CS 50: Introduction to Computer Science
Harvard University
Scribe Notes
Tova Wiener

Announcements: 00:00 – 6:00

Welcome to Week 8 – there was no class on Monday.
Final project specs are online.
If you did not get quiz 1 back, see your TF.

Revealing efforts of a section that got into the photo finding game in Problem Set 4.
Looking at pictures from that section.
Chris's section won. Two other students recognized for extensive picture taking.

What's coming up 6:00 – 7:00

This week's goals are twofold: To introduce you to compression (focus of PS6) and trees and linked lists. This will bring us to a nice end for C. Friday we will look under the hod just to make you more familiar with what you are doing. Monday will be about Secure Coding.

Huffman Coding Theory: 7:00 – 33:00

What is the motivation for something like Morse code? Similar to what we try to do in representing characters by computers – ASCII. The broblem with ASCII is perhaps that you are wasting bits. Each character is 8 bits. 'A' is 65 bits total, we only needed 7 bits total. And, if most of our messages do not need weird punctuation, we could get away with fewer bits. If we only needed 27 characters, like in speller, we would only need 5 bits total for each. But there are certainly letters of the alphabet that are more common than others. It seems foolish for the most common letters we take 5, or 8, bits, but we also spend that much for less common letters. Intuitively, it would be nice to use the fewest number of bits for the common letters. Morse code works like that also. To send an 'e' across the wire takes only a small amount of effort. It is the shortest of all possible codes. 'i' is pretty common, and thus it too is short. However, the problem is that there could be ambiguity. 'e' 't' looks like an 'a'. That is unfortunate because the goal was to go first. The way to tell the difference is to pause in between characters. But that gap itself is like sending a piece of information, and it slows down the whole process. Better would be a system that is immediately decodable, and you can just read it immediately.

Huffman Coding is a canonical example of exactly that. The goal is to compress information on disks, using as few bits as possible to represent each character. The most common characters will be encoded with the fewest bits. The bits stored on disk are no longer ASCII characters, but rather "Huffman" bits. A human can only look at the file after reversing the process.

Look at example with random selection of A-E. They each come up with different frequency. The goal is to figure out which bit sequences of 0s and 1s to represent in each character than ASCII gives.

Huffman coding says to make single node trees with relative frequencies in each root. So we now have five “roots”. The algorithm says to take the two smallest nodes, and connect them via a parent node, and add their frequency to the parent’s node. Arbitrarily but consistently, we are saying that the branch going to the left is a 0, and the right is a 1. We now apply this algorithm recursively. Ultimately, we are just adding the frequencies, so we should see a 1 at the top. Now, we now how to compress a value – follow the branches of the tree to any given letter. If you are optimizing the common case, and spending more in the uncommon case, that gives a mathematically better (smaller) result.

If we wanted to create a program to compress, it would need to read through the text and count the letters, storing the frequency in an array. It would then create a tree, and use that tree to assign values to each letter.

If the characters in the input text have the same frequency, and has all possible ASCII characters, then this does not help you. Huffman exploits the fact they do not all use the same number of symbols, and they do not use all 256 letters.

There is a catch. We need to save the tree itself along with the compressed files. If you just save the coding, you will not know how to decode. So it could be worse if you compress a very small file.

We encounter a problem when there are equal frequencies. However, because of the optimization, it will be equivalent. If you simply store the frequencies, and not the actual encodings, there needs to be a consistent way to know what branch to take.

We will leave it to you to implement in C.

Bitwise Operators: 33:00

There is a new set of syntax rules for bit comparison, addition and modification.

For example:
& is the bitwise “and” operator (as opposed to && for Booleans)

1 & 1 = 1
1 & 0 = 0
0 & 1 = 0
0 & 0 = 0

The same thing happens for “or”. These are called truth tables.

~ is 1’s complement. 0 becomes 1, and 1 becomes 0.

The ^ is the exclusive “or” symbol. The answer is true if only one of the two bits are a 1.

So:

0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0

What can we do with these tricks?
For example, we could use these to print out integers in binary. Say we had three in decimal. That is 11 in binary, with 29 0s to the left of it. How would our program print this out? We will print out the integer by iterating over the bits. So we create a for loop, which starts at:

int i = sizeof(int) * 8 – 1 (31)

We will start with a 1 in binary. The left shift operator (<) can move our one bit to the left ‘i’ times. So the first time, this will move this 31 times – all of the way to the left. We then “and” (&) this with our number, 3. We do this bit by bit. In this case, we have 10…..00 and 00…..11. These will have no overlap, so at every location we will get false, and so a 0 will be printed. We then move the 1 30 times to the left, and we have the same thing.
But, on the last iteration, we actually find some to match. In that case, a 1 would be printed. If at any point it finds true, that means a bit was set in the int passed in. This would print out the binary equivalent of our number.