

scA 为 1, scB 为 0; 如果 a-b 的值为负数, 那么 scA 为 0, scB 为 1。scA 和 scB 必有一个为 1, 另一个必为 0。所以 `return a * scA + b * scB;`, 就是根据 a-b 的值的状况, 选择要么返回 a, 要么返回 b。

但方法一是有局限性的, 那就是如果 a-b 的值出现溢出, 返回结果就不正确。

第二种方法可以彻底解决溢出的问题, 也就是如下代码中的 `getMax2` 方法。

```
public int getMax2(int a, int b) {
    int c = a - b;
    int sa = sign(a);
    int sb = sign(b);
    int sc = sign(c);
    int difSab = sa ^ sb;
    int sameSab = flip(difSab);
    int returnA = difSab * sa + sameSab * sc;
    int returnB = flip(returnA);
    return a * returnA + b * returnB;
}
```

解释一下 `getMax2` 方法。

如果 a 的符号与 b 的符号不同 (`difSab==1,sameSab==0`), 则有:

- 如果 a 为 0 或正, 那么 b 为负 (`sa==1,sb==0`), 应该返回 a;
- 如果 a 为负, 那么 b 为 0 或正 (`sa==0,sb==1`), 应该返回 b。

如果 a 的符号与 b 的符号相同 (`difSab==0,sameSab==1`), 这种情况下, a-b 的值绝对不会溢出:

- 如果 a-b 为 0 或正 (`sc==1`), 返回 a;
- 如果 a-b 为负 (`sc==0`), 返回 b;

综上所述, 应该返回 `a * (difSab * sa + sameSab * sc) + b * flip(difSab * sa + sameSab * sc)`。

只用位运算不用算术运算实现整数的加减乘除运算

【题目】

给定两个 32 位整数 a 和 b, 可正、可负、可 0。不能使用算术运算符, 分别实现 a 和 b 的加减乘除运算。

【要求】

如果给定的 a 和 b 执行加减乘除的某些结果本来就会导致数据的溢出, 那么你实现的

函数不必对那些结果负责。

【难度】

尉 ★★★☆☆

【解答】

用位运算实现加法运算。如果在不考虑进位的情况下， $a \oplus b$ 就是正确结果，因为 0 加 0 为 0(0&0)，0 加 1 为 1(0&1)，1 加 0 为 1(1&0)，1 加 1 为 0(1&1)。

例如：

a: 001010101

b: 000101111

无进位相加，即 $a \oplus b$: 001111010

在只算进位的情况下，也就是只考虑 a 加 b 的过程中进位产生的值是什么，结果就是 $(a \& b) \ll 1$ ，因为在第 i 位上只有 1 与 1 相加才会产生 $i-1$ 位的进位。

例如：

a: 001010101

b: 000101111

只考虑进位的值，即 $(a \& b) \ll 1$: 000001010

把完全不考虑进位的相加值与只考虑进位的产生值再相加，就是最终的结果。也就是说，一直重复这样的过程，直到进位产生的值完全消失，说明所有的过程都加完了。

例如：

a: 001010101

b: 000101111

上边两值的 \oplus 结果: 001111010

上边两值的 $\&\ll 1$ 结果: 000001010

上边两值的 \oplus 结果: 001110000

上边两值的 $\&\ll 1$ 结果: 000010100

上边两值的 \oplus 结果: 001100100

上边两值的 $\&\ll 1$ 结果: 000100000

上边两值的 \wedge 结果: 001000100

上边两值的 $\&\ll 1$ 结果: 001000000

上边两值的 \wedge 结果: 000000100

上边两值的 $\&\ll 1$ 结果: 010000000

上边两值的 \wedge 结果: 010000100

上边两值的 $\&\ll 1$ 结果: 000000000

最后 $\&\ll 1$ 结果为 0, 则过程终止, 返回 010000100。具体请参看如下代码中的 add 方法。

```
public int add(int a, int b) {
    int sum = a;
    while (b != 0) {
        sum = a ^ b;
        b = (a & b) << 1;
        a = sum;
    }
    return sum;
}
```

用位运算实现减法运算。实现 $a-b$ 只要实现 $a+(-b)$ 即可, 根据二进制数在机器中表达的规则, 得到一个数的相反数, 就是这个数的二进制数表达取反加 1 (补码) 的结果。具体请参看如下代码中的 negNum 方法。实现减法运算的全部过程请参看如下代码中的 minus 方法。

```
public int negNum(int n) {
    return add(~n, 1);
}

public int minus(int a, int b) {
    return add(a, negNum(b));
}
```

用位运算实现乘法运算。 $a*b$ 的结果可以写成 $a*2^0*b_0+a*2^1*b_1+\dots+a*2^i*b_i+\dots+a*2^{31}*b_{31}$, 其中, b_i 为 0 或 1 代表整数 b 的二进制数表达中第 i 位的值。举一个例子, $a=22=000010110$, $b=13=000001101$, $res=0$ 。

a: 000010110

b: 000001101

res: 000000000

b 的最右侧为 1，所以 $res=res+a$ ，同时 b 右移一位，a 左移一位。

a: 000101100

b: 000000110

res: 000010110

b 的最右侧为 0，所以 res 不变，同时 b 右移一位，a 左移一位。

a: 001011000

b: 000000011

res: 000010110

b 的最右侧为 1，所以 $res=res+a$ ，同时 b 右移一位，a 左移一位。

a: 010110000

b: 000000001

res: 001101110

b 的最右侧为 1，所以 $res=res+a$ ，同时 b 右移一位，a 左移一位。

a: 101100000

b: 000000000

res: 100011110

此时 b 为 0，过程停止，返回 $res=100011110$ ，即 286。

不管 a 和 b 是正、负，还是 0，以上过程都是对的，因为都满足 $a*b=a*2^0*b_0+a*2^1*b_1+\dots+a*2^i*b_i+\dots+a*2^{31}*b_{31}$ 。具体请参看如下代码中的 multi 方法。

```
public int multi(int a, int b) {
    int res = 0;
    while (b != 0) {
        if ((b & 1) != 0) {
            res = add(res, a);
        }
        a <<= 1;
        b >>= 1;
    }
    return res;
}
```

用位运算实现除法运算，其实就是乘法的逆运算。先举例说明一种最普通的情况， a 和 b 都不为负数，假设 $a=286=100011110$ ， $b=22=000010110$ ， $res=0$ ：

a: 100011110

b: 000010110

res: 000000000

b 向~~左~~右位移 31 位、30 位、……、4 位时，得到的结果都大于 a 。而当 b 向~~左~~右位移 3 位的结果为 010110000，此时 $a \geq b$ 。根据乘法的范式，如果 $b * res = a$ ，则 $a = b * 2^0 * res_0 + b * 2^1 * res_1 + \dots + b * 2^i * res_i + \dots + b * 2^{31} * res_{31}$ 。因为 b 在向~~左~~右位移 31 位、30 位、……、4 位时，得到的结果都比 a 大，说明 a 包含不下 $b * 2^{31} \sim b * 2^4$ 的任何一个，所以 $res_4 \sim res_{31}$ 这些位置上应该都为 0。而 b 在向~~左~~右位移 3 位时， $a \geq b$ ，说明 a 可以包含一个 $b * 2^3$ ，即 $res_3 = 1$ 。接下来看剩下的 a ，即 $a - b * 2^3$ ，还能包含什么。

a: 001101110

b: 000010110

res: 000001000

b 向右位移 2 位之后为 001011000，此时 $a \geq b$ ，说明剩下的 a 可以包含一个 $b * 2^2$ ，即 $res_2 = 1$ ，然后让剩下的 a 减掉一个 $b * 2^2$ ，看还能包含什么。

a: 000010110

b: 000010110

res: 000001100

b 向右位移 1 位之后大于 a ，说明剩下的 a 不能包含 $b * 2^1$ 。 b 向右位移 0 位之后 $a = b$ ，说明剩下的 a 还能包含一个 $b * 2^0$ ，即 $res_0 = 1$ 。当剩下的 a 再减去一个 b 之后，结果为 0，说明 a 已经完全被分解干净，结果就是此时的 res ，即 000001101=13。

以上过程其实就是先找到 a 能包含的最大部分，然后让 a 减去这个最大部分，再让剩下的 a 找到次大部分，并依次找下去。

以上过程只适用于当 a 和 b 都不是负数的时候，所以，如果 a 和 b 中有一个为负数或者都为负数时，可以先把 a 和 b 转成正数，计算完成后再看 res 的真实符号是什么就可以。

具体请参看如下代码中的 `div` 方法，`sign` 方法是判断整数 n 是否为负，负数返回 `true`，否则返回 `false`。

```
public boolean isNeg(int n) {
    return n < 0;
}
```

```

public int div(int a, int b) {
    int x = isNeg(a) ? negNum(a) : a;
    int y = isNeg(b) ? negNum(b) : b;
    int res = 0;
    for (int i = 31; i > -1; i = minus(i, 1)) {
        if ((x >> i) >= y) {
            res |= (1 << i);
            x = minus(x, y << i);
        }
    }
    return isNeg(a) ^ isNeg(b) ? negNum(res) : res;
}

```

除法实现还剩非常关键的最后一步。以上方法可以算绝大多数的情况，但我们知道 32 位整数的最小值为-2147483648，最大值为 2147483647，最小值的绝对值比最大值的绝对值大 1，所以，如果 a 或 b 等于最小值，是转不成相对应的正数的。可以总结一下：

- 如果 a 和 b 都不为最小值，直接使用以上过程，返回 div(a,b)。
- 如果 a 和 b 都为最小值，a/b 的结果为 1，直接返回 1。
- 如果 a 不为最小值，而 b 为最小值，a/b 的结果为 0，直接返回 0。
- 如果 a 为最小值，而 b 不为最小值，怎么办？

第 1~3 情况处理都比较容易，对于情况 4 就棘手很多。我们举个简单的例子说明本书是如何处理这种情况的。为了方便说明，我们假设整数的最大值为 9，而最小值为-10。当 a 和 b 属于[0,9]的范围时，我们可以正确地计算 a/b。当 a 和 b 都属于[-9,9]时，我们可以计算，也就是情况 1；当 a 和 b 都等于-10 时，我们也可以计算，就是情况 2；当 a 属于[-9,9]，而 b 等于-10 时，我们也能计算，就是情况 3；当 a 等于-10，而 b 属于[-9,9]时，如何计算呢？

1. 假设 a=-10，b=5。
2. 计算(a+1)/b 的结果，记为 c。对本例来讲就是-9/5 的结果，c=-1。
3. 计算 c*b 的结果。对本例来讲，-1*5=-5。
4. 计算 a-(c*b)，即-10-(-5)=-5。
5. 计算(a-(c*b))/b 的结果，记为 rest，意义是修正值，即-5/5=-1。
6. 返回 c+rest 的结果。

也就是说，既然我们对最小值无能为力，那么就把最小值增加一点，计算出一个结果，然后根据这个结果再修正一下，得到最终的结果。

除法运算的全部过程请参看如下代码中的 divide 方法。

```

public int divide(int a, int b) {

```

```

    if (b == 0) {
        throw new RuntimeException("divisor is 0");
    }
    if (a == Integer.MIN_VALUE && b == Integer.MIN_VALUE) {
        return 1;
    } else if (b == Integer.MIN_VALUE) {
        return 0;
    } else if (a == Integer.MIN_VALUE) {
        int res = div(add(a, 1), b);
        return add(res, div(minus(a, multi(res, b)), b));
    } else {
        return div(a, b);
    }
}

```

整数的二进制表达中有多少个 1

【题目】

给定一个 32 位整数 n ，可为 0，可为正，也可为负，返回该整数二进制表达中 1 的个数。

【难度】

尉 ★★☆☆

【解答】

最简单的解法。整数 n 每次进行无符号右移一位，检查最右边的 bit 是否为 1 来进行统计。具体请参看如下代码中的 `count1` 方法。

```

public int count1(int n) {
    int res = 0;
    while (n != 0) {
        res += n & 1;
        n >>= 1;
    }
    return res;
}

```

如上方法在最复杂的情况下要经过 32 次循环，下面看一个循环次数只与 1 的个数有关的解法，如下代码中的 `count2` 方法。

```

public int count2(int n) {

```