**Program to print the value of the Kth Bit in a given integer**

**Solution 1**

n=int(input("enter a integer"))

k=int(input("enter the bit pos"))

mask= 1 << k

masked\_number=n & mask

bitvalue=masked\_number >> k

print(bitvalue)

**Program to check whether bit is set or not**

n=int(input("enter a integer"))

k=int(input("enter the bit pos"))

if (n & (1 << k)) >>k:

print("set")

else:

print("not set")

or

n=int(input("enter an integer"))

b=int(input("enter the bit to test for set contd"))

if n & (1 << b):

print("set")

else:

print("not set")

**Program to check whether LSB bit is set or not**

n=int(input("enter a integer"))

if n&1:

print("LSB set")

else:

print("LSB not set")

or

n=int(input("enter a integer"))

print("Value of LSB is {:} of {:}".format((n&1),n))

**Program to print all bit values of a given integer**

n=int(input("enter an integer"))

k=n.bit\_length()-1

while k>=0:

if (n & (1<<k)) >>k:

print("1")

else:

print("0")

k=k-1

n=int(input("enter an integer"))

ki=n.bit\_length()-1

for k in range(0, ki+1):

if (n & (1<<ki-k)) >>ki-k:

print("1")

else:

print("0")

n=int(input("enter an integer"))

ki=n.bit\_length()-1

for k in range(ki, -1,-1):

if (n & (1<<k)) >>k:

print("1")

else:

print("0")

n=int(input("enter a integer"))

for k in range(n.bit\_length()-1,-1,-1):

if (n & (1 << k)) >>k:

print("1 ")

else:

print("0 ")

LSB to MSB

n=int(input("enter a integer"))

for k in range(0,n.bit\_length(),1):

if (n & (1 << k)) >>k:

print("1 ")

else:

print("0 ")

**Program to set the kth bit of an integer**

n=int(input("enter an integer"))

k=int(input("enter the bit to set"))

mask=1 << k

nn= n | mask

print(n,nn)

print(bin(n),bin(nn))

or

n=int(input("enter an integer"))

k=int(input("enter the bit to set"))

nn= n | (1 << k)

print(n,nn)

print(bin(n),bin(nn))

**Program to unset / clear the kth bit of an integer**

n=int(input("enter an integer"))

k=int(input("enter the bit to unset"))

nn=n&~(1<<k)

print("number before unset is",format(n,"08b"))

print("new number is {:08b}".format(nn))

or

n=int(input("enter an integer"))

k=int(input("enter the bit to unset"))

nn=n & ~(1 << (k - 1))

print(n,nn)

print("old number is",format(n,"08b"))

print("new number is {:08b}".format(nn))

print("old number is {1:032b} new number is {0:032b}".format(nn,n))

**Program to toggle the kth bit of an integer**

n=int(input("enter an integer"))

k=int(input("enter the bit to unset"))

nn=n^(1<<k)

print(n,nn)

print("old number is",format(n,"08b"))

print("new number is {:08b}".format(nn))

print("old number is {1:032b} new number is {0:032b}".format(nn,n))

**Program to turn off the rightmost 1 bit to 0**

n=int(input("enter an integer"))

k=int(input("enter the bit to unset"))

nn=n & (n-1)

print(n,nn)

print("old number is",format(n,"08b"))

print("new number is {:08b}".format(nn))

**Program to turn on the rightmost 0 bit to 1**

n=int(input("enter an integer"))

k=int(input("enter the bit to unset"))

nn=n | (n+1)

print(n,nn)

print("old number is",format(n,"08b"))

print("new number is {:08b}".format(nn))

**Program to turn off the trailing 1’s to 0’s**

n=int(input("enter an integer"))

k=int(input("enter the bit to unset"))

nn=n & (n+1)

print(n,nn)

print("old number is",format(n,"08b"))

print("new number is {:08b}".format(nn))

**Program to turn on the trailing 0’s to 1’s**

n=int(input("enter an integer"))

k=int(input("enter the bit to unset"))

nn=n | (n-1)

print(n,nn)

print("old number is",format(n,"08b"))

print("new number is {:08b}".format(nn))

**Swap two nibbles in a byte**

**A nibble is a four-bit aggregation, or half an octet. There are two nibbles in a byte.**

**Given a byte, swap the two nibbles in it. For example 100 is be represented as 01100100 in a byte (or 8 bits). The two nibbles are (0110) and (0100). If we swap the two nibbles, we get 01000110 which is 70 in decimal.**

n=int(input("enter an integer"))

nn= (n & 0x0F)<<4 | (n & 0xF0)>>4

print(n,nn)

print("old number is",format(n,"08b"))

print("new number is {:08b}".format(nn))

print("old number is {1:032b} new number is {0:032b}".format(nn,n))

100 is 01100100 in binary. The operation can be split mainly in two parts

1) The expression “x & 0x0F” gives us last 4 bits of x. For x = 100, the result is 00000100. Using bitwise ‘<<' operator, we shift the last four bits to the left 4 times and make the new last four bits as 0. The result after shift is 01000000.

2) The expression “x & 0xF0” gives us first four bits of x. For x = 100, the result is 01100000. Using bitwise ‘>>’ operator, we shift the digit to the right 4 times and make the first four bits as 0. The result after shift is 00000110.

At the end we use the bitwise OR ‘|’ operation of the two expressions explained above. The OR operator places first nibble to the end and last nibble to first. For x = 100, the value of (01000000) OR (00000110) gives the result 01000110 which is equal to 70 in decimal.

# Compute the minimum or maximum of two integers without branching

Use XOR and bitwise AND operator

Minimum of x and y will be

y ^ ((x ^ y) & -(x < y))

It works because if x < y, then -(x < y) will be all ones, so r = y ^ (x ^ y) & ~0 = y ^ x ^ y = x. Otherwise, if x >= y, then -(x < y) will be all zeros, so r = y ^ ((x ^ y) & 0) = y. On some machines, evaluating (x < y) as 0 or 1 requires a branch instruction, so there may be no advantage.

To find the maximum, use

x ^ ((x ^ y) & -(x < y));

# Compute the sum of two integers using only bit operators

Sum of two bits can be obtained by performing XOR (^) of the two bits. Carry bit can be obtained by performing AND (&) of two bits.

n1=int(input("enter first number"))

n2=int(input("enter second number"))

while n2:

carry = n1 & n2

n1 = n1 ^ n2

n2 = carry << 1

print(n1)

Above is simple [Half Adder](http://en.wikipedia.org/wiki/Adder_%28electronics%29#Half_adder) logic that can be used to add 2 single bits. We can extend this logic for integers. If x and y don’t have set bits at same position(s), then bitwise XOR (^) of x and y gives the sum of x and y. To incorporate common set bits also, bitwise AND (&) is used. Bitwise AND of x and y gives all carry bits. We calculate (x & y) << 1 and add it to x ^ y to get the required result.