Introduction to Computer Science – Honors I

CS 181 – Fall 2013

Assignment 2 (due 9/25)

5 pages

Complete the following subjects and submit your answers/code electronically through moodle. Place all your files in a directory named your surname, underscore, and your initial, all in letter case. For example, if your name is John Smith, place all your files in smith_j/. Each programming assignment subject should be placed in its own directory immediately under this directory. For example, subject Hamming Code should be in smith_j/hammingcode/. Answers to non-programming assignments should be placed in file answers.pdf (or answers.txt or answers.docx) under the top-level directory (ex: smith_j/). Archive the top-level directory with zip and submit as file A2_surname_initial.zip (ex: A2_smith_j.zip). DO NOT submit any .class files.

Under the Stevens Honor System, you are required to sign the Honor pledge in all material that you hand in for grading. The Stevens Honor pledge is shown below:

"I pledge my honor that I have abided by the Stevens Honor System."

Please copy the pledge to the top of your answer sheet and type your name. Programming assignments should also include the above in a comment block at the top of your source file(s).

```
/*
 * QuickSort.java
 *
 * "I pledge my honor that I have abided by the Stevens Honor System."
 *
 * John Smith (c) 2013
 *
 */
```

Maximum points = 70.

Subjects:

1. Hamming code (place under directory *hammingcode*, 50 points)

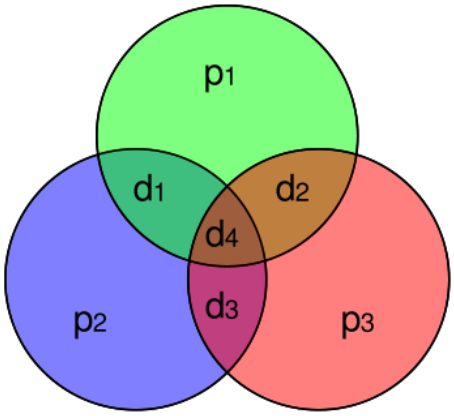
A parity bit is an extra 0 or 1 attached to a byte to help detect if an error has occurred. With even parity, each byte together with its parity bit will contain an even number of 1s. If the byte itself contains an odd number of 1s, then the parity bit is set to 1, to make the total number even. Otherwise the parity bit is set to 0. Obviously, if any one of the bits gets flipped, the number of 1s will be odd and we can determine that the byte contains an error.

When two bits in a byte flip, either from 0 to 1 or from 1 to 0, the parity check fails to detect the error. Also parity check can detect the flip of a single bit, but it cannot determine which bit is incorrect. Hamming codes are improvement over single bit parity check as they can not only detect but also correct single bit error. Hamming codes can also detect (but cannot correct) two-bit error.

Encode:

Hamming(7,4) encodes 4 bits of data into 7 bits by adding 3 parity bits. Hamming codes use even parity. The rules of Hamming(7,4) is shown in the following table:

Bit position		**1**	**2**	**3**	**4**	**5**	**6**	**7**
Encoded data bits		**p1**	**p2**	**d1**	**p3**	**d2**	**d3**	**d4**
Parity bit coverage	**p1**	X		X		X		X
	p2		X	X			X	X
	p3				X	X	X	X



Here, d1, d2, d3, d4 are the 4-bit data and p1, p2, p3 are the parity bits. From this table it is obvious that parity bit p1 is calculated from data bits d1, d2 and d4 and all data bits are covered by at least 2 parity bits.

Let's work through an example:
Suppose, we want to use Hamming(7,4) to encode the byte 1011.

The first thing we will do is split the byte into two Hamming code data blocks, 1011 and 0001.
We expand the first block on the left to 7 bits: _ _ 1 _ 0 1 1.
The first missing bit (bit 1) is 0, because adding bits 3, 5 and 7 gives an even number (2).
The second missing bit (bit 2) is 1, because adding bits 3, 6 and 7 gives an odd number (3).
The last missing bit (bit 4) is 0, because adding bits 5, 6 and 7 gives an even number (2).
This means our 7 bit block is: 0 1 1 0 0 1 1

We expand the second block to 7 bits using similar logic, giving: 1 1 0 1 0 0 1

7 bits do not make a byte, so we add a leading 0 to each code block. So the encoded final bytes are 0 0 1 1 0 0 1 1, 0 1 1 0 1 0 0 1

Error Detection and Correction:
Calculate e1, e2, and e3 as follows:
e1 = p1 + d1 + d2 + d4
e2 = p2 + d1 + d3 + d4
e3 = p3 + d2 + d3 + d4

If e1, e2, e3 are calculated to be even numbers, then there is no error. We can recover the original data block by simply pulling out the data bits from the code block. The data bits of 0 1 1 0 0 1 1 are in locations 3, 5, 6, and 7, giving 1011.

But if the calculated values (e1, e2, e3) produce odd value, then that indicates an error.

1. If only one value is odd, that means the corresponding parity bit is flipped.
2. If two values are odd, that means the common data bit they covered is flipped.
3. If all three values are odd, that means, the only common data bit of all three, d4 is flipped.

For example, suppose we send our code block 0 1 1 0 0 1 1 out across the Internet, and accidentally bit 5 gets flipped, giving: 0 1 1 0 **1** 1 1.

e1 = 0 + 1 + 1 + 1 = 3 (odd number)
e2 = 1 + 1 + 1 + 1 = 4 (even number)
e3 = 0 + 1 + 1 + 1 = 3 (odd number)

Since both e1 and e3 are odd, the data bit covered by only parity p1 and p3 bits, (d2) is flipped. So we can correct it by flip d2 back and then pull out the data bits to get the original data block.

Programming Assignment Deliverable

You are to implement a Hamming(7,4) code in Java. Your class should be named Hamming. It should contain the following public methods:

public static void encode (byte data)
This method should get a byte value through its parameter, split it to 2 blocks of 4 bits each, encode each of the 4-bits of the value with the corresponding hamming code, and print the two resulting 7-bit blocks as integers. (20 points if encoding works correctly.)

public static void decode (byte hammingCode)
This method should take as an input a byte of Hamming(7,4) code words. It should decode the 7 least-significant bits of the byte, pulling out the data bits and correcting errors if present. Finally, it should print the (potentially) corrected 4-bit integer in decimal. (20 points if decoding works correctly.)

public static void main(String []args)
The program should also include a main() method that reads the action to perform and a value from keyboard. You can use the Scanner class to read a string and a byte from keyboard. The string can be "encode" or "decode", and it should perform the corresponding hamming-code operation with the byte of the input. (10 points if you process the input correctly.)

Example:
Run the program to encode.
Input: encode 213
Output: 85 37
Because 213 in binary is 1101 0101, after encoding it is 0101 0101 and 0010 0101, in decimal 85 37.

Run the program to decode. If you are using an IDE "decode 85" is the command-line argument that should be passed to your program.
Input: decode 85
Output: 13
Because 85 in binary is 0101 0101, after decoding it is 1101, in decimal 13.

2. Cone volume and area (10 points)

Write a class named Cone with two fields (height and radius). Define two six public methods of the class:

public void setHeight(double h)
This method will take a double as input and set the height using that value.

public void setRadius(double r)
This method will take a double as input and set the radius using that value.

public double getHeight()
This method should return the height of the cone.

public double getRadius()
This method should return the radius of the cone.

public double volume()

This method should calculate the volume of the cone using the values of height and radius of the object and return the volume.

Public double totalSurfaceArea()
This method should calculate the total surface area of the cone using the values of height and radius of the object and return the area.

public static void main(String []args)
The program should also include a main() method that will receive keyboard input, invoke these methods, and print the volume and surface area of the cone.

(place the Cone.java file under directory *conevolume,* 10 points)

3. Operator precedence and associativity (10 points)

Place parentheses in the appropriate place of the following expressions to specify the order of evaluation using operator precedence rules. (4 x 2.5 = 10 points)
For example:
Given: 1+2*2,
Answer: 1 + (2*2).

a. a = b < c && d == e || f >= c
b. a = b * c / d + e % f + ++g
c. a | 4 + c >> b & 7
d. a = b ==1 & --c < 100