Bitwise Operators

We commonly use the binary operators && and ||, which take the logical and and logical or of two boolean expressions.

Since boolean logic can work with single bits, C provides operators that work on individual bits within a variable.

As we learned earlier in the semester, if we store an int in binary with the value 47, its last eight binary bits are as follows:

00101111

Similarly, 72 in binary is

01001000.

Bitwise operators would take each corresponding bit in the two input numbers and calculate the output of the same operation on each set of bits.

For example, a bitwise AND is represented with a single ampersand sign: &. This operation is carried out by taking the and of two bits. If both bits are one, the answer is one. Otherwise, the answer is zero.

Here is the bitwise and operation on 47 and 72:

```
  0 0 1 0 1 1 1 1
& 0 1 0 0 1 0 0 0
---------------------
  0 0 0 0 1 0 0 0 (which has a value of 8.)
```

Thus, the following code segment has the output 8:

```
int x = 47, y = 72;
int z = x & y;
printf("%d", z);
```

Here is a chart of the other bitwise operators:

Function	Operator	Meaning
and	&	1&1 = 1, rest = 0
or	|	0 | 0 = 0, rest = 1
xor	^	1 ^ 0 = 0 ^ 1 = 1 0 ^ 0 = 1 ^ 1 = 0
Not (unary operator)	~	~0 = 1, ~1 = 0

Now, let's calculate the other bitwise operations between 47 and 72:

```
  0 0 1 0 1 1 1 1
| 0 1 0 0 1 0 0 0
---------------------
  0 1 1 0 1 1 1 1 (which has a value of 111.)

  0 0 1 0 1 1 1 1
^ 0 1 0 0 1 0 0 0
---------------------
  0 1 1 0 0 1 1 1 (which has a value of 103.)
```

Two's Complement

In order to understand exactly how the bitwise not operator works on integers, we must understand exactly how a signed integer is stored inside the computer. Regular binary notation, which we learned earlier, is used to store unsigned integers (these don't allow for negative values and are used less frequently than regular ints).

Two's Complement is used to store regular ints. The storage scheme is almost identical to regular binary, except for the meaning of the most significant bit. An int is stored using 32 bits. For an unsigned number, each of these bits are place-holders with value 2^{31}, 2^{30}, 2^{29}, …, 2^2, 2^1, and 2^0. Essentially, if the bit at location i is 1, then we contribute 2^i to the value of the number. (If it's 0, nothing changes.) In two's complement, we only change the most significant bit to mean -2^{31}.

For example, if all of the bits were 1 in a regular int:

11111111 11111111 11111111 11111111

Then, the value of this number would be:

$$-2^{31} + 2^{30} + 2^{29} + 2^{28} + \ldots + 2^2 + 2^1 + 2^0 = -1.$$

From here, we can get to other negative values by turning some of the bits off. For example, it's fairly easy to see that -5 would be represented as follows:

11111111 11111111 11111111 11111011

Taking into account two's complement, we can calculate the effect of the bitwise not operator.

Consider calculating ~x, where x = 47, from our previous example.

Here is all 32 bits of x:

00000000 00000000 00000000 00101111

Flipping each bit, we get:

11111111 11111111 11111111 11010000

Using a bit of logic, we can see that if we were to have all 1's the value would be -1. But, now that we've turned off the bits that add up to 47, our new value will be -48.

In essence, we see that in the typical case, when x is positive, ~x is equal to –x-1.

Hopefully you can see that if x is negative, then ~x is ALSO –x-1. (Thus, ~~x always equals x, as you might imagine.)

Left and Right Shift Operators

The left-shift operator is <<.
The right-shift operator is >>.

When we left-shift a value, we must specify how many bits to left-shift it. What a left-shift does is move each bit in the number to the left a certain number of places. In essence, so long as there is no overflow, a left-shift of one bit multiplies a number by two (since each bit will be worth twice as much).

It follows that a left shift of 2 bits multiplies a number by 4 and a left shift of 3 bits multiples a number by 8. In general, a left-shift of k bits multiplies a number by 2^k.

Using bitwise operators to iterate through subsets

Imagine solving the following problem with brute force:

Given an array of values, such as {9, 3, 4, 5, 12}, does there exist a subset of values in the array that adds up to a target, say 22?

Our goal would be simply to try EACH possible subset of the array, add the values and see if we get the target. If we think about binary and look at the binary values from 0 to 31, we have:

00000	**01000**	**10000**	**11000**
00001	**01001**	**10001**	**11001**
00010	**01010**	**10010**	**11010**
00011	**01011**	**10011**	**11011**
00100	**01100**	**10100**	**11100**
00101	**01101**	**10101**	**11101**
00110	**01110**	**10110**	**11110**
00111	**01111**	**10111**	**11111**

If we assume that 0 means, "don't put this number in the set" and 1 means, "put this number in the set, then these 32 listings represent all possible subsets of a set of 5 values.

Thus, our idea is as follows:

Loop from 0 to 31, for each value and calculate the sum of the corresponding subset. For example, since 13 is 01101, this means that the subset we want to add up is array[3], array[2] and array[0]. We are using the most significant bit in the number to correspond to the last array slot and the least significant bit in the number to correspond to index 0 in the array. In this example, when we are considering 13, the values we add are 9, 4 and 5 to obtain 18.

Here is the code that does this:

```
int i, j;
   int n = 5;
   int array[] = {9, 3, 4, 5, 12};
   int target = 22;

   for (i=0; i < (1 << n); i++) {

      int sum = 0;
      for (j=0; j < n; j++)
         if ( (i & (1 << j)) != 0 )
            sum += array[j];

      if (sum == target)
         printf("Can add up to the target!\n");
   }
```

Notes:

1) Remember that a left-shift of n bits multiplies by 2^n, so the value of 1 << n for this example is $2^5 = 32$, as desired.

2) The j loop is going through each array element, trying to decide whether or not to add it. The value 1 << j has only one bit set to 1, it's the bit at location j.

3) If we do a bitwise and with a number of the form 000…001000…, then our answer will either be all 0s OR it will be the number itself. Basically, all of the 0s cancel out the other 31 bits. The one 1 isolates that particular bit, which is exactly what we want.

Knapsack Problem and Bitwise Operators

In general, bitwise operators are good with dealing with subsets. The knapsack problem is a more general version of the subset sum problem. In this problem, you are given items which have both weights and values, as well as a target total weight. The goal is to find the subset of values that, in total, weighs the target total weight (or in some cases is less than or equal to the target), that has maximal value. Thus, we can adjust the solution above somewhat. Instead of returning whether or not we can achieve a particular target, we could simply return the maximal value achieved of any subset equal (or less than or equal to that target). This is exactly what is done in the posted problem, Knapsack. Here is heart of that code:

```
int n, i, maxWeight;
int weights[MAXSIZE];
int values[MAXSIZE];

scanf("%d%d", &n, &maxWeight);
for (i=0; i<n; i++) scanf("%d%d", &weights[i], &values[i]);

int bestValue = 0, bestSubset = 0, subset;

for (subset=1; subset<(1<<n); subset++) {

    int thisValue = 0, thisWeight = 0;

    for (i=0; i<n; i++) {
        if ((subset & (1<<i)) != 0) {
            thisValue += values[i];
            thisWeight += weights[i];
        }
    }

    if (thisWeight > maxWeight) continue;

    if (thisValue > bestValue) {
        bestValue = thisValue;
        bestSubset = subset;
    }
} // end for subset
```

Correct Answer Recovery Problem and Bitwise Operators

This was a neat problem I made up for a high school programming contest. In the problem you are given the test results of a T/F exam of your students. (Namely, just a frequency chart of how many students got each number of problems correct.) In addition, you know what each student answered on each question. The only issue is that you lost the correct answers for the exams and don't know which students got which grades. Based on the data you have, you want to calculate the number of different correct test answers that are possible.

The way to solve this problem is to try each possible answer key. For each potential answer key, grade each student's exam. Then, check if this set of frequencies matches the overall results of the students. If so, we count that potential answer key, otherwise we don't.

The cool thing about this problem is grading the exam. We can store the student's answers (the exam had 15 or fewer questions) as a single integer. So, for example, the integer

011101010101111

Represents an answer of true on questions 0-3, 5, 7, 9, and 11-13 and an answer of false on questions 4, 6, 8, 10 and 14. (Recall that bit 0 is all the way over to the right.)

Now, imagine if the answer key was:

110110110001101

To do the grading, take the bitwise xor of these two:

011101010101111
110110110001101 ⊕

101011100100010 (1 is an incorrect response!)
So, all we have to do is count the number of bits equal to 1 in the binary representation of the xor of the two values and that equals the total number of questions the student got wrong. Finally subtract this from the total number of questions and we get how many questions the student got correct!!!

Here is the heart of the code to the solution to that problem:

```
for (i=0; i<(1<<numQuestions); i++) {

     int thisFreq[MAXQUESTIONS+1];
     for (j=0; j<=MAXQUESTIONS; j++) thisFreq[j] = 0;

     for (j=0; j<numStudents; j++)
         thisFreq[numQuestions-numBitsOn(i^answers[j])]++;

     res += possible(thisFreq, freq, numQuestions+1);
}

int numBitsOn(int n) {

    int res = 0;

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

int possible(int* freq1, int* freq2, int length) {

    int i;
    for (i=0; i<length; i++)
        if (freq1[i] != freq2[i])
            return 0;

    return 1;
}
```

Babysitter Problem and Bitwise Operators

In this problem you are given several possible jobs to take. Each job occurs on some set of days and has some payment for the full job. You would like to choose a set of jobs such that they do NOT conflict (no two jobs scheduled for the same day) that maximizes your payment.

What we'll do for this problem is a bit of overkill, but it will calculate every subset of jobs we can do and the payment for each of those subsets.

The key idea is as follows:

Let a bitmask stand for a subset of jobs we've done. So, something like 0001011, represents that we've done jobs 0, 1 and 3, the bits that are on, in this number. These jobs must correspond to a non-overlapping set of days, let's say this is 111000111, so between the three jobs, you would have to work on days 0, 1, 2, 6, 7 and 8. Let's assume the payment for this subset of jobs is $500.

Now, consider job 4. Bit 4 is off in 0001011, so we could maybe add it to this subset. Let's say this job is on days 3 and 5 and pays $300. The bitmask of days for this job is 000101000. To see if we can add this job or not, we would take the bitwise and of this number and our current set of days we're working:

111000111 & 000101000 = 000000000

This means, if we were to add this job, there would be no day we would be working two jobs!!!

If we were to add this job, then our new set of days we are working is just the bitwise or of the two:

111000111 | 000101000 = 111101111

And the total amount of money we would make is $500 + $300. Finally the corresponding subset of jobs we would be working is:

0011011, since we added job 4 to our old set of jobs 0, 1 and 3.

In the posted code, the bitmask of the subset of jobs is used as an array index, and both the corresponding amount for that subset of jobs and the bitmask for the days worked are stored in two different arrays: money and dayList, respectively.