Solution exercice 1.4

1) 31
2) 19
3) 16
4) 14
5) 2
6) 4
7) 144
8) 4
9) 23
10) -1 … si décalage arithmétique (cas de gcc), INT_MAX si décalage logique
11) -3 … si décalage arithmétique, INT_MAX – 2 si décalage logique (*)
12) 1 NB ~n = -n – 1, si n signé et ~n = <*valeur max du domaine*> – n, si n non signé
13) 6 … car, en vertu de la priorité des opérateurs, interprété comme 6 | (5 & 4)

(*) n >> m revient bien à faire une division par 2^m non-euclidienne si n unsigned (que >> soit arithmétique ou logique), ou si n signed et >> arithm, ou si n signed, n >= 0 et >> logique... mais pas si n signed, n < 0 et >> logique.

Solution exercice 1.6

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#define INT_SIZE (sizeof(int) * 8)

typedef unsigned short ushort;

int* setBit(ushort pos, ushort bitValue, int* n);

int main(void) {

   int n = 0;
   printf("n = %d\n", n);
   printf("n = %d\n", *setBit(0, 1, &n));
   printf("n = %d\n", *setBit(0, 0, &n));
   printf("n = %d\n", *setBit(1, 1, &n));

   for (ushort pos = 0; pos < INT_SIZE; ++pos)
      setBit(pos, 1, &n);
   printf("n = %d\n", n);

   int m = -1;
   printf("m = %d\n", *setBit(31, 0, &m));

   return EXIT_SUCCESS;
}

int* setBit(ushort pos, ushort bitValue, int* n) {
   assert(pos < INT_SIZE);
   assert(bitValue == 0 || bitValue == 1);
   { // Variante 1
      const int MASK_1 = 1 << pos,
                MASK_2 = bitValue << pos;
      *n = (*n & ~MASK_1) | MASK_2;
   }
//    { // Variante 2
//       const int MASK = 1 << pos;
//       *n ^= (-bitValue ^ *n) & MASK;
//    }
//    { // Variante 3
//       const int MASK = 1 << pos;
//       *n = bitValue ? *n | MASK : *n & ~MASK; // ... mais usage de ?:
//    }
   return n;
}

// n = 0
// n = 1
// n = 0
// n = 2
// n = -1
// m = 2147483647
```