Q 13.) Copy set bits in a range, toggle set bits in a range:

1) Copy set bit :- we have two numbers A and B, and we want to copy the bits of B to A for a given range L to R from LSB to MSB.

→ L starts from 1 and R goes till 32.

Ex →

```
          8 7 6 5 4 3 2 1
A =       0 1 0 1 0 1 1 0
B =       1 0 0 1 1 0 0 1
L =       3
R =       7
```

if we create a mask like

mask = 0 1 1 1 1 1 0 0

and do Bitwise AND with A, we can extract set bits of A in given range.

```
A    = 0 1 0 1 0 1 1 0
mask = 0 1 1 1 1 1 0 0
       0 1 0 1 0 1 0 0
```

→ Now If do Bitwise OR of (A & mask) with B, we will get our answer.

→ Creating mask is a crutial parts, there can be multipk ways to create the mask.

① left shift 1 till (r-l+1)

```
mask     = 0 0 1 0 0 0 0 0
(mask-1) = 0 0 0 1 1 1 1 1
```

Now left shift it by (l-1) times.

(mask-1) << (l-1) = 0 1 1 1 1 0 0

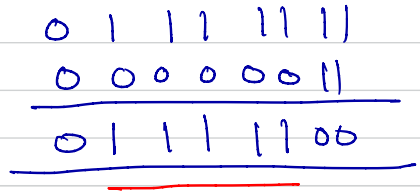
(2) Second way to creat mask is :-

right mask = $(1 << r) - 1$

= 0 1 1 1 1 1 1 1 1

left mask = $1 << (l-1) - 1$

= 0 0 0 0 0 0 0 1 1




```cpp
#include<iostream>
using namespace std;

int CopySetBit(int X, int Y, int l, int r){

 //only 1 can overflow in case of r = 32 and l=1
 //we need to take 1ll
   int maskLength = (1ll<<(r-l+1)) - 1;

    int mask = ((maskLength)<<(l-1)) & X;

    Y = Y | mask;
    return Y;
    }

int main()
{
    unsigned int A,B,l,r;
    cin>>A>>B>>l>>r>>l;

    cout<<CopySetBit(A,B,l,r);
    return 0;
}
```

2.) Toggle bits in range :-

→ Mask computation is same as copy bits in range

There are two ways above.

① `mask = (1 << (r-l+1)) - 1;`
`mask = (mask << (l-1));`

② `mask = ((1<<r)-1) ^ ((1<<(l-1))-1).`

→ Now after computing the mask, if we do XOR of mask with num, we will get toggle bits in range L to R.

`num = num ^ mask.`

ex- `1 1 0 1 1 0 1 1` , i = 2, j = 6

mask `0 0 1 1 1 1 1 0`

`1 1 1 0 0 1 0 1` → result.

```
int toggleBits(int N , int L , int R) {
        int mask = (1<<(R-L+1))-1;
        mask = (mask<<(L-1));
        int result = N ^ mask;
    }
```

Q 14.) Divide two integers without using Multiplication, Division and mod operator:

ex- Dividend = 10
Divisor = 3
Quotient = $\left\lfloor \frac{10}{3} \right\rfloor = 3$.

Approach 1: Repeated subtraction :-

Keep subtracting the divisor from dividend until dividend becomes less than divisor.
So, dividend will get reduced to became remainder & no. of times will subtract divisor from dividend will become the quotient.

Eg. $10 - 3 = 7$
$7 - 3 = 4$
$4 - 3 = 1 < 3$
} (3) $q = 3$

Time Complexity = O(dividend)
Space Complexity = O(1)

Approach 2 :-

As every number can be represented in base 2 (0 or 1), represent the quotient in binary form by using the shift operator as given below:

1. Determine the most significant bit in the divisor. This can easily be calculated by iterating on the bit position i from 31 to 1.
2. Find the first bit for which ***divisor << i*** is less than dividend and keep updating the i^{th} bit position for which it is true.
3. Add the result in the *temp* variable for checking the next position such that **(temp + (divisor << i))** is less than the **dividend**.
4. Return the final answer of the quotient after updating with a corresponding sign.

Q 15.) SINGLE NUMBER II

Given an integer array nums where every element appears thrice times except for one, which appears exactly once. Find the single element and return it.

ex – nums = [1, 1, 5, 1]
output = 5

Method 1: Brute Force

→ We can run two for loops and count the elements, if the count of any element is 1, after iterating the complete array, it will be our answer.

```cpp
//brute force
    int singleNumber(vector<int>& nums) {
        int nsize = nums.size();

        for(int i=0;i<nsize;i++){
             int count = 0;
            for(int j=0;j<nsize;j++){
                {
                    if(nums[i]==nums[j]){
                        count++;
                    }
                }
            }
            if(count==1){
                return nums[i];
            }
        }
        return -1;
    }
```

TC = $O(N^2)$

Method - 2 : Sorting + linear traversal

Ex - 5 4 2 4 4 5 5

After Sorting :- 2 4 4 4 5 5 5

→ We will have three conditions :-
1. Left Boundary condition
2. Right Boundary condition
3. other than Boundary element

1. Left Boundary condition :-
ex - 2 4 4 4 5 5
→ check if (num[0] != num[1])

2. Right Boundary condition :-
ex - 4 4 4 5 5 5 6
→ check if (nums[n-2] != nums[n-1])

3. Somewhere except Boundary
ex - 1 1 1 2 3 3 3

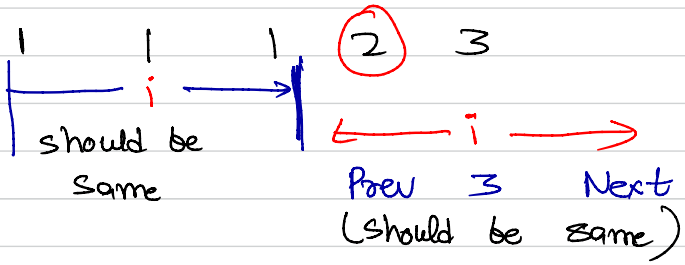
run a while loop, start comparing from index 1,

```
{ if (nums[i] != nums[i-1])
        return nums[i-1];
}
i+=3;
```

1 1 1 2 3 3 3
i
i

→ if current element is same as previous, then surely its next element will also be the same.

→ If we increament by 3, and its prev elem doesn't equal to current element then prev element is ans.



```cpp
//better approach : Sorting + linear traversal
    int singleNumber(vector<int>& nums) {
        int n = nums.size();
        sort(nums.begin(),nums.end());
        if(n<3) return nums[0];

        //check boundary conditions
        //2 4 4 4 5 5 5
        if(nums[0]!=nums[1]) return nums[0];
        //another boundary condition
        // 4 4 4 5 5 5 6
        if(nums[n-2]!=nums[n-1]) return nums[n-1];
        int i=1;
        while(i<n){
            if(nums[i]!=nums[i-1]){
                return nums[i-1];
            }
            i+=3;
        }
        return -1;
    }
```

$TC = O(N \log N) + O(N)$

$O(N \log N)$ ↓ sorting

$O(N)$ ↓ Traversing

Method 3: Unordered_map

→ We can store the count of elements in unordered_map and traverse it.

```cpp
//3rd approach : unordered_map : TC : O(N) , SC : O(N)
    int singleNumber(vector<int>& nums) {
        int n = nums.size();
        unordered_map<int,int> mp;
        for(int i=0;i<n;i++){
            mp[nums[i]]++;
        }

        for(auto it : mp){
            if(it.second==1){
                return it.first;
            }
        }

    return -1;
    }
```

TC = O(N)
SC = O(N)

Method 4: Counting Set Bits in Every Number array

ex - [2 2 2 3 5 5 5]

0 1 0 } 2
0 1 0
0 1 0
0 1 1 ← 3
1 0 1 } 5
1 0 1
1 0 1

→ Check for all 32 bits.
→ Use shift operation to find if current bit is set or not.
→ if the number is only appearing once, the set bit at that particular position will not be multiple of 3. so add that in ans.

result ↓ leftshift ↓

i = 0, leftshift = 1, numberofsetbits = 4, result = 0 + 1 = 1
i = 1, leftshift = 2, numberofsetbits = 4, result = 1 + 2 = 3
i = 2, leftshift = 4, numberofsetbits = 3, result = 3 + 4 = 7

i = 31

```cpp
//4th approach : counting Set bits in every number array
//TC: O(32N) every case not better than 3rd approach
    int singleNumber(vector<int>& nums) {
        int n = nums.size();
        int result = 0;

    int leftshift, numberofsetbits;

    // Iterate through every bit
    for(int i = 0; i <32; i++) {

        numberofsetbits = 0;
        leftshift = (1 << i);
        for (int j = 0; j < n; j++) {
            if (nums[j] & leftshift)
                numberofsetbits++;
        }
        if ((numberofsetbits % 3) != 0)
            result+= leftshift;
    }

    return result;
    }
```

Method : 5 Bit Manipulation:

→ We will use two variables ones and twos.
Ones will store the element, which is occuring only once.
twos will store the element, which is occuring twice.
→ when an element is occuring thrice, ones and twos will store 0.

ex- [2 2 2 3]

i=0, ones = 0, element = 2 (10), twos = 0 (00)

```
ones = (ones ^ ele) & (~twos);
twos = (twos ^ ele) & (~ones);
```

ones = 0 0
^ 1 0
1 0
& 1 1
1 0

ones = 2

twos = 0 0
^ 1 0
1 0
& 0 1
0 0

twos = 0

i=1, ones = 1 0
^ 1 0
0 0
& 1 1
0 0

ones = 0

twos = 0 0
^ 1 0
1 0
& 1 1
1 0

twos = 2

i=2, ones = 0 0
^ 1 0
1 0
& 0 1
0 0

twos = 1 0
^ 1 0
0 0
& 1 1
0 0

i=3, ones = 0 0
^ 1 1
1 1
& 1 1
1 1

ones = 3

twos = 0 0
^ 1 1
1 1
& 0 0
0 0

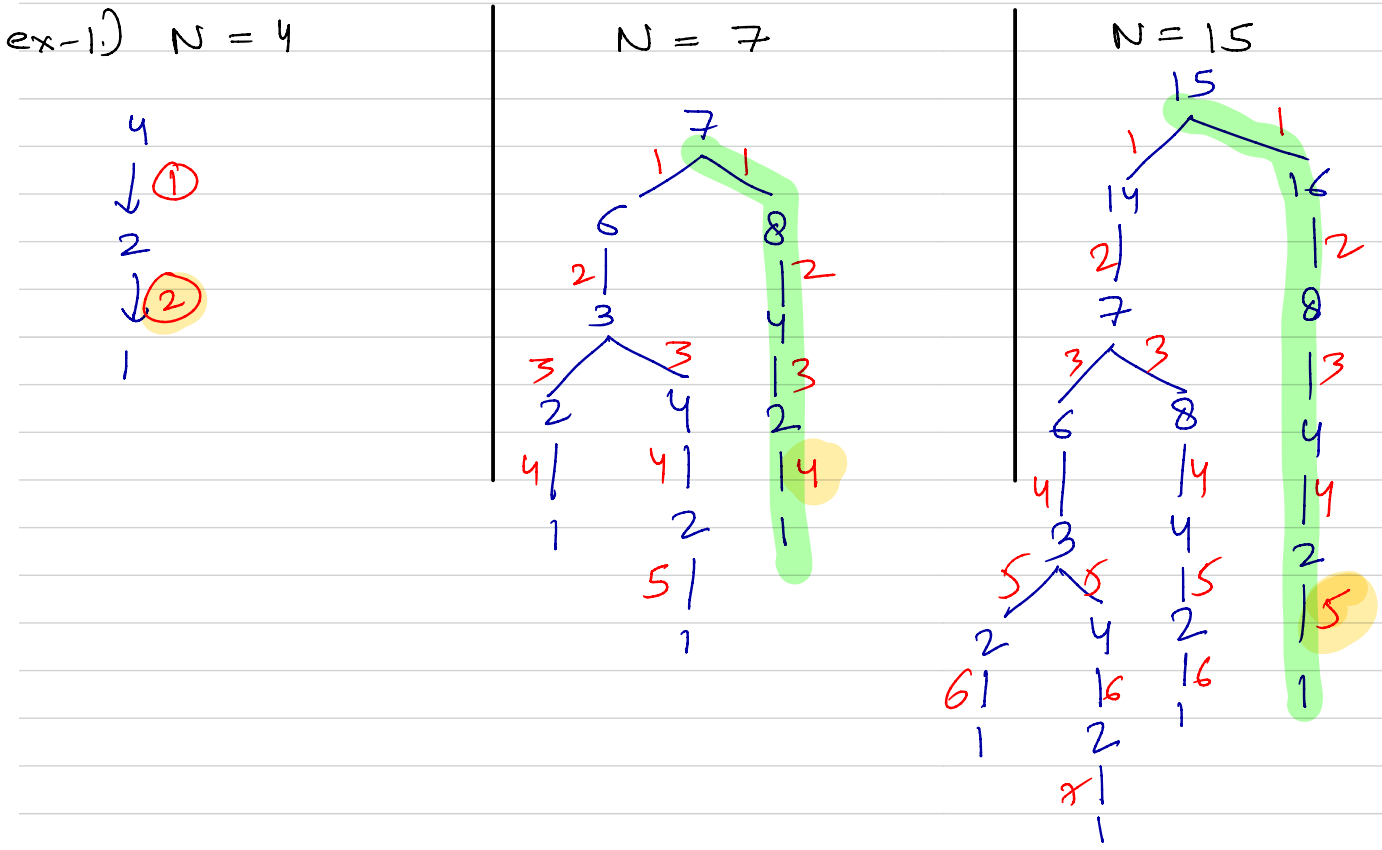
twos = 0

```cpp
//5th approach : Bit manipulation (Not intuitive)
//TC: O(N)
//our answer will store in ones
    int singleNumber(vector<int>& nums) {
        int ones =0;
        int twos =0;

        for(auto ele: nums){
            ones = (ones^ele) & (~twos);
            twos = (twos^ele) & (~ones);
        }
        return ones;
    }
```

Q 16.) Reduce a Number to 1:

Given a number, our task is to reduce the given number N to 1 in the minimum number of steps.
We can perform two types of operations in each step:

Operation 1: If the number is even then divide the number by 2
Operation 2: If the number is odd, then we are allowed to either (N+1) or (N-1).



Method 1:

We can use recursion untill n=1, for even numbers, we can return 1 + funct(n/2),
for odd numbers, we can return : 1 + min(funct(n-1),funct(n+1)).

```c
int minways(int n)
{
    if (n == 1)
        return 0;
    else if (n % 2 == 0)
        return 1 + minways(n / 2);
    else
        return 1 + min(minways(n - 1),minways(n + 1));
}
```

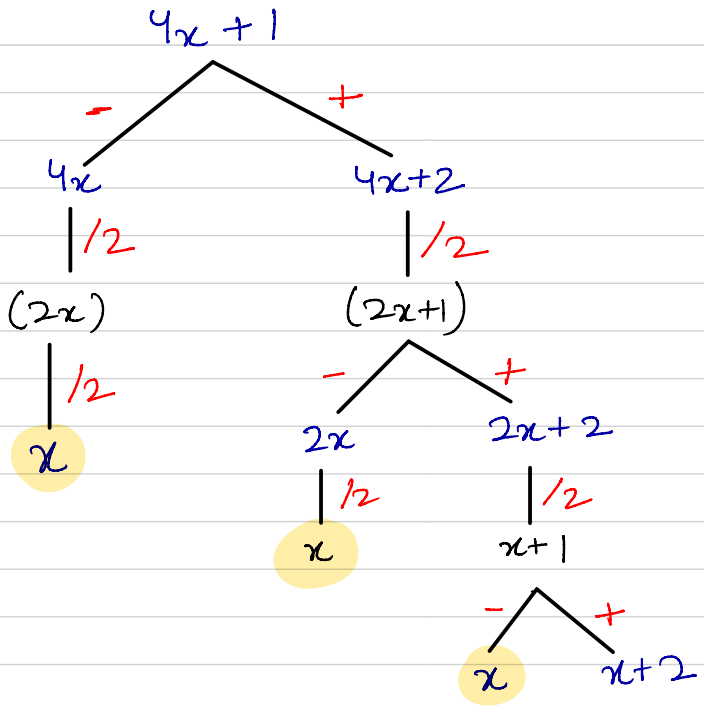
Method 2: Using Bit Manipulation

Even number can be represent as 4x + 0, i.e. 4x
Odd number can be represent as 4x+1 and 4x+3.

for Even number, we know that we need to divide it by 2,
but for odd number we have two cases(4x+1, 4x+3)and two operations (n-1,n+1).So We need to identify, which case is suitable for which operation.

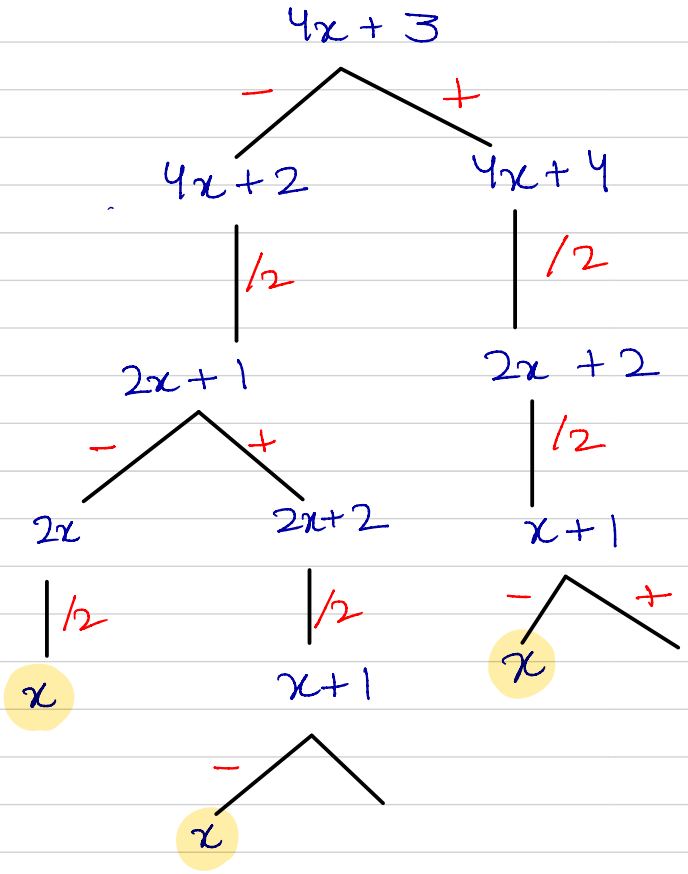
→ Lets take some examples :-

1) Take 4x+1 case first,



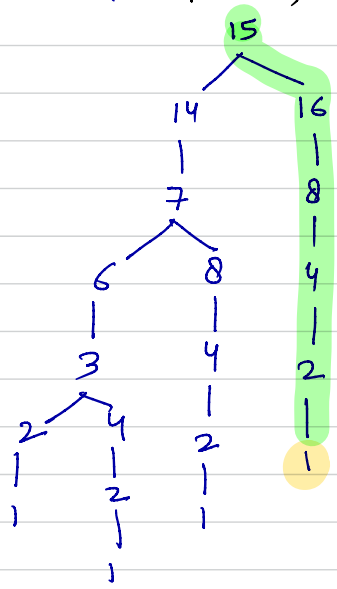
→ for case 4x+1, we will reach to the 1, in minimum steps using n-1 at every step having odd number.

2.) Take $4x+3$ case

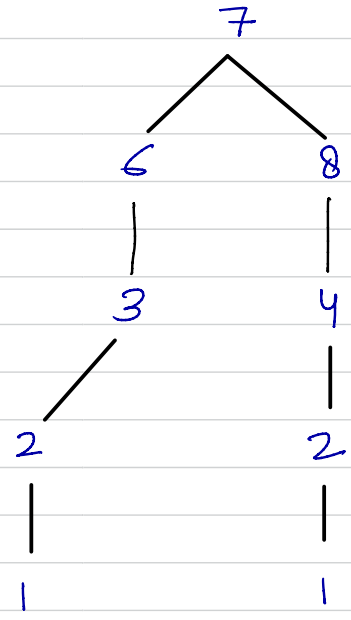


⇒ In above case if $x-1$ will give min answer then both − & + takes same step, but if $x+1$ will give min ans then + path takes min step, so + path is better than −.

ex- N=15,



ex - N=7



3.) If N=3 :-

If N=3, n-1 will give minimum steps.

Now, N%2, can be represented as ((n&1)==0)

(N%4==3) can be represented as ((N&3)==3)

(N%4==1) ⟹ ((N&3)==1)

NOTE:- N+1 can be overflow, so take long.

```
1 int countSteps(int n)
2 {
3   int count = 0;
4   while (n > 1) {
5       count++;
6
7       // num even, divide by 2
8       if ((n&1)==0)
9           n>>=1;
10
11      // num odd, n%4 == 1 or n==3(special edge case),
12      else if ((n & 3) == 1 || (n==3) )
13          n -= 1;
14
15      // num odd, n%4 == 3, increment by 1
16      else if((n&3) == 3)
17          n += 1;
18  }
19 return count;
20 }
21
```

Q 17.) Detect if two integers have opposite sign:

→ Signed integers in computer are store in 2's complement form. where MSB bit represent the sign of the number
1 → Negative integer
0 → Positive integer

So, if two integers have opposite sign, the XOR of two number will give negative number, so

```
if((A ^ B) < 0)
    "Opposite sign";
else
    "Not opposite sign";
```

```cpp
int main()
{
    int num1,num2;
    cin>>num1>>num2;

    if((num1^num2)<0){
        cout<<"Opposite Sign"<<endl;
    }
    else cout<<"Not Opposite sign"<<endl;

    return 0;
}
```

Q 18.) Add 1 to an integer:

→ We need to add one to a given number without using +, −, ×, /, ++, −− .... operator.

Method 1:- flip all the bits after rightmost 0 bit.
→ flip the rightmost 0 bit.
⇒ we will get our answer.

ex − 5

101

↓

110

= 6

ex − 11

1011

↓

1100

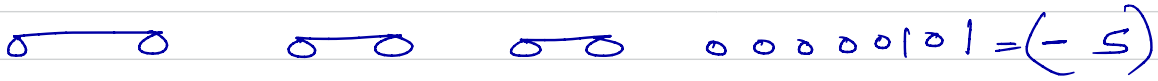
= 12

```
int addOnetoint(int num)
{   int leftshift = 1;

    // Flip all the set bits until we find a 0
    while(num & leftshift)
    {   num = num ^ leftshift;
        leftshift <<= 1;
    }

    // flip the rightmost 0 bit
    num = num ^ leftshift;
    return num;
}
```

METHOD-2:- Let's say x is the numerical value of a number. then $\sim x = -(x+1)$ (in 2's complement)
so to get $(x+1) \Rightarrow -(\sim x)$.

ex- 4

2's complement of $\sim x$ =



$-(\sim x) = -(-5) = 5$

```
1 int addOnetoint(int num)
2 {
3     return (-(~num));;
4 }
```

Q 19.) Find Xor of a number without using XOR operator:

Method 1: Traverse all bits one by one.
for every pair of bits, check if both are same or not.

```
int XOR(int num1, int num2)
{
    int res = 0;

    // Assuming 32-bit Integer
    for (int i = 31; i >= 0; i--)
    {
    bool bit1 = num1 & (1 << i);
    bool bit2 = num2 & (1 << i);

    bool XoredBit = (bit1 & bit2) ? 0 : (bit1 | bit2);

    res <<= 1;
    res |= XoredBit;
    }
    return res;
}
```

Method 2: we can use the property of XOR.

$$a \wedge b = a\bar{b} + b\bar{a}$$

```
1 int XOR(int num1, int num2)
2 {
3     int res = (num1 & (~num2)) | ((~num1 )& num2);
4     return res;
5 }
```

Q 20.) Determine if two integers are equal without using comparison and arithmetic operators

Method 1: Using XOR :-

```
if (a ^ b)
    "Not same";
else
    "Same";
```

```cpp
void areSame(int num1, int num2)
{
    if (num1 ^ num2)
        cout << "Not Same";
    else
        cout << "Same";
}
```

Method :- 2

```
if ((a & ~b) == 0)
    "same";
else
    "Not same";
```

```cpp
void areSame(int num1, int num2)
{
    if ((num1 & (~num2))==0)
        cout << "Same";
    else
        cout << "Not  Same";
}
```

Q 21.) Find minimum or maximum of two integers without using branching

Method 1 :-

Let us assume 'b' is minimum and 'a' is maximum among 'a' and 'b'.('a' < 'b') is the comparison we will be using. We will calculate minimum and maximum as follows:

- We can write the minimum as 'b' ^ (('a' ^ 'b') & - ('a' < 'b')).
  - If 'b' is minimum 'a' < 'b' comes out to be all zeroes. ('a' ^ 'b') & '0' comes out to '0'. Therefore expression value comes out to be 'b' finally which is the minimum.
  - If 'a' is minimum 'a' < 'b' comes out to be all ones. ('a' ^ 'b') & '1' comes out to ('a' ^ 'b'). Therefore expression value comes out to be 'a' finally which is the minimum.

- We can write maximum as 'a' ^ (('a' ^ 'b') & - ('a' < 'b')).
  - If 'a' is maximum 'a' < 'b' comes out to be all zeroes. ('a' ^ 'b') & '0' comes out to '0'. Therefore expression value comes out to be 'a' finally which is the maximum.
  - If 'b' is maximum 'a' < 'b' comes out to be all ones. ('a' ^ 'b') & '1' comes out to ('a' ^ 'b'). Therefore expression value comes out to be 'b' finally which is the maximum.

Method-2

- We can write the minimum as 'b' + (('a' - 'b') & (('a' - 'b') >> (noOfBitsInInt - 1))). On right shifting 'a' - 'b' by 1 less than no of bits in int we get the most significant bit.
  - If 'b' is minimum 'a' - 'b' comes out to be non-negative and on right shifting, we get '0'. ('a' - 'b') & '0' comes out to '0'. Therefore expression value comes out to be 'b' finally which is the minimum.
  - If 'a' is minimum 'a' - 'b' comes out to be negative and on right shifting, we get '1'. ('a' - 'b') & '1' comes out to ('a' - 'b'). Therefore expression value comes out to be 'a' finally which is the minimum.

- We can write maximum as 'a' - (('a' - 'b') & (('a' - 'b') >> (noOfBitsInInt - 1))). On right shifting 'a' - 'b' by 1 less than no of bits in int we get most significant bit.
  - If 'a' is maximum 'a' - 'b' comes out to be non-negative and on right shifting, we get '0'. ('a' - 'b') & '0' comes out to '0'. Therefore expression value comes out to be 'a' finally which is the maximum.
  - If 'b' is maximum 'a' - 'b' comes out to be negative and on right shifting, we get '1'. ('a' - 'b') & '1' comes out to ('a' - 'b'). Therefore expression value comes out to be 'b' finally which is the maximum.

Reference - codestudio

LEETCODE PROBLEMS:

Q 1.) Find missing and repeating number / Set mismatch:

You have a set of integers s, which originally contains all the numbers from 1 to n. Unfortunately, due to some error, one of the numbers in s got duplicated to another number in the set, which results in repetition of one number and loss of another number.

You are given an integer array nums representing the data status of this set after the error.

Find the number that occurs twice and the number that is missing and return them in the form of an array.

```cpp
vector<int> findErrorNums(vector<int>& arr) {
        int n  = arr.size();
        vector<int> temp(n+1,0);
       vector<int> ans(2,0);
       for(int i=0;i<n;i++){
           temp[arr[i]]++;
       }
       for(int i=1;i<=n;i++){
           if(temp[i]==0){
               ans[1] =i;
           }
           if(temp[i]==2){
               ans[0] = i;
           }
       }
       return ans;
```

Method -2 :

Ex - 1 1 2 3 4 6

sum of N numbers $= N \times \frac{(N+1)}{2} = S$

Summation of square of a number $= \frac{n \times (n+1) \times (2n+1)}{6} = P$

$\Rightarrow$ S - array elements

$(1+2+3+4+5+6) - (1+2+3+4+6+1)$

$5 - 1$

$= x - y$

$\Rightarrow S^2$ - (array elements)^2

$= 1^2+2^2+3^2+4^2+5^2+6^2-(1^2+2^2+3^2+4^2+6^2+1^2)$

$= 25 - 1 \quad (x^2 - y^2)$

$= 24$

$$\left[\begin{array}{l} x^2-y^2 = 24 \\ x-y = 4 \end{array}\right] \rightarrow (x+y)(x-y) = 24$$

$\Rightarrow (x+y)(x-y) = P$

$\Rightarrow (x+y) \cdot S = P$

$\Rightarrow (x + x-S) \cdot S = P$

$\Rightarrow 2x = \frac{P}{S} + S$

$\Rightarrow x = \left(\frac{P}{S} + S\right)/2;$

$\Rightarrow y = x - S$

$x - y = S$

$y = x - S$

```cpp
vector<int>missing_repeated_number(const vector<int> &nums) {
    long long int len = nums.size();

    long long int S = (len * (len+1) ) /2;
    long long int P = (len * (len +1) *(2*len +1) )/6;
    long long int missingNumber=0, repeating=0;

    for(int i=0;i<nums.size(); i++){
       S -= (long long int)nums[i];
       P -= (long long int)nums[i]*(long long int)nums[i];
    }

    missingNumber = (S + P/S)/2;

    repeating = missingNumber - S;

    vector <int> ans;

    ans.push_back(repeating);
    ans.push_back(missingNumber);
    return ans;
}
```

Method - 3: Using XOR

ex- 1 1 2 3 4 6

→ Do XOR of all the elements and elements from (1 to N).

⇒ $(1 \wedge 1 \wedge 2 \wedge 3 \wedge 4 \wedge 6) \wedge (1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 \wedge 6)$

⇒ $1 \wedge 5$

1 ↓ Repeating ↓ x

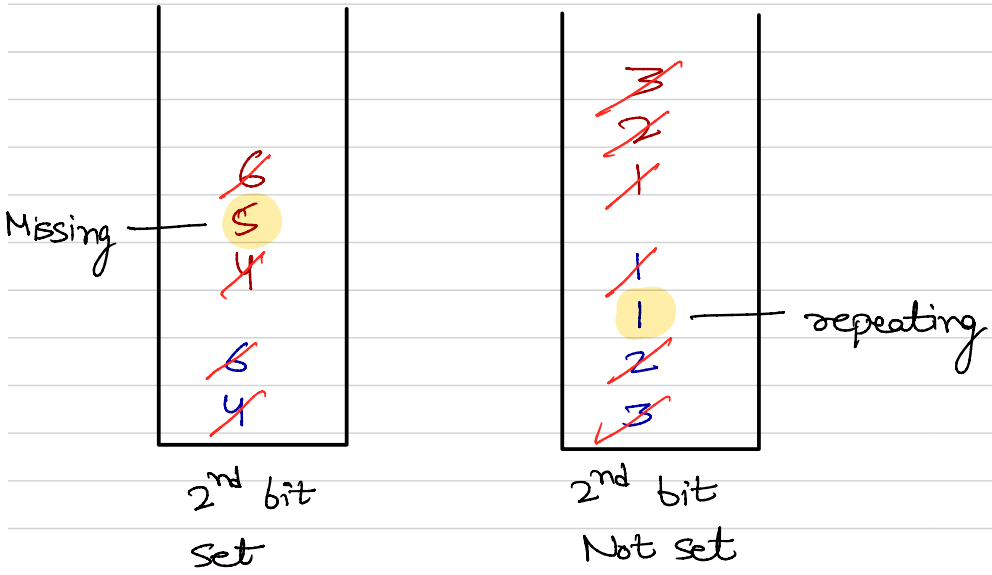
5 ↓ missing ↓ y

⇒ $x \wedge y = 4 = 100$

→ XOR of 1 and 0 is only 1, if XOR of any bit is set means the index bit is either set on X or on Y, but not on both.

→ We can take any set bit, but here we will take rightmost set bit and divide the elements based on the set bit or unset bit at the Rightmost set bit in XOR.

→ $x \wedge y = 4$ (100) 2^{nd} bit is set.



```cpp
//third method : Optimal Approach using XO
    vector<int> findErrorNums(vector<int>& nums) {
      int xor1;

    /* Will have only single set bit of xor1 */
    int setbitposition;

    int x = 0; // missing
    int y = 0; // repeated
    int n = nums.size();

    xor1 = nums[0];

    /* Get the xor of all numsay elements */
    for (int i = 1; i < n; i++)
        xor1 = xor1 ^ nums[i];

    /* XOR the previous result with numbers from 1 to n */
    for (int i = 1; i <= n; i++)
        xor1 = xor1 ^ i;

    /* Get the rightmost set bit in setbitposition */
    setbitposition = xor1 & ~(xor1 - 1);

    for (int i = 0; i < n; i++) {
        if (nums[i] & setbitposition)
            /* nums[i] belongs to first set */
            x = x ^ nums[i];
        else
            /* nums[i] belongs to second set */
            y = y ^ nums[i];
    }

    for (int i = 1; i <= n; i++) {
        if (i & setbitposition)
            /* i belongs to first set */
            x = x ^ i;
        else
            /* i belongs to second set */
            y = y ^ i;
    }

    // NB! numbers can be swapped, maybe do a check ?
    int x_count = 0;
    for (int i=0; i<n; i++) {
        if (nums[i]==x)
            x_count++;
    }

    if (x_count==0)
        return {y, x};

    return {x, y};
    }
```

Q 2.) Maximum Product of Word Lengths (Amazon, google)

Given a string array `words`, return *the maximum value of* `length(word[i]) * length(word[j])` *where the two words do not share common letters*. If no such two words exist, return `0`.

Example 1:

```
Input: words = ["abcw","baz","foo","bar","xtfn","abcdef"]
Output: 16
Explanation: The two words can be "abcw", "xtfn".
```

Example 2:

```
Input: words = ["a","ab","abc","d","cd","bcd","abcd"]
Output: 4
Explanation: The two words can be "ab", "cd".
```

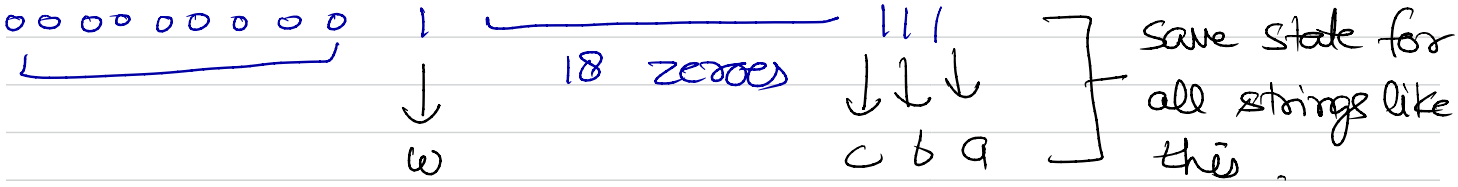
Example 3:

```
Input: words = ["a","aa","aaa","aaaa"]
Output: 0
Explanation: No such pair of words.
```

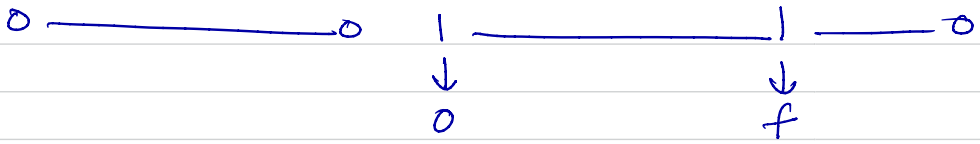
ex- words = ["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]

→ Set the bit corresponding to particular character in the string.

abcw →
- a = set 0th bit
- b = set 1st bit
- c = set 2nd bit
- w = set 22nd bit (119 - 27 = 22)



foo
- f - set 5th bit
- o - set 14th bit
- o - set 14th bit



⇒ Do & of states of both strings, if no character is common, then find the max of ans & word size of both string.

```cpp
int maxProduct(vector<string>& words) {
  int n = words.size();
  int ans = 0;
  vector<int> state(n);
  for(int i=0;i<n;i++){

      for(char ch:words[i]){
        //set the bits corresponding to the particular character
          state[i] |= 1<<(ch-'a');
      }

      for(int j=0;j<i;j++){
          //if no common letter between two strings, then find max
          if(!(state[i] & state[j])){
              int currans = words[i].size()* words[j].size();
              ans = max(ans,currans);
          }
      }
  }
        return ans;
    }
```

Q 3.) Concatenation of Consecutive Binary Numbers

Given an integer n, return the decimal value of the binary string formed by concatenating the binary representations of 1 to n in order, modulo 109 + 7.

ex- N = 1, "1"
output = 1

N = 3,
1 = 1
2 = 10
3 = 11
after concatenation : 11011
output = 27.

ex- N = 4, Range will be from 1 to 4
String = 11011100
⇓ decimal value % (10^9+7)
220

Method 1:-
1. Create String
2. Parse string and find the decimal value.

NOTE :- String creation as well as parsing will consume a lot of time.
⇒ String concatenation is a costly process.

→ We can optimize the above approach.

NOTE - If we observe, appending binary string at the end of an existing string will cause a left-shift effect by the size of the appended string.

ex= N=4

OBSERVATION

N	String	Calculation	Value
1	1	—	1
2	1 10	1×2^2 + 2 (2 — size of appended string; 2 — value of appended string)	6
3	1 10 11	$6 \times 2^2 + 3$	27
4	1 10 11 100	$27 \times 2^3 + 4$	220

→ if previously calculated value = P

current Number = x

∴ No. of digits in x = $1 + \log_2(x)$ = D

∴ New decimal value = $P \times 2^D + x$

$(P << D) + x$

```
int concatenatedBinary(int n) {
        long long int val = 0;
        int i=1;
        int mod = 1e9 + 7;
        while(i<=n){
            int digits = (1+log2(i));
            val = ((val<<digits)%mod + i)%mod;
            i+=1;
        }
        return val;
    }
```

TC = O(N)

Q 4.) Check if a String Contains all binary codes of size k

Given a binary string `s` and an integer `k`, return `true` *if every binary code of length* `k` *is a substring of* `s`. Otherwise, return `false`.

Example 1:

```
Input: s = "00110110", k = 2
Output: true
Explanation: The binary codes of length 2 are "00", "01", "10" and "11". They can be all found as substrings at indices 0, 1, 3 and 2 respectively.
```

Example 2:

```
Input: s = "0110", k = 1
Output: true
Explanation: The binary codes of length 1 are "0" and "1", it is clear that both exist as a substring.
```

Example 3:

```
Input: s = "0110", k = 2
Output: false
Explanation: The binary code "00" is of length 2 and does not exist in the array.
```

Method 1 :- We can generate all the substrings of size K and compare one by one.

→ It will be very costly.

→ There will be 2^K substrings, so TC will be $O(2^K \times n)$.

Method 2 :- We can check all the substrings in the given string itself.

→ If we generate all the substrings of size K and store them in set, and check the size of set.

→ If size of set == 2^K, then we can say, we can generate all the substrings of size K, hence return true, else false.

ex - 10 0 0 10 1 1 1 0, K = 3

set = {100, 000, 001, 010, 101, 011, 111, 110}

↳ set size = 2^K = 8, so, TRUE.

ex - 0 1 0 0 1 0 1 1, K = 3

set = {010, 001, 011, 100, 101}

set size = 5 ≠ 2^K, so return FALSE

```cpp
bool hasAllCodes(string s, int k) {
    int n = s.size();
    if(k>n) return false;

    unordered_set<string> set;
    for(int i = 0; i <= n - k; i++)
        set.insert(s.substr(i, k));
    return set.size() == (1 << k);
    }
```

Q 5.) Find the Duplicate Number

Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only one repeated number in nums, return this repeated number.

You must solve the problem without modifying the array nums and uses only constant extra space.

Method 1: Sorting (NlogN)

→ We can sort the array, start comparing the array elements.

ex - 3 1 2 4 6 5 6 7

Sorting :- 1 2 3 4 5 6 6 7

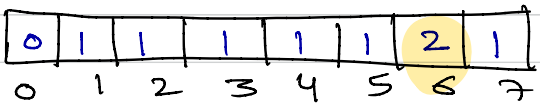
TC = O(NlogN)
SC = O(1)

```cpp
int findDuplicate(vector<int>& nums) {
        int n =  nums.size();
        int ind=0;
        sort(nums.begin(),nums.end());

        for(int i =1;i<n;i++)
        {
            if(nums[i]==nums[i-1])
            {
                return ind =  nums[i];
                break;
            }
        }
        return ind;
        }
```

Method 2 :- Use extra space / Hashing

→ Store the count of every element in a vector.
→ Iterate through the count vector

ex- 3 1 2 4 6 5 6 7



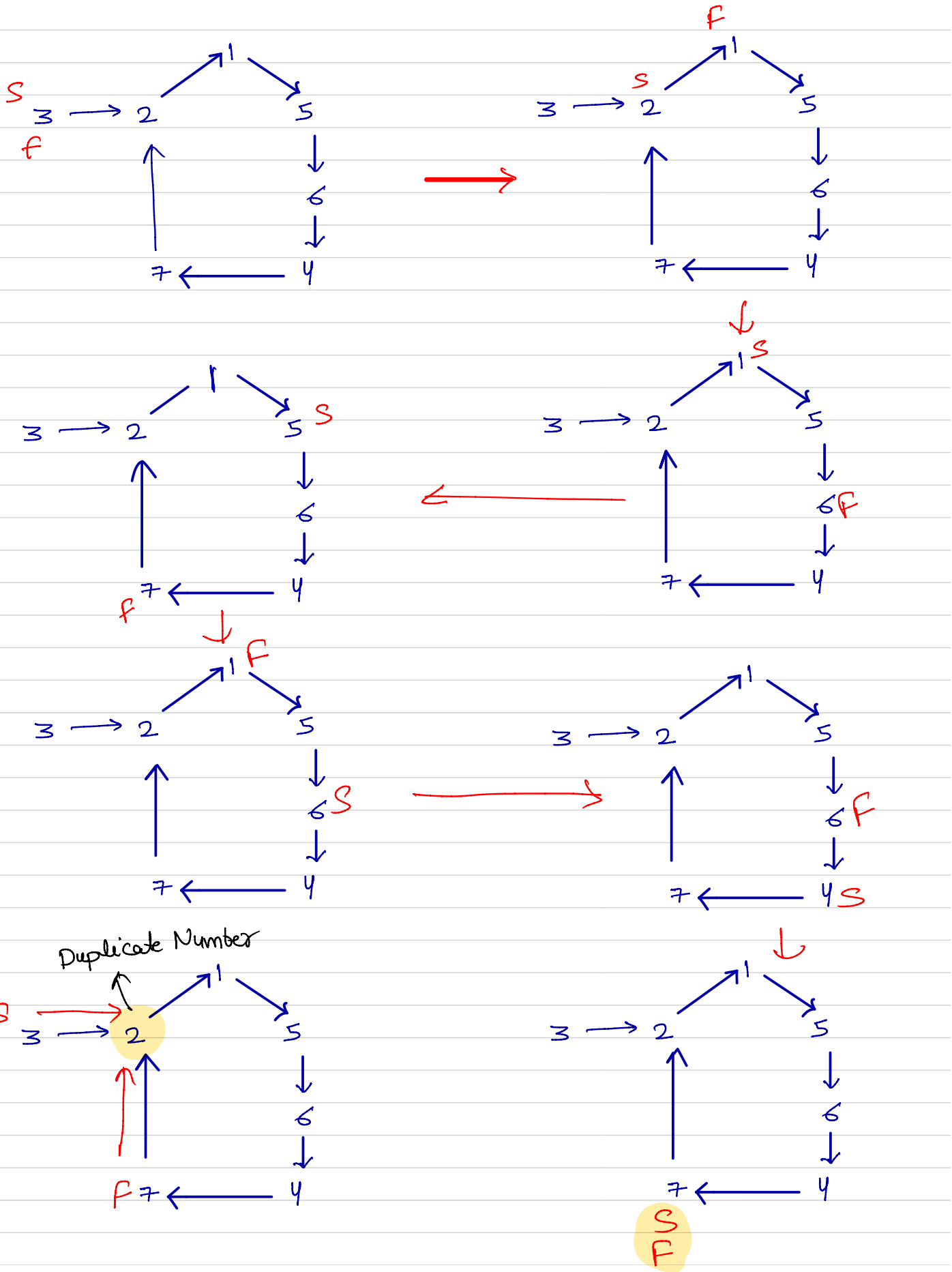
TC = O(N)
SC = O(N)

```cpp
int findDuplicate(vector<int>& nums) {
        int n =  nums.size();
        int ind=0;
        vector<int> cnt(n+1,0);
        for(int i=0;i<n;i++){
            cnt[nums[i]]= cnt[nums[i]]+1;
        }
        for(int i=1;i<=n;i++){
            if(cnt[i]>1) return i;
        }
    return 0;
    }
```

Method 3 : Linked List (slow/fast) cycle method :-

ex -

3	5	1	2	7	6	4	2
0	1	2	3	4	5	6	7



```cpp
int findDuplicate(vector<int>& nums) {
        int slow = nums[0];
        int fast = nums[0];
       do{
           slow = nums[slow];
           fast = nums[nums[fast]];
       } while(slow!=fast);

        slow = nums[0];
        while(slow!=fast){
            slow = nums[slow];
            fast = nums[fast];
        }
        return slow;
    }
```

TC = O(N)
SC = O(1)

believe in the process

Kapil Yadav