Beware of bugs in the above code; I have only proved it correct, not tried it.
Donald E. Knuth

Learning Objectives

1. Using bitwise logical and shifting operations in a high-level language like C.

Work that needs to be handed in

1. modify `extractMessage.c` as described in Problem 1.
2. modify `countOnes.c` by implementing a function that counts the number of 1s in an unsigned integer without using loops or conditionals, as described in Problem 2.

Note: the `picture.bmp` *file (and this handout) can be retrieved from:*
https://subversion.ews.illinois.edu/svn/sp13-cs398/_shared/

Problems

1. **Steganography**

   Steganography[1] is the science of concealing a secret message within a "carrier" message in a way that makes it hard to discern the presence of the hidden information (for example by writing in "invisible ink"). Here we will look at a simple method of hiding a text message within an image, so that the image looks almost unchanged. Given an image (in the BMP format[2]) containing a secret message, your task is to write a C program to decipher the message.

   Each pixel in a BMP image is encoded with 24 bits as a triple (blue, green, red), where each component is an 8-bit quantity that can take values from 0 to 255. Thus, a red pixel is represented as (0, 0, 255) whereas a mixture of blue and green could be represented as (100, 200, 0). We've given you functions to manipulate BMP files (see `bmp.h` and `bmp.c`), as well as an example program (`sample.c`) illustrating how to use these functions.

   Messages are encoded in BMP files according to the following scheme:

   (a) First, each letter in the message is encoded as a `char` according to its ASCII code[3]. The message is terminated by a `char` with value 0. Thus, the message can be viewed as a sequence of bits.

   (b) Next, the message bits are encoded across several adjacent pixels, beginning with pixel (0, 0), followed by pixel (1, 0), (2, 0), . . . etc. If the image is k pixels wide, then pixel ($k-1$,0) is followed by pixel (0,1),(1,1),(2,1),...

   (c) Message bits are stored in the *least* significant bit (LSB) of the `green` components of pixels. For example, a message beginning with the letter H (ASCII code 0x48, i.e. 01001000 in binary) is encoded as follows:

   - pixel (0,0): LSB(`green`) = 0
   - pixel (1,0): LSB(`green`) = 1
   - pixel (2,0): LSB(`green`) = 0
   - pixel (3,0): LSB(`green`) = 0
   - pixel (4,0): LSB(`green`) = 1, . . .

   The sample BMP file `picture.bmp` contains the following message encoded according to the above scheme: `Hello, bitworld!` The sample BMP file can be found in the `_shared` directory on the SVN server.

   What you need to do:

   In the `extractMessage.c` file, there is a function that accepts two parameters (the name of a BMP file and a message buffer). Re-write this function to decode the hidden message encoded in the BMP file according to the above scheme and place this message into the provided buffer (**including the terminating NULL char**) . To do this, your code must use bitwise operations in C.

[1] Steganography: http://en.wikipedia.org/wiki/Steganography
[2] BMP file format: http://astronomy.swin.edu.au/~pbourke/dataformats/bmp/
[3] ASCII codes: http://www.asciitable.com

2. **Count Ones**

In this problem, you will write an efficient C function to count the number of bits that are equal to one in an unsigned integer. Your function must be of the following form:

```
unsigned int countOnes(unsigned int input)
```

Here is a simple (but *inefficient*) way to solve the problem:

```
unsigned int countOnesDumb(unsigned int input) {
  int i, count;
  count = 0;
  for(i = 0; i < 8 * sizeof(unsigned int); i++) {
    if ((input & 1) != 0) {
        count++;
    }
    input = input >> 1;
  }
  return count;
}
```

This is inefficient, because it takes $O(n)$ operations[4] on n-bit numbers to count the number of ones in an n-bit integer. We want you to write a function that performs only $O(log2n)$ operations on n-bit numbers for this task. Here is a description of this efficient algorithm for 32-bit integers (you can assume that integers are 32 bits long):

Algorithm:

(a) Treat each integer as 32 1-bit counters. Pair adjacent counters off, and add each pair. The result will be 16 2-bit counters.

(b) Take these 16 2-bit counters and pair adjacent counters off. Add the pairs of counters to produce 8 4-bit counters.

(c) Keep pairing and adding in this fashion until you get a single 32-bit counter, which will contain the result.

Is this really efficient? Think about step 1 of this algorithm. For each pair of bits, we need to perform one addition operation - this means $n/2$ additions for an n-bit number. For step 2 we need to perform one addition operation for each of the 16 2-bit counters - this means $n/4$ additions are required for this step. We can continue this trend until we are left with a single 32-bit counter. After all steps are complete we are left an algorithm that requires $O(n)$ operations!

The trick is to achieve $O(log2n)$ is to do all these additions together (in parallel). In what follows, we'll show you how to do this with an example. To keep things simple, we'll work with 8-bit integers.

Doing things efficiently:

Suppose the input integer is 11001101. We treat this as eight 1-bit counters. First, lets isolate every alternate counter in this integer. We can do this by AND-ing the number with the bit-pattern 01010101 (i.e. 0x55)

[4] Recall from CS 125/173/225: $O(n)$ means at most $c \cdot n$, for some constant $c > 0$

```
     1 1 0 0 1 1 0 1
AND  0 1 0 1 0 1 0 1
--------------------
     0 1 0 0 0 1 0 1    odd counters
```

Next we get the other counters, this time by AND-ing with the bit-pattern 10101010 (i.e. 0xAA)

```
     1 1 0 0 1 1 0 1
AND  1 0 1 0 1 0 1 0
--------------------
     1 0 0 0 1 0 0 0    even counters
```

Now we need to pair-up adjacent counters and add them. We can do this by first RIGHT-SHIFTING the even counters by 1 to align them with the odd counters, and then simply adding the two numbers:

```
     0 1 0 0 0 1 0 0    even counters RIGHT-SHIFTED by 1
  +  0 1 0 0 0 1 0 1    odd counters
--------------------
     1 0 0 0 1 0 0 1
```

This completes step 1 of the algorithm. It takes 4 operations (two ANDs, one RIGHT-SHIFT and one + operation) in the case of 8-bit integers, and will similarly take 4 operations in the case of 32-bit integers.

Moving on to step 2, we need to treat the number 10001001 as four two-bit counters as follows:

```
   1 0 0 0 1 0 0 1
   \ / \ / \ / \ /
 =  2   0   2   1
```

We will need an appropriate bit-pattern to pick out alternate counters like before:

```
0 0 0 0 0 0 0 1 odd counters
```

and

```
1 0 0 0 1 0 0 0 even counters
```

Again, we align the counters (this time by right-shifting the even counters by 2) and add them:

```
     0 0 0 0 0 0 0 1
  +  0 0 1 0 0 0 1 0
--------------------
     0 0 1 0 0 0 1 1
  =  \     / \     /
       2       3
```

Finally, we treat 00100011 as two 4-bit counters (as shown above) and add them to get the final answer: 5.