DataLab

```
int bitXor(int x, int y) { // 1
  return ~(x & y) & ~(~x & ~y);
}
int tmin(void) {    // 2
  return 0x01 << 31;
}
int negate(int x) {  //5
  return (~x+1);
}
```

这三个题都比较简单，是基本概念的考察，没什么可以谈论的。要注意的话，可能就是受限的操作符，要让你进行一些变换。

题目三 isTmax(int x)

如何判断一个数是不是Tmax呢?

> 如果一个数是Tmax的话，那么Tmax + 1 = ~(Tmax)。
> 但是如果这个数是-1的话 那么-1 + 1 与~（-1）也为True。我们需要排除这样的情况。

```
int isTmax(int x){
    int tmin = x + 1;
    int eq = tmin ^ (~x);
    return (!eq) && (!!tmin);
}
```

> ^可以比较两个数是否相同。

题目四 allOddBits(int x)

本题的题意是如果x的所有奇数位都为1，那么就返回1，反之为0。

所以思路就是构建一个t ， t的所有奇数位都是1，然后(x&t) == t 即可。

```
int allOddBits(int x){
    int val = 0xAA; // 8bit 1010 1010
    int aval = 0xAA << 8 | 0xAA; // 16 bit
    int bval = aval << 16 | aval; // 32 bit
    int res = (x & bval) ^ bval;
    return !res;
}
```

> 通过&可以屏蔽一些位的影响

题目六 isAsciiDigit(int x)

如果 x在 0x30 与 0x39之间，则返回1，反之返回0。 **也就是说，这个题要让我们使用一些运算符，来实现><的运算。**

可以想到 `x < y` 等价于 `x - y < 0` 用伪代码描述如下：

```
def isAsciiDigit(num: int) -> bool:
    return x <= 0x39 and x >= 0x30
# 相当于
def isAsciiDigit(num: int) -> bool:
    return x - 0x39 <= 0 and x - 0x30 >= 0
```

根据这个可以编写程序： `~0x30 + 1` 是因为本题不可以使用减法操作符。

```
int isAsciiDigit(int x){
    int sign1 = x + (~0x3a + 1);
    int sign2 = x + (~0x30 + 1);
    return (sign1 >> 31) && !(sign2 >> 31);
}
```

> 取0x39 会导致 x = 0x39时过不了测试，根据提示改成0x3a就可以了，细想一下确实是这样的。
> 或者也可以将上面的第二行改写成 int sign1 = 0x39 + (~x+1)

题目7 conditional(int x,int y,int z)

`same as x ? y : z` 这个题要让我们通过一些受限的操作符，来实现分支操作。

用伪代码描述如下

```
if x:
    return y
else:
    return z
```

进而我们可以编写代码

```
int conditional(int x,int y,int z){
    int flag = !!x;  // get condeitionl expression
    int bry = flag & y;
    int brz = (!flag) & z;
    return bry | brz;
}
```

运行发现，有问题。

问题出在&上，我们想当然的认为&与&&的行为一致了。

根据课本的说法，如果我们能获得一个全1的二进制表示A，那么A&&val与A&val的结果是一致的。全一的二进制用补码来说就是-1。

flag的结果是0或者1，根据我们课本上对于加法逆元概念的阐述，获取-1的方式为， `-1 = (~x +1)` ,而0的逆元为它本身，也就是 `0 = ~0 + 1` 。进而修改代码。

```
int conditional(int x,int y,int z){
    int flag = !!x;  // get condeitionl expression
    int cond = (~flag + 1);
    int bry = cond & y;
    int brz = (~cond) & z; //此处也进行了修改，原因同上,是为了取得一致的行为。
    return bry | brz;
}
```

我们成功的通过了测试。

题目八 isLessOrEqual(int x,int y)

题目如下：if x <= y then return 1, else return 0

根据我们题目六的思路，这个题可以改写成x - y <= 0，然后判断符号位进行解决。并且我们要吸取教训，由于存在0，且0的符号位与正数的符号位相同，所以写成y - x >= 0比上面的写法好实现一点。

我们尝试编写代码。(本题依然不允许使用减法操作符，所以我们依然要进行一个转换)

```
int isLessOrEqual(int x, int y) {
    int sign = (y + (~x+ 1)) >> 31;
    return !sign;
}
```

好像这样就可以了，试着运行下。
成功的出错了！这个题相比题6，由于xy是补码，所以要判断可能发生的溢出，所以改写下代码。

```
int isLessOrEqual(int x, int y) {
    int sign = (y + (~x+ 1)) >> 31;
    int sx = x >> 31;
    int sy = y >> 31;

    int case1 = (!sx) & sy;   // x > 0 , y < 0 当这种情况时，需要返回False
    int case2 = (sx & (!sy)); // x < 0, y > 0 显然是返回True

    return   case2 | ((!case1) & !sign);
}
```

这样就解决了这个题目了。

题目九 logicalNeg(int x)

这个题目是使用~ & ^ | + << >>，去构造一个！操作。根据我们题目7的思路，我们知道逻辑操作在操作数为全0或者全1的时候，与按位操作有相似的行为。

所以我们可以开始写这个题了。

```
int logicalNeg(int x){
    int nx = ~x + 1; //首先尝试获得全0或者全1，如果x是0，那么nx = 0，如果x不是0，那么 nx与x 的符号相
    int case1 = (x | nx) >> 31;
    // 此处的 >> 为算术右移，所以我们获得是111..111全1 或者0。通过对结果加一，我们就可以可得到Neg的行为
    return (case1+1);
}
```

这个题，做的时候不会做，我偷看题解了。 一开始没想到这种算术右移。

题目十 howManyBits(int x)

说实话，这个题没做出来，寄了，我也是偷看了答案才写的。

思路

howManyBits，我们首先考虑正数，因为负数相对麻烦点。对于正数而言，howManyBits换句话说就是，对于从左到右的一个数字序列来说，我们需要找最左侧的一个非0数字
也就是说，要找到00..001....中的那个1

find it

如何找到这个1呢？我偷看了答案，答案采用了一种很巧妙的方式。
现在我们要回顾下find这个操作了，我们学过很多搜索算法，如果我们对 列表中元素值之间的关系以及元素的存储顺序一无所知。在最差情形中，我们必须遍历L中的每 一个元素才能确定L是否包含e。也就是说 find操作的复杂度为O(len(n)),与列表的长度相关。
回到这个问题，我们真的对这个01序列的关系与顺序一无所知吗?
很显然不是这样的，所以我们可以优化find it，做一个二分查找。
这样的我们就解决了 find it，的过程，最后将这个第一个非0数字的长度+一个符号位的长度，就是howManyBits的结果了。

负数的情况

负数的情况可以将负数做一次取反操作变成正数处理。尽管数值不对应，但是bits数是相同的。这地方不再赘述了，看代码就可以了。

代码

```
int howManyBits(int x) {

  int cnt_0,cnt_1,cnt_2,cnt_4,cnt_8,cnt_16;
  int sign = x >> 31;
  int flag;

  x = (sign & (~x)) | (~sign & x);  // -x => +x

/*  binary search */
  flag = !!(x >> 16);
  cnt_16 = flag << 4;
  x = x >> cnt_16; //move 16 bit,if flag equals 1, else x not changed

  flag = !!(x >> 8);
  cnt_8 = flag << 3; //move 8 bit,if flag equals 1,else x not changed
  x = x >> cnt_8;

  flag = !!(x >> 4);
  cnt_4 = flag << 2;//move 4 bit,if flag equals 1,else x not changed
  x = x >> cnt_4;

  flag = !!(x >> 2);
  cnt_2 = flag << 1;//move 2 bit,if flag equals 1,else x not changed
  x = x >> cnt_2;

  flag = !!(x >> 1);
  cnt_1 = flag << 0;//move 1 bit,if flag equals 1,else x not changed
  x = x >> cnt_1;

  cnt_0 = x;

  return cnt_16 + cnt_8 + cnt_4 + cnt_2 + cnt_1 + cnt_0 + 1;
}
```

题目11 unsigned floatScale2(unsigned *uf*)

通过上面的操作，我们使用几个受限的操作符，实现了减法，条件，比较，查找等操作。
现在所有的操作都解禁了，我们可以很轻松的写代码了。关于浮点数的问题没有什么很难做的，基本是

考察浮点数的理解。
这个题是返回浮点数*2的结果。这个题没有什么好说的，理解了浮点数的表示即可做出来。

```
unsigned floatScale2(unsigned uf) {
  unsigned fsign = uf >> 31;
  unsigned fexponent = (uf >> 23) & (0xff);
  unsigned ffraction = uf & (0x7fffff);

  if (fexponent >= 1 && fexponent <= 254)
    fexponent = (fexponent + 1);
  else if (fexponent == 0)
      if (ffraction == 0)
          return uf;
      else
          ffraction <<= 1;
  else if (fexponent == 255)
        return uf;
  uf = (fsign << 31) | (fexponent << 23) | ffraction;
  return uf;
}
```

题目12 int floatFloat2Int(unsigned uf)

```
int floatFloat2Int(unsigned uf) {
    unsigned sign = uf >> 31;
    unsigned Exp = (uf >> 23) & (0xff);
    unsigned fraction = uf & (0x7fffff);

    int E = Exp - 127;
    int result;

    fraction |= 0x00800000; //使M = 1 + f，即将第24位置1，这地方需要一定的理解。
    if (E < 0)
        return 0;
    else if (E > 31)
        return 0x80000000u;
    else if (0<= E && E <= 23)
        result =  (fraction >> (23 - E)); //此处发生了舍入操作。
    else if (E > 23 && E <= 31)
        result =  (fraction << (E - 23));
    if (sign == 1){
        result = ~(result) + 1;
    }
    return result;
}
```

题目13 floatPower2(int x)

```
unsigned floatPower2(int x) {
  unsigned E = x + 127;
  unsigned fraction = 0;
  unsigned uf = 0;
  if (x > 127){
    uf = 0xff << 23;
  }else if (x < -149){
    uf = 0;
  }else if (x <= 127 && x >= -126){
    uf = E << 23;
  }
  else if (x < -126){
    fraction = 0x1 << (x + 149);
    uf = fraction;
  }
  return uf;
}
```