Bit-Manipulations

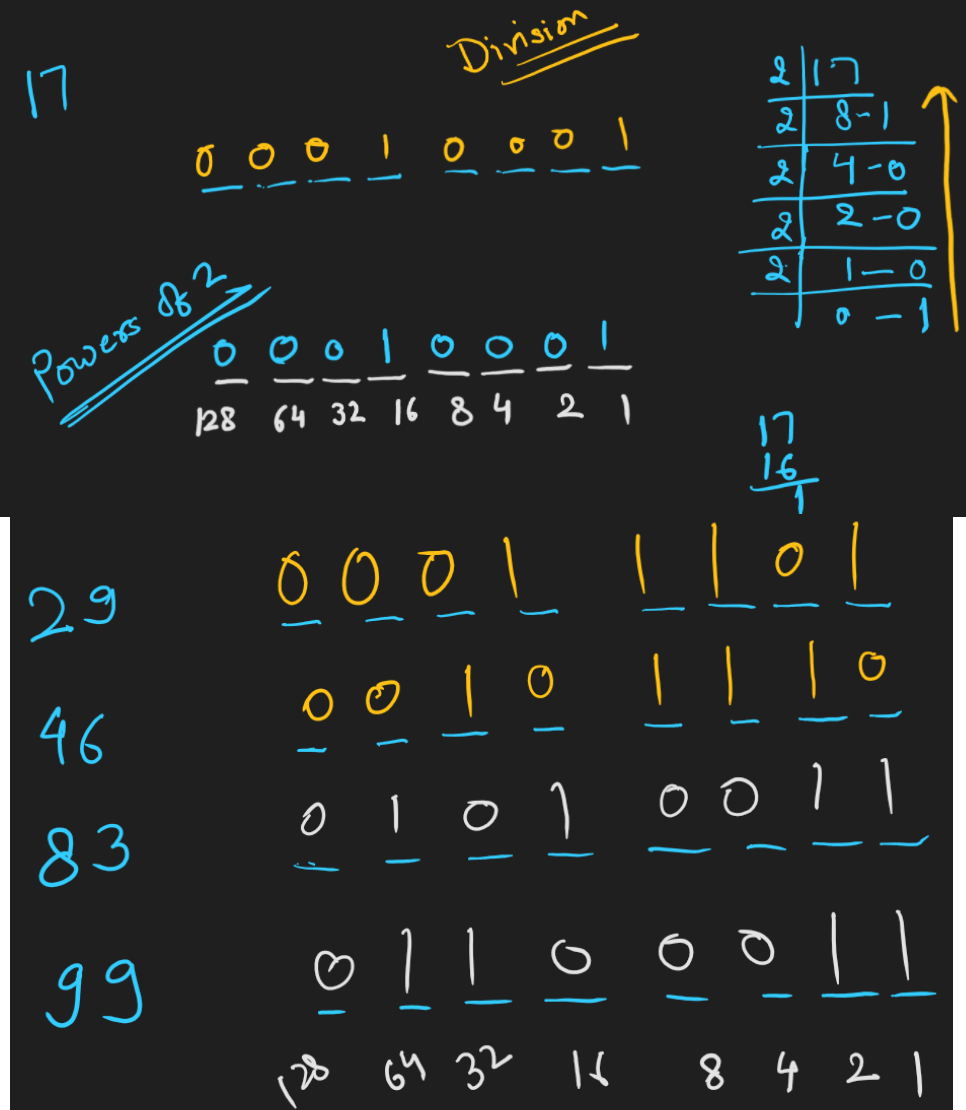
How is an integer stored in computer memory?

int a = 17;

Stored as binary number of how many bits?

4 Bytes => 32-bits

For now, Let's assume a data type with only 8-bits



How -ve Numbers are represented in computer memory?

int a = -17;

ASSUME a is 8-bit number

-*X* = ~*X* + 1

-ve of a number is = One’s Complement + 1 = Two’s Complement

MSB (Most Significant Bit)

* In a K-Bit Number, the Leftmost Bit is called MSB.

LSB (Least Significant Bit)

* In a K-Bit Number, the Right Most Bit is called LSB.

SIGNED NUMBERS (can represent both +ve and -ve)

* MSB value is taken as -ve power of 2.

UNSIGNED NUMBERS

* MSB value is taken as +ve power of 2.

Converting DECIMAL to BINARY is easy.

+ve number ===> Direct Conversion using Div or Power of 2 Method

-ve Number ===> 2s Complement Method or Power of 2 with MSB as -ve value

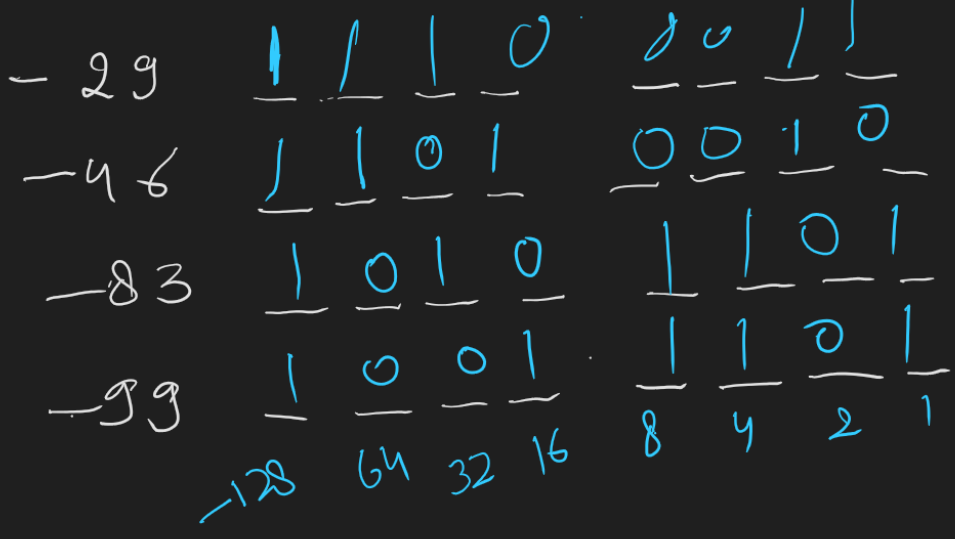
BINARY TO DECIMAL

We need to know if the number is SIGNED or UNSIGNED.

Given an 8-Bit Binary Number, convert it to decimal

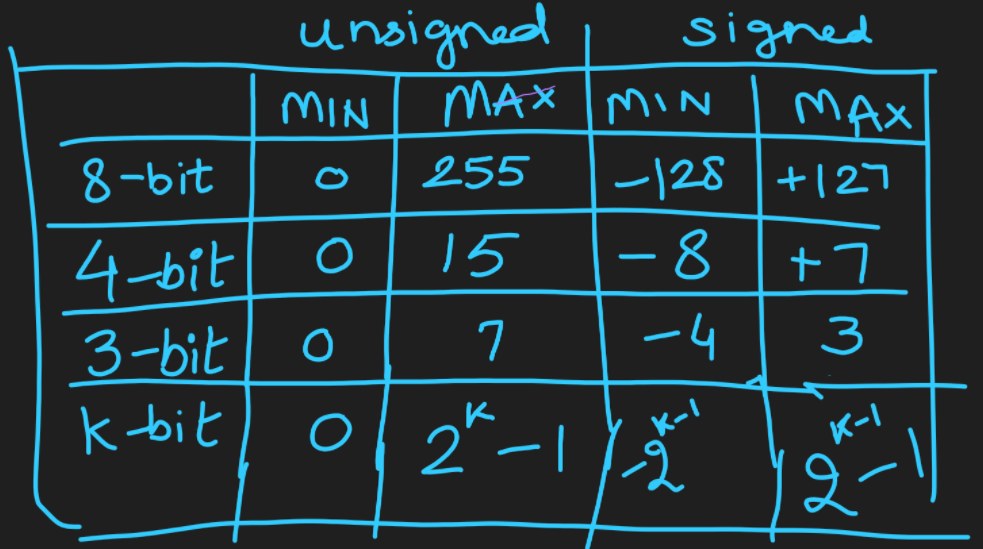
1001 1101 ===> (SIGNED) ===>-128+16+8+4+1=-128+29=-99

1001 1101 ===> (UN-SIGNED) ===> 128 + 29 =157



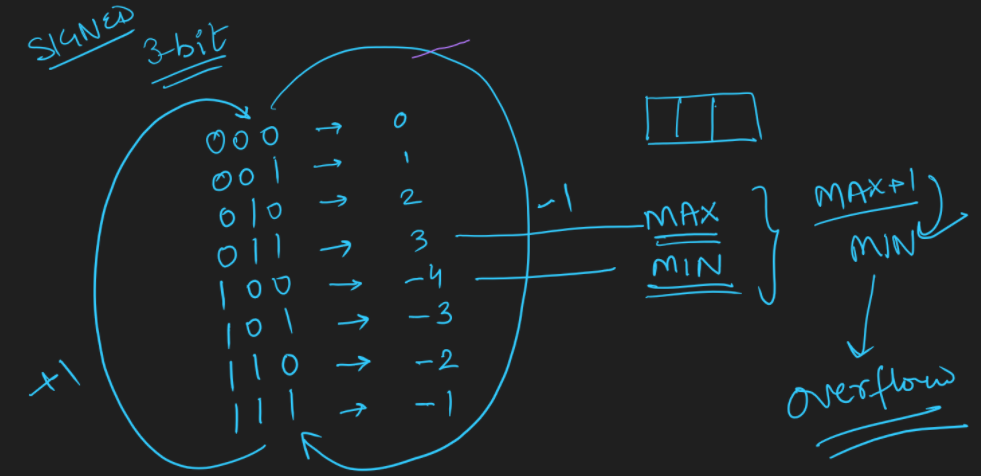
### FASTER WAY TO GET 2’s Complement

* Start from the right side and keep writing digits till we encounter first 1.
* After that flip everything.
  + 00101100 ==> 11010100
* JAVA doesn’t have any unsigned numbers.
* C/C++/C# has signed and unsigned int separate.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BITS | UNSIGNED | | SIGNED | |
| MIN | MAX | MIN | MAX |
| 32 | 0 | 232-1 | -231 | 231 -1 |
| 4,294,967,295 | -2,147,483,648 | 2,147,483,647 |
| **≈ 4 X 109** | **≈ -2 X 109** | **≈ 2 X 109** |
| 64 | 0 | 264-1 | 263 | 263-1 |
| 18,446,744,073,709,551,615 | -9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |
| **≈ 18 X 1018** | **≈ -9 X 1018** | **≈ 9 X 1018** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 210 | 1024 | **≈** | 1000 | 103 |
| 232 | 4 X 210 X 210 X 210 | 4 X 1000 X 1000 X 1000 | 4 X 109 |



# BITWISE OPERATORS

1. Bitwise AND
2. Bitwise OR
3. Bitwise XOR
4. Bitwise One's COMPLEMENT
5. Bitwise LEFT SHIFT
6. Bitwise RIGHT SHIFT

1, 2, 3. Bitwise AND, OR, XOR

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Numbers | | 4Bit | A|B | A&B | A^B |
| A | 3 | 0011 |  |  |  |
| B | 9 | 1001 |  |  |  |
| A | 7 |  |  |  |  |
| B | 2 |  |  |  |  |
| A | 12 |  |  |  |  |
| B | 14 |  |  |  |  |

1. Bitwise not

int main(void) {

int A = 5;

int B = ~A;

*cout* << B << "\n";

return 0;

}

* Negative of a number is it’s 2’s complement.

if A = 5, then **-A** 🡺2’s Complement🡺(1’s Complement + 1) 🡺(~A+1)

-A = ~A + 1

~A = -A – 1

If A = 5; B = ~A 🡺 -5-1 🡺 -6

1. Bitwise LEFT SHIFT

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Variable | Value | 8 Bit Representation | | | | | | | |
| X | 23 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| X = X << 1 |  | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |

**X = X << K = X \* 2K**

1. Bitwise RIGHT SHIFT

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Variable | Value | 8 Bit Representation | | | | | | | |
| X | 23 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| X = X >> 1 |  | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

**X = X >> K = X / 2K**

## PROPERTIES OF XOR

* *a* ^ *a* = 0
* *a* ^ 0 = *a*
* *a* ^ b = b ^ *a* ====> ASSOCIATIVE
* *a* ^ b ^ c = (*a* ^ b) ^ c = *a* ^ (b ^ c) = (*a* ^ c) ^ b ===> COMMUTATIVE

**[PROBLEM]** Given n (<=20), return 2n.

APPROACH 1: Write a loop ‘n’ times and multiply with 2. TC: n

APPROACH 2: Using bitwise XOR. TC: 1

int twoPowerN(int n) {

return (1 << n);

}

long long fourPowerN(int n) {

return (1LL << (2 \* n));

}

* 1 is INTEGRAL CONSTANT (32-bit)
* 1L is LONG CONSTANT (64-bit) in JAVA/C#
* 1LL is LONG LONG constant in C++

## Some Points

* BITWISE OPERATORS have lower precedence so when using them keep in mind to use brackets.
* Usually when values go beyond 1018, We need to either use STRING data type.
* Problem author would ask us to print the answer by doing MODULO. => C#, JAVA :> Biglnteger and BigDoubIe which internally use Strings.
* PYTHON automatically switches to strings when int values grow big.
* C/C++ - We need to use strings for big data values.

**[PROBLEM]** Given a number n (64-bit), check if its kth bit (<=60) is set or not. Right most bit is called the 0th bit.

EXAMPLE:

N = 34 (0010 0010)

K = 1 ===> yes

K = 2 ===> no

k = 5 ===> yes

bool isKthBitSetV1(long long n, int k) {

return (n & (1LL << k)) != 0; // 'true' if set.

}

bool isKthBitSetV2(long long n, int k) {

return ((n >> k) & 1) != 0; // 'true' if set.

}

bool isKthBitSetV3(long long n, int k) {

return (n | (1LL << k)) == n; // 'true' if set.

}

int main(void)

{

*ios\_base*::*sync\_with\_stdio*(false);

*cout*.*tie*(nullptr);

*cin*.*tie*(nullptr);

int n, k; *cin* >> n >> k;

*cout* << *std*::*boolalpha* << isKthBitSetV1(n, k) << "\n";

*cout* << *std*::*boolalpha* << isKthBitSetV2(n, k) << "\n";

*cout* << *std*::*boolalpha* << isKthBitSetV3(n, k) << "\n";

return 0;

}

**[PROBLEM]** Given a number n (64-bit), set its kth bit (<=60) and return the updated number. Right most bit is called the 0th bit.

N = 34 (0010 0010)

N=34, k=2 ===> 38

N=34, k=1 ===> 34

N=34, k=0 ===> 35

long long setKthBit(long long n, int k) {

return (n | (1LL << k));

}

## [PROBLEM] Clear Kth Bit in Number

Given T pairs of two integral values N and K, print N by clearing Kth bit in N.  Right most bit is bit 0, second from right is considered bit 1 and so on.  Clearing bit means we need to make Kth bit as 0 in binary representation of given number N.

**Can you do it for T complexity.  Each test case should be O(1)**

**Input Format**

Line 1:  Integer T - Number of test cases.  
Next T lines represent two space separated numbers N  K per line.

**Constraints**

1 <= T <= 1000000  
0 <= K <= 63  
0 <= N <= 1018

**Output Format**

For each given N and K, print single value back which is N after setting Kth bit in it.

**Sample Input 0**

5

10 2

15 2

3 0

3 1

7 2

**Sample Output 0**

10

11

2

1

3

**Explanation 0**

There are 5 test cases given here:  
Binary form of 10 => 1010 => Clearing second bit leaves it as is => so answer is 10.  
Binary form of 15 => 1111 => Clearing second bit makes it 1011 => so answer is 11.  
Binary form of 3 => 11 => Clearing 0th bit makes it 10 => So answer is 2.

APPROACH 1:

long long clearKthBitV1(long long n, int k) {

return n & ~(1LL << k);

}

APPROACH 2:

bool isKthBitSetV1(long long n, int k) {

return (n & (1LL << k)) != 0; // 'true' if set.

}

long long clearKthBitV2(long long n, int k) {

if (isKthBitSetV1(n, k))

return n - (1LL << k);

return n;

}

APPROACH 3:

long long clearKthBitV3(long long n, int k) {

if (isKthBitSetV1(n, k))

return n ^ (1LL << k);

return n;

}

[PROBLEM] Given an array of size N. Tell if all numbers from 0 to 9 are present in it or not.

A= {3, 1, 8, 0, 0, 9, 2, 2, 2, 2,4,4, 4, 4, 5, 5, 5, 5}

ANS = NO.

A= {1,2, 3, 1, 2, 3,4, 6, 5, 7, 9, 1, 0, 8}

ANS = YES.

bool allDigitsPresent(*vector*<int> arr) {

}

Approach 1: (Sort and Check)

1. Sort the Array in ASC order.
2. Loop over the array to check if all 0 to 9 are there or not.

TC: nlogN + N

SC: 1

Approach 2: (Set)

1. Take a Set
2. Loop over the array
   1. If numbers are in range 0 to 9
      1. Insert in Set
3. return set.length() == 10;

TC: NlogN

SC: 10

Approach 3: (Array Of Int or Bool)

1. Take an array B of size 10 (indexed 0 to 9)
2. Fill every value with 0
3. Loop over the array
   1. If numbers are in range 0 to 9
      1. B[a[i]] = 1
4. Loop over B if all are 1s return true
5. Else return False

TC: N + 10

SC: 10

Approach 4: (Take one integer as SET)

1. Take X = 0
2. for i = [0, N-1]
   1. if(a[i] >=0 && a[i] <= 9)
      1. X = setKthBit(x, a[i])
3. return X == 0b1111111111; // Binary Number
4. return X == (1 << 10) -1; // Binary Number

TC: N

SC: 1

void setKthBit(int& n, int k) {

n = n | (1LL << k);

}

bool allDigitsPresent(*vector*<int> arr) {

int x = 0;

for(auto i : arr)

if (i >= 0 && i <= 9)

{

setKthBit(x, i);

}

return x == 0b1111111111;

}

int main(void) {

*ios\_base*::*sync\_with\_stdio*(false);

*cout*.*tie*(nullptr);

*cin*.*tie*(nullptr);

*vector*<int> A = { 3, 1, 8, 0, 0, 9, 2, 2, 2, 2,4,4, 4, 4, 5, 5, 5, 5 };

*vector*<int> B = { 1,2, 3, 1, 2, 3,4, 6, 5, 7, 9, 1, 0, 8 };

*cout* << *std*::*boolalpha* << allDigitsPresent(A) << "\n";

*cout* << *std*::*boolalpha* << allDigitsPresent(B) << "\n";

return 0;

}

* From the 4th point of “APPROACH 4” return X == (1 << 10) -1; // Binary Number

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Bit Position | | | | | | | | | | |
| Powers Of 2 | Value | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 21 - 1 | 1 |  |  |  |  |  |  |  |  |  |  | 1 |
| 22 - 1 | 3 |  |  |  |  |  |  |  |  |  | 1 | 1 |
| 23 - 1 | 7 |  |  |  |  |  |  |  |  | 1 | 1 | 1 |
| 24 - 1 | 15 |  |  |  |  |  |  |  | 1 | 1 | 1 | 1 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 210 - 1 |  |  | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**[PROBLEM]** Given a string, tell if it has all lowercase characters in it. Can you solve this problem with 1 int as SET?

<https://codeforces.com/problemset/problem/520/A>

#include <iostream>

using namespace *std*;

void setKthBit(int& n, int k)

{

n = n | (1LL << k);

}

bool containsAllLowerCaseChars(*string* s)

{

int x = 0;

for (auto i : s)

{

if (i >= 'a' || i <= 'z')

setKthBit(x, i - 'a');

}

return x == ((1 << 26) - 1);

}

int main(void) {

*ios\_base*::*sync\_with\_stdio*(false);

*cout*.*tie*(nullptr);

*cin*.*tie*(nullptr);

*string* s; *cin* >> s;

// Test string

// nishiththequickbrownfoxjumpsoverthelazydogsdf

// thequickbrownfoxjumpsoverthelazydasdfasd

// thequickbrownfoxjumpsoverthelazydog

*cout* << *boolalpha* << containsAllLowerCaseChars(s) << "\n";

return 0;

}

## Unique Number In Five Multiplier

Given an array having all numbers that are repeated multiple of 5 times except one. We need to find the number.

**Input Format**

First line of input contains T - number of test cases. Its followed by 2T lines, the first line contains N - the size of the array (of the form 5X + 1) and second line contains the elements of the array.

**Constraints**

1 <= T <= 300  
1 <= N <= 1 + 104  
-109 <= A[i] <= 109

**Output Format**

For each test case, print the number which occurs only once, separated by new line.

**Sample Input 0**

2

11

7 5 7 5 9 7 5 7 5 7 5

6

10 10 10 10 11 10

**Sample Output 0**

9

11

**Explanation 0**

9 occurs one time. Rest all 7 and 5 occur 5 times each.  
11 occurs one time. 10 occurs 5 times.

**APPROACH 1:**

Using MAP

TC: NlogN = O(NlogN)

SC: N/5 + 1 = O(N)

**APPROACH 2:**

Sort and Solve

TC: NlogN + N

SC: 1

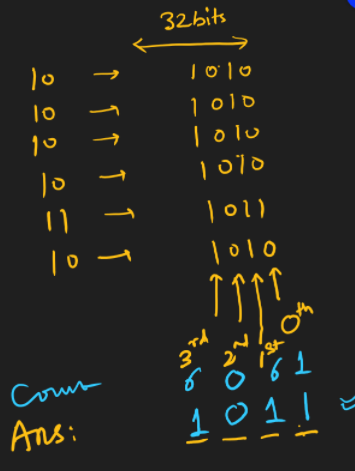
bool checkBit(long long n, int k) { **APPROACH 3:**

return (n & (1LL << k)) != 0;

}

long long setKthBit(int& n, int k) { **TC: 32\*N = O(N)**

return n = n | (1LL << k); **SC: 1**

}

int FiveMultiplier(*vector*<int> arr) {

int ans = 0;

for (int bitPos = 0; bitPos < 32; bitPos++) {

int setBitsCount = 0;

for (auto k = 0; k < arr.*size*(); k++) {

if (checkBit(arr[k], bitPos))

setBitsCount++;

}

if (setBitsCount % 5 != 0)

setKthBit(ans, bitPos);

}

return ans;

}

int main(void) {

*ios\_base*::*sync\_with\_stdio*(false);

*cout*.*tie*(nullptr);

*cin*.*tie*(nullptr);

int t; *cin* >> t;

while (t--) {

int n; *cin* >> n;

*vector*<int> arr(n);

for (auto i = 0; i < n; i++)

*cin* >> arr[i];

*cout* << FiveMultiplier(arr) << "\n";

}

return 0;

}

## Subsets Generation

TC: N\*2N

SC: 1

*List*< *List*<Integer> > subsets(*List*<lnteger> v)

{

*List*<*List*<Integer>> allSubsets = new ArrayList<>();

int totalSubsets = (1 << v.*size*());

for (int i = 1; i < totalSubsets; i++) {

*List*<Integer> curSubset = new ArrayList<>();

for (int bitPos = 0; bitPos < v.*size*(); bitPos++) {

if (checkBit(i, bitPos))

curSubset.*add*(v.*get*(bitPos));

}

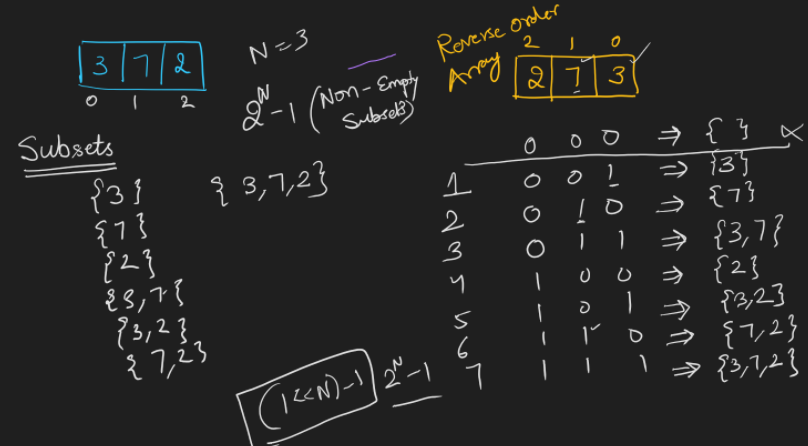
alISubsets.*add*(curSubset);

}

// For Lexicographical order, sort allSubsets, which is list of list

return allSubsets;

}



#include <iostream>

#include <vector>

#include <algorithm>

using namespace *std*;

bool checkBit(long long n, int k) {

return (n & (1LL << k)) != 0;

}

*vector*<*vector*<int>> subsets(*vector*<int> v) {

*vector*<*vector*<int>> allSubSets;

int totalSubSets = 1 << v.*size*();

for (auto i = 1; i < totalSubSets; i++) {

*vector*<int> currSubset;

for (auto bitPos = 0; bitPos < v.*size*(); bitPos++) {

if (checkBit(i, bitPos))

currSubset.*push\_back*(v[bitPos]);

}

allSubSets.*push\_back*(currSubset);

}

return allSubSets;

}

bool compare(*vector*<int>& a, *vector*<int>&b) {

*sort*(a.*begin*(), a.*end*());

*sort*(b.*begin*(), b.*end*());

return *lexicographical\_compare*(a.*begin*(), a.*end*(), b.*begin*(), b.*end*());

}

int main(void) {

int t; *cin* >> t;

while(t--) {

int n; *cin* >> n;

*vector*<int> li(n);

for (auto i = 0; i < n; i++)

*cin* >> li[i];

auto ans = subsets(li);

*sort*(ans.*begin*(), ans.*end*(), compare);

for (auto i : ans) {

for (auto j : i)

*cout* << j << " ";

*cout* << "\n";

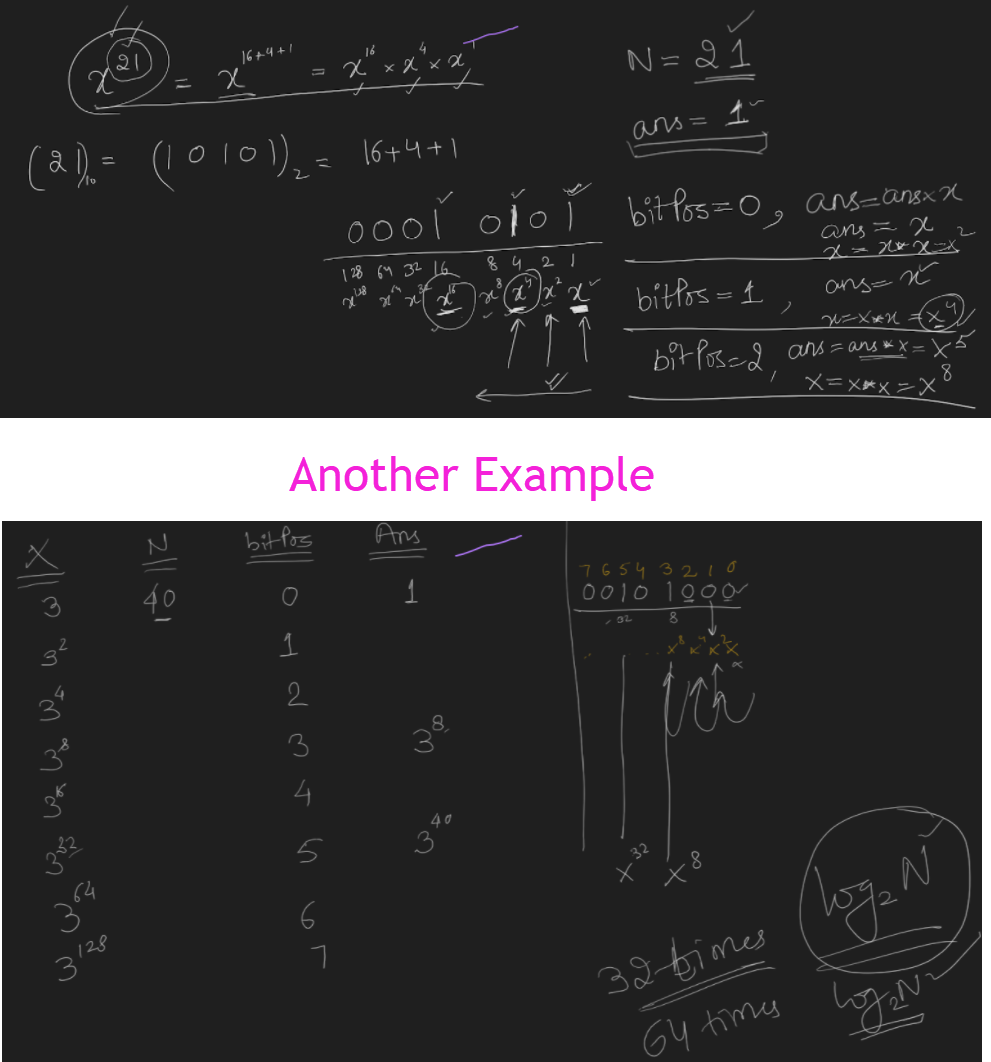
}

}

return 0;

}

## X Power N using Bit Manipulation (IogN)



|  |
| --- |
| // TC: 64  // SC: 1  long long xPowerNModD(long long x, int n, int d) {  long long ans = 1;  for (int bitPos = 0; bitPos < 64; bitPos++) {  if (checkBitPos(n, bitPos))  ans = (ans%d \* x%d) % d;  x = (x%d \* x%d) % d;  }  return ans;  }  // TC: log2N  // SC: 1  long long xPowerNModD(long long X, int n, int d) {  long long ans = 1;  while (n != 0) {  if ((n & 1) != 0)  ans = (ans%d \* x%d) % d;  X = (x%d \* x%d) % d;  n = n >> 1;  }  return ans;  } |

long long xPowerNModD(long long x, int n, int d)

{

long long ans = 1;

for (auto i = 0; i < 64; i++)

{

if (checkBit(n, i))

ans = ((ans % d) \* (x %d)) % d;

x = ((x % d) \* (x %d)) % d;

}

return ans;

}

int main(void)

{

long long x;

int n, d;

*cin* >> x >> n >> d;

*cout* << *ceil*(*log2*(n)) << "\n";

*cout* << xPowerNModD(x, n, d) << "\n";

return 0;

}

[PROBLEM] Find number of set bits in a given number.

* n = (n & (n-1)) RESETS RIGHTMOST SET BIT

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Bit Position | | | | | | | |
|  | Value | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| n | 100 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| n-1 | 99 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| n & (n-1) |  | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

#include <iostream>

using namespace *std*;

int main(void) {

*ios\_base*::*sync\_with\_stdio*(false);

*cout*.*tie*(nullptr);

*cin*.*tie*(nullptr);

long long n, num; *cin* >> num;

n = num;

int c = 0;

while (n != 0) {

c++;

n = (n & (n - 1));

}

*cout* <<"Number of set bits in "<< num << " is " << c << ".\n";

return 0;

}

<https://leetcode.com/problems/add-binary/>

<https://leetcode.com/problems/subsets/>

<https://leetcode.com/problems/subsets-ii/>

<https://leetcode.com/problems/single-number/>

<https://leetcode.com/problems/single-number-ii/>

<https://leetcode.com/problems/reverse-bits/>

<https://leetcode.com/problems/number-of-1-bits/>

<https://leetcode.com/problems/bitwise-and-of-numbers-range/>

<https://leetcode.com/problems/power-of-two/>

<https://leetcode.com/problems/single-number-iii/>

<https://leetcode.com/problems/missing-number/>

<https://leetcode.com/problems/counting-bits/>

<https://leetcode.com/problems/hamming-distance/>

<https://leetcode.com/problems/number-complement/>