datalab实验报告

李俊霖 2021201709

问题1 bitXor

题目描述

用位运算表示异或 ^ .

思路&优化过程

采用离散数学的主析取范式和主合取范式的思想。
刚开始使用主析取范式展开，得到解决方式1，需花费8个符号:

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

采用主合取范式展开，优化到7个符号:

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

问题2 thirdBits

题目描述

构造类似于 ……`001001001001` 的二进制串

思路&优化过程

初始思路

构造出0x09，即 `00001001` ，然后按照位数依次左移6、12、24个位得到目标:

```
int thirdBits(void)
{
  int x = 0x09;
  x = x + (x << 6);
  x = x + (x << 12);
  x = x + (x << 24);
  return x;
}
```

优化思路

先构造出0x49，即 `01001001` ,然后向左移动两次9个和18个位即可拼接成所需二进制串:

```
int thirdBits(void)
{
    int x = 0x49;
    x = x + (x << 9);
    x = x + (x << 18);
    return x;
}
```

问题3 fitsShort

题目描述

判断一个 `int` 数是否可以表示成16位的有符号整数。（是否超出16位有符号整数的范围）

思路

- 若这个数为正数（右移补0）， `x>>15` 取出这个数16-32位， `x>>31` 可以取出这个数的符号位0（ `000...000` ），如果没超过16位的范围，符号位和16-32应该同为0。
- 若这个数为负数（右移补1）， `x>>15` 取出这个数16-32位， `x>>31` 可以取出这个数的符号位1（ `111...111` ），如果没超过16位的范围，符号位和16-32应该同为1。

```
int fitsShort(int x)
{
    return !((x >> 15) ^ (x >> 31));
}
```

问题4 isTmax

题目描述

判断是不是最大的二进制int整数，即 `01111111...1`

思路&优化过程

基本思路 （6步）

最大int整数+1得到 `100000...000` ， 与其本身相加会得到 `111111...111` 。但值得注意的是 `-1` ，即 `11111...111` 也会有此性质，需要把这种情况排除，方法是判断这个数 + 1取反是否为0。

```
int isTmax(int x)
{
  int m = x + 1;
  x = ~(m + x) + (!m);

  return !x;
}
```

优化（5步）

最大int整数+1得到 `100000...000` ， 这个数加自己相加会得到 `000000...000` ， 用这种方法可以节省一个符号。

```
int isTmax(int x)
{
    int m = x + 1;
    return !((m + m) | (!m));
}
```

问题5 fitsBits

题目描述

判断一个 `int` 数是否可以表示成 `n` 位的有符号整数。（是否超出 `n` 位有符号整数的范围）

基本思路

思路和问题三基本相同，问题在于构造出 `n-1` 。

- 版本1 用 `n + (~1 + 1)` 代表 `n - 1` 。

```
int fitsBits(int x, int n)
{
  int i = n + (~1 + 1);
  return !((x >> i) ^ (x >> 31));
}
```

- 优化版本1.0 用 `n + (~1)` 代表 `n - 1` 。
- 优化版本2.0 用 `n + 31` 代表 `n - 1` 。

```
int fitsBits(int x, int n)
{
    // int i = n + (~0); //~0 means -1，6步
    int i = n + 31; // means j减1，  5步
    return !((x >> i) ^ (x >> 31));
}
```

问题6 upperBits

题目描述

设置一个数的前 n 位为1。

思路和优化

- 开始想用long long，发现不行。
- 注意判别输入0的情况，不输出任何1。
- 用 !!(n) 来判断输入是否为0，然后移到最左，再相对应右移 n-1 位。
- n-1 有多种写法
    - 1.0版 n + (~1 + 1)
    - 2.0版 n + (~1)
    - 3.0版 n + 31
      代码如下：

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

问题7 anyOddBit

题目描述

判断是否有任何奇数位的数字为1。

思路

构造掩码 m = 10101010...10101010 ,即 AAAAAAAA ， 然后掩码与数字与运算即可判断。

```
int anyOddBit(int x)
{
    int m = 0xAA + (0xAA << 8);
    m = m + (m << 16);
    return !!(m & x);
}
```

问题8 byteSwap

题目描述

交换第n个和第m个字节。

思路

- 实现交换，想到用异或的思想。
- 异或交换原理
  - 与自身异或为0；
  - 与0异或还是自身；
  - 具有交换律；
  - 具有结合律。
- 首先，将字节扩展到位。每个byte代表8个bit，左移3位即乘8。
- x右移相应的位数得到的末8位要交换的那个字节。
- 异或交换，保留后八位。
- 再把bits1分别移到需要交换的两个字节处
  - 对于每一个字节，异或两次为0，异或一次为自身。
  - 与原数异或之后，该位置的字节异或了两次，另一个字节异或了一次，因此留的是另一个字节，实现交换。

代码如下：

```
int byteSwap(int x, int n, int m)
{
    //将字节扩展到位
    int n_bit = n << 3;
    int m_bit = m << 3;

    //异或交换
    int bits1 = ((x >> n_bit) ^ (x >> m_bit));
    //保留后八位（1个字节）
    bits1 = bits1 & 0xFF;

    //再把bits1分别移到需要交换的两个字节处
    int ans = x ^ (bits1 << n_bit) ^ (bits1 << m_bit);
    return ans;
}
```

问题9 absVal

题目描述

输出给定数字的绝对值

思路

思路1

- 关键在于怎么实现选择的结构，>0不变，<0取反
- `x >> 31` 取出符号位（全0或全1）
- 如果为1，负数，则取反+1（`(~x) + 1`）
- 如果为0，正数，不变。
- 若x为负数，`x >> 31` 为0xFFFFFFFF，`(x >> 31)^x` 可以把取反；若x为正数，`x >> 31` 为0，`(x >> 31)^x` 操作后x不变。
- `num & a` 是一个调整的构造，在x>0时为0，x<0时为1。
  代码如下：

```
int absVal(int x)
{
    int num = x >> 31;
    int a = 0x1;
    return (num ^ x) + (num & a); // 4步
}
```

问题10 divpwr2

题目描述

一个数除以2的指数，向0取整。

思路

- 正数右移n为即得到结果。
- 负数右移n位结果不同，因为取整规则不同，所以我们要构造一个偏移值。
- 找规律得到偏移值。-5/2^2 =-1.25->-1,正确；-6/2^2 =-1.5->-2,应该为-1;-7/2^2 =-1.75->-2,但也应该为-1.我们整体给一个 + 3的偏移值之后，他的值四舍五入之后就等于向0取整的值了。
- 所以偏移值为2^n-1。

```
int divpwr2(int x, int n)
{
    int s = x >> 31;
    int b = (1 << n) + ~0;
    x = x + (b & s);
    return x >> n;
}
```

问题11 float_neg

题目描述

如果是NAN，返回参数；否则返回-f。

思路

- NAN：如果指数位区域全位1，且小数位不为0，这个数表示为不是一个数。形如 `0111 1111 1....` 。
- 让这个数与 `0111 1111 1..111` 做与运算，如果前面重合且后23位不为零，则满足NAN情况。
- 计算-f直接把最高位取反即可（异或1取反）。

```
unsigned float_neg(unsigned uf)
{
    if ((uf & 0x7FFFFFFF) > 0x7F800000)
        return uf;
    return (uf ^ 0x80000000);
}
```

问题12 logicalNeg

题目描述

用位运算实现逻辑非！。

思路

- 非0返回0，0返回1
- 利用特性：任何数取反+1，最高位一定和本身相反；但0取反1，还是0。

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

问题13 bitMask

问题描述

构造中间几位为1的掩码，如 `00000111100000` 。

思路

- 分别构造两个高位为0和低位为0的掩码，两个与运算得到目标掩码。
- 用 `0xFFFFFFFF` 来得到两个高位为0和低位为0的掩码。
- 低位为0的掩码直接左移相应位数。
- 高位为0的掩码在高位往前一位 + 1，直接把高位的1全部进1变成0。
- 在构造高位0掩码时，不能直接 `a << (highbit + 1)` ，否则当 `highbit` 为 `31` 时，这是个未定义行为。

```
int bitMask(int highbit, int lowbit)
{
    int x = ~0;
    int a = 0x1;
    int low = x << lowbit;
    int hi = x + (a << highbit << 1);
    return low & hi;
}
```

问题14 isGreater

问题描述

如果x>y，返回1；否则返回0。

思路

- 通过 `x >> 31` 和 `y >> 31` 分别获取x、y的符号位。
- 符号位不等的情况下，只需判断：若x负y正，返回0,；其余返回1
- 符号位相等的情况下，需要判断大小。
  - 注意只有x>y，才返回1，即x>=y+1,才返回1。
  - 注意到，-y = ~y + 1,因此~y + x = x - y - 1,如果x - y - 1 >= 0,即x>=y-1,即x>y,返回1，其余返回0。
  - 因此可以直接用 `(~y + x) >> 31` 来判断。
- 符号位相等和不等种情况用或（|）来连接。

```
int isGreater(int x, int y)
{
    //获取x、y的符号位
    int sign_x = x >> 31;
    int sign_y = y >> 31;

    //符号位不等
    int sign_not_equal = sign_x & !sign_y;

    //符号位相等的情况下，判断大小
    //其中-y = ~y + 1,因此~y + x = x - y - 1,如果x - y - 1 >= 0,即x>=y-1,即x>y,返回1，其余返回0.
    int sign_equal = (!(sign_x ^ sign_y)) & ((~y + x) >> 31);
    return !(sign_equal | sign_not_equal);
}
```

问题15 logicalShift

问题描述

向右移动n位，前面补0.

思路

- 向右移动，与一个高n位为0、其他位1的掩码进行与运算，消掉高位可能出现的1
- 重点在于怎么构造形如 `0...0011...11` 的掩码。
- 构造1： `m = (~0) + (1 << (~n) << 1)` ,类似bitmask题目。(7步)

```
int logicalShift(int x, int n)
{
    int ans = x >> n;
    int m = (~0) + (1 << (~n) << 1);
    return ans & m;
}
```

问题16 satMul2

问题描述

计算一个数×2，如果溢出输出最大/最小值。

思路

- 正常情况下，左移一位即为×2。
- 如果溢出，即最高位符号位和次高位不相同，左移之后符号位取反，用 `bit_2 = ans >> 31` 取出次高位。
- 用异或 `x ^ ans` 可以得到最高位和次高位是否相同，作为分支的依据。

- 如果相同，notflow。
- 如果不相同，溢出，则需要考虑溢出的最大还是最小。
  - 如果次高位为1（bit_2为111……），最高位0，则正数溢出，应该输出01111……
  - 如果次高位为0（bit_2为000……），最高位1，则负数溢出，应该输出10000……
  - 归纳可知，可以用bit_2和10000……异或实现对输出的控制。
    代码：

```
int satMul2(int x)
{
    int ans = x << 1;
    int bit_2 = ans >> 31;
    int same = (x ^ ans) >> 31;
    int notflow = (~same) & ans;
    int t = 1 << 31;
    int isflow = (same) & (t ^ bit_2);
    return notflow + isflow;
}
```

问题17 subOK

题目描述

判断两数相减会不会超出 `int` 表示范围。

思路

- `x-y` 用 `x + (~y + 1)` 来表示。
- 溢出只有x为正y为负，x为负y为正两种情况，即xy异号，用 `x^y` 来判断。
- 若溢出，结果和x是异号的，同样用异或判断。
- 因此如果x和y异号和x和result异号，则溢出。
- 返回0或1，用！来调整。

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

问题18 trueThreeFourths

问题描述

求一个数×3/4的值，向0舍入。

解决思路

关键在于怎么区分正负数和处理负数时向上取整的问题。

- 整数部分：
  - 除以4，再将结果乘3即可。
- 小数部分：
  - 对于正数，获取除以4的余数（0,1,2,3），乘三再除以4取整即可判断是否需要舍入。
  - 对于负数，获取除以4的余数，还要加上3才可以向上取整。
    - 加3小技巧： `sgn = (x >> 31) & 3` ，顺便可以判断正负号。
      代码如下：

```
int trueThreeFourths(int x)
{
    //除以4
    int x1 = x >> 2;
    // x2获取除以4的余数 0,1,2,3
    int x2 = x & 3;
    //乘以4
    int integer = (x1 + x1 + x1);
    // 向零取整，意味着>=0时向下取整，<0时向上取整，即在/4前+3
    // 正数不动（本来就是向下取整），负数加3，再整体除以4
    int sgn = (x >> 31) & 3;
    int  fraction = (x2 + x2 + x2 + sgn) >> 2;
    return integer + fraction;
}
```

问题19 isPower2

题目描述

判断一个数是不是2的倍数。特别的，负数和零都不是2的倍数。

思路

- 2的倍数具有的性质：第n位为1，其余位都为0。这个数-1会得到0~n-1位为1，其余位为0的数，在和数字本身与运算会把所有的1都消掉，变成全0。
- 然而x=0也具有上述性质，因此需要排除。
- 凡是负数，即最高位为1的数，都排除。
- 因此，首先构造 `flag = ~(x >> 31)` 再和 `(!!x)` 做与运算，排除负数和0的情况。
- `ans = ~(x & (x + (~0)))` 即为判断2的倍数的性质。
- `ans` 和 `flag` 做与运算可以得到结果。

```
int isPower2(int x)
{
    int flag = ~(x >> 31);
    flag = flag & (!!x);
    int ans = !(x & (x + (~0)));
    return ans & flag;
}
```

问题20 float_i2f

题目描述

将一个int类型整数转成float的浮点数形式。

思路

按照浮点数的表示基本规则，分为三段（符号位、阶码、尾数）

- 首先，对0进行特判。
- 把负数变为正数方便后续处理。
- 符号位：用 `sign = x & 0x80000000` 获取符号位。
- 阶码位
  - 找到除最高位之外的第一个1，times是这个1的从左往右数的第几个
  - 阶码：127+32-times
  - 这个数左移23放到浮点数的阶码位置。
- 尾数：
  - 原数右移9位，放到尾数位置。
  - 考虑尾数数字右移9位之后的舍入问题
    - 如果后9位大于1 0000 0000，要进一位
    - 如果后10位是11 0000 0000，也要进一位
    - 其他情况不进位
- 把三段相加拼接即可。
  完整代码：

```
unsigned float_i2f(int x)
{

    unsigned sign = x & 0x80000000;
    unsigned ans = x;
    if (x == 0)
        return 0;
    //如果是负数，变为正数
    if (x < 0)
    {
        ans = -x;
    }
    unsigned times = 0;
    unsigned mask = 0x80000000;
    unsigned t;
    //找到除最高位之外的第一个1，times是这个1的从左往右数的第几个
    //得到的ans是1之后的小数位的数字
    while (1)
    {
        t = ans;
        ans = ans << 1;
        times++;
        if (t & mask)
            break;
    }
    //下面考虑尾数数字右移9位之后的舍入问题
    unsigned flag;
    //如果后9位大于1 0000 0000，要进一位
    if ((ans & 0x01FF) > 0x0100)
        flag = 1;
    //如果后10位是11 0000 0000，也要进一位
    else if ((ans & 0x03FF) == 0x0300)
        flag = 1;
    //其他情况不进位
    else
        flag = 0;
    //把三段拼接起来（尾数：原数右移9位；阶码：127+32-times）
    int anss = sign + (ans >> 9) + ((159 - times) << 23) + flag;
    return anss;
}
```

问题21 howManyBits

问题描述

判断x需要多少位的补码表示。

解决思路

- 用异或的思想，即 `check = x ^ (x << 1)` 可以得到从低到高位最后一个有意义的1（或者说是最后一个1）。

- 下面转化为如何找到 check 里这个最高位的1在哪个位置，采用二分查找的思想。
- 对 check 右移16位，查看其$17_{32位是否为零。如果不为零，则代表最高位在16}$32位中，将 check 再右移十六位，下一步是查看在$17_{32位中的哪一位；如果不为零，则代表最高位在1}$16位中，将 check 不再右移，下一步是查看在1~16位中的哪一位。
- 下面右移8位，将check分成1-8位和9-16位查找，重复上述思路。
- 右移4位，将check分成1-4位和5-8位查找，重复上述思路。
- 以此类推，直到将check分成1位和2位。
- 最后的答案应该 + 1，因为无论是任何数都至少有1位（即使是0）。

```
int howManyBits(int x)
{
    int check = x ^ (x << 1);
    int bit_16_to_32 = !!(check >> 16) << 4;
    check >>= bit_16_to_32;
    int bit_8_to_16 = !!(check >> 8) << 3;
    check >>= bit_8_to_16;
    int bit_4_to_8 = !!(check >> 4) << 2;
    check >>= bit_4_to_8;
    int bit_2_to_4 = !!(check >> 2) << 1;
    check >>= bit_2_to_4;
    int bit_1_to_2 = !!(check >> 1);
    int ans = 1 + bit_16_to_32 + bit_8_to_16 + bit_4_to_8 + bit_2_to_4 + bit_1_to_2;
    return ans;
}
```

问题22 float_half

问题描述

输出0.5*一个浮点数的值，如果输入的是NAN则返回原值。

解决思路

对于规范化浮点数，乘0.5即为阶码-1；对于非规范化浮点数，乘0.5即为尾数右移一位，并考虑舍入问题。

- 首先用 uf & 0x80000000 获取符号位。
- 特判NAN：如果阶码的八位全为1，则直接返回原值。

```
if ((uf & 0x7FFFFFFF) >= 0x7F800000)
        return uf;
```

- 用 e = uf & 0x7F800000 获取八位阶码。
- 如果阶码>1，直接减掉阶码部分的1即可。

```
if (e > 0x00800000)
        return (uf - 0x00800000);
```

- 如果阶码<=1，需要尾数部分整体右移一位，注意需要先 `int shift = uf ^ sign` 把符号位踢掉。
- 考虑舍入问题，右移一位只需要考虑最后两位：如果后两位为10，或00，直接右移舍去；如果为01，向偶数舍入，也是直接右移舍去；如果为11，向偶数舍入需要进一位，所以先 + 1，再舍去。判断语句： `(uf & 0x3) == 0x3` 。
- 后来发现更好的特判NAN办法：在获取阶码之后，直接比对阶码是否全为1：

```
if (e == 0x7F800000)
        return uf;
```

最终代码：

```
unsigned float_half(unsigned uf)
{
    unsigned sign = uf & 0x80000000;
    unsigned e = uf & 0x7F800000;
    if (e == 0x7F800000)
        return uf;
    if (e > 0x00800000)
        return (uf - 0x00800000);
    else if (e <= 0x00800000)
    {
        int shift = uf ^ sign;
        int l = ((uf & 0x3) == 0x3);
        return (sign + ((shift + l) >> 1));
    }
}
```

总结与体会

- 位运算操作
  相对于一般的高级语言运算，使用位运算需要我们对底层算法逻辑更加明晰。通过这次的 实验，我对位运算的操作、补码的表示、浮点数的表示有了更深入的了解和认识，也对他们之间的运算逻辑更为熟悉。
- 要关注边界条件和特殊值
  datalab中的很多题，我在刚开始写的时候感觉思路很清晰，但总是无法通过。后来发现是很多边界条件我都没有考虑在内，导致边界的值输出不正确，比如0、全1、第32位等等。对边界的考虑的全面性在平时写代码中也是很关键的，可以提高代码的容错性和是程序的逻辑更清晰有效，通过这次实验，我也提高了写代码过程中的边界条件敏感性。

非常感谢柴老师课上的耐心讲解，感谢助教在我完成本次实验中提供的帮助！