

The UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

Comp 120 Computer Organization

Spring 2005

Problem Set #1

Issued Thursday, 1/20/05; Due Thursday, 1/27/05

Homework Information: Some of the problems are probably too long to be done the night before the due date, so plan accordingly. Late homework will not be accepted. Feel free to get help from others, but the work you hand in should be your own.

Problem 1. “Miss Information?”

In lecture we learned that information resolves uncertainty, and that information is measured in units of *bits*. In order to uniquely identify one of N equally likely alternatives, $\log_2 N$ bits of information must be communicated. This is equivalent to saying that $\log_2 N$ bits are needed to encode any particular choice. If we learn a fact that only narrows down a list of possible alternatives to a choice of M candidates ($M < N$), we say that we’ve received $\log_2(N/M)$ bits of information. For example, if we start with 4 equally likely alternatives, we’ll need 2 bits to provide a unique encoding for each of the 4 alternatives. If we learn a fact that narrows the number of alternatives down to 2, that fact has conveyed $\log_2(4/2) = 1$ bit of information.

- (A) You’re given a standard deck of 52 playing cards that you start to turn face up, card by card. How many bits of information do you get when you learn what the first card is? The fifth card? The last card?
- (B) After returning all 52 cards back to the deck and shuffling, you turn over the first card, you notice that it is red. How many bits of information in conveyed in finding out the card’s color? You then notice that the card is a face card. How many *additional* bits are conveyed by this second observation? Next you notice that the card is a king; how many bits of information are conveyed by this? How many remaining bits must be resolved to uniquely identify the card?

Next we consider the situation when the N alternatives are *not* equally likely. It still takes $\log_2 N$ bits to encode any particular choice, but with clever encoding the *average* number of bits needed to encode a choice might be smaller. For example, suppose there were three alternatives (“A”, “B”, and “C”) with the following probabilities of being chosen:

$$\begin{aligned} p(\text{“A”}) &= 0.8 \\ p(\text{“B”}) &= 0.1 \\ p(\text{“C”}) &= 0.1 \end{aligned}$$

We might encode the choice of “A” with the bit string “0”, the choice of “B” with the bit string “10” and the choice of “C” with the bit string “11”.

- (C) If we record the results of making a sequence of choices by concatenating in left-to-right order the bit strings that encode each choice, what sequence of choices is represented by the bit string “00101001100000”?

- (D) What is the expected length of the bit string that encodes the results of making 1000 choices? What is the length in the worst case? How do these numbers compare with $1000 \cdot \log_2(3/1)$?

Intuitively it seems that we get less information when we learn of a likely choice and more information when we learn of an unlikely choice. This intuition is reflected in the following formula for the average number of bits of information (Entropy) received about a choice made from N alternatives when each alternative has a (possibly different) probability p_i :

$$\text{Bits of information} = - \sum_{i=1}^N p_i \log_2(p_i)$$

- (E) How much information do you receive, on average, from a single flip of a crooked coin where $p_{tail} = 0.3$ and $p_{head} = 0.7$? Is this more or less than the amount of information you receive from the flip of a fair coin?
- (F) File lengths on a computer are often reported in terms of *bytes* where each byte contains 8 bits. What bounds can you place on the amount of information in a file that is 12183 bytes long? If you learn that the file is a departmental telephone directory, does that affect your estimate on the amount of information in the file? Briefly explain.
- (G) File compression programs are supposed to be *lossless*, i.e., they should preserve all the information in your file. A popular compression program can compress the file in part (E) to a file of length 3412. Is this consistent with your answer to part (E)? Explain.

Problem 2. Modular Arithmetic and 2's Complement Representation

Most computers choose a particular *word length* (measured in bits) for representing integers and provide hardware that performs operations on word-size operands. Many current generation processors have word lengths of 32 bits. Restricting the size of the operands and the result to a single word means that the arithmetic operations are actually performing arithmetic modulo 2^{32} .

- (A) How many different values can be encoded in a 32-bit word?

Almost all modern computers use a 2's complement representation for integers since the 2's complement addition operation is the same for both positive and negative numbers. In 2's complement notation, one negates a number by complementing each bit in its representation (i.e., changing 0's to 1's and vice versa) and adding 1. By convention, we write 2's complement integers with the most-significant bit (MSB) on the left and the least-significant bit (LSB) on the right. Also by convention, if the MSB is 1, the number is negative; otherwise it's non-negative.

- (B) Please use a 32-bit 2's complement representation to answer the following questions. What's the representation for 0? For the most positive integer that can be represented? For the most negative integer that can be represented? What are the decimal values for the most positive and most negative integers? What do you get if you negate the largest negative integer (given both the binary and decimal values)?

(C) Since writing a string of 32 bits gets tedious, it's often convenient to use hexadecimal notation where a single digit in the range 0—9 or A—F is used to represent adjacent groups of 4 bits (starting from the left). Give the corresponding 8-digit hexadecimal encoding for each of the following numbers:

(C.1) 37_{10}

(C.2) -32768_{10}

(C.3) $11011110101011011011111011101111_2$

(C.4) $10101011101011011100101011111110_2$

(C.5) -1_{10}

(D) Calculate the following using 6-bit 2's complement arithmetic (which is just a fancy way of saying to do ordinary addition in base 2 keeping only 6 bits of your answer). Remember that subtraction can be performed by negating the second operand and then adding it to the first operand.

(D.1) $13 + 10$

(D.2) $18 - 15$

(D.3) $15 - 18$

(D.4) $27 - 6$

(D.5) $-6 - 15$

(D.6) $21 + (-21)$

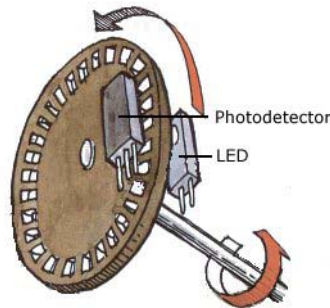
(D.7) $31 + 12$

Explain what happened in the last addition and in what sense your answer is “right”.

(E) At first blush “Complement and add 1” doesn't seem to be an obvious way to negate a number. Give a brief explanation of why it is “obvious” if you think about it for a minute. Hint: recall that a number and its negative have the following relationship, $0 = A + (-A)$, and think about what that implies about the 2's complement representation for $-A$.

Problem 3. Rotary Shaft Encoders

Rotary shaft encoders are used to measure the position and/or movement of a shaft or axle. For example there are two such encoders inside the standard mouse: one for “X” motion and another for “Y” motion. A low-cost implementation involves a slotted opaque disk, a light-emitting diode (LED) and a photodetector:



As the shaft turns, the disk rotates and the photodetector outputs a stream of pulses as the light from the LED is alternately blocked by the disk or allowed to pass by a slot.

- (A) The simple implementation above detects motion but one cannot tell in which direction the shaft is rotating. Adding another set of slots and a second LED/photodetector pair can solve this problem. Design a pattern for the second set of slots that allows the direction of rotation to be determined. Draw a diagram showing how both sets of slots are arranged and the pulse streams produced by the photodetectors as the disk rotates clockwise and counterclockwise. Hint: arrange the slots so that the pulse streams form a two-bit counter that counts up in one direction and down in the other.
- (B) The absolute position of the shaft can be established by adding additional sets of slots, LEDs and photodetectors until the desired resolution is achieved. For example, an arrangement of three slots will provide for eight possible shaft positions. Using the “natural” counting sequence

000, 001, 010, 011, 100, 101, 110, 111

to encode the position might seem obvious, but can lead to the following difficulty. As the shaft is rotating, signals from the photodetector array aren’t perfectly synchronized due to slight mechanical imperfections or differing electrical delays. This leads to momentary “glitches” in the readout; for example, as the shaft transitions between positions “001” and “010” there may be a period during which the readout appears as “000” or “011.”

Describe a different 3-bit encoding sequence that overcomes this difficulty.

- (C) Generalize your coding scheme from part (B) and explain how to generate an N-bit code that avoids “glitches” in the readout as the shaft rotates.

Problem 4. Error Detection and Correction

To protect stored or transmitted information one can add check bits to the data to facilitate error detection and correction. One scheme for detecting single-bit errors is to add a *parity* bit:

$$b_0 \ b_1 \ b_2 \ \dots \ b_{N-1} \ p$$

When using *even parity*, p is chosen so that the number of “1” bits in the protected field (including the p bit itself) is even; when using *odd parity*, p is chosen so that the number of “1” bits is odd. *In the remainder of this problem assume that even parity is used.*

To check parity-protected information to see if an error has occurred, simply compute the parity of information (including the parity bit) and see if the result is correct. For example, if even parity was used to compute the parity bit, you would check if the number of “1” bits was even.

- (A) If an error changes one of the bits in the parity-protected information (including the parity bit itself), the parity will be wrong, i.e., the number of “1” bits will be odd instead of even. Which of the following parity-protected bit strings has a detectable error?

- (1) 11101101111011011
- (2) 11011110101011011
- (3) 10111110111011110
- (4) 00000000000000000

Detecting errors is useful, but it would also be nice to correct them! To build an *error correcting code* (ECC) we’ll use additional check bits to help pinpoint where the error occurred. There are many such codes; a particularly simple one for detecting and correcting single-bit errors arranges the data into rows and columns and then adds (even) parity bits for each row and column. The following arrangement protects nine data bits:

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	p_{row0}
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	p_{row1}
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	p_{row2}
p_{col0}	p_{col1}	p_{col2}	

A single-bit error in one of the data bits ($b_{i,j}$) will generate two parity errors, one in row i and one in column j . A single-bit error in one of the parity bits will generate just a single parity error for the corresponding row or column. So after computing the parity of each row and column, if both a row *and* a column parity error are detected, inverting the listed value for the appropriate data bit will produce the corrected data. If only a single parity error is detected, the data is correct (the error was one of the parity bits).

- (B) Give the correct data for each of the following data blocks protected with the row/column ECC shown above.

- | | | | | | | | |
|-----|------|-----|------|-----|-----|-----|------|
| (1) | 1011 | (2) | 1100 | (3) | 000 | (4) | 0111 |
| | 0110 | | 0000 | | 111 | | 1001 |
| | 0011 | | 0101 | | 10 | | 0110 |
| | 011 | | 100 | | | | 100 |

- (C) The row/column ECC can also detect many double-bit errors (i.e., two of the data or check bits have been changed). Characterize the sort of double-bit errors the code does *not* detect.

The row/column ECC shown above requires a number of parity bits proportional to the square root of the number of data bits. The *Hamming single-error-correcting code* requires approximately $\log_2(N)$ check bits to correct single-bit errors. Start by renumbering the data bits with indices that *aren't* powers of two:

Indices for 16 data bits = 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21

The idea is to compute the check bits choosing subsets of the data in such a way that a single-bit error will produce a set of parity errors that uniquely indicate the index of the faulty bit:

p_0 = even parity for data bits 3, 5, 7, 9, 11, 13, 15, 17, 19, 21

p_1 = even parity for data bits 3, 6, 7, 10, 11, 14, 15, 18, 19

p_2 = even parity for data bits 5, 6, 7, 12, 13, 14, 15, 20, 21

p_3 = even parity for data bits 9, 10, 11, 12, 13, 14, 15

p_4 = even parity for data bits 17, 18, 19, 20, 21

Note that each data bit appears in at least two of the parity calculations, so a single-bit error in a data bit will produce at least two parity errors. When checking a protected data field, if the number of parity errors is zero or one, the data bits are okay (exactly one parity error indicates that one of the parity bits was corrupted). If two or more parity errors are detected then the errors identify exactly which bit was corrupted.

- (D) If the parity calculations involving p_0 , p_2 and p_3 fail, assuming a single-bit error what is the index of the faulty data bit? Let e_0, e_1, \dots, e_4 indicate the presence (or lack of) of an error in p_0, p_1, \dots, p_4 ($e_i = 1$ if there's a parity error in the subset protected by p_i , 0 otherwise). What is the binary representation, in terms of e_0, e_1, \dots, e_4 , of the index of the faulty bit?
- (E) What is the relationship between the index of a particular data bit and the check subsets in which it appears? Hint: consider the binary representation of the index.
- (F) As with the row/column ECC, the Hamming SECC doesn't detect all double-bit errors. Characterize the types of double-bit errors that will not be detected. Suggest a simple addition to the Hamming SECC that allows detection of all double-bit errors.

Problem 5. Huffman Coding

On an architectural dig in somewhere between Raleigh and Chapel Hill, researchers have discovered a deck of fossilized punch cards containing the source for a circa 1960 program called Retspan. The program was invented to promote sharing of the digitized Elvis tunes enjoyed by the genetic ancestors of modern programmers. Retspan was designed to run on the then-current *decimal* computers, whose unit of storage is the decimal digit rather than the bit. Each tune is encoded as a stream of equally-probable single digits (0 through 9) and transmitted to the receiver, where the stream is decoded to reproduce the scratchy audio signal.

(A) How much information, in bits, does each transmitted digit convey?

Looking closely at the source code, you notice that each digit d read by the receiver is processed by first computing the value $f(d)$, as follows:

$f(d)$ = the largest prime factor of $d+1$, i.e.
If $d+1 = 1$ then 1
Else if $d+1 = 2, 4, \text{ or } 8$ then 2
Else if $d+1 = 3, 6, \text{ or } 9$ then 3
Else if $d+1 = 5 \text{ or } 10$ then 5
Else 7.

All further processing reflects only the value $f(d)$; thus information lost during the conversion from d to $f(d)$ is not used to reproduce the tune. Having attended the first two Comp 120 lectures, you recognize that some of the information transmitted between Retspan users is unnecessary. For example, the digits 1 and 3 are transmitted as distinct symbols, despite the fact that their distinction is lost by the above conversion and hence not needed.

You begin thinking about writing an improved version of Retspan that works by transmitting only $f(d)$ rather than each digit d . It occurs to you that this approach both reduces the alphabet of transmitted symbols from 10 to 5, and introduces asymmetries in the probabilities of each symbol transmitted (i.e., they are not equally likely).

(B) What is the amount of information conveyed about $f(d)$ by a transmitted “0”? By a transmitted “1”?

(C) What is the *average* amount of information conveyed about $f(d)$ by each transmitted digit?

You recall a brief mention of Huffman coding in lecture, and suspect that this coding technique could improve the efficiency of your improved Retspan system by eliminating some of the redundancy. Intrigued by your suspicion that this technique could be applied to Retspan transmissions, you search the web for information on Huffman coding. Huffman’s insight was that a variable-length binary decoding tree can be constructed by a simple greedy algorithm as follows:

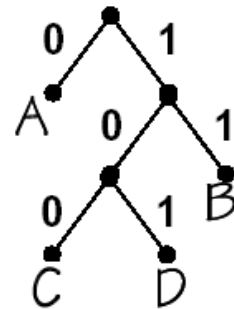
1. Begin with the set S of symbols to be encoded as binary strings, together with the probability $P(x)$ for each symbol x . The probabilities sum to 1, and measure the frequencies with which each symbol appears in the input stream. In the example from lecture, the initial set S contains the 11 symbols and associated probabilities.

2. Repeat the following steps until there is only 1 symbol left in S:
 - a. Choose the two members of S having *lowest* probabilities. Choose arbitrarily to resolve ties. In the example from lecture, '2' and '12' might be the first nodes chosen.
 - b. Remove the selected symbols from S, and create a new node of the decoding tree whose children (sub-nodes) are the symbols you've removed. Label the left branch with a "0", and the right branch with a "1". In the first iteration of the lecture example, the bottom-most internal node (leading to '2' and '12') would be created.
 - c. Add to S a new symbol (e.g., "2-or-12" in our example) that represents this new node. Assign this new symbol a probability equal to the sum of the probabilities of the two nodes it replaces.

When S contains a single symbol, the decoding tree is complete. It contains the essential information necessary to specify how each of the original symbols is to be represented as a binary string.

- (D) How many iterations of step 2 will it take to generate a decoding tree for a set of n symbols? [*HINT*: This is easy!].
- (E) Create a Huffman decoding tree for the efficient coding of $f(d)$ as variable-length binary strings. Present your results as in the example below, i.e. as a table listing probabilities and binary encodings of the five transmitted $f(d)$ symbols (1, 2, 3, 5, and 7) as well as a binary decoding tree.

<i>choice_i</i>	<i>p_i</i>	<i>encoding</i>
" A "	1/2	0
" B "	1/6	11
" C "	1/6	100
" D "	1/6	101



Huffman Decoding Tree