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. “This is a Computer Science class?”

Computers are not the only systems that manipulate and store information. During the past 50 years the field of biology has experienced a revolution that was prompted by the discovery of the structure DNA molecule and its role in genetics. The information content of every biological system is encoded in its DNA sequence as a series of four nucleic acids— cytosine, guanine, adenine, and thymine, which I will abbreviate as C, G, A, and T. Genes are merely subsequences of DNA that encode assembly instructions for a particular protein. Proteins are the primary molecular machines that perform the tasks of life. Within genes, trios of nucleic acids, called codons, are used to encode one of 20 amino acids, which are the building blocks of proteins. This coding is described in the following table:

<table>
<thead>
<tr>
<th>1st base</th>
<th>2nd base</th>
<th>3rd base</th>
<th>4th base</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>TTT Phenylalanine</td>
<td>T C A G</td>
<td>TTT Phenylalanine</td>
</tr>
<tr>
<td>C</td>
<td>CTT Leucine</td>
<td>C C A G</td>
<td>CTT Leucine</td>
</tr>
<tr>
<td>A</td>
<td>ATT Isoleucine</td>
<td>A C T G</td>
<td>ATT Isoleucine</td>
</tr>
<tr>
<td>G</td>
<td>GTT Valine</td>
<td>G G A C</td>
<td>GTT Valine</td>
</tr>
</tbody>
</table>

Notice that every protein is initiated with the same amino acid (Methionine) and that there are three codes reserved for terminating the protein sequence.
(A) Assuming that all nucleic acids are equally likely, how many bits of information are represented in a 3-nucleic acid sequence? What are the minimal number of bits necessary to encode one of 20 possible amino acids and a stop code? Do these two numbers agree? If not, speculate on a reason why they might differ.

(B) Assume that all codons occur with equal frequency. If you are told that a given codon represents some amino acid, how many bits of information are conveyed? Likewise, how many bits are conveyed if you find out that some codon encodes the stop message? How many bits are conveyed by discovering that a codon encodes Serine? (Look over the table entries carefully!)

(C) Suppose you are told that a given unknown codon with at least one T nucleotide, how many bits of information were you given? Suppose you find out that the same unknown codon encodes a stop code. How many additional bits does this new information add?

All 20 amino acids share a common chemical substructure, but are distinguished by a unique side chain. Amino acids are usually classified by the properties of their side chains. For example the side chains of Lysine, Histidine, and Arginine are chemical bases.

(D) How many bits are conveyed by finding out that a given unknown amino acid is a base? How many bits are conveyed by finding out that a given codon represents an amino acid that has a basic side chain? In this second case, how many additional bits are conveyed by resolving that the given codon represents Lysine?

Mutations occur in DNA when one nucleic acid is randomly substituted for another. DNA substitution mutations are of two types. Transitions are interchanges of purines (A and G) or of pyrimdines (C and T), which involve nucleotides of similar shape. Transversions are interchanges between purines and pyrimdines, which involve exchanging dissimilar chemical structures. Although there are twice as many possible transversions, transition mutations occur at a significantly higher frequency than transversions. It is often the case that a transition mutation in the third position of a codon does not change the amino acid that is coded for.

(E) When told that a given transition mutation causes a modification in the resulting protein but does not change its overall chain length how many bits are conveyed by this information? (This one is a little tricky)

Next we consider case where codon alternatives are not equally likely. In human genes we observe that Glycine coding codons appear with the following frequencies:

\[
p(\text{"GGG"}) = 0.24 \\
p(\text{"GGA"}) = 0.14 \\
p(\text{"GGT"}) = 0.12 \\
p(\text{"GGC"}) = 0.50
\]

(F) Clearly in nature’s encoding scheme the prefix “GG” is used to encode for Glycine. How many bits of information are conveyed by the codon’s last nucleic acid?

(G) Suppose you are collecting information about every occurrence of a Glycine codon within the human genome, and you plan to keep track of each coding variation. There are four possibilities, which could be trivially encoded using 2 bits. However, what is the entropy
of this set of codes? On average, how many bits would be wasted by using the fixed
length 2-bit coding scheme?

You might instead encode the choice of “GGC” with the bit string “0”, the choice of “GGG”
with the bit string “10”, the choice of “GGA” with the bit string “110” and the choice of “GGT”
with the bit string “111”.

(H) If we record Glycine encoding codon sequences by concatenating these bit strings in left-
to-right order, what sequence of choices is represented by the bit string
“0011011101010”?

(I) 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 does this number compare with
1000*\log_2(4/1)?

**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?

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:

(D)

(C.1) \(411_{10}\)
(C.2) \(-65536_{10}\)
(C.3) \(101110101101100001010000010001_{2}\)
(C.4) \(1010101110101011110110101100111_{2}\)
(C.5) \(-1_{10}\)
(E) 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) $20 + 10$
(D.2) $19 - 15$
(D.3) $15 - 19$
(D.4) $27 - 9$
(D.5) $-9 - 17$
(D.6) $21 + (-21)$
(D.7) $31 + 12$

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

(F) At first blush “Complement and add 1” doesn’t seem like 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. 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:

\[ b_0 \ b_1 \ b_2 \ldots \ b_{n-1} \ p \]

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:
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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>0000</td>
<td>111</td>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>0101</td>
<td>10</td>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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 $p_0$, $p_1$, and $p_3$ fail, assuming a single-bit error what is the index of the faulty data bit? Let $e_0$, $e_1$, ..., $e_4$ indicate the presence (or lack of) of an error in $p_0$, $p_1$, ..., $p_4$ ($e_i = 1$ if there’s an parity error in the subset protected by $p_i$, 0 if not).
otherwise). What is the binary representation, in terms of \( e_0, e_1, \ldots e_4 \), 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 4. 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 1950’s program called Senuti. The program was invented to promote sharing of the Elvis recordings enjoyed by the genetic ancestors of modern programmers. Senuti 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 read by the receiver is processed by first computing the value \( f(d) \), as follows:

```python
def f(digit):
    if (digit > 5) then
        return 8
    else if (digit > 2 )
        return 4
    else if  (digit > 0)
        return 2
    else
        return 1
```

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 three Comp 411 lectures, you recognize that some of the information transmitted between Senuti users is unnecessary. For example, the digits 6 and 9 are transmitted as distinct symbols, despite the fact that their distinction is lost by the given conversion and hence not needed.

You begin thinking about writing an improved version of Senuti 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 4, 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 \( d \) when \( f(d) \) outputs a “1”? When it outputs an “8”?

Comp 411 - Fall 2007 - 6 - Problem Set #1
(C) What is the *average* amount of information conveyed about d with each output of f(d)?

You recall a brief mention of Huffman coding in lecture, and suspect that this coding technique could improve the efficiency of your improved Senuti system by eliminating some of the redundancy. Intrigued by your suspicion that this technique could be applied to Senuti 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 four symbols and associated probabilities in the above table.

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, C and D 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”.
   c. Add to S a new symbol 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 above, i.e. as a table listing probabilities and binary encodings of the four transmitted f(d) symbols (1, 2, 4, and 8) as well as a binary decoding tree.