```
/*
 * CS:APP Data Lab
 *
 * <Please put your name and userid here>
 * name- Riti Patel
 UID- 706254474
 * bits.c - Source file with your solutions to the Lab.
 *          This is the file you will hand in to your instructor.
 *
 * WARNING: Do not include the <stdio.h> header; it confuses the dlc
 * compiler. You can still use printf for debugging without including
 * <stdio.h>, although you might get a compiler warning. In general,
 * it's not good practice to ignore compiler warnings, but in this
 * case it's OK.
 */

#if 0
/*
 * Instructions to Students:
 *
 * STEP 1: Read the following instructions carefully.
 */

You will provide your solution to the Data Lab by
editing the collection of functions in this source file.

INTEGER CODING RULES:

  Replace the "return" statement in each function with one
  or more lines of C code that implements the function. Your code
  must conform to the following style:

  int Funct(arg1, arg2, ...) {
      /* brief description of how your implementation works */
      int var1 = Expr1;
      ...
      int varM = ExprM;

      varJ = ExprJ;
      ...
      varN = ExprN;
      return ExprR;
  }

  Each "Expr" is an expression using ONLY the following:
  1. Integer constants 0 through 255 (0xFF), inclusive. You are
      not allowed to use big constants such as 0xffffffff.
  2. Function arguments and local variables (no global variables).
  3. Unary integer operations ! ~
  4. Binary integer operations & ^ | + << >>

  Some of the problems restrict the set of allowed operators even further.
  Each "Expr" may consist of multiple operators. You are not restricted to
  one operator per line.

  You are expressly forbidden to:
  1. Use any control constructs such as if, do, while, for, switch, etc.
  2. Define or use any macros.
  3. Define any additional functions in this file.
```

4. Call any functions.
5. Use any other operations, such as &&, ||, -, or ?:
6. Use any form of casting.
7. Use any data type other than int. This implies that you
   cannot use arrays, structs, or unions.

You may assume that your machine:
1. Uses 2s complement, 32-bit representations of integers.
2. Performs right shifts arithmetically.
3. Has unpredictable behavior when shifting if the shift amount
   is less than 0 or greater than 31.

EXAMPLES OF ACCEPTABLE CODING STYLE:

```
/*
 * pow2plus1 - returns 2^x + 1, where 0 <= x <= 31
 */
int pow2plus1(int x) {
   /* exploit ability of shifts to compute powers of 2 */
   return (1 << x) + 1;
}

/*
 * pow2plus4 - returns 2^x + 4, where 0 <= x <= 31
 */
int pow2plus4(int x) {
   /* exploit ability of shifts to compute powers of 2 */
   int result = (1 << x);
   result += 4;
   return result;
}
```

NOTES:

1. Our checker requires that you do NOT define a variable after
   a statement that does not define a variable.

   For example, this is NOT allowed:

```
int illegal_function_for_this_lab(int x, int y) {
  // this statement doesn't define a variable
  x = x + y + 1;

  // The checker for this lab does NOT allow the following statement,
  // because this variable definition comes after a statement
  // that doesn't define a variable
  int z;

  return 0;
}
```

2. Use the dlc (data lab checker) compiler (described in the handout) to
   check the legality of your solutions.
3. Each function has a maximum number of operations (! ~ & ^ | + << >>)
   that you are allowed to use for your implementation of the function.
   The max operator count is checked by dlc. Note that assignment ('=')
   is not counted; you may use as many of these as you want without penalty.
4. Use the btest test harness to check your functions for correctness.
5. The maximum number of ops for each function is given in the

```
     header comment for each function.

/*
 * STEP 2: Modify the following functions according the coding rules.
 *
 *   IMPORTANT. TO AVOID GRADING SURPRISES:
 *   1. Use the dlc compiler to check that your solutions conform
 *      to the coding rules.
 *   2. Use btest to formally verify that your solutions produce
 *      the correct answers.
 */


#endif

/*
 * tmin - return minimum two's complement integer
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 4
 *   Rating: 1
 */
int tmin(void) {
  return 1 << 31;
}

/*
 * isTmax - returns 1 if x is the maximum, two's complement number,
 *          and 0 otherwise
 *   Legal ops: ! ~ & ^ | +
 *   Max ops: 10
 *   Rating: 1
 */
int isTmax(int x) {
  return !((x^(~(x+1)))|!(~x));
}

/*
 * bitXor - x^y using only ~ and &
 *   Example: bitXor(4, 5) = 1
 *   Legal ops: ~ &
 *   Max ops: 14
 *   Rating: 1
 */
int bitXor(int x, int y) {
  int notX = ~(x & ~y);
  int notY = ~(~x & y);
  return ~(notX & notY);
}

/*
 * sign - return 1 if positive, 0 if zero, and -1 if negative
 *   Examples: sign(130) = 1
 *             sign(-23) = -1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 10
 *   Rating: 2
 */
int sign(int x) {
  int sign = x>> 31;
```

```
    int z = !x;
    return (~z& (sign|1))| (z &0);
}

/*
 * fitsBits - return 1 if x can be represented as an
 *            n-bit, two's complement integer.
 *   1 <= n <= 32
 *   Examples: fitsBits(5, 3) = 0
 *             fitsBits(-4, 3) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 15
 *   Rating: 2
 */
int fitsBits(int x, int n) {
    int z= 32 +(~n+1);
    return !(x^((x<<z)>>z));
}

/*
 * byteSwap - swaps the nth byte and the mth byte
 *   Examples: byteSwap(0x12345678, 1, 3) = 0x56341278
 *             byteSwap(0xDEADBEEF, 0, 2) = 0xDEEFBEAD
 *   You may assume that 0 <= n <= 3, 0 <= m <= 3
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 25
 *   Rating: 2
 */
int byteSwap(int x, int n, int m) {
   int z = 0xFF;
   int mShift = m << 3;
   int shift = n << 3;



   int nbyte = (z << shift)&x;
   int mbyte = (z << mShift)&x;
   int c = (z << shift) | (z << mShift);


   nbyte = (nbyte >> shift)&z;
   mbyte = (mbyte >> mShift)&z;

   nbyte = nbyte << mShift;
   mbyte = mbyte << shift;

   c = ~c & x;

   return c | mbyte | nbyte;
}

/*
 * isAsciiDigit - return 1 if 0x30 <= x <= 0x39
 *                (ASCII codes for characters '0' to '9')
 *   Examples: isAsciiDigit(0x35) = 1
 *             isAsciiDigit(0x3a) = 0
 *             isAsciiDigit(0x05) = 0
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 15
```

```
 *   Rating: 3
 */
int isAsciiDigit(int x) {
  int sign = 1 << 31;
   int upper = ~(sign | 0x39);
   int lower = ~0x30;


   upper= sign & (upper+x) >> 31;
   lower = sign & (lower+1+x) >> 31;

   return !(upper | lower);
}

/*
 * conditional - same as x ? y : z
 *   Example: conditional(2, 4, 5) = 4
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 16
 *   Rating: 3
 */
int conditional(int x, int y, int z) {
  x = !!x;
  x = ~x+1;
  return (x & y) | (~x & z);
}

/*
 * subtractionOK - Determine if can compute x-y without overflow
 *   Examples: subtractionOK(0x80000000, 0x80000000) = 1
 *             subtractionOK(0x80000000, 0x70000000) = 0
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 3
 */
int subtractionOK(int x, int y) {
  int res = x + (~y + 1);
  int same = x ^ y;
  int resS = res ^ x;

  return !((same & resS) >> 31);
}

/*
 * rotateRight - Rotate x to the right by n
 *   Can assume that 0 <= n <= 31
 *   Example: rotateRight(0x87654321, 4) = 0x18765432
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 25
 *   Rating: 3
 */
int rotateRight(int x, int n) {
  int moving= x << (32 + (~n +1));
  int tMax = ~(1 << 31);
  tMax = tMax >> (n+((~1)+1));
  x = x >> n;
  x = x & tMax;
  return x | moving;
}
```

```
// Below are the extra credit problems (4 pts in total)
// 2 points each (1 correctness pt + 1 performance pt)

/*
 * bitParity - returns 1 if x contains an odd number of 0's
 *   Examples: bitParity(5) = 0
 *             bitParity(7) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 */
int bitParity(int x) {
  return 2;
}

/*
 * greatestBitPos - return a mask that marks the position of the
 *                  most significant 1 bit. If x == 0, return 0
 *   Example: greatestBitPos(96) = 0x40
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 70
 */
int greatestBitPos(int x) {
  return 2;
}
```