**OOP Computer Science**

**Problem Set 1 Binary and 2s Complement**

**Objective:**

To refresh, reinforce concepts of using a Binary Number system, learn and understand how to represent negative binary numbers using 2s Complement, expose to some Java data type/data width basics.

**Key Concepts:**

* Binary conversion to/fr hex, decimal, base64
* Operations of binary numbers
* Storing properties using binary digits
* Java arrays (instead of lists)
* Java Integer class and max/min values
* **note**: for conversion code, we will manually do the conversion algorithm (there is an extremely convenient shortcut that Python provides; although that's nice to know, we will not use that as our primary conversion method).

**Sections:**

**A. Binary Refresher**

**B. Negative Binary Numbers**

**C. 1s Complement**

**D. 2s Complement**

**E. Finale**

**F. Java Objects/Classes**

**G. Challenge**

**Your summary:**

**(This space intentionally blank)**

**Problem Set 1 Binary and 2s Complement**

**Part A Binary Refresher**

A.1.1. Fill in (Write out) the 3-bit **Addition Table (by hand, no overflow)**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **+** | **000** | **001** | **010** | **011** | **100** | **101** | **110** | **111** |
| **000** | **000** | **001** | **010** | **011** | **100** | **101** | **110** | **111** |
| **001** | **001** | **010** | **011** | **100** | **101** | **110** | **111** | **000** |
| **010** | **010** | **011** | **100** | **101** | **110** | **111** | **000** | **001** |
| **011** | **011** | **100** | **101** | **110** | **111** | **000** | **001** | **010** |
| **100** | **100** | **101** | **110** | **111** | **000** | **001** | **010** | **011** |
| **101** | **101** | **110** | **111** | **000** | **001** | **010** | **011** | **100** |
| **110** | **110** | **111** | **000** | **001** | **010** | **011** | **100** | **101** |
| **111** | **111** | **000** | **001** | **010** | **011** | **100** | **101** | **110** |

A.1.2. Write a Python program to output a **4-bit Addition Table**. There are several different techniques you could use; think of them and choose one.

**Code:**

**def make\_add\_table(size):**

*#empty table*

**table = []**

*#iterate through rows and columns*

**for i in range(2\*\*size):**

**table\_row = []**

**for j in range(2\*\*size):**

*#sum current row and col value*

**sum = i+j**

*#blank output*

**num = ""**

*#break decimal number down and build up num variable with*

*#binary digits*

**while sum > 0:**

**val = sum & 1**

**num = str(val) + num**

**sum = sum >> 1**

*#zero pad to bit size*

**while len(num) < size:**

**num = "0" + num**

**table\_row.append(num[-sum:])**

**table.append(table\_row)**

**return table**

**Print out the 4-bit Addition Table. We will use it later.**

A.2. Convert to decimal:

A.2.1. (0100)2 = **4**

A.2.2. (1011)2 = **11**

A.2.3. (1111)2 = **15**

A.2.4. (0001 1100) = **28**

A.3. Convert from decimal

A.3.1. (5)10 = **0101**

A.3.2. (13)10 = **1101**

A.3.3. (15)10 = **1111**

A.4. Convert to hex

A.4.1. (0100)2 = **4**

A.4.2. (1011)2 = **b**

A.4.3. (1111)2 = **f**

A.4.4. (0001 1100) = **1c**

A.5. Convert from hex

A.5.1. (0A)16 = **(0000 1010)2 (10)10**

A.5.2. (1C)16 = **(0001 1100)2 (28)10**

A.5.3. (9F)16 = **(1001 1111)2 (159)10**

A.5.4. (ff)16 = **(1111 1111)2 (255)10**

A.5.5. (9fcc e39d 3477 90dd 453c 378a bc9a 332f f990 2311 9101)16 = ? (j/k)

A.6. Why do we like multiples of 4-bit binary when converting to/fr hex?

**- We like like 4 bit chunks because the maximum value for 4 bits is 16, which is the base of hexadecimal. Because of this 1 grouping of 4 bits can be converted to a hexadecimal character with ease. If a grouping is larger than 4 bits convert 4 at a time starting from LSB. If the number of bits remaining is less than 4 bits, zero padding will get you back to 4.**

A.7. Why is 8-bit (or multiples of 8bit) so important?

**- Because 8 bits are a btye, which can also be represented as two hexadecimal characters. A byte is the size in which we store a single ASCII character, and is the “standard” grouping size of bits.**

A.8. Why is hex so convenient for HTML Color Codes? Explain.

**- Hexadecimal is important for color codes because traditional RGB color values go from 0-255 which can easily be represented by one byte, or in a more readable way, two hex characters. So we can create an RGB color value that is 6 hexadecimal characters long, with 2 characters apiece representing R, G, and B. This makes looking at colors from 24 bits, down to just 6 hex characters.**

A.9. Why is Base64 used?

**- Base 64 is similar to hex, as it takes multiple bits and encodes in in a character, but base 64 is 6 bits to a character as compared to hex’s 4. The 6 bits are the numbers 0-63 and are represented with normal numbers, lower case letters, uppercase letters, and + and /. Base 64 takes the compression of hex up a notch and can capture a lot of information in less characters. For instance we use base 64 to encode long numbers, such as an RSA key which can be many thousands of bits, into a concise string of characters.**

A.10\*. Why is Base58 used?

**- Base 58 remove characters that look similar to each other like lower case L and upper case I. It is most commonly used in bitcoin addresses, as you want to be as accurate as possible when writing the address down, base 58 removes characters that you could mix up and possibly confuse with one another.**

CODE: Write Java class to convert binary to hex.

**class BinToHex:**

**hexLookup = ['1', '2', '3', '4', '5', '6', '7', '8', \**

**'9', 'a', 'b', 'c', 'd', 'e', 'f']**

**def \_\_init\_\_(self, bits):**

*#input is a binary number*

**self.bits = bits**

**self.hexVals = ""**

**def convert(self):**

*#break off bits until inputted bit value is zero*

**while self.bits > 0:**

*#capture the 4 rightmost bits*

**currVal = self.bits & 0xf**

*#insert new value to output by indexing*

*#char to a list of possible chars*

**self.hexVals = self.hexLookup[currVal-1] + self.hexVals**

*#shift the value by the 4 bits you just examined*

**self.bits = self.bits >> 4**

**B. Negative Binary Numbers**

Since we only have bits to represent data (ie., we do not have a place for a negative sign), the simplest solution is to use the "high order bit" (ie., the left-most bit) as a flag to indicate negative or positive. 1 is negative, 0 is positive.

If we are using a binary number system that allows for negative numbers, we will call the binary numbers "**signed**". If we are using a binary number system that does not allow for negative numbers (ie., positive or zero only) we will call the numbers "**unsigned**".

B.0. Negate the following numbers using "Sign Bit" technique

B.0.1. (0011 1100) **(1011 1100)**

B.0.2. (0111 1001) **(1111 1001)**

B.0.3. (1101 1011) **(0101 1011)**

B.1. Given unsigned 8-bit binary, ie. "normal" binary numbers

B.1.1. How many total numbers can you represent? **256**

B.1.2. What is the largest number you can represent? (Give both binary and decimal representation) **(255) (1111 1111)**

B.1.3. What is the smallest number you can represent? (Give both binary and decimal representation) **(0) (0000 0000)**

B.1.4. If you count all the numbers from (1.3) thru (1.2), how many numbers is that? **256**

B.1.5. Does that match 1.1? **Yes.**

B.2. Given signed 8-bit binary numbers, using the "**Sign Bit**" technique

B.2.1. How many total numbers can you represent? **255**

B.2.2. What is the smallest number you can represent? (Give both binary and decimal representation)**-127**

B.2.3. What is the largest number you can represent? (Give both binary and decimal representation)**127**

B.2.4. If you count all the numbers from (2.2) thru (2.3), how many numbers is that? **25**

B.2.5. Does that match 2.1? **No**

B.2.6. How far off are you? Do you know why/can you find the problem area? **I am one number off, the problem is that a signed bit number has both a positive and negative 0.**

B.2.7. Does negating a negative number result in the expected positive number? **Yes**

Simple Addition of Signed Binary

B.3. Given signed 8-bit binary numbers using the "Sign Bit" technique

B.3.1. (0001 1011)**(27)** + (0010 0001)**(33)** = **(0011 1100) (60)**

B.3.2. (0000 0001)**(1)** + (0011 0011)**(51)** = **(0011 0100) (52)**

B.3.3. (0011 1001)**(57)** + (1001 0001)**(-17)** = **(1100 1010) (-74)**

B.3.4. (1100 0011)**(-67)** + (0011 0010)**(50)** = **(1111 0101) (-117)**

B.3.5. (0000 0000)**(0)** + (1000 0000)**(-0)** = **(1000 0000) (-0)**

B.3.6. (1111 1111)**(-127)** + (0000 0001)**(1)** = **(0000 0000) (0)**

B.3.7. (1001 1100)**(-28)** + (1000 0100)**(-4)** = **(0010 0000) (32)**

B.3.8. (1010 0001)**(-33)** + (1010 0001)**(-33)** = **(0100 0010) (66)**

B.3.9. (1000 0000)**(-0)** + (0000 1000)**(8)** = **(1000 1000) (-8)**

B.4. Some of those additions "work out". If you convert the binary numbers into decimal THEN add them, it should match the number if you add the binary numbers first THEN convert to decimal.

B.4.1. Which of (3.1-3.9) works and which ones don't? **1 and 2 work, but 3 through 9 don’t work.**

B.4.2. Any thoughts on what causes the broken ones? **All of the broken ones have one or more negative signed bits.**

**B.C. CODE**:

We will use signed binary int as a 4-char **array** using Sign Bit. Write a Java class to do the following:

B.C.1. Take signed decimal int (proper domain), convert to 4-char array, print.

B.C.2. Take 4bit char array and output the decimal value.

B.C.1. Take 4bit char array and negate it; output the result.

B.C.2. Take two signed binary char arrays and add them; output the result.

**class SignedBit():**

**def \_\_init\_\_(self, val):**

*#4 bit array, val domain is 7 to -7*

**self.val = abs(val)**

*#little endian bit array*

**self.bitArray = []**

*#convert val to bit array*

**while self.val > 0:**

*#take next bit*

**remainder = self.val & 1**

**print(remainder)**

*#add bit to bitArray as str*

**self.bitArray.append(str(remainder))**

*#remove that bit*

**self.val = self.val >> 1**

**while len(self.bitArray) < 3:**

**self.bitArray.append("0")**

**if val == abs(val):**

**self.bitArray.append("0")**

**else:**

**self.bitArray.append("1")**

**def \_\_add\_\_(self, other):**

**resultArray = []**

**carry = 0**

**for (a,b) in zip(self.bitArray, other.bitArray):**

*#build up a counter if it encounters 1s*

*#if counter is 3, it places a 1*

*#if counter is 2, it carries a 1*

*#if counter is 1, it places a 1*

**counter = carry**

**counter += 1 if a == "1" else 0**

**counter += 1 if b == "1" else 0**

**resultArray.append("1" if counter == 1 or counter == 3 else "0")**

**carry = 0 if counter < 2 else 1**

**return resultArray**

**def negate(self):**

*#if signed bit is 0 change to 1 and vice versa*

**if self.bitArray[-1] == "0":**

**self.bitArray[-1] = "1"**

**else:**

**self.bitArray[-1] = "0"**

**C. 1s Complement**

Another technique is to just "negate" ALL the bits of a number.

eg, in 8 bit signed, 1s Complement

The negative of (0110 0011) is (1001 1100)

C.1. Negate the following binary numbers using 1st Complement:

C.1.1. (0011 0110) = **(1100 1001)**

C.1.2. (1100 1001) = **(0011 0110)**

C.1.3. (1101 0000) = **(0010 1111)**

C.2.1. How many total numbers can you represent?

C.2.2. What is the smallest number you can represent? (both binary and decimal) C.2.3. What is the largest number you can represent? (both binary and decimal)

C.2.4. Count all the numbers from (2.2) thru (2.3); how many numbers is that?

C.2.5. Does that match 2.1?

C.2.6. How far off are you? Do you know why/can you find the problem area?

C.2.7. Does negating a negative number result in the expected positive number?

**C.C. CODE**:

We will use signed binary int as a 4-char array using **1s Complement**. Write Java Class to do the following:

B.C.1. Take signed decimal int (proper domain), convert to 4char array and print.

B.C.2. Take 4bit char array and output the decimal value.

B.C.1. Take 4bit char array and negate it, output the result.

B.C.2. Take two signed binary ints and add them, output the result.

**D. 2s Complement**

2s Complement is a different method, but as with 1s Complement and Sign Bit, the high-order bit will determine the sign of the number.

In general (ie., in any number system) Explain how Subtraction is related to addition. Now explain how you can find a number by using Subtraction (again, in general).

Now, using your 4-bit binary addition table (with no overflow) from A.1:

D.1. What is the largest (decimal) number representable in 4-bit signed?

D.2. What is the smallest (decimal) number representable in 4-bit signed?

D.3. Explain how to do addition using the table, eg. 4 + 3 = ?

D.4. Explain how to do a subtraction using the table, eg., 7 - 3 = ?

D.5. How does overflow impact 3.3 and 3.4?

Now let's try to find a negative number. We'll do that by using the relationship that addition and subtraction have to each other.

D.6. Given the equation: **4 = 3 - X**. Re-write it as an ADDITION equation.

D.7. Solve for X in 3.6.

So now, if we use our 4-bit addition table, we can solve our addition problem.

**Step 1** : find the number **4** as binary (0100) on the left side of the table.

**Step 2** : Stay on that row and then go to the right and find the number **3** in binary (0011).

**Step 3** : Go to the top of that column to find out which column in binary that number **3** is in. Write that number down.

**Step 4** : That is the answer.

D.8. What is the answer?

D.9. Repeat with **3 = 2 - X**.

D.9.1. Do you get the same number as in 3.8?

D.9.2. Does it make sense? why?

D.10. Repeat with 5 = 3 - X

D.11. Repeat with 5 = 2 - X

D.12. Explain what is going on and why this seems to work

D.13. Is there a negative-zero(-0) value?

D.14. Fill in the table for 4-bit 2s complement negative numbers:

|  |  |
| --- | --- |
| **Decimal Value** | **Binary Representation** |
| **-1** |  |
| **-2** |  |
| **-3** |  |
| **-4** |  |
| **-5** |  |
| **-6** |  |
| **-7** |  |

D.15. Convert the equation to binary and solve

D.15.1. -1 + 1 = ?

D.15.2. -1 - 1 = ?

D.15.3. -6 + 1 = ?

D.15.4. -6 - 1 = ?

D.15.5. 7 + 1 = ?

D.15.6. 5 + 5 = ?

D.15.7. -5 - 2 = ?

D.16. In 4-bit UNSIGNED int, what is **MAX\_VALUE**?

D.16. In 4-bit signed 2s Complement int, what is **MAX\_VALUE** ?

D.17. In 4-bit signed 2s Complement int, what is **MIN\_VALUE** ?

D.19. In 16-bit UNSIGNED int, what is **MAX\_VALUE**?

D.18. In 16-bit signed 2s Complement int, what is **MAX\_VALUE** ?

D.18. In 16-bit signed 2s Complement int, what is **MIN\_VALUE** ?

D.20. In 32-bit UNSIGNED int, what is MAX\_VALUE?

D.18. In 32-bit signed 2s Complement int, what is **MAX\_VALUE** ?

D.18. In 32-bit signed 2s Complement int, what is **MIN\_VALUE** ?

D.19. In Java, you can use Integer.MAX\_VALUE and Integer.MIN\_VALUE to get the maximum value and min values. Integer.SIZE gives number of bits.

D.19.1. What is MAX/MIN/SIZE of Integers

D.19.2. What is MAX/MIN/SIZE of Short

D.19.3. What is MAX/MIN/SIZE of Long

D.19.4. What is MAX/MIN/SIZE of Byte

We will use signed binary int as a 4-bit **string** using **2s Complement**. Write Java code to do the following:

B.C.1. Take signed decimal int (proper domain), convert to 4bit string and print.

B.C.2. Take 4bit string and output the decimal value.

B.C.1. Take 4bit string and negate it, output the result.

B.C.2. Take two signed binary ints and add them, output the result.

**E. Finale**

Now consider Sign Bit, 1s Complement, and 2s Complement.

E.1. What are the major differences between them.

E.1.1. What are the major weaknesses of Sign Bit and 1s Complement.

E.1.2. Why is 2s Complement generally the method used?

E.1.3. What is the interesting properties that 2s Complement has that could possibly cause unusual results in 16bit signed ints?

**F. Objects and Classes in Java**

Java uses 2s Complement for ints. Note there is no **unsigned** in Java.

F.1. What is max of Integer + 1?

F.2. What is min of Integer - 1?

F.3. Look at the following code snippet and explain the results:

**byte myByte = -1;**

**int num = ((int) myByte) & 0xff;**

**System.out.println(byte + " => " + num);**

(hint: what happens if you don't include the 0xff mask?)

Write Java code for the class Bin4 to handle 4-bit binary.

F.4. Write addition code to add two Bin4 numbers

F.5. Write to\_decimal to output the decimal value of the binary

F.6. Write to\_hex to output the hexadecimal value of the binary

F.7. Write a **Bin8** class to handle 8-bit binary.

**G. Challenge Section**

G.1. Challenge : Write Java code to convert a bit string into Base64.

G.2. Challenge : Write Java code to convert a bit string into Base58.

G.3. Write a Permission Java class, that contains 3 bits in an array. The highorder bit is "Read", the middle bit is "Write", and the low order bit is "eXecute".

The class should have the following methods:

* enableRead()
* disableRead()
* enableWrite()
* disableWrite()
* enableExecute()
* disableExecute()
* permissionValue()
* toString()

The permissionValue() should output the decimal value of the octal number (base-8) represented by the bits. toString() should output the permissions as "RWX", eg., Read and Write permission but no Execute permissions is "RW"

G.4. Write a UgoPermission Java class that contains "RWX" permissions for each of "**User**", "**Group**", and "**Other**".

The class should have the following methods:

* setPermissionValue(int userValue, int groupValue, int otherValue)
* toString()

setPermissionValue use userValue to set the user read, write, and execute bits, and similar for Group and Other.

toString() outputs all the permissions values as in G.3.