Isaac Denny

CSC 2410

9 September 2024

# Lab 3 Review

End of Lab Questions:

1. The difference between & and && is how they are used: the bitwise AND operator operates on primitive values on the bit level, while the conditional && operator evaluates either to TRUE or FALSE (1 or 0 respectively) based on the condition.
2. 5 | 3 = 7. On the bit level, that is 0101 | 0011 = 0111. To calculate this, we look at each bit one at a time. The 0th bit (indexed 0 from the right) of 5 is 1, and from 3 is 1, so 1 or 1 evaluates to 1. Likewise the 1st bit from 5 is 0, and from 3 is 1, so 0 or 1 is 1. The same occurs for the final bit and evaluates to 0111 which is 7 in base 10
3. The bitwise NOT on 00001111 evaluates to 11110000 by simply flipping the bits
4. XOR with any number and itself is 0. A ^ A = 0, that is 1010 ^ 1010 = 0 for each bit
5. With any string of 8 bits you can clear the 3rd bit performing a bitwise AND on it with a bitmask with the 3rd bit as a 0, and every other bit as a 1. You can derive this mask with ~8 (~1000).
6. The bitwise OR operation is helpful in this case because it leaves the other values alone if you use a mask with 0's in place of the bits you do not want to change. Any bits that are 1 in that mask will always evaluate to 1, because anything or 1 is always 1

7. A bitmask is a string of bits that can be used in a bitwise operation to change the effect of the operation. More specifically, it can define which bits need to be affected and which bits need to be ignored in the operation.

Review:

For this semester, I have really tried to spend time thinking through the operations and examples before beginning to code. I have left most of my thought process in a block comment at the beginning of each function. This helps me a lot when it comes time to code because my thoughts are in order and I have structured my response to be aware of any edge cases or bumps I can encounter while coding (it also prompts optimization for larger programs). This is, of course, a lot more useful for larger algorithms, but still helped a bit when I got to the circular bit shifting.

I was able to quickly move through each of the functions without many bumps until I reached the circular bit shifting operations. I have seen problems like this before– "Rotate Array" on leetcode– but never for bits, and I was unsure of the best way to proceed. I originally tried to derive a bitmask to find the overflowed bits and place them on the other side of the integer, but it became obvious that would not work, because the overflowed bits are not stored on the integer, and it would still require me to move the bits to the beginning of the string. I figured using a long to store the overflowed int would be out of scope for the problem, so that led me to think about how I could get the left-most bits to the other side in order and without overflowing them. I then realized I could find the distance they needed to be shifted by subtracting the shift amount from the length of the bitstring (32 for ints). After I had this result, I could bitwise OR with the normally-shifted result and get the final shifted bitstring.