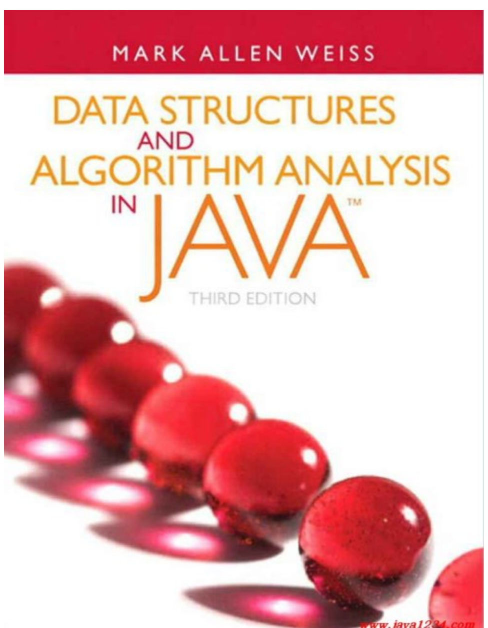
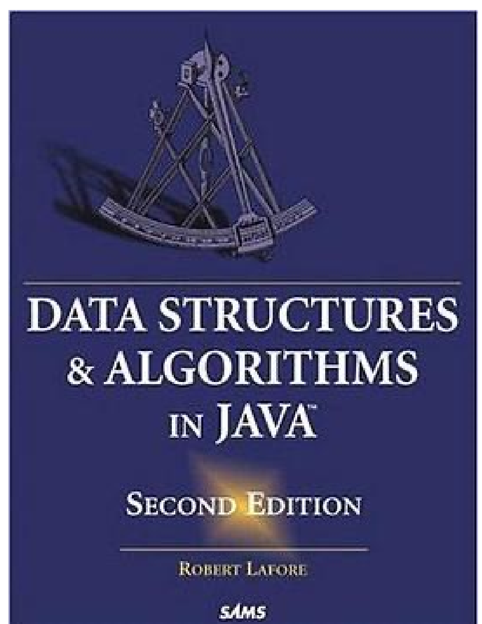
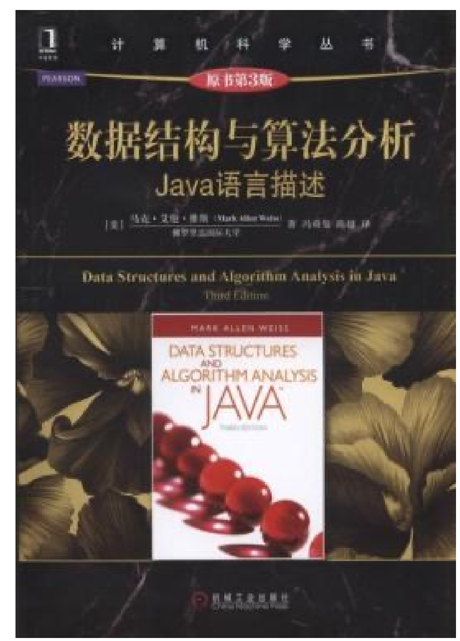
Topic 10 –
Bit Manipulation





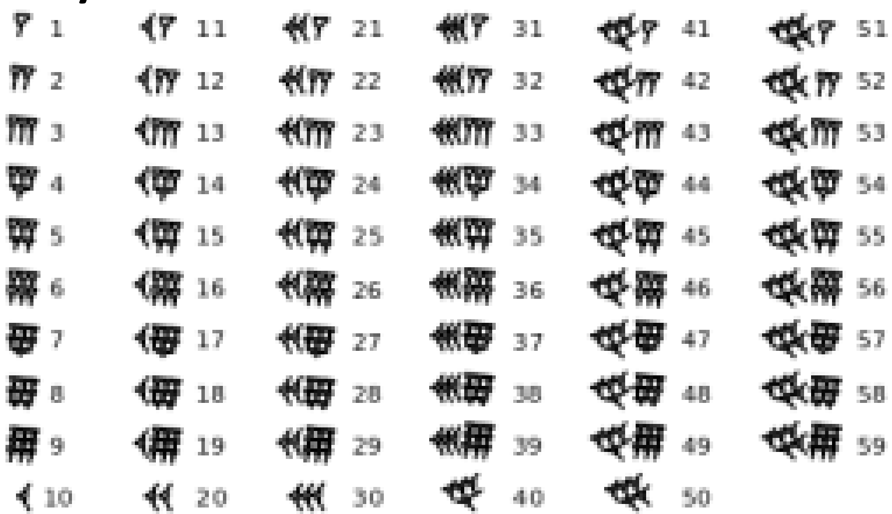


Topics

- Introduction
- Programming Revision
- Methods and Objects
- Arrays and Array Algorithms
- Big O Notation
- Sorting Algorithms
- Stacks and Queues
- Linked Lists
- Recursion
- **Bit Manipulation**

Positional number system

- The Hindu-Arabic numeral system, which is base 10, is the most commonly used system in the world
- The system evolved in India around 300BC, with zero identified about 1,000 years later, popularized by Fibonacci and spreading across Europe by around 1400
- The ancient Babylonians used a **base 60** system
- The Mayans used a **base 20** system



Positional number system

- There have been arguments for a base 12 (dozenal) system

Positional number system

- In **base $\mathbf{10}$, $\mathbf{1004}$** means
  - 4 units
  - 0 10s
  - 0 10^2s
  - 1 10^3s

  ➔$4 + 0 \times 10 + 0 \times 10^2 + 1 \times 10^3 = 1004$ (in base 10 – obviously!)
- In **base $\mathbf{12}$, $\mathbf{1004}$** means
  - 4 units
  - 0 12s
  - 0 12^2s
  - 1 12^3s

  ➔ $4 + 0 \times 12 + 0 \times 12^2 + 1 \times 12^3 = 1732$ (in base 10)

Converting base

- To convert a number in base 10 to any other base, we need to figure out how many units of each power it has
- E.g. convert 1004 from base 10 to base 12
  - Find how many 12^3s it has: **0**
  - Find out how many 12^2s it has: **6** with remainder 140
  - Find out how many 12s it has: **11** with remainder 8
  - Find out how many units it has: **8**
  - So the answer is 6-11-8 or 6elv8

  - 1004 % 12 = **8** ➔ 1004 – 8 = 996 ➔ 996 / 12 = 83
  - 83 % 12 = **11.** ➔ 83 – 11 = 72 ➔ 72 / 12 = 6
  - 6 % 12 = **6** ➔ 6 – 6 = 0 ➔ 0 / 12 = 0

Converting base

- In mathematics and computing, **hexadecimal** (also base 16, or hex) is a positional numeral system with a radix, or base, of 16. It uses sixteen distinct symbols, most often the symbols $0 - 9$ to represent values zero to nine, and A, B, C, D, E, F (or alternatively a, b, c, d, e, f) to represent values ten to fifteen
- Convert $\mathbf{9A3}$ to decimal base 10
  - 9A3 =
  - $9 \times 16^2 = 2304$
  - $10 \times 16^1 = 160$
  - $3 \times 16^0 = 3$
  - Total $= 2304 + 160 + 3 = 2467$

Bit representation

- In Java an **int** is represented as **32 bits**
- Starting from the right, each bit represents **increasing powers of 2**
- The leftmost bit is **special**.
- It is **negative**, and represents -2^{31}, which is $-2{,}147{,}483{,}648$
- The effective range is therefore from $2{,}147{,}483{,}647$ to $-2{,}147{,}483{,}648$
- There are several operators for directly manipulating the bit representation

Bit representation

This (leftmost) is negative – if the value of this bit is 1 it counts as - 2^31

2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- The int value of this 32-bit string is -2^31 + 2^30 + 2^29 which is -536,870,912

Big Endian vs Little Endian

- Little Endian is used by Java, also PowerPC, ARM, iPhone, Xbox 360 and PS3 and encodes the bytes in this order:

1st byte / 2nd byte

2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0

3rd byte / 4th byte

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Big Endian is used by Intel (C / C++ depends on the system) and encodes bytes in this order:

1st byte / 2nd byte

2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

3rd byte / 4th byte

2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}	2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}
0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0

Picking out a byte

- Say you want to pick out the second byte in a 32-bit integer, you can do it as follows

```
int num = 3534464; //we want byte 2 of this
int filter = Integer.parseInt
    (“00000000111111110000000000000000”,2);
int secondbyte = num & filter;
```

- The bits in the position of a 0 are lost, the bits in the position of a 1 are preserved

Bit representation

Value	**Bit representation**			
0	00000000	00000000	00000000	00000000
-1	11111111	11111111	11111111	11111111
-2	11111111	11111111	11111111	1111111**0**
65535	00000000	00000000	11111111	11111111
-65535	11111111	11111111	00000000	0000000**1**

Bitwise AND operator (&)

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

- Take the two numbers, convert into bit form, follow the above rules to combine them, and then convert back to an integer value

```
int result = 7 & 3;
7 = ...111
3 = ...011
7 & 3 = ...011 = 3
```

```
int result = 6 & 3;
6 = ...110
3 = ...011
7 & 3 = ...010 = 2
```

Bitwise OR operator (|)

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

- Take the two numbers, convert into bit form, follow the above rules to combine them, and then convert back to an integer value

```
int result = 11 | 8;
11 = ...1011
8  = ...1000
11 | 8 = ...1011 = 11
```

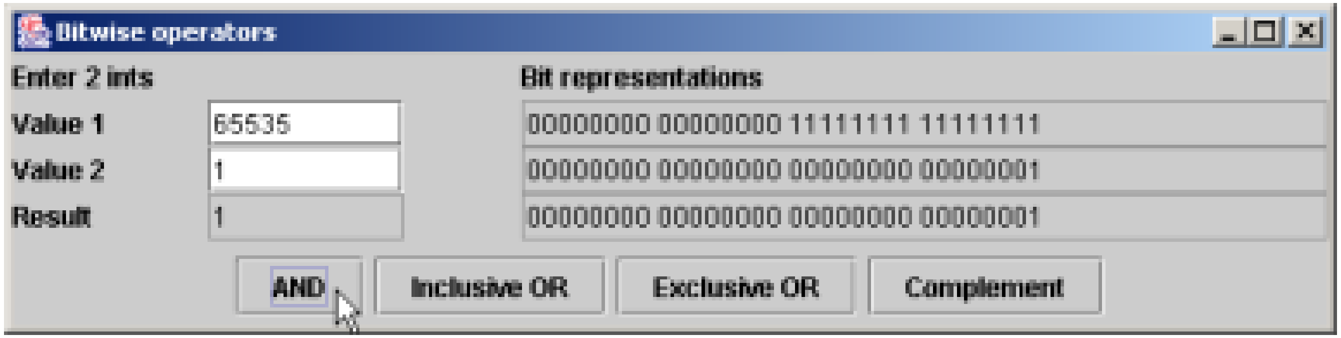
```
int result = 11 | 6;
11 = ...1011
6  = ...0110
11 | 8 = ...1111 = 15
```

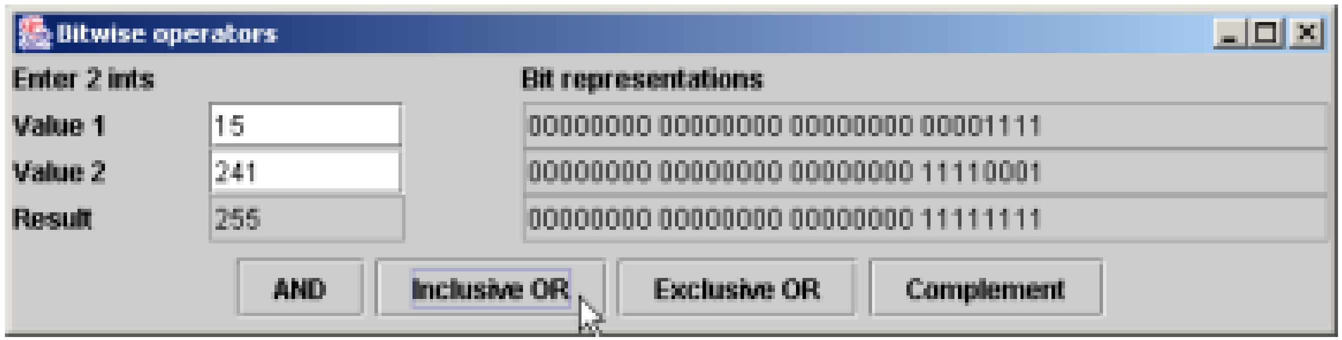
Bitwise XOR operator (^)

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	**1**	**0**

- Take the two numbers, convert into bit form, follow the above rules to combine them, and then convert back to an integer value

```
int result = 15 ^ 6;
15 = ...1111
6   = ...0110
15 ^ 6 = ...1001 = 9
```


Bitwise operators
Enter 2 ints
Bit representations
Value 1
65535
00000000 00000000 11111111 11111111
Value 2
1
00000000 00000000 00000000 00000001
Result
1
00000000 00000000 00000000 00000001
AND
Inclusive OR
Exclusive OR
Complement


Bitwise operators
Enter 2 ints
Bit representations
Value 1
15
00000000 00000000 00000000 00001111
Value 2
241
00000000 00000000 00000000 11110001
Result
255
00000000 00000000 00000000 11111111
AND
Inclusive OR
Exclusive OR
Complement


Bitwise operators
Enter 2 ints
Bit representations
Value 1
139
00000000 00000000 00000000 10001011
Value 2
199
00000000 00000000 00000000 11000111
Result
76
00000000 00000000 00000000 01001100
AND
Inclusive OR
Exclusive OR
Complement


Bitwise operators
Enter 2 ints
Bit representations
Value 1
21845
00000000 00000000 01010101 01010101
Value 2
Result
-21846
11111111 11111111 10101010 10101010
AND
Inclusive OR
Exclusive OR
Complement

Question

- What is 2 & 2?

A)	0
B)	1
C)	2
D)	3
E)	4

- What is 8 | 7?

A)	7
B)	9
C)	11
D)	13
E)	15

- What is 11 ^ -3?

A)	-10
B)	-5
C)	1
D)	8
E)	11

Bitwise left shift (<<)

- Take the number, convert into bit form, shift the bits to the left and fill in the spaces on the right with 0s

  int result = 15 << 3;
  - 15 = 00000000 00000000 00000000 00001111
  - 15 << 3 = 00000000 00000000 00000000 01111000
  - 15 << 3 = 120 (15 * $\mathbf{2^3}$)

  int result = 116 << 5;
  - 116 = 00000000 00000000 00000000 01110100
  - 116 << 5 = 00000000 00000000 00001110 10000000
  - 116 << 5 = 3,712 (116 * $\mathbf{2^5}$)

Bitwise signed right shift (>>)

- Take the number, convert into bit form, shift the bits to the right and fill in the spaces on the left with 1s if it's a negative number, 0s otherwise

  int result = 116 >> 3;
  - 116 = 00000000 00000000 00000000 01110100
  - 116 >>3 = 00000000 00000000 00000000 00001110
  - 116 >>3 = 14 which is (around 116 / 2^3)

  int result = -116 >> 3;
  - -116 = 11111111 11111111 11111111 10001100
  - -116 >>3 = 11111111 11111111 11111111 11110001
  - -116 >>3 = -15 which is (around -116 / 2^3)

Bitwise unsigned right shift (>>>)

- Take the number, convert into bit form, shift the bits to the right and fill in the spaces with 0s no matter what

  int result = 116 >>> 3;
  - 116 = 00000000 00000000 00000000 01110100
  - 116 >>>3 = 00000000 00000000 00000000 00001110
  - 116 >>>3 = 14 which is (around 116 / 2^3)

  int result = -116 >> 3;
  - -116 = 11111111 11111111 11111111 10001100
  - -116 >>>3 = 00011111 11111111 11111111 11110001
  - -116 >>>3 = 536,870,897

Bitwise complement (~)

- Take the number, convert into bit form, flip every 1 to a 0 and vice versa
- This operation on n is the same as (n*-1) - 1

  int result = ~116;

  - 116 = 00000000 00000000 00000000 01110100
  - ~116 = 11111111 11111111 11111111 10001011
  - ~116 = -117

Question

- What is 2 << 3?

A)	2
B)	4
C)	8
D)	16
E)	32

- What is 8 >> 2?

A)	0
B)	1
C)	2
D)	4
E)	8

Question

- What is -11 >>> 1 given that -1 >>> 1 is 2,147,483,647?

A)	-2,147,483,647
B)	-2,147,483,631
C)	-15
D)	2,147,483,642
E)	2,147,483,647

- What is ~7?

A)	-6
B)	-7
C)	-8
D)	-9
E)	-10

Optional

Interview question #1

- Explain what the following code does:

  ((n & (n-1) == 0)

- What does it mean if A & B == 0?
  - It means that A and B never have a 1 bit in the same place

- So if n & (n-1) == 0, then n and n-1 never share a 1

Interview question #1

- What does n-1 look like (as compared with n)?
- Try doing subtraction by hand in base 2

	1	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**
–										**1**
=	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**1**

- When you subtract 1 from a number you look at the least significant bit
- If it's a 1 you change it to a 0 and you're done
- If it's a zero, you must borrow from a larger bit, so you go to increasingly larger bits, changing each from a zero to a 1, until you find a 1
- You flip that to a 0 and you're done

Interview question #1

- So what does n & (n – 1) indicate?
- n and (n – 1) must have no 1s in common
- Therefore all the Xs below must be zeroes

	X	**X**	**X**	**X**	**X**	**X**	**1**	**0**	**0**	**0**
–										**1**
=	**X**	**X**	**X**	**X**	**X**	**X**	**0**	**1**	**1**	**1**

- So the number n must look like this: 00001000
- n is therefore a power of two
- Therefore, ((n & (n-1)) == 0) checks if n is a power of 2

Interview question #2

- Subtract two numbers without using minus

```
int first = 10;
int second = 3;
```

- Bitwise complement gives you the negative version of a number - 1

```
result = first + ~second + 1;
```

Interview question #3

- Add two numbers together without using +, - , * or /

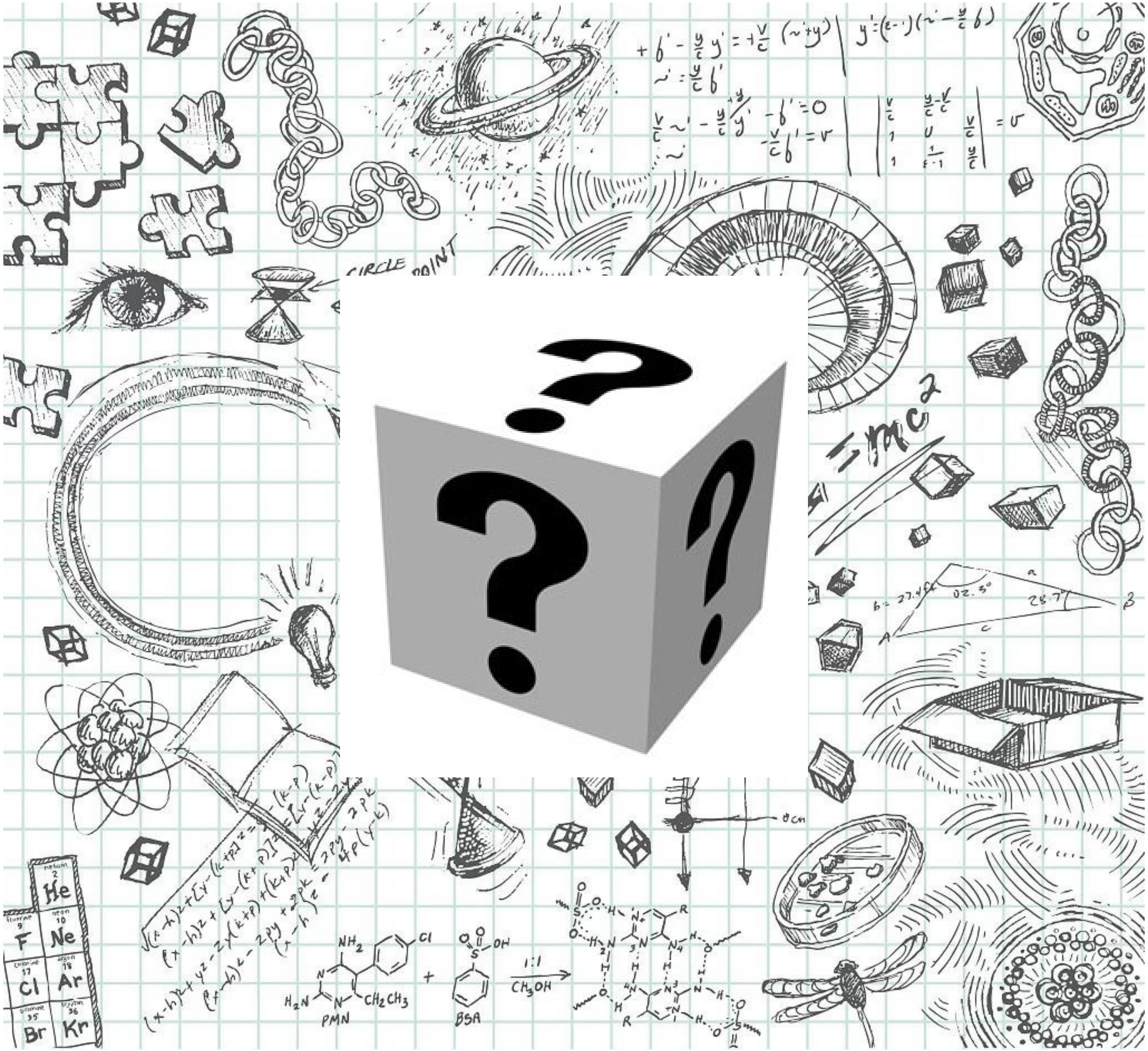
```
public int addition(int a, int b) {
        if(b==0) {
                return a;
        } else {
                sum = a^b;
                carry = (a&b)<<1;
                return addition(sum,carry);
        }
}
```

1st call	bit representation	
a=4,	... 0100	
b=6,	... 0110	
XOR,	... 0010	(2) sum
a&b,	... 0100	
(a&b)<<1,	... 1000	(8) carry
2nd call		
a=2,	... 0010	
b=8,	... 1000	
XOR,	... 1010	(10) sum
(a&b)<<1,	... 0000	(0) carry
return 10		

Question

- What is ~4 << ((5 & 3) | 1)?

A)	-19
B)	-10
C)	1
D)	7
E)	13


CIRCLE
POINT
He
F
Ne
Cl
Ar
Br
Kr
PMN
BSA
1:1
CH3OH