04-32位

所选bomb：bomb3

Phase 1:

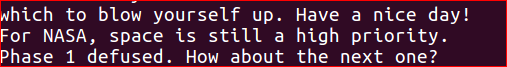


374：4(%esp) <- 0x8049938,

376,374:(%esp) <- 0x8(%ebp),

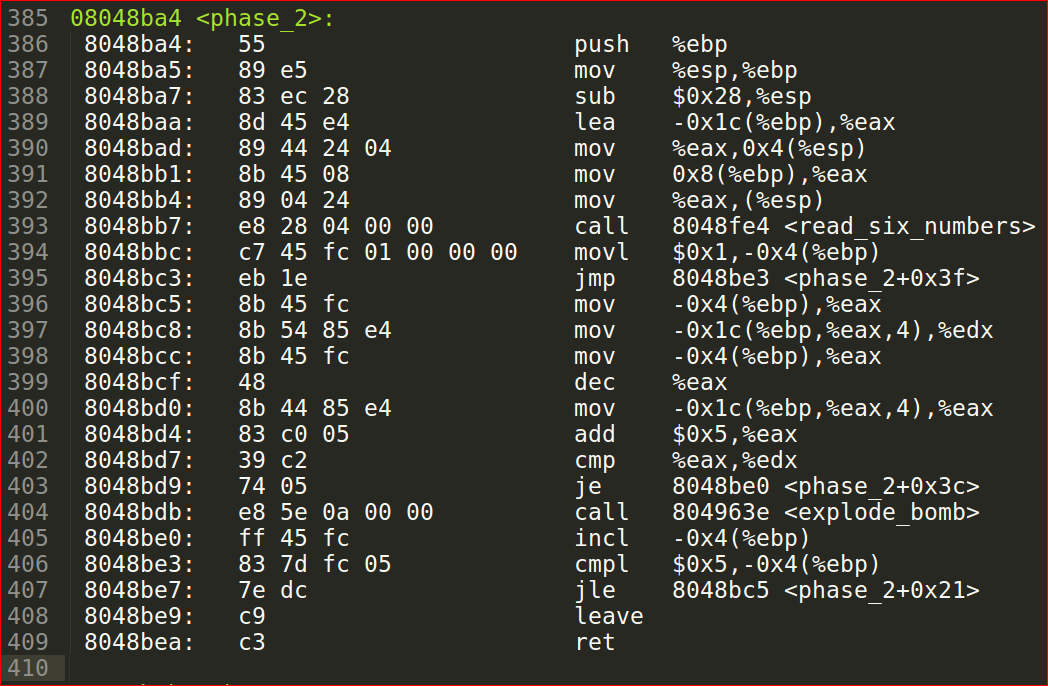
378,379比较，若不同则bomb！

比较的是4(%esp)和(%esp)，即0x8049938和0x8(%ebp)两个字符串指针指向的内容。

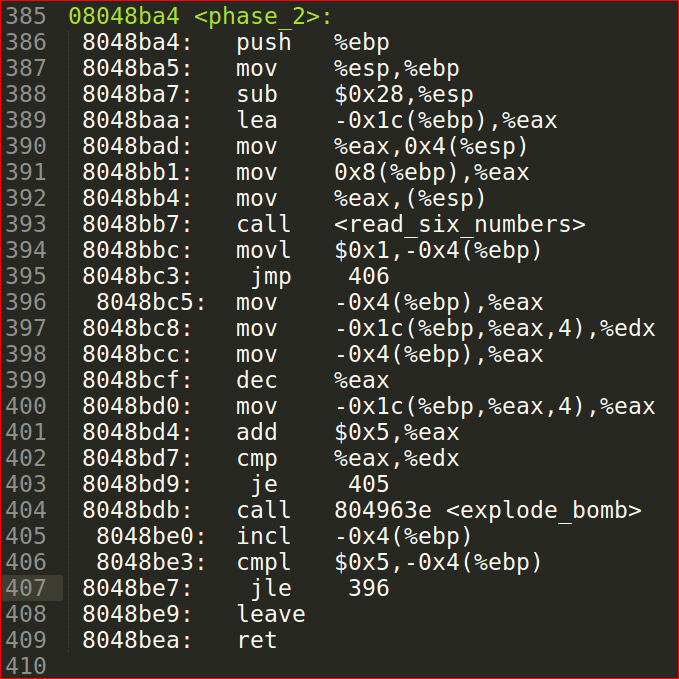
用gdb通过x/s 0x8049938 读取其字符串，得到 

这一串字符，即为所求。phase 1 完成。

phase 2：



经修改



因为需要6个数字，所以估计至少要判断6次，

显然有循环结构

顺序应是

407 -> 396

403 -> 405

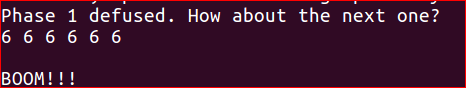
407 -> 396

403 -> 405

407 -> 396

403 -> 405

但是如果第一次跳到406时jg，即M[R(ebp)-4] > 5,

则直接leave，但我的input会不会存道M[R(ebp)-4]上呢？试了一下6 6 6 6 6 6 这组数

不对，注意到，前面M[R(ebp)-4]只有在405处加一，在394初始化为1，所以估计是要在402判断6个数。

只需观察eax，edx

第一次到396， R(eax) = 1,

397 R(edx) = M[R(ebp)-12+4]=M[R(ebp)-8],

398 R(eax) = 1

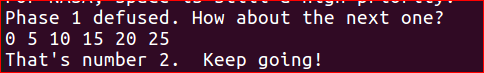
399 R(eax) = 0

400 R(eax) = M[R(ebp)-12+R(eax)\*4] = M[R(ebp)-12]

401 R(eax) += 5

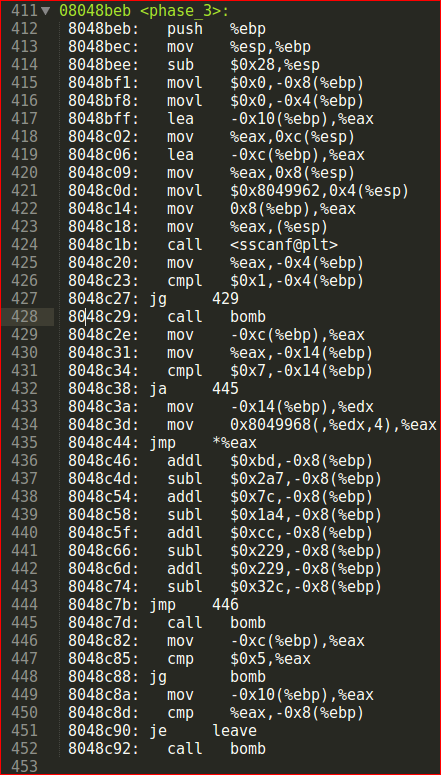
402 cmp eax edx

所以我猜测，这六个数是递增5的数列。



果然！phase 2 完成！

phase 3: 代码修改得：



看到 sscanf， 必定要用参数，而通过x/s0x8049962,

得到参数格式

本题是顺序流程，其中有switch结构

417,418: 将第二个参数arg2存入M[R(ebp)-0x10].

419,420: 将第一个参数arg1存入M[R(ebp)-0xc].

421：“%d %d”

425,425:输入数据个数必须大于1，

429,430:将第一个参数arg1存入M[R(ebp)-0x14]

431:arg1大于7则bomb，必须小于等于7，

433,434:R(eax) = M[arg1 \* 4 + 0x8049968].

switch到436-443某一行，M[R(ebp)-8]得到一个值。

446,447,448:arg1必须小于5,

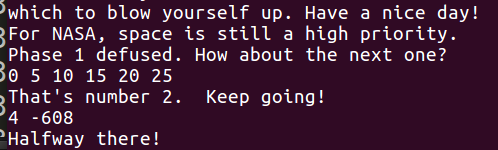
450: arg2 必须等于M[R(ebp)-8]，

可以取arg1 = 4，这样可以少算一点。。

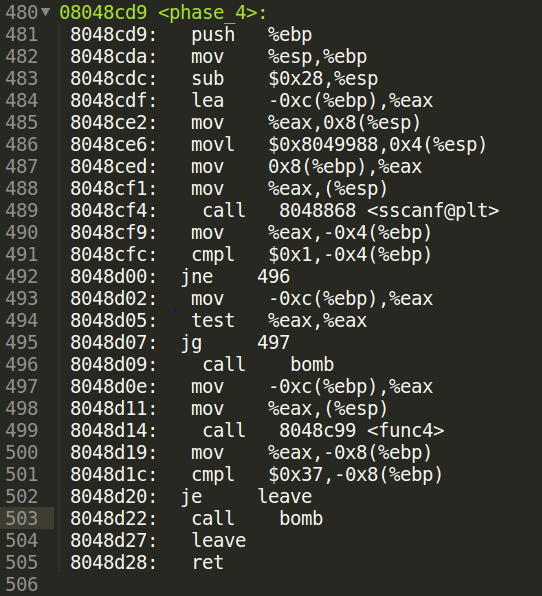
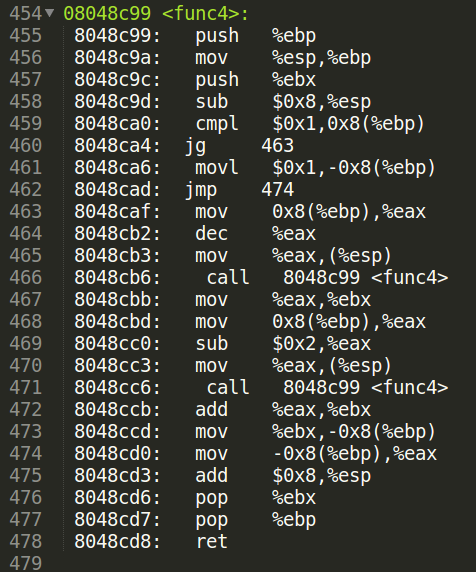
转换成十六进制，得到8048c5f,M[R(ebp)-8]初始值为0，

计算得，0+0xcc+0x229-0x229-0x32c = 204-812=-608

输入4， -608 得到

 phase 3完成！

phase 4：



先看phase\_4:

输入一个参数arg，保存在M[R(ebp)-0xc].

495: arg必须大于0

499: call func4

501: func4返回值必须等于0x37 =55.

现在看func4

460：如果M[R(ebp)+0x8] 小于等于1，则返回1

否则，jmp 463，再次调用func4（M[R(ebp)+0x8]-）

467：这次func返回值存在R(ebx)

468-471：调用func（M[R(ebp)+0x8] -2）

472：R(ebx)+=返回值

473：M[R(ebp)+0x8] = R(ebx)

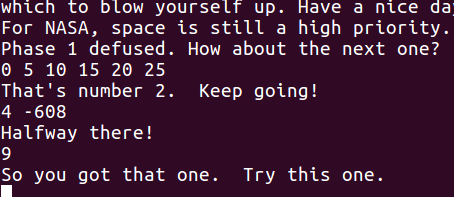
（也就是从“否则”开始，返回

func4(M[R(ebp)+0x8] - 1)+func(M[R(ebp)+0x8]-2)

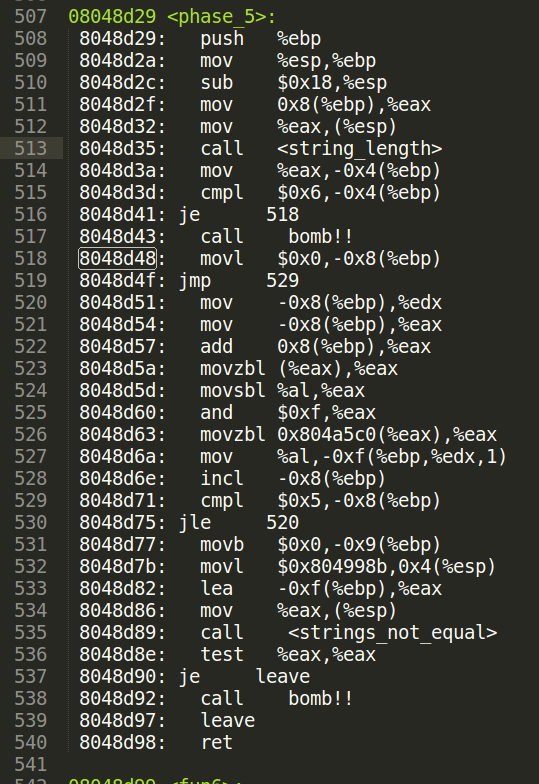
所以func4（n）=func4（n-1）+func4(n-2)

所以是斐波那契数列！

结果是55，得n=9！！！

phase 4完成！

phase 5：（修改后）



本题输入的是字符串，

511: M[R(ebp)+8]存储字符串地址，记为pointer

516: strlen(arg)必须等于6

518: M[R(ebp)-8] 像是一个counter，记为count吧

529: count = 0 <= 5 -> 520

520: R(edx) = count

522: R(eax) = count + pointer

523,524,525: R(eax) = pointer[counter]的低四位

526: R(eax) = M[R(eax) + 0x804a5c0],

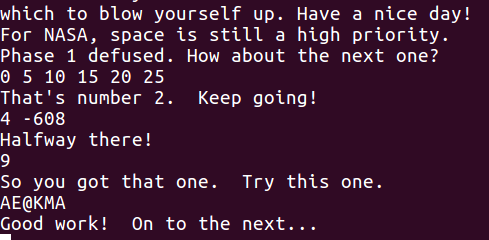
527: M[R(ebp)-0xf+R(edx)] = al;



只要第四位分别是1,5,0,11,13,1即可

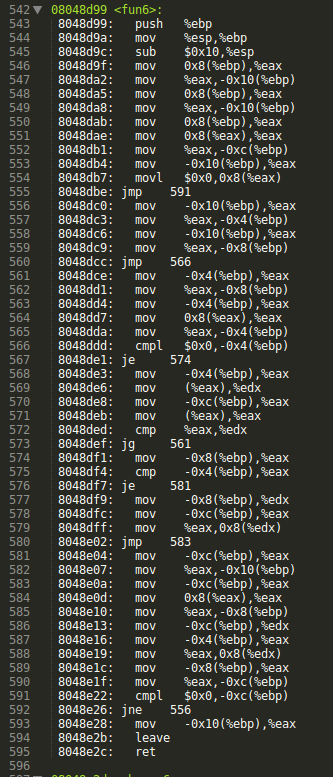
查ascii表，’@’为0100 0000,

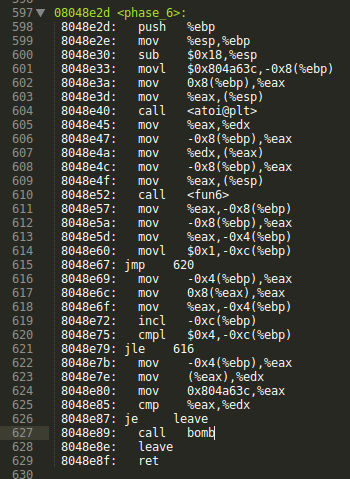
所以取 AE@KMA



phase 5 完成！！！

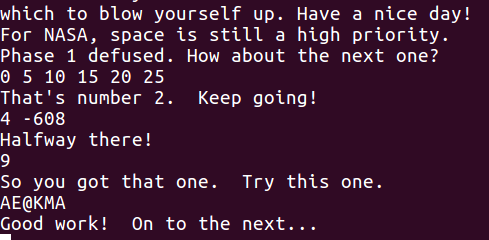
phase 6：





最终。。。此题实在是蒙圈了。。。查了一下，可能是链表。

由于时间精力没有分配好，此题只能选择放弃，还有后面的secret\_phase也只好放下。

实验总结：第二个实验投入的精力与时间远超第一个实验。由于期末精力分配不合理，只能作罢。

暂时完成了5个phase。

收获还是很大的，学会了基本的阅读汇编代码的能力。有一个感触就是这几个实验的创新设计。有趣的同时又能强迫学生学会各种东西，比如gdb调试。

还有一个很大的感触就是

上课听懂≠理解，

“认真”地看一遍书≠掌握。

只有一个字一个字的看，一道题一道题的做，才会发现问题，也才会真正的理解并掌握。我在做实验的过程中就发现了很多很多的理解漏洞，通过练习题与反复阅读，弥补了一些。我发现，每读一次，理解真的会加深一层。

最后，“计算机系统详解”真的是一门让人收获巨大的课。也许有一天，科大也会像CMU一样将其设为必修课。

最后，感谢吴俊敏老师的引路！