1. Evaluate the following and determine if they have a positive overflow, negative overflow, or a normal operation (Final answers in decimal):
   a. $B2T_6(001111) + B2T_6(000001)$
      001111
      +000001
      = 010000 => 16
   b. $B2T_6(111011) + B2T_6(100101)$
      111011
      +100101
      = 1100000 => negative overflow
   c. $B2T_8(11101111) + B2T_8(10110001)$
      11101111
      +10110001
      = 110100000 => negative overflow
   d. $B2T_8(11000011) + B2T_8(11110001)$
      11000011
      +11110001
      = 1001110100 => negative overflow
   e. $B2T_{10}(0101001100) + B2T_{10}(0111000111)$
      0101001100
      +0111000111
      = 1100010011 => positive overflow
   f. $B2T_{10}(1101001100) + B2T_{10}(0111000111)$
      1101001100
      +0111000111
      = 10100010011 => negative overflow

2. Write code for a function mul3div4 that, for integer argument x, computes (3 ∗ x/4) but follows the **bit-level integer coding rules** (see above). Your code should replicate that the computation 3*x can cause an overflow.

```
int mul3div4(int x) {
    int mul = x + (x << 1);  // 3 * x using bit shifting
    return mul >> 2;  // divide by 4 using bit shifting
}
```

3. Write a function with the following prototype:

   /* Determine whether arguments can be subtracted without overflow*/
   int tsub_ok(int x, int y);

   This function should return 1 if the computation x-y does not overflow.

```
int tsub_ok(int x, int y) {
    int sign_x = (x >> 31) & 1; // isolate sign of x
    int sign_y = (y >> 31) & 1; // isolate sign of y
    int sign_diff = ((x - y) >> 31) & 1;  // isolate sign of difference
    return !((sign_x ^ sign_y) & (sign_x ^ sign_diff));  // signs match
}
```