Contest 1: A... Try HackMD (https://hackmd.io?utm_source=view-page&utm_medium=logo-nav)

Quick Revision Videos

https://drive.google.com/drive/folders/1Ww4p2egf541Vz8JUj0KTVBCyqYFZ1RMB?usp=share_link (https://drive.google.com/drive/folders/1Ww4p2egf541Vz8JUj0KTVBCyqYFZ1RMB?usp=share_link)
https://drive.google.com/file/d/1zEJN4CuoTR7K6b0ecEiKzEP1Rzp3jHoR/view?usp=share_link (https://drive.google.com/file/d/1zEJN4CuoTR7K6b0ecEiKzEP1Rzp3jHoR/view?usp=share_link)

Arrays 1: One Dimensional

Problem 1 - Find Maximum Subarray Sum

Problem Statement

Given an integer array A, find the maximum subarray sum out of all the subarrays.

```
How other than this, can Kadane's be asked ?


  Imagine you're a financial analyst tracking the daily stock prices of a company. Your t
  In this scenario, you want to determine the duration (number of consecutive days) in wh
```

Examples

Example 1:
For the given array A with length N,

Index	0	1	2	3	4	5	6
Array	-2	3	4	-1	5	-10	7

Output:

```
Max Sum: 11
Subarray: 3 4 -1 5
```

Optimized Solution - Kadanes Algorithm

Solution:

Say we maintain two variables, max_sum and cur_sum. max_sum keeps track of maximum sum found till now and cur_sum keeps track of the current sum.

We will consider elements one by one. If there's a positive element, we can just add it to our currsum, but a negative element is considered only if the overall sum is positive because in the future if positives come, they may further increase this positivity(sum).

If after adding a negative element, the curr_sum becomes < 0, then we shall not carry it forward since it will further decrease any number that'll be added to it, curr_sum will be set to 0 in this case.

Example -

```
A[ ] = { -2, 3, 4, -1, 5, -10, 7 }
```

Answer array: 3, 4, -1, 5

Explanation:
3+4 = 7
7 + (-1) = 6 (still positive)
6+5 = 11 (higher than 7)

Dry Run

```
    0   1    2    3  4  5   6  7   8
{ -20, 10, -20, -12, 6, 5, -3, 8, -2 }
```

i	A[i]	currSum	maxSum	
0	-20	-20	-20	reset the currSum = 0 and do not take it forward since adding a negative will make it more negative and adding a positive will reduce positivity of that element.
1	10	10	10	
2	-20	10 + (-20)= -10	10	reset currSum = 0
3	-12	-12	10	reset currSum = 0
4	6	6	10	
5	5	6 + 5	11	
6	-3	11 - 3 = 8	11	Keep currSum as 8 only since if we find a positive, it can increase the sum
7	8	8 + 8 = 16	16	
8	-2	16 - 2 = 14	16	Keep currSum as 14 only since if we find a positive, it can increase the sum

Final maxSum = 16

Pseudocode

```
int maximumSubarraySum(int[] arr, int n) {
       int maxSum = Integer.MIN_VALUE, currSum = 0;

       for (int i = 0; i <= n - 1; i++) {
           currSum += arr[i];

           if (currSum > maxSum) {
           maxSum = currSum;
           }

           if (currSum < 0) {
           currSum = 0;
           }
       }

       return maxSum;
   }
```

Time and Space Complexity

TC - O(n)
SC - O(1)

Problem 2 - Perform multiple Queries from index i to j

Given an integer array A such that all the elements in the array are 0. Return the final array after performing multiple queries

`Query: (i, j, x)` : Add x to all the elements from index i to j

Given that `i <= j`

Example:

Let's take an example, save we have the zero-filled array of size 7 and the queries are given as
q1 = (1, 3, 2)
q2 = (2, 5, 3)
q3 = (5, 6, -1)

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Arr[7]	0	0	0	0	0	0	0
V1		2	2	2			
V2			3	3	3	3	
V3						-1	-1
Ans	0	2	5	5	3	2	-1

Optimized Solution

Idea

- We can add the value at the starting index and subtract the same value just after the ending index which will help us to only carry the effect of **+val** till a specific index.
- Following the index(k) where we have done **-val**, the effect will neutralise.

Pseudocode:

```
zeroQ(int N, int start[ ], int end[ ], int val[ ]){
    long arr[N] = 0;
    for(int i = 0; i < Q; i++){

        //ith query information: start[i], end[i], val[i]
        int s = start[i], e = end[i], v = val[i];

        arr[s] = arr[s] + v;

        if(e < n - 1){
            arr[e + 1] = arr[e + 1] - v;
        }
    }

    //Apply cumm sum a psum[] on arr
    for (i = 1; i < N; i++){
        arr[i] += arr[i - 1];
    }

    return arr;
}
```

Time and Space complexity for optimised solution

TC: O(Q + N)
SC: O(1)

Problem 3 - Rain Water Trapping

Given N buildings with height of each building, find the rain water trapped between the buildings.

Example Explanation

Example:
arr[] = {2, 1, 3, 2, 1, 2, 4, 3, 2, 1, 3, 1}

We now need to find the rainwater trapped between the buildings

> **Image Not Showing**
> Possible Reasons
> - The image was uploaded to a note which you don't have access to
> - The note which the image was originally uploaded to has been deleted
>
> Learn More → (https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg)

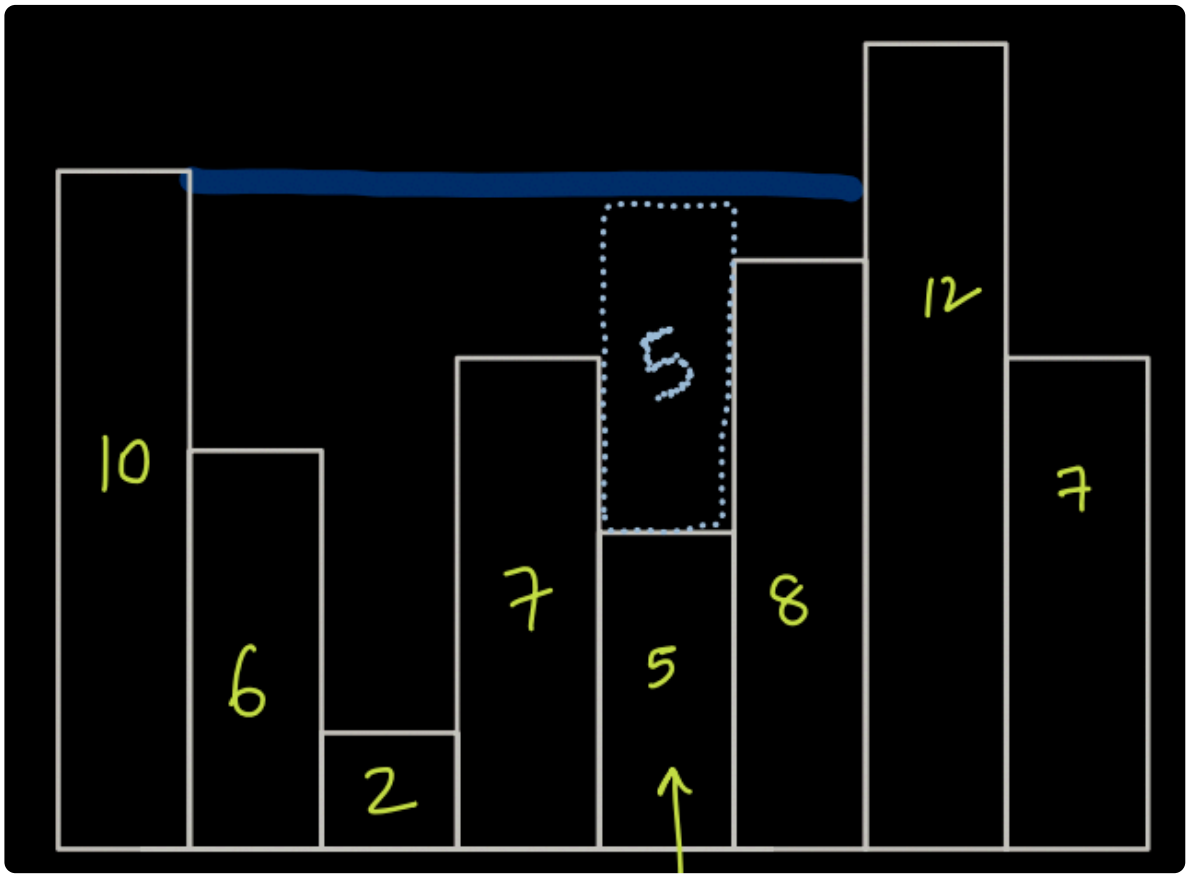
Ans: 8

Idea:

If we get units of water stored over every building, then we can get the overall water by summing individual answers.

Water over every building depends on the height of the minimum of the largest buildings on either sides.

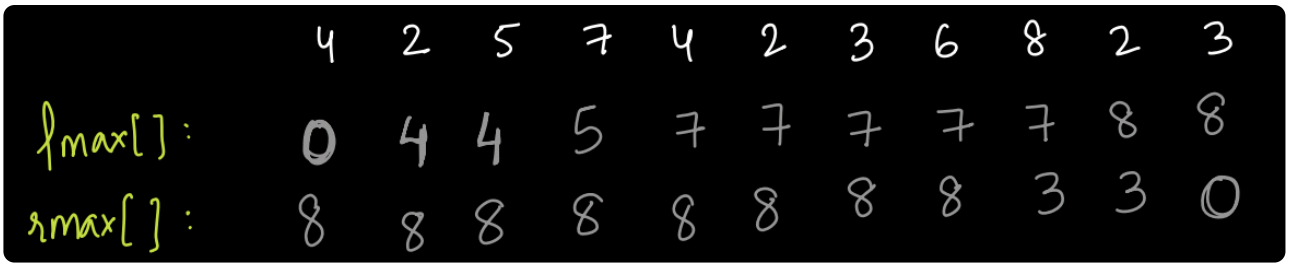
Example:
Water stored over building 5 depends on minimum of the largest building on either sides.
i.e, min(10, 12) = 10
Water stored over 5 is 10 - 5 = 5 units.



Optimised Approach

We can store the max on right & left using carry forward approach.

- We can take 2 arrays, lmax[] & rmax[].
- Below is the calculation for finding max on left & right using carry forward approach.
- This way, we don't have to find max for every element, as it has been pre-calculated.



Pseudocode

```
ans = 0;

int lmax[N] = {0};
for(int i = 1; i < N; i++) {
    lmax[i] = max(lmax[i - 1], A[i - 1]);
}

int rmax[N] = {0};
for(int i = N - 2; i >= 0; i--) {
    rmax[i] = rmax[i + 1], A[i + 1];
}

for(int i = 1; i < N - 1; i++) {
    water = min(lmax[i], rmax[i]) - A[i];

    if(water > 0) {
        ans += water;
    }
}
```

Time & Space

Time - O(N) {Since we have precalculated lmax & rmax}
Space - O(N)

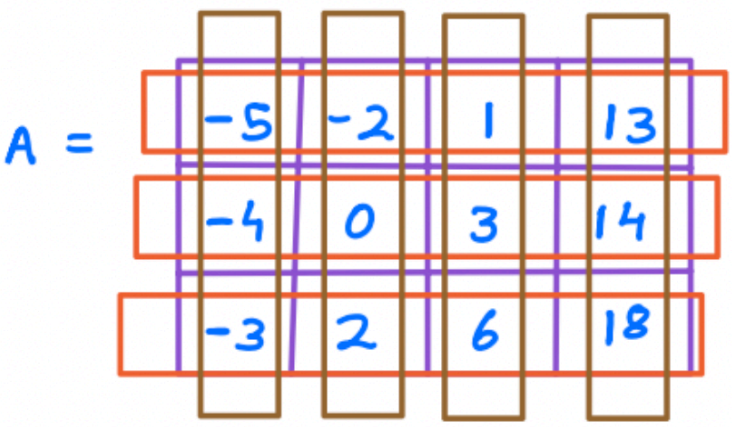
Arrays 2: Two Dimensional

Problem 1 - Search in rowwise and colwise sorted matrix

Given a row wise and column wise sorted matrix, find out whether element **k** is present or not.

Example

Observe that rows and columns are both sorted.



Test Case 1

13 => Present (true)

Test Case 2

15 => Not Present (false)

Idea

- We shall exploit the property of the matrix being sorted.
- Start with the cell from where we can decide the next step.
  Example:

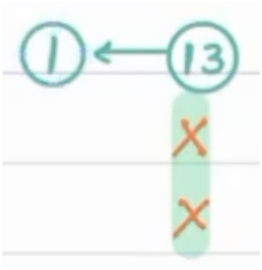
-5	-2	1	13
-4	0	3	14
-3	2	6	18

Search for: 0

Say we stand at **top right cell i.e, 13**.
Now,**13 > 0**, should we go left or down ? Can we decide ?
Yes, if we move down the elements are > 13, but we are looking for an element < 13, so we should move left.

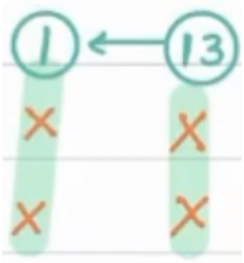
It means, all elements below 13, can be neglected.

Move Left

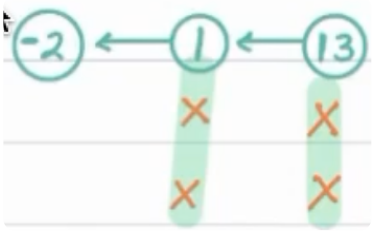


Now, where shall we move ?

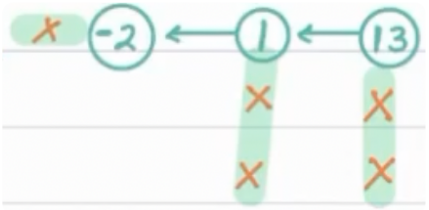
Since, **1 > 0**, again all elements below 1 are greater than 1, hence can be neglected.



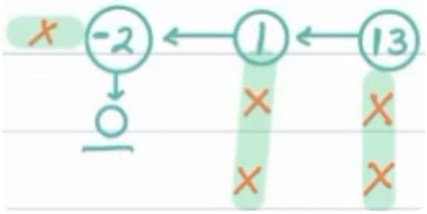
Move Left



Now, **-2 < 0**, all elements on left of -2 are lesser than -2, hence can be neglected.



Move Down



Approach

- We can start at top right cell.
- If A[i][j] < K, move down, else move left.
- Repeat until the element is found, our the search space is exhausted.

NOTE: We could have also started at bottom left cell.

Pseudocode

```
int i = 0, j = M - 1
while(i < N && j>= 0){
    if(arr[i][j] == K){
        return true;
    } else if(arr[i][j] < K){
        i++; //move down; next row
    } else{
        j--; //move left; previous column
    }
}
return false;
```

Time & Space Complexity

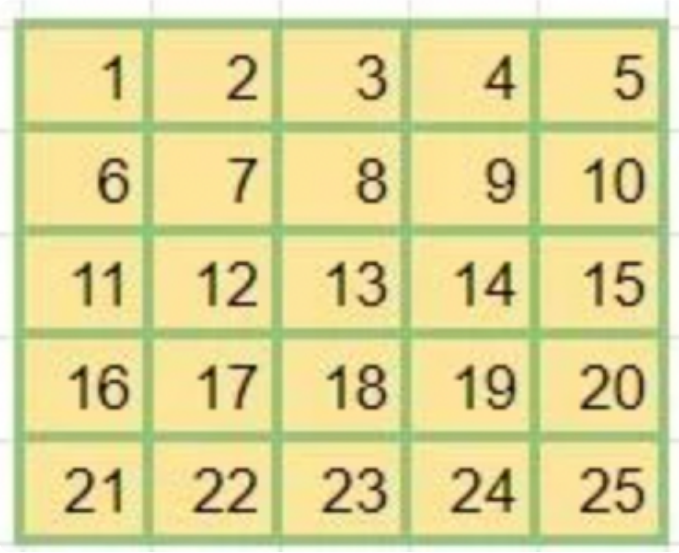
TC: O(M+N) since at every step, we are either discarding a row or a column. Since total rows+columns are N+M, hence Iterations will be N+M.
SC: O(1)

Problem 2 - Print Boundary Elements

Given an matrix of N X N i.e. Mat[N][N], print boundary elements in clockwise direction.

Example:

```
N = 5
```



Output: [1, 2, 3, 4, 5, 10, 15, 20, 25, 24, 23, 22, 21, 16, 11, 6]

Approach

- Print N-1 elements of first row from left to right
- Print N-1 elements of last column from top to bottom
- Print N-1 elements of last row from right to left
- Print N-1 elements of first column from bottom to top

Pseudocode

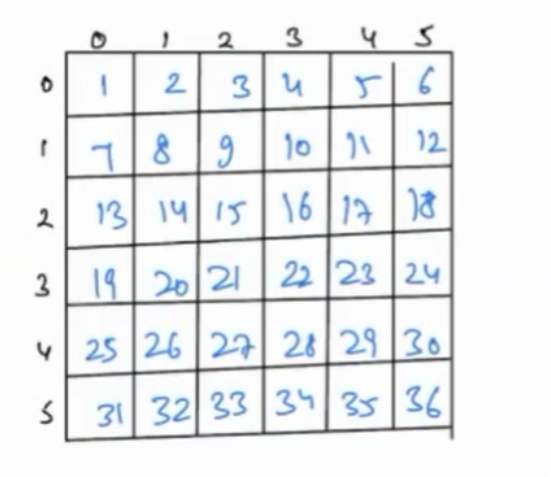
```
function printBoundaryElements(Mat[][],N) {
    i=0 j=0

    for(idx = 1 ; idx < N ; idx++ ){
       print(Mat[i][j] + ",")
       j++
    }

    for(idx = 1 ; idx < N ; idx++ ){
       print(Mat[i][j] + ",")
       i++
    }

    for(idx = 1 ; idx < N ; idx++ ){
       print(Mat[i][j] + ",")
       j--
    }

    for(idx = 1 ; idx < N ; idx++ ){
       print(Mat[i][j] + ",")
       i--
    }
}
```

Problem 3 - Spiral Matrix

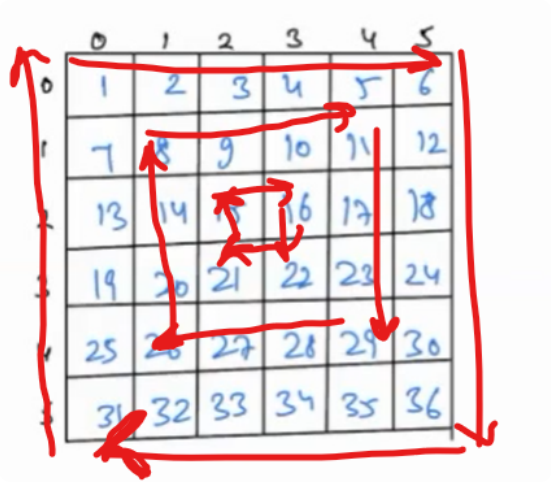
Given an matrix of N X N i.e. Mat[N][N]. Print elements in spiral order in clockwise direction.

Example

```
N = 5
```



Here is the depiction to understand the problem better:



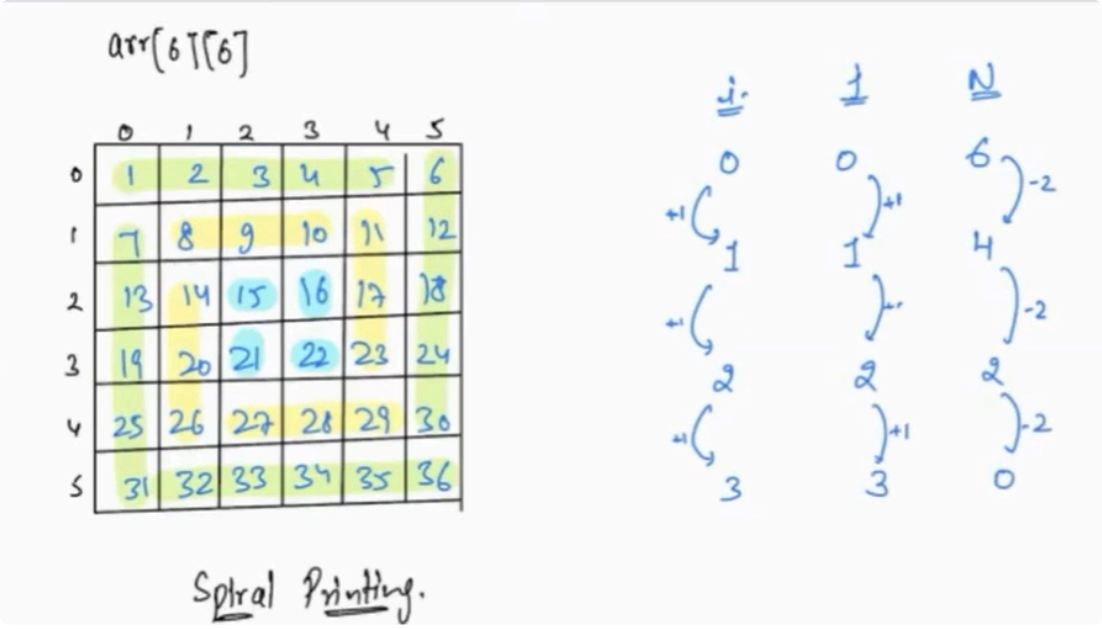

```
Solution = [1,2,3,4,5,6,12,18,24,30,36,35,34,33,32,31,25,19,13,7,8,9,10,11,17,23,29,28,
```

The red arrow represents direction of traversal(clockwise) and fashion in which elements are traversed.

Approach

- We can break the problem into several boundary printing problem discussed above
- So first print boundary of matrix of N x N

- Then we print boundary of next submatrix with top left element being (1,1) and Bottom right element being (N-2,N-2).
- After every boundary, to print the next boundary, N will be reduced by 2 and i & j will be incremented by 1.
- We do this till matrix of size least size is reached.



Boundaries of submatricies are highlighted in different color.

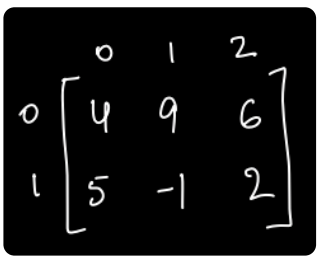
Edge Case

Will the above code work if matrix size is 1 ?
No, since the loops run N-1 times, therefore we have to handle it separately.

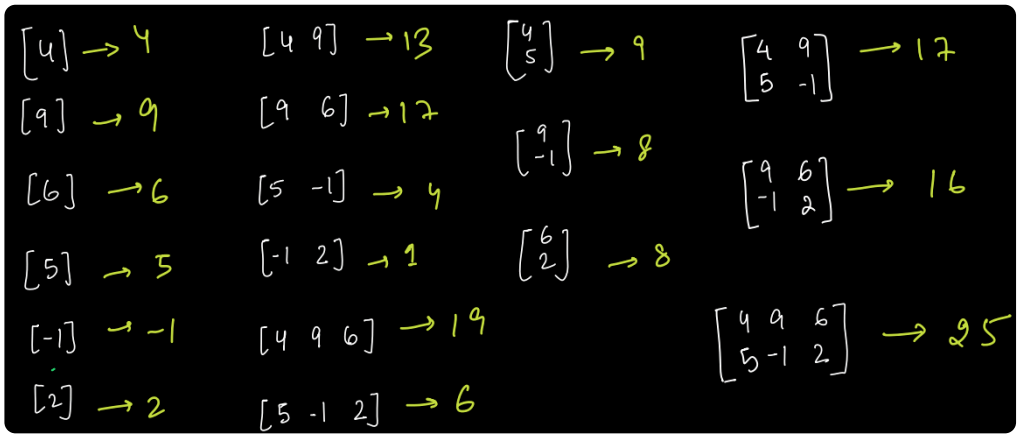
Problem 4 - Sum of all Submatrices Sum

Given a matrix of N rows and M columns determine the sum of all the possible submatrices.

Example:



All Possible sub-matrices are -



Total Sum = 166

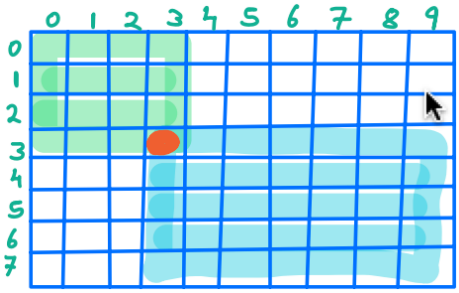
Approach

This question can be solved using **contribution technique**.

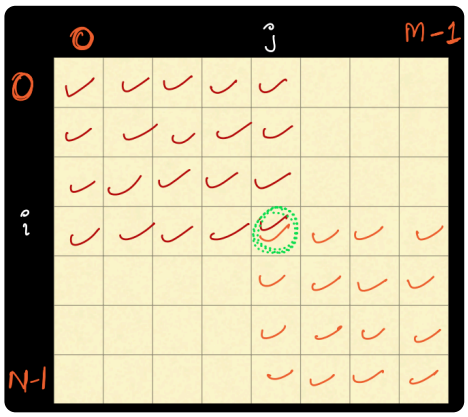
We will find that in how many submatrices a particular element is part of, then we just have to add up the individual results.

In what all submatrices, a particular element is part of ?

Let's look at the red cell in below figure.
If we combine all the top left cells(marked with green color) with all the bottom right cells(marked with blue color), then in all those submatrices, the red cell will be present.



How to find the number of TL cells and BR cells in which (i,j) is part of.



TOP LEFT:

rows: [0 i]

cols: [0 j]

total cells = (i+1) * (j+1)

BOTTOM RIGHT:

rows: [i N-1]

cols: [j M-1]

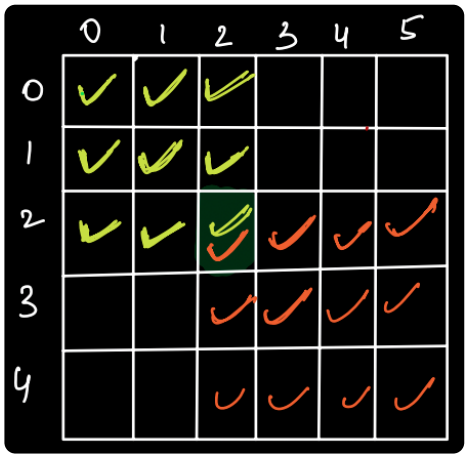
total cells = (N-i) * (M-j)

Now, to find the total submatrices of whish (i,j) is part of -

contribution of (i,j) = TOP LEFT * BOTTOM RIGHT

Every top left cell can be combined with every bottom right cell.

Example



For (2,2)

TOP LEFT:

3 * 3 = 9

BOTTOM RIGHT

(5-2) * (6-2) = 3 * 4 = 12

Total matrices of which (2,2) is part of 9 * 12.

Pseudocode

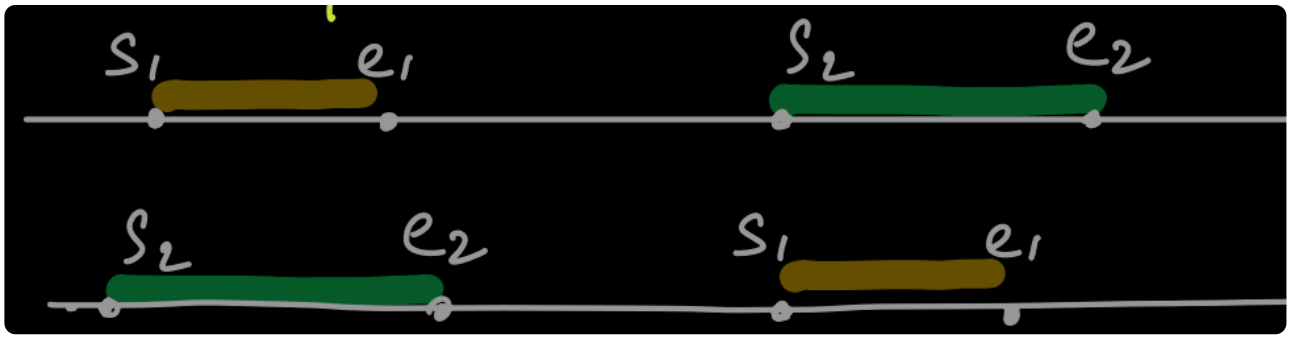
```
total = 0
for(int i=0; i<N; i++) {
    for(int j=0; j<M; j++) {

        top_left = (i+1) * (j+1);
        bottom_right = (N-i) * (M-j);

        contribution = A[i][j] * top_left * bottom_right;

        total += contribution
    }
}
return total
```

Arrays 3 - Interview Problems

Merge Intervals

Non-Overlapping Condition

Say there are two Intervals, I1 {s1 e1} & I2 {s2 e2}
Then the condition for them to not overlap is -




```
if(s2 > e1 || s1 > e2)
```

So, if above condition is not followed, it says that Intervals are overlapping!

How to merge overlapping Intervals ?

[I1.start , I1.end] & [I2.start , I2.end]
After merging -
[min(I1.start, I2.start) , max(I1.end, I2.end)]

Problem 1 Merge sorted Overlapping Intervals

Given a sorted list of overlapping intervals, sorted based on start time, merge all overlapping intervals and return sorted list.

Input:

Interval[] = {(0,2), (1,4), (5,6), (6,8), (7,10), (8,9), (12,14)}

Output:

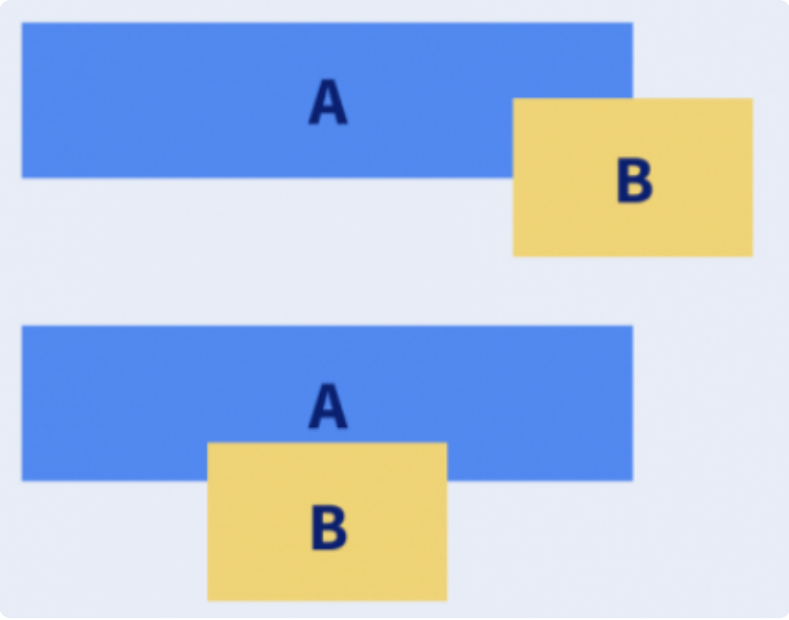
{(0,4), (5,10), (12,14)}

Explanation:

Interval 1	Interval 2		Answer Interval List
0-2	1-4	Overlapping	0-4
0-4	5-6	Not Overlapping	0-4, 5-6
5-6	6-8	Overlapping	0-4, 5-8
5-8	7-10	Overlapping	0-4, 5-10
5-10	8-9	Overlapping	0-4, 5-10
5-10	12-14	Not Overlapping	0-4, 5-10, 12-14

The Array Is Sorted Based on Start Time. What Is the Overlapping Condition?

Say start time of A < start time of B



Overlapping Condition -

If start of B <= end of A

Approach

- Create an array to store the merged intervals.
- If the current and ith intervals overlaps, merge them. In this case, set merged interval as current interval.
- Else, insert the current interval to answer array since it doesn't overlap with any other interval and update the current Interval to ith Interval.

Pseudocode

```
//Already a class/structure will be present for Interval
//We will only need to create an answer array of type Interval

list<Interval> ans;

// Current Segment
    int cur_start = A[0].start, cur_end = A[0].end;

    for (int i = 1; i < A.size(); i++) {

        // if i'th interval overlaps with current interval
        if (A[i].start <= cur_end) {
            // merging them
            cur_end = max(cur_end, A[i].end);
        }
        else {
            //adding current interval to answer.
            //create a new Interval
            Interval temp(cur_start, cur_end); //if struct is declared, otherwise if cl
            ans.push_back(temp);

            // update cur Interval to ith
            cur_start = A[i].start;
            cur_end = A[i].end;
        }
    }
    Interval temp(cur_start, cur_end);
    ans.push_back(temp);
    return ans;
```

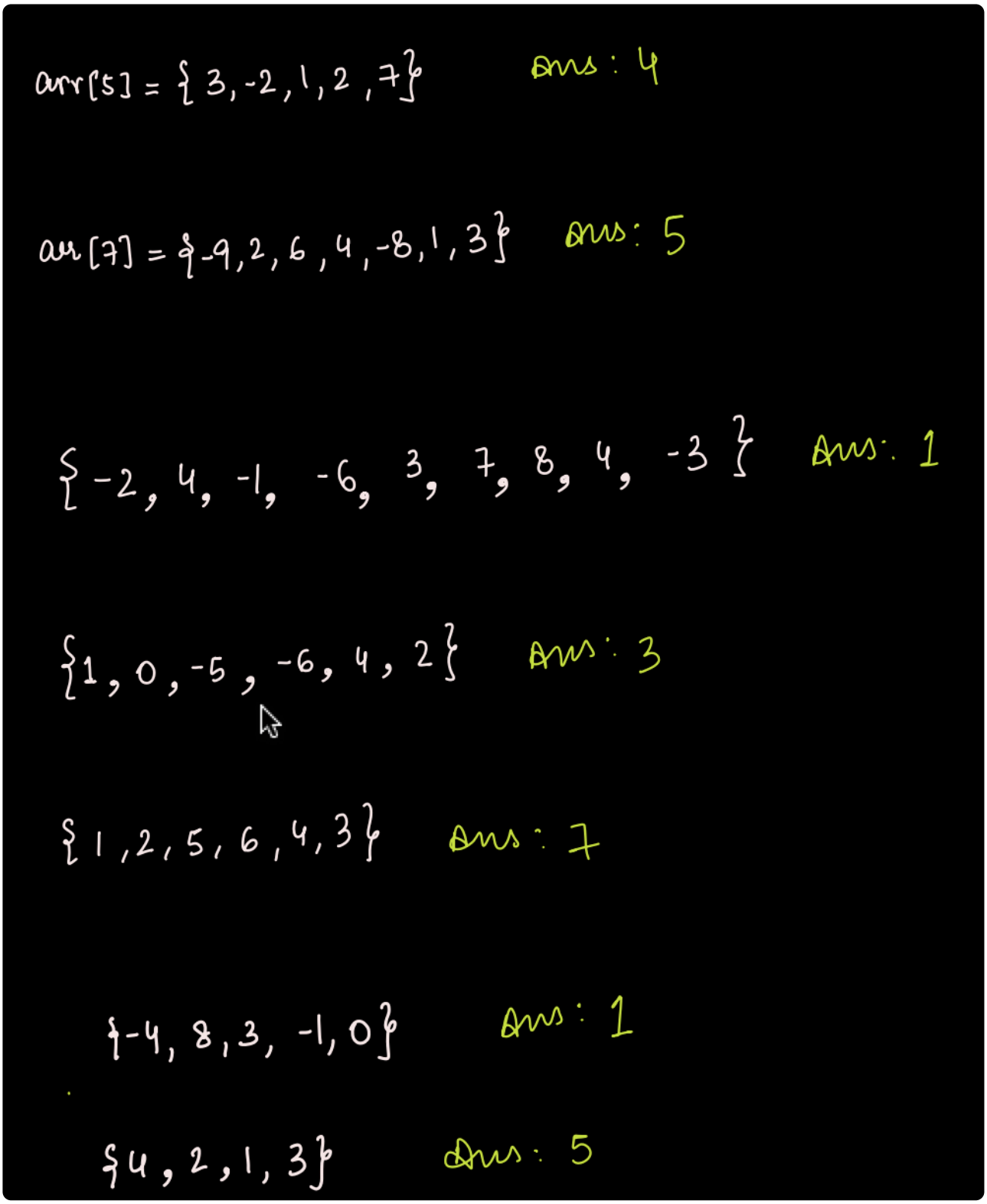
Time and Space Complexity

- TC - O(N)
- SC - O(1)

Problem 2 - Find First Missing Natural Number

Given an unsorted array of integers, Find first missing Natural Number.

Examples



Solution Idea

Can we utilise the fact that answer can be out of 1 to N+1?
If any number other than 1 to N is present, then missing is out of 1 to N only.
If all elements from 1 to N are present, then it will be N+1.

Say we start checking if 1 is present or not, we somehow want to mark the presence of 1.
Now, since we can't use extra space, so we can use indices to mark the presence of a number.

Index 0 can be used to mark presence of 1.
Index 1 can be used to mark presence of 2.
Index 2 can be used to mark presence of 3.
so on...

Now how do we mark the presence ?

One Solution:

```
We can set element at that index as negative.
```

But what if negative number is part of the input?
We can just change negative number to a number that is out of our answer range. It can be N+2.

A[ ] = {4, 0, 1, -5, -10, 8, 2, 6}

```
Since N = 8, answer can only be within range 1 to 9, so we can set A[i] <=0 as 10
```

A[ ] = {4, 10, 1, 10, 10, 8, 2, 6}

A[0]=4, set index 3 as negative

```
A[ ] = {4, 10, 1, -10, 10, 8, 2, 6}
```

A[1]=10 Ignore!
A[2]=1, set index 0 as negative

```
A[ ] = {-4, 10, 1, -10, 10, 8, 2, 6}
```

A[3]=10 Ignore!
A[4]=10 Ignore!
A[5]=8, set index 7 as negative

```
A[ ] = {-4, 10, 1, -10, 10, 8, 2, -6}
```

A[6]=2, set index 1 as negative

```
A[ ] = {-4, -10, 1, -10, 10, 8, 2, -6}
```

A[7]=6, set index 5 as negative

```
A[ ] = {-4, -10, 1, -10, 10, -8, 2, -6}
```

NOTE: Since we are marking elements as negative, so when checking presence of a certain number, we'll have to consider the absolute value of it.

Pseudocode

```
for(int i = 0; i < N; i++) {
    if(A[i] <= 0) {
        A[i] = N + 2;
    }
}

for(int i = 0; i < N; i++) {
    int ele = abs(A[i]);

    if(ele >= 1 && ele <= N) {
        int idx = ele - 1;
        A[idx] = -1 * abs(A[i]);
    }
}

for(int i = 0; i < N; i++) {
    if(A[i] > 0)return i + 1;
}
return N + 1;
```

Time and Space Complexity

- **TC -** O(N)
- **SC -** O(1)

Bit Manipulation 1

Truth Table for Bitwise Operators

Below is the truth table for bitwise operators.

a	b	a&b	a|b	a^b	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Left Shift Operator (<<)

- The left shift operator (<<) shifts the bits of a number to the left by a specified number of positions.
- The left shift operator can be used to multiply a number by 2 raised to the power of the specified number of positions.

In general, it can be formulated as:

```
a << n = a * 2^n
1 << n = 2^n
```

Right Shift Operator (>>)

- The right shift operator (>>) shifts the bits of a number to the right by a specified number of positions.
- When we right shift a binary number, the most significant bit (the leftmost bit) is filled with 0.
- Right shift operator can also be used for division by powers of 2.

In general, it can be formulated as:

```
a >> n = a/2^n
1 >> n = 1/2^n
```

Basic AND XOR OR Properties

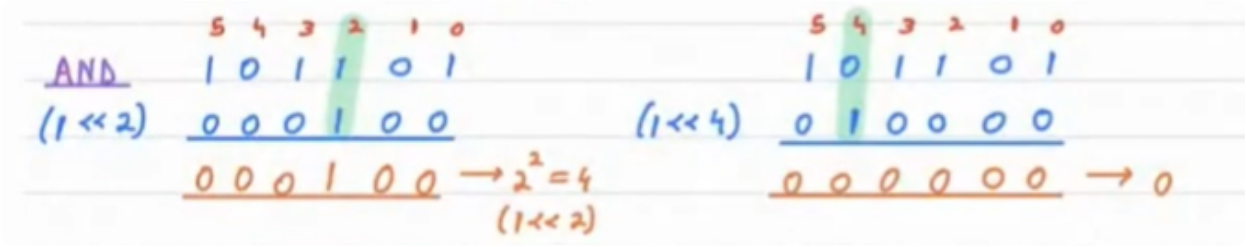
Even/Odd Number

- In binary representation, if a number is even, then its least significant bit (LSB) is 0.
    - **A & 1 = 0** (if A is even)
- Conversely, if a number is odd, then its LSB is 1.
    - **A & 1 = 1** (if A is odd)
- **A ^ 0 = A** (for all values of A)
- **A ^ A = 0** (for all values of A)

Check if ith bit is set or not - AND(&) Operator

```
X = (N & (1<<i))
```

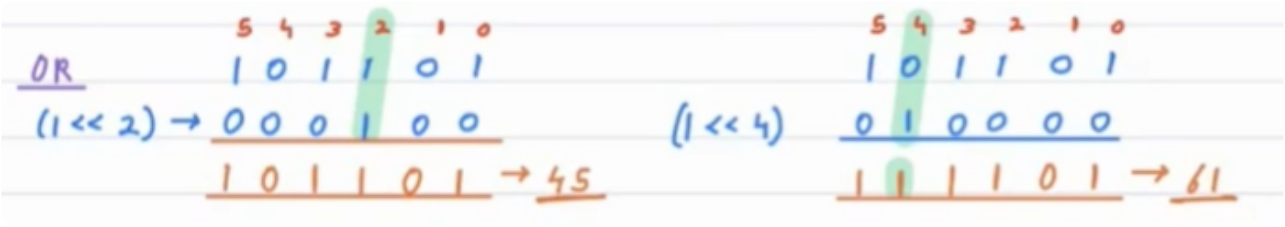
Example



Set ith Bit - OR(|) Operator

```
N = (N | (1<<i))
```

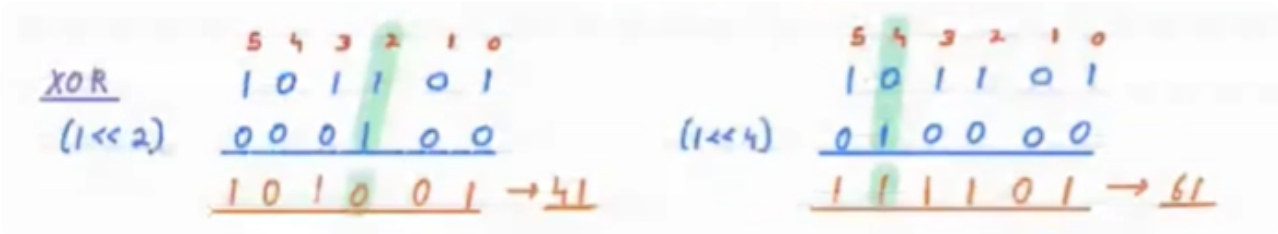
Example



Toggle ith Bit - XOR(^) Operator

```
N = (N ^ (1<<i))
```

Example



UNSET the i^{th} bit of the number N if it is set.

```
1  Function unsetbit(int N, int i){
2      int X = (N & (1<<i));
3      if(checkbit(N,i)){
4          N = (N ^ (1<<i));
5      }
6  }
```

Problem - Count the total number of SET bits in N

Given an integer **N**, count the total number of SET bits in **N**.

Iterate over all the bits of integer(which is maximum 32) and check whether that bit is set or not. If it is set then increment the answer(initially 0).

```
function checkbit(int N, int i){
    if(N & (1<<i)){
        return true;
    }
    else{
        return false;
    }
}

function countbit(int N){
    int ans = 0;
    for(i = 0; i < 32; i++){
        if(checkbit(N, i)){
            ans = ans + 1;
        }
    }
    return ans;
}
```

Here, checkbit function is used to check whether the i^{th} bit is set or not.

Bit Manipulation 2

Problem 1 - Single number 1

We are given an integer array where every number occurs twice except for one number which occurs just once. Find that number.

Example:

Input: [4,5,5,4,1,6,6]
Output: 1

```
only 1 occurs single time
```

Approach 1:

Since ^ helps to cancel out same pairs, we can use it.
Take XOR of all the elements.

Pseudocode

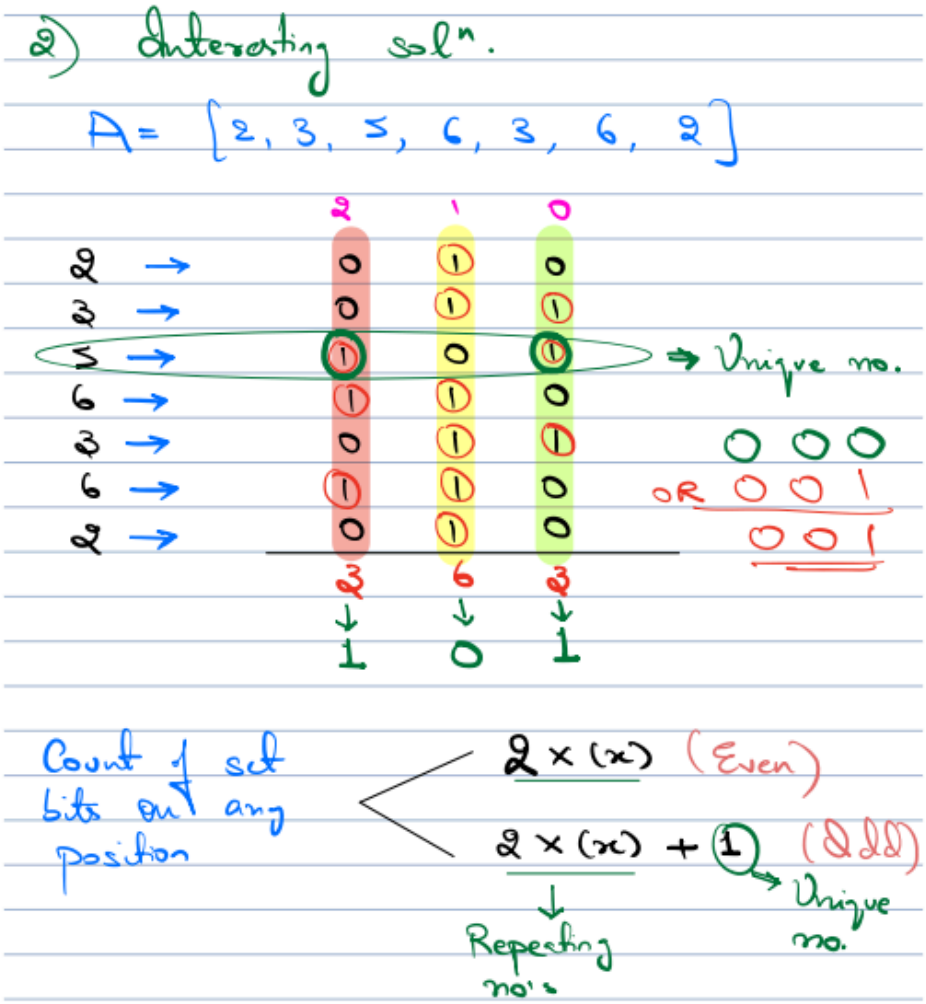
```
int x = 0;

for (int i = 0; i < arr.size(); ++i) {
    x = x ^ arr[i]; // XOR operation
}

print(x);
```

Approach 2:

Interesting Solution!
Bitwise operators work on bit level, so let's see how XOR was working on bits.
For that, let's write binary representation of every number.

Observations:

For every bit position, if we count the number of 1s the count should be even because numbers appear in pairs, but it can be odd if the bit in a single number is set for that position.



- We will iterate on all the bits one by one.
- We will count the numbers in the array for which the particular bit is set
- If the count is odd, in the required number that bit is set.

Pseudocode

```
int ans = 0;

for(int i = 0; i < 32; i++) { // go to every bit one by one
    int cnt = 0;

    for(int j = 0; j < arr.size(); j++) { // iterate on array

       // check if ith bit is set
       if((arr[j]& (1<<i))cnt++;
    }

    if(cnt & 1) // If the count is odd
        ans = ans | 1 << i; // set ith bit in ans
}

print(ans);
```

Time and Space Complexity

- **TC -** O(N)
- **SC -** O(1)

Similar Question:

Given an integer array, all the elements will occur thrice but one. Find the unique element.

Problem 2 - Single number 3

Given an integer array, all the elements will occur twice except two. Find those two elements.

Input: [4, 5, 4, 1, 6, 6, 5, 2]

Output: 1, 2

Solution Approach:

- Suppose if two unique numbers are **a** and **b**. Their XOR is **c**.
- We will find the position of any set bit in XOR c, it will denote that this bit is different in a and b.

- Now, we divide the entire array in two groups, based upon whether that particular bit is set or not.
- This way a and b will fall into different groups.
- Now since every number repeats twice, they will cancel out when we take XOR of the two groups individually leaving a and b.

Pseudocode

```
    int xorAll = 0;

    // XOR of all numbers in the array
    for (int i = 0; i < N; i++) {
        xorAll ^= A[i];
    }

    // Find the rightmost set bit position
    // Note: Any other bit can be used as well
    int pos;

    for(pos = 0; pos < 32; pos++)
    {
        if(checkbit(xorAll,pos))
            break;
    }

    num1 = 0;
    num2 = 0;

    // Divide the array into two groups based on the rightmost set bit
    for (int i = 0; i < N; i++) {
        if (checkbit(A[i], pos)) {
            num1 ^= A[i];
        } else {
            num2 ^= A[i];
        }
    }

   print(num1);
   print(num2);
```

Follow Up Question

Given an array A of length N where all the elements are distinct and are in the range [1, N+2]. Two numbers from the range [1, N+2] are missing from the array A. Find the two missing numbers.

Problem 3 - Maximum and pair

Given N array elements, choose two indices(i, j) such that **(i != j)** and **(arr[i] & arr[j])** is maximum.

Input: [5, 4, 6, 8, 5]

Output: [0, 4]

If we take the **&** of 5 with 5, we get 5 which is the maximum possible value here. The required answer would be their respective indices i.e. **0,4**

Observation

1. When bit is set in both the numbers, that bit in their **&** will be 1
2. For answer to be maximum, we will want the set bit to be present towards as left as possible.
3. This indicates that we should start processing the numbers from MSB.

Optimized Solution

- Iterate from the Most significant bit to Least significant bit and for all the numbers in the array, count the numbers for which that bit is set
- If the count comes out to be greater than 1 then pairing is possible, so we include only the elements with that bit set into our vector. Also, set this bit in your answer.
- If the count is 0 or 1, the pairing is not possible, so we continue with the same set and next bit position.

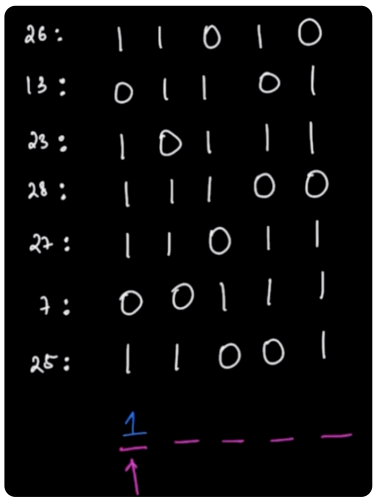
Dry Run

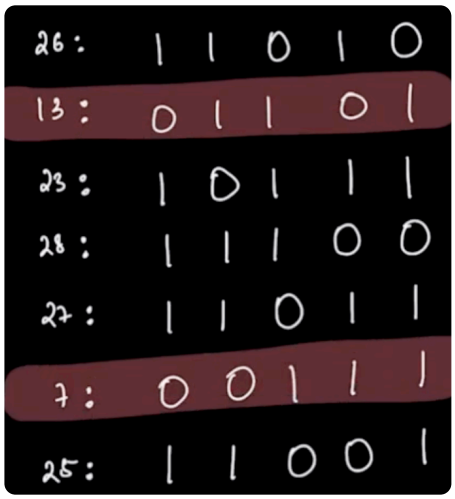
Example: { 26, 13, 23, 28, 27, 7, 25 }

26: 1 1 0 1 0
13: 0 1 1 0 1
23: 1 0 1 1 1
28: 1 1 1 0 0
27: 1 1 0 1 1
07: 0 0 1 1 1
25: 1 1 0 0 1

1. Let's start with MSB, **at position 4**, there are 5 numbers with set bits. Since count is >=2, we can form a pair. Therefore, in answer 1 will be present at this position.
   ans:

1	_	_	_	_

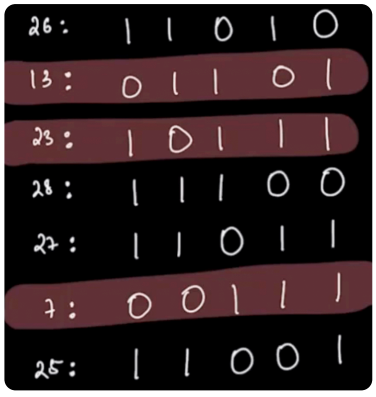




2. At position 3, there are 4 numbers with set bits(which haven't been cancelled). Since count is >=2, we can form a pair. Therefore, in answer 1 will be present at this position.
   ans:

1	1	_	_	_

We will remove all numbers where 0 is present.
[23 gets removed or is set to 0]



3. At position 2, there is 1 number with set bit. Since count is less than 2, we can't form a pair. Therefore, in answer 0 will be present at this position.
   ans:

1	1	0	_	_

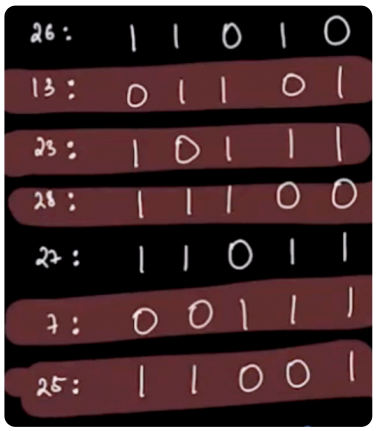
We will NOT remove any number.

4. At position 1, there are 2 numbers with set bits. Since count is >=2, we can form a pair. Therefore, in answer 1 will be present at this position.

ans:

1	1	0	1	_

We will remove all numbers where 0 is present.
[28 and 25 gets removed or are set to 0]



5. At position 0, there is 1 number with set bit. Since count is <2, we can't form a pair. Therefore, in answer 0 will be present at this position.

ans:

1	1	0	1	0

We will NOT remove any number.

We are done and answer final answer is present in variable ans.

Pseudocode

```
 int ans = 0;

for(int i = 31; i >= 0; i--)
{
    //count no. of set bits at ith index
    int count = 0;

    for(int j = 0; j < n; j++) {
        if(arr[j] & (1 << i))
            cnt++;
    }

    //set that bit in ans if count >=2
    if(count >= 2) {
        ans = ans | (1 << i);

        //set all numbers which have 0 bit at this position to 0
        for(int j = 0; j < n; j++) {
            if(arr[j] & (1 << i) == 0)
                arr[j] = 0;
        }

    }
}

print(ans);

//The numbers which cannot be choosen to form a pair have been made zero

```

Time and Space Complexity

- **TC -** O(N)
- **SC -** O(1)

Follow Up Questions

1. Find maximum & of triplets then we will do for count>=3 and for quadruples as count >= 4 and so on ...
2. Calculate the Count of Pairs for which bitwise & is maximum
    - Do exactly as above and then traverse on the array and find the number of elements which are greater than 0. Required answer will be NC2 or N(N-1)/2

Recursion

Definition

1. Function calling itself.

2. A problem is broke into smaller problem(subproblem) and the solution is generated using the subproblems.

Example

Here's an example of how recursion can be used to calculate the sum of the first **N** natural numbers:

```
sum(N) = 1 + 2 + 3 + 4 + ..... + N-1 + N
sum(N) = sum(N-1) + N
```

Here, **sum(N-1)** is smaller instance of same problem that is **sum(N)**.
sum(N-1) is known as a subproblem.

How to write a recursive code?

We can solve any recursive problem using below magic steps:

1. **Assumptions :** Decide what the function does for the given problem.
2. **Main logic :** Break the problem down into smaller subproblems to solve the assumption.
3. **Base case :** Identify the inputs for which we need to stop the recursion.

Problem Statement

Given a positive integer N, find the factorial of N.

Quiz

If N = 5, find the factorial of N.

- [ ] 100
- [x] 120
- [ ] 150
- [ ] 125

Solution :

Assumptions: Create a function which -

- Takes an integer value `N` as parameter.
- Calculates and returns N!
- return type should be integer.

Main logic:

- The factorial of N is equal to N times the factorial of (N-1)
- We made assumption that sum(N) calculates and return factorial of N natural number. Similarly, sum(N-1) shall calculate and return the factorial of N-1 natural number.

  `factorial(N) = N * factorial(N-1)`

Base case: The base case is when N equals 0, i.e., `0! = 1`

For example,
When we perform, `factorial(N) = N * factorial(N-1)` , value of N keeps on decreasing by 1.

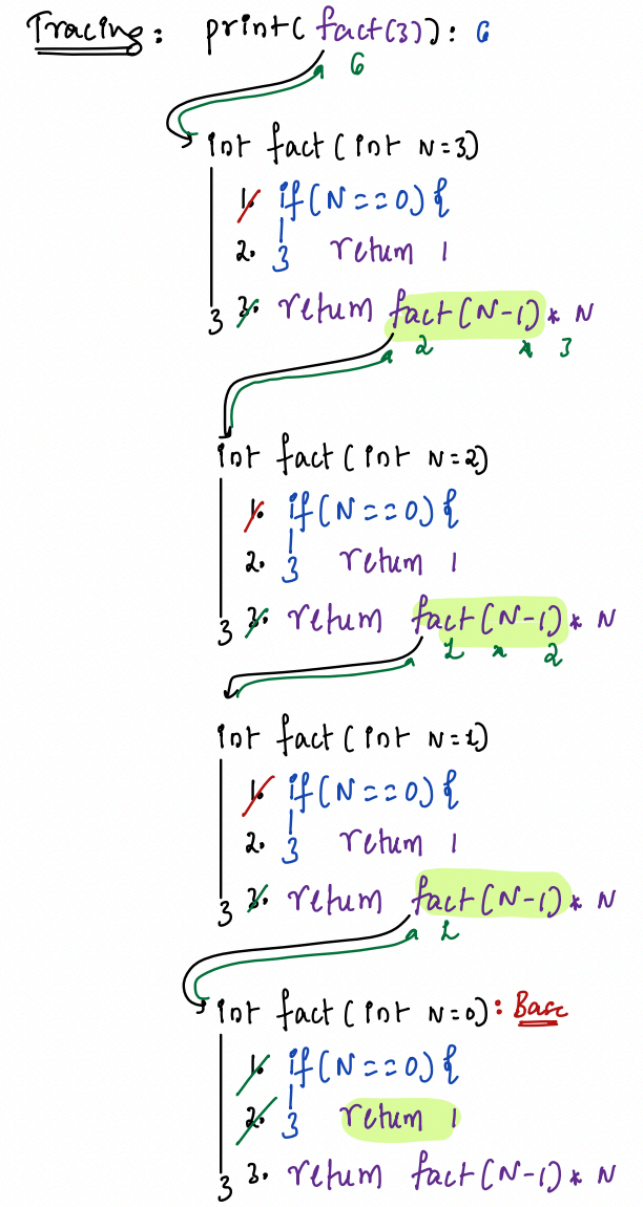
`N -> (N-1) -> (N-2) ........ 2 -> 1`

for N = 1 as well as 0, the factorial is 1.
So, we can write base case for N = 0 which will also cover for N = 1.

Pseudocode

```
int factorial(int N) {
    // base case
    if (N == 0) {
        return 1;
    }
    // recursive case
    return N * factorial(N-1);
}
```

Dry Run



Problem Statement

The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding numbers.

The sequence goes like this:

N	0	1	2	3	4	5	...
Fibonacci(N)	0	1	1	2	3	5	...

Given a positive integer N, write a function to compute the Nth number in the Fibonacci sequence using recursion.

Example

```
Input = 6
Output = 8
```

Quiz

If N = 7, find the Nth number in the fibonacci sequence.

- [ ] 8
- [ ] 5
- [x] 13
- [ ] 10

Solution

Assumptions: Create a function which -

- Takes an integer value `N` as parameter.
- calculates and returns N^{th} number in the fibonacci sequence.
- return type should be integer.

Main logic: Recursively compute the $(N-1)^{th}$ and $(N-2)^{th}$ numbers in the sequence and add them together to get the nth number.

```
fibonacci(N) = fibonacci(N - 1) + fibonacci(N - 2)
```

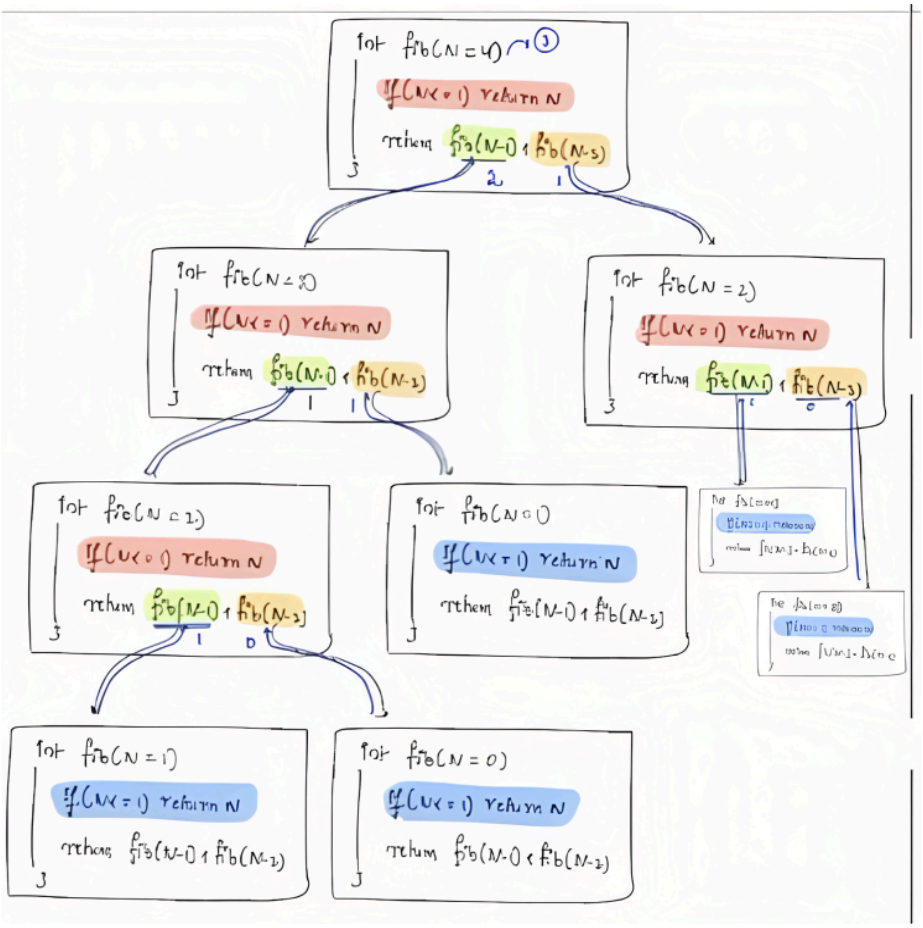
Base case: If N is 0 or 1, return the corresponding value in the Fibonacci sequence.

- Since, according to the definition of fibonacci sequence, the smallest two possible values of N are `N = 1` & `N = 2`, therefore, stop recursion as soon as N becomes 0 or 1.

Pseudocode

```
int fibonacci(int N) {
    // base case
    if (N == 0 || N == 1) {
        return N;
    }

    // recursive case
    return fibonacci(N - 1) + fibonacci(N - 2);
}
```

Function Call Tracing



Problem Statement

Given two integers **a** and **n**, find $\mathbf{a^n}$ using recursion.

Input

```
a = 2
n = 3
```

Output

```
8
```

Explanation

2^3 i.e, 2 * 2 * 2 = 8.

Brute Force Approach

The above problem can be redefined as:

```
a ^ n = a * a * a.....* a (n times).
```

The problem can be broken into subproblem as:

```
a ^ n = a ^ (n-1) * a
```

So we can say that pow(a,n) is equivalent to

```
pow(a,n) = pow(a,n-1) * a
```

Here, pow(a,n) is the defined as `a^n` .
We have seen how the problem is broken into subproblems. Hence, it can be solved using recursion.
Below is the algorithm:

- Assumption step:
  - Define a recursive function pow(a,n).
- Main Logic:
  - Define a recursive case: if **n** > 0, then calculate the pow(a,n-1).
  - Return a * pow(a,n-1).
- Base Case:
  - Base condition: if **n** = 0, then return 1.

Pseudocode

```
function pow(int a, int n){
    if(n == 0) return 1;
    return a * pow(a,n-1);
}
```

Optimized Approach

We can find pow(a,n) by one more relation :
$a^n = a^{n/2} * a^{n/2}$ (when n is even)
$a^n = a^{n/2} * a^{n/2} * a$ (when n is odd)

Below is the algorithm:

- Assumption Step:
  - Define a recursive function **pow(a,n)**.
- Main Logic:
  - Calculate **pow(a,n/2)** and store it in a variable **p**.

- if n is odd, then return **p * p * a**.
- else return **p * p**.

- Base Condition:
  - if **n = 0**, then return **1**.

Pseudocode

```
Function pow(int a, int n){
    if(n == 0) return 1;

    int p = pow(a, n/2);

    if(n % 2 == 0) {
        return p * p;
    }
    else {
        return p * p * a;
    }
}
```

Please do dry run, recursion is more about dry run then code.

Tower of Hanoi

There are n disks placed on tower A of different sizes.

Goal

Move all disks from tower A to C using tower B if needed.

Constraint

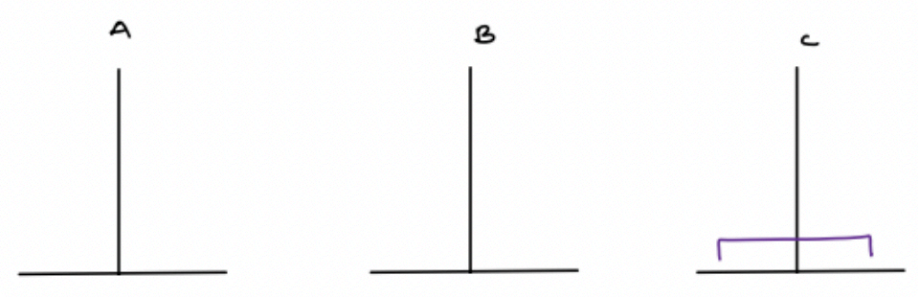
- Only 1 disk can be moved at a time.
- Larger disk can not be placed on a small disk at any step.

Print the movement of disks from A to C in minimum steps.

Example 1

Input: N = 1

Explanation:

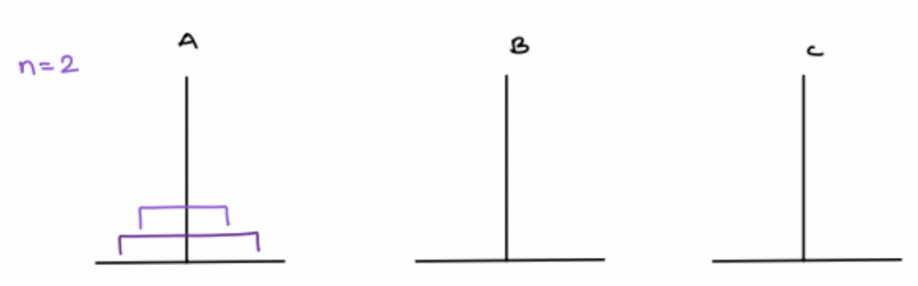


Output:

1: A -> C

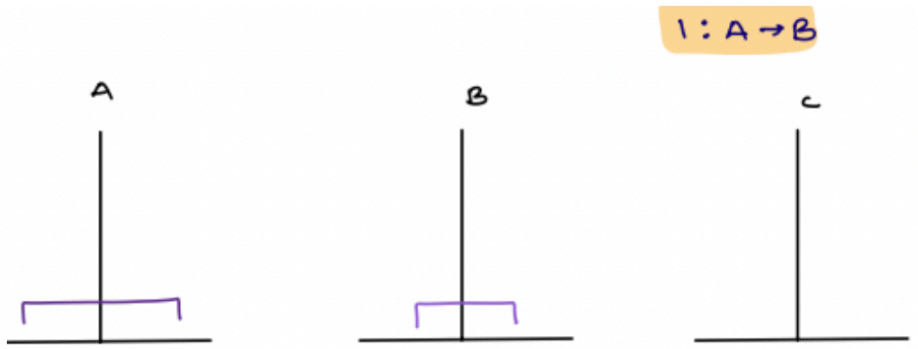
Example 2

Input: N = 2

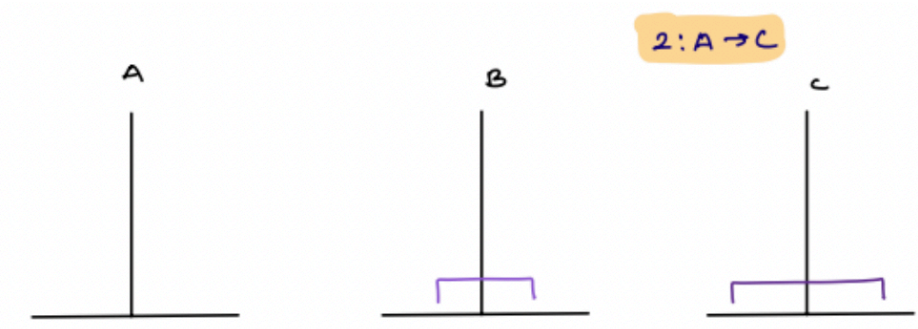


Explanation:

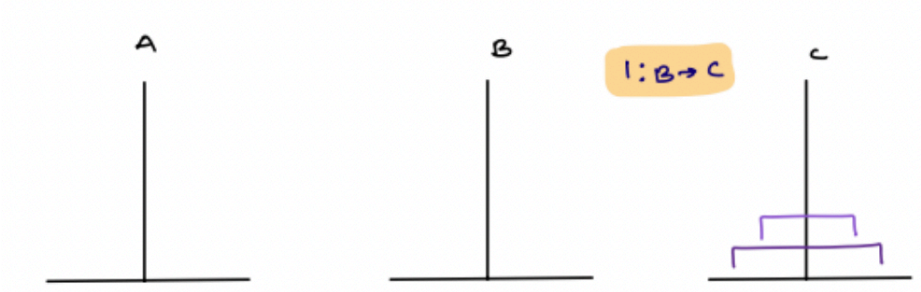
1: A -> B



2: A -> C



1: B -> C



Output:

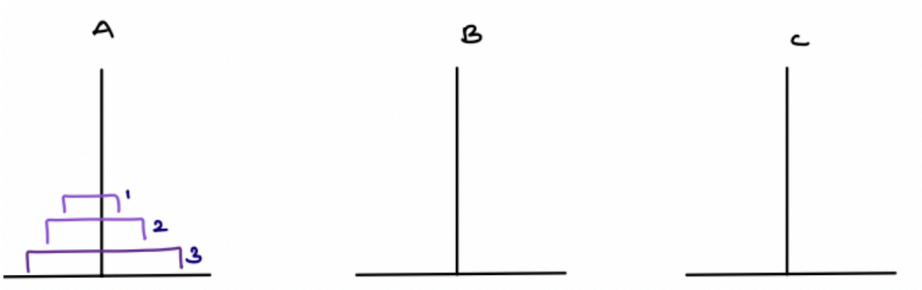
1: A -> B

2: A -> C

1: B -> C

Example 3

Input: N = 3

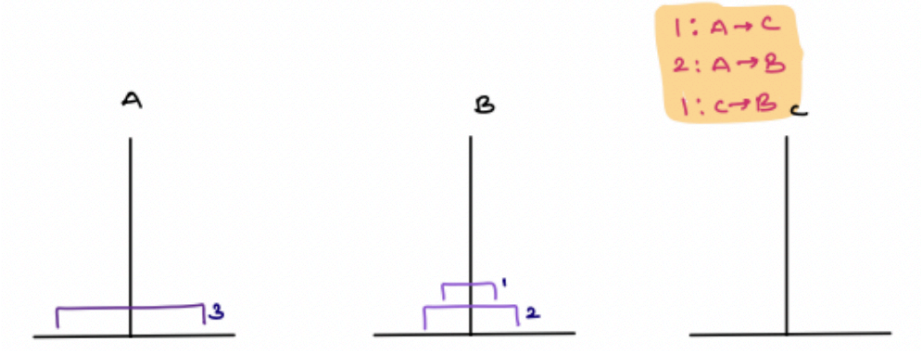


Explanation:

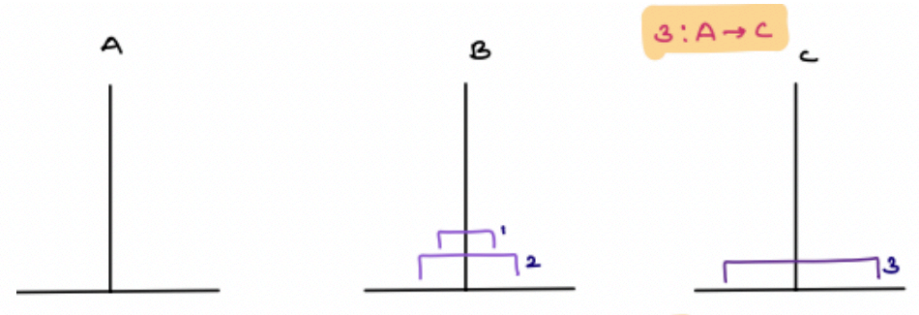
1: A -> C

2: A -> B

1: C -> B



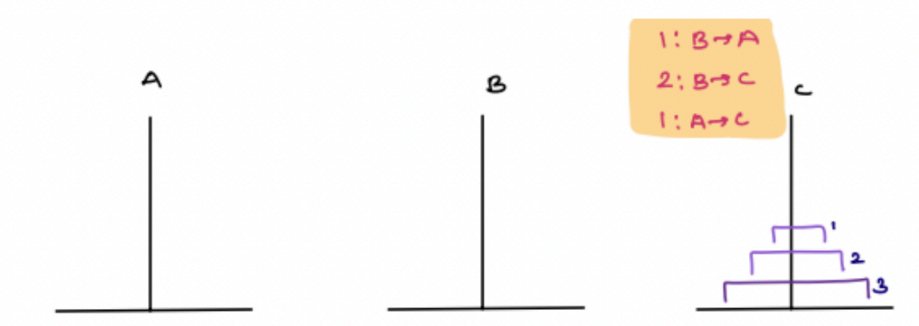
3: A -> C



1: B -> A

2: B -> C

1: A -> C



Output:

1: A -> C

2: A -> B

1: C -> B

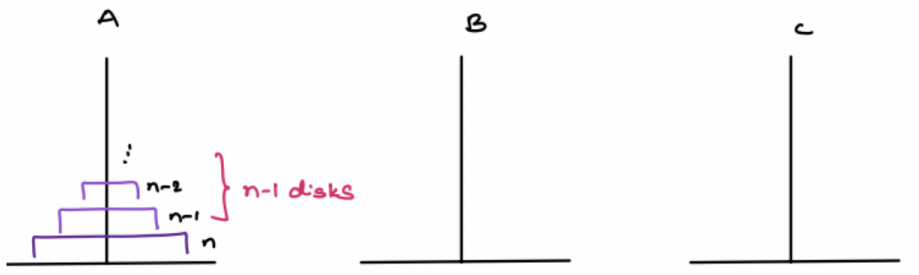
3: A -> C

1: B -> A

2: B -> C

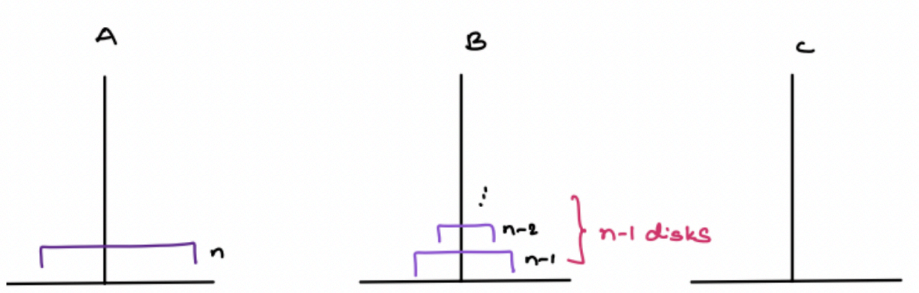
1: A -> C

n disks



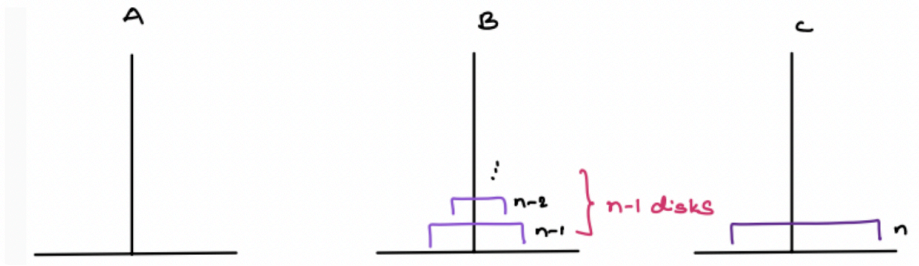
Step 1:

Move (n-1) disks from A to B



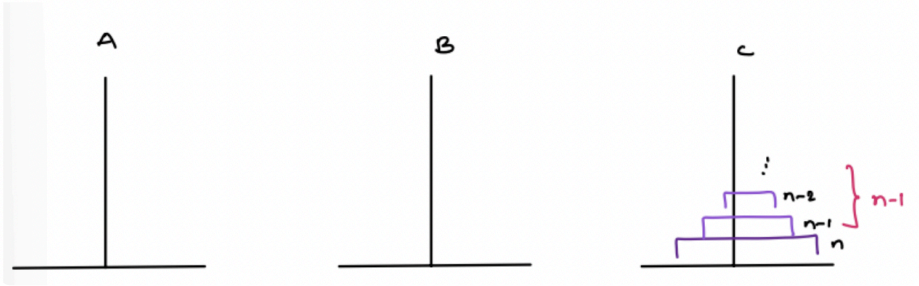
Step 2:

Move n disk to C



Step 3:

Move (n-1) disks from B to C

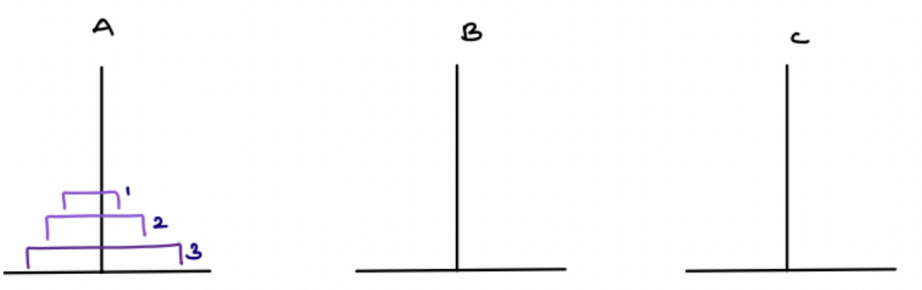


Pseudocode

```
                src  temp  dest
void TOH(int n, A,    B,   C){
    1. if(n==0)
         return;
    2. TOH(n-1, A, C, B); // move n-1 disks A->B
    3. print(n:A->C); // moving n disk A->C
    4. TOH(n-1, B, A, C); // move n-1 disks B->C
}
```

Dry Run

n = 3



Tracing:

n src tmp dst

TOH(3 A B C) x x x x

n src tmp dst

2: TOH(2 a c b) : x x x x

n src T D

2: TOH(1, a, b, c) : x x x x

2: TOH(0, a, c b) : return

3: print(1: a → c)

4: TOH(0, b, a, c) : return

print(2: a → b)

n s T D

4: TOH(1, c, a, b) : x x x x

2: TOH(0, c, b, a) return

print(1: c → b)

4: TOH(0, a, c, b) · return

3 : print(3: A → c)

n s T D

4 : TOH(2, B, A, c) x 2 3 4

Output:

1: A -> C

2: A -> B

1: C -> B

3: A -> C

1: B -> A

2: B -> C

1: A -> C

Maths 1 : Modular Arithmetic & GCD

A % B = Remainder when A is divided by B

Range of A % B will be within **[0, B - 1]**

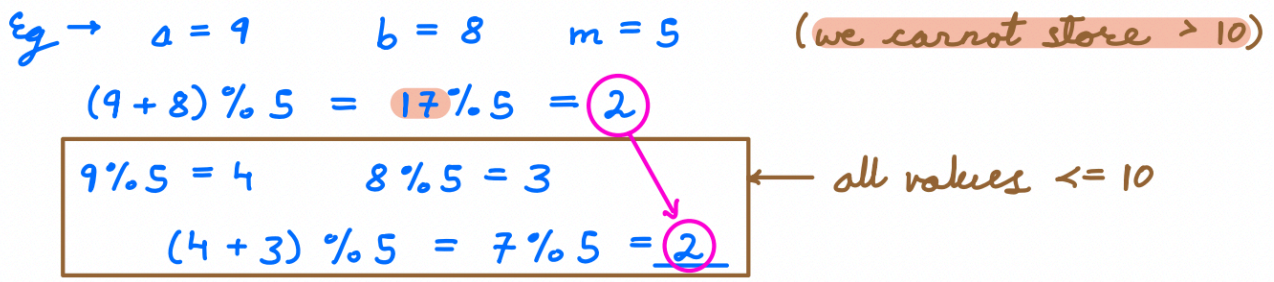
Why do we need Modular Arithmetic(%) ?

The most useful need of `%` is to limit the range of data. We don't have unlimited range in **Integer** OR **Long**, hence after a certain limit, we cannot store data. In such cases, we can apply mod to limit the range.

Rules for applying mod on Arithmetic Operations

1. `(a + b) % m = (a % m + b % m) % m`

Example:



2. `(a * b) % m = (a % m * b % m) % m`

3. `(a + m) % m = (a % m + m % m) % m = (a % m) % m = a % m`

4. `(a - b) % m = (a % m - b % m + m) % m`

This extra **m** term is added to ensure that the result remains within the range of **0 to m-1** even if the intermediate result of **(a % m - b % m) is negative**. This guarantees that the final remainder is a non-negative value.

Example:

Let's take **a = 7**, **b = 10**, and **m = 5**.

$(a - b)\ \%\ m\ =\ (7 - 10)\ \%\ 5\ =\ \ -3\ \%\ 5 = -3$ (which is not in the range 0 to 4)

Now we can simly do

(-3 + 5) % 5 = 2 (now value is in the range 0 to 4)

5. `(a ^ b) % m = ( (a % m) ^ b ) % m`

(a raise to power b)

Quiz

Evaluate :

$(37^{103} - 1)\%12$

Choices

- [ ] 1
- [ ] 0
- [ ] No Idea
- [ ] 10

Explanation:

The answer is 0.

$=>\ (37^{103} - 1)\%12$

$=>\ ((37^{103}\ \%12) - (1\%12) + 12)\%12$

$=>\ (((37\%12)103\ \%12) - 1 + 12)\ \%12$

$=>\ (1 - 1 + 12)\%12 = 12\%12 = 0$

GCD Basics

- GCD - Greatest Common Division
- HCF - Highest Common Factor
- GCD(A, B) - Greatest factor that divides both a and b

If we have `GCD(A, B) = x`

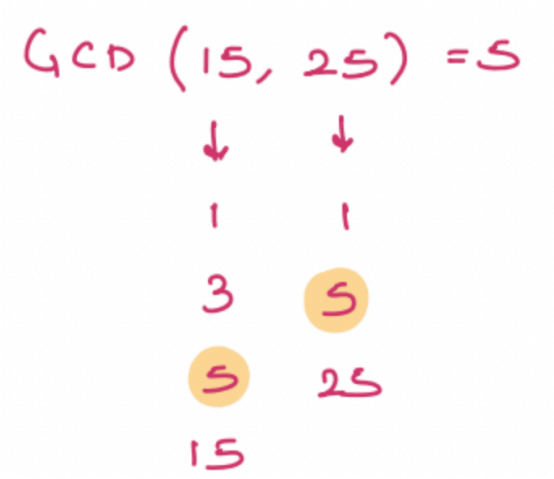
This implies:-

- A % x = 0
- B % x = 0

and hence `x` is the highest factor of both A and B

Example - 1

GCD(15, 25) = 5



Example - 2

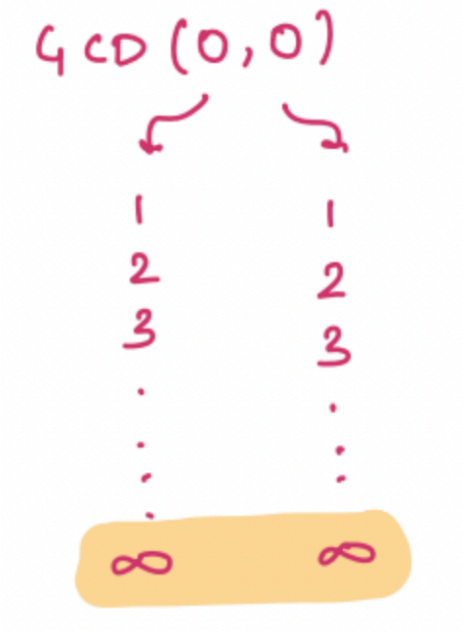
GCD(12, 30) = 6

Example - 3

GCD(0, 4) = 4

Example - 4

GCD(0, 0) = Infinity



Properties of GCD

Property-1

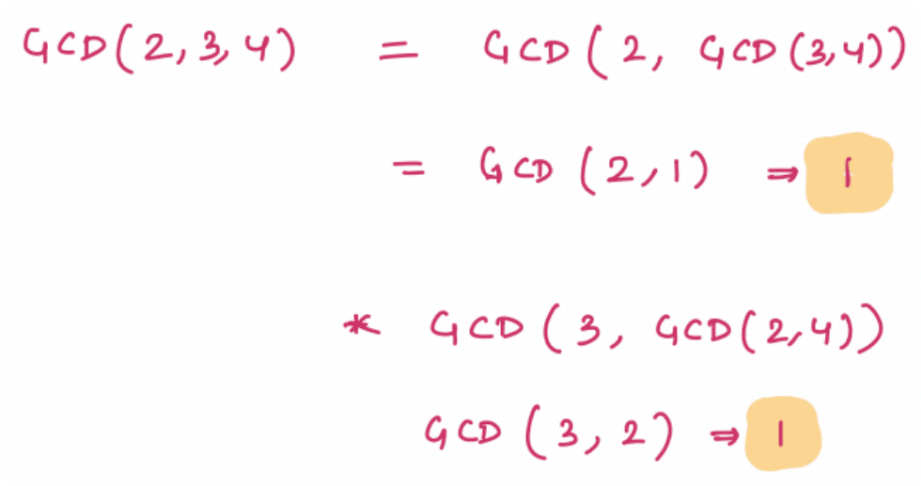
GCD(A, B) = GCD(B, A)

Property-2

GCD(0, A) = A

Property-3

GCD(A, B, C) = GCD(A, GCD(B, C)) = GCD(B, GCD(C, A)) = GCD(C, GCD(A, B))



Property - 4

Given `A >= B > 0`,
GCD(A, B) = GCD(A - B, B)

> Note: The proof isn't asked in Interviews. If you want to study, please stay back.

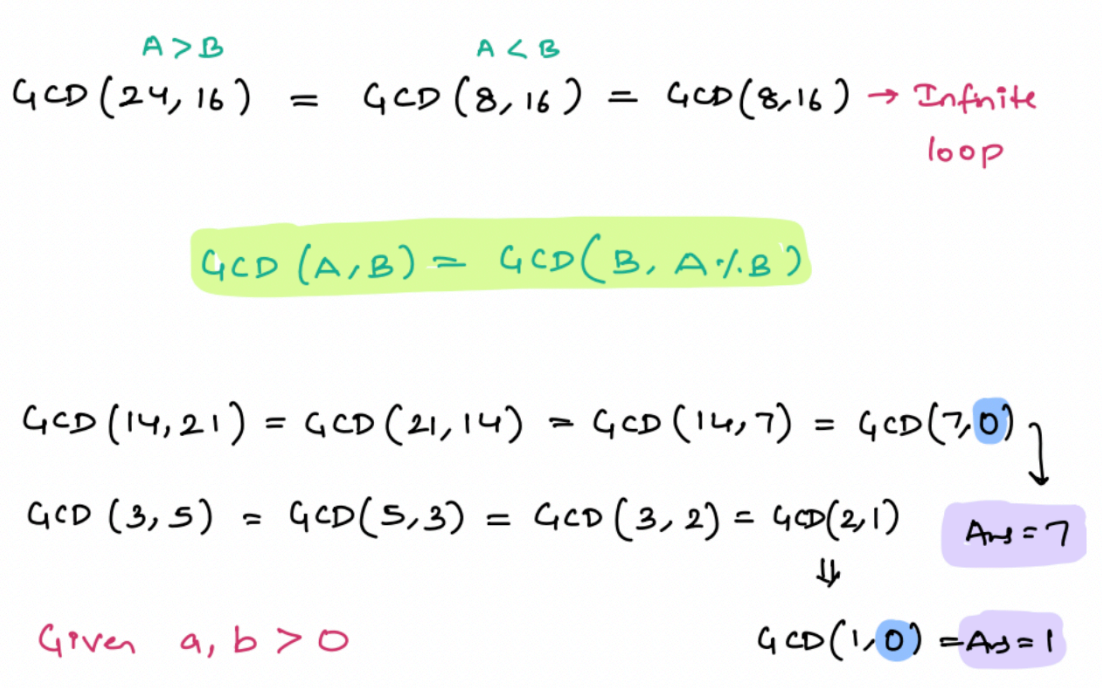
Example:

> **Image Not Showing**
> Possible Reasons
> - The image was uploaded to a note which you don't have access to
> - The note which the image was originally uploaded to has been deleted
>
> Learn More → (https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg)

Property - 5

GCD(A, B) = GCD(A % B, B)

Problem : Write a function to find GCD (A, B)



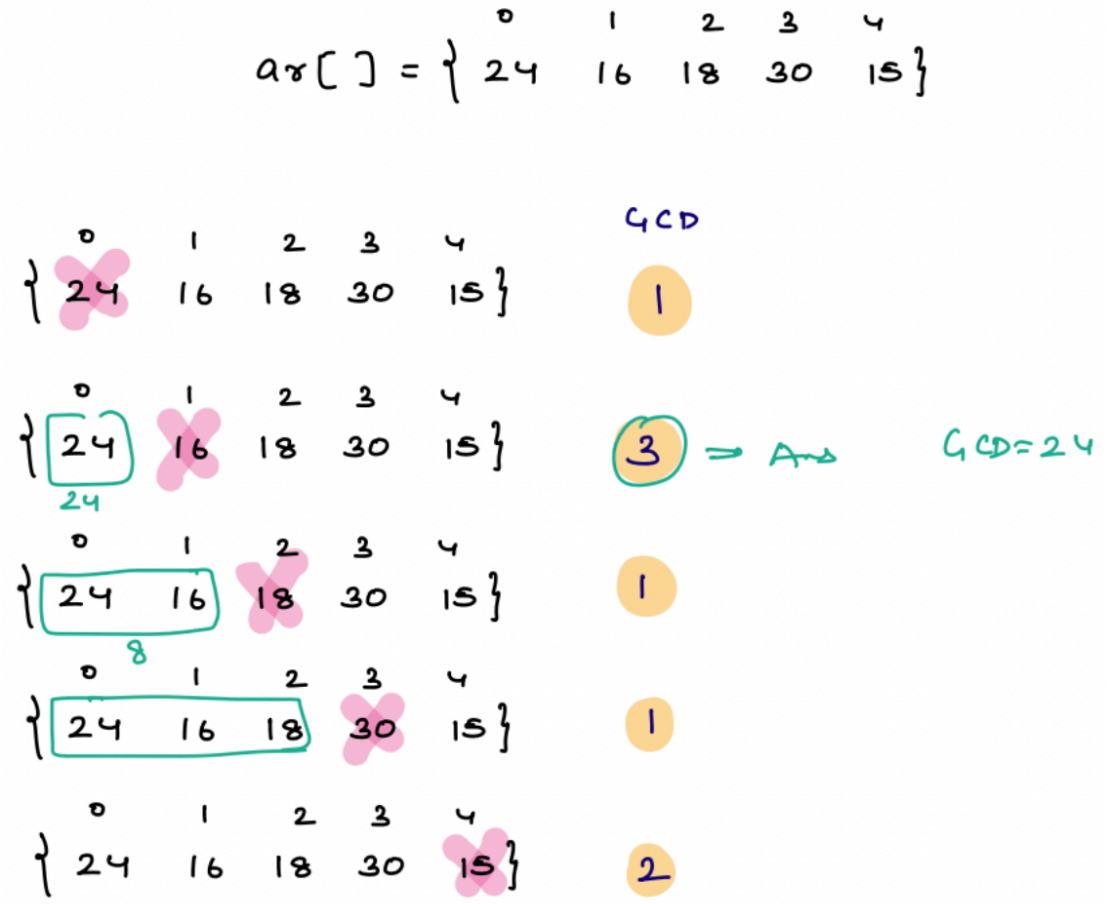
Suppose we have two positive numbers a, b then:

```
int gcd(a, b) {
    if (b == 0) {
        return a;
    }
    return gcd(b, a % b);
}
```

Problem

Given arr[N] elements , we have to delete one element such that GCD of the remaining elements becomes maximum.

Example:



Brute Force Approach

The brute approach for this problem will be to delete arr[i], and then calculate the GCD for all the remaining elements. This will be repeated for all the elements.

TC : $O(n^2 \log max)$

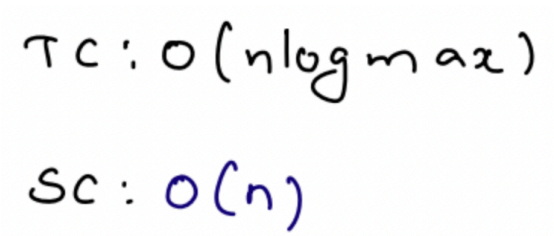
Optimal Approach: Prefix Array

Approach:

- Idea is to find the GCD value of all the sub-sequences of length (N – 1) and removing the element which is not present in the sub-sequence with that GCD. The maximum GCD found would be the answer.
- To find the GCD of the sub-sequences optimally, maintain a `prefixGCD[]` and a `suffixGCD[]` array using single state dynamic programming.
- The maximum value of GCD(`prefixGCD[i - 1]` , `suffixGCD[i + 1]`) is the required answer.

The implementation is given below:

```
// Recursive function to return gcd of a and b
    static int gcd(int a, int b) {
        if (b == 0)
            return a;
        return gcd(b, a % b);
    }

    static int MaxGCD(int a[], int n) {

        // Prefix and Suffix arrays
        int Prefix[] = new int[n + 2];
        int Suffix[] = new int[n + 2] ;

        Prefix[1] = a[0];
        for (int i = 2; i <= n; i += 1) {
            Prefix[i] = gcd(Prefix[i - 1], a[i - 1]);
        }

        Suffix[n] = a[n - 1];
        for (int i = n - 1; i >= 1; i -= 1) {
            Suffix[i] = gcd(Suffix[i + 1], a[i - 1]);
        }

        // If first or last element of the array has to be removed
        int ans = Math.max(Suffix[2], Prefix[n - 1]);

        // If any other element is replaced
        for (int i = 2; i < n; i += 1) {
            ans = Math.max(ans, gcd(Prefix[i - 1], Suffix[i + 1]));
        }
        return ans;
    }
```



Revision - Hashing

Problem 1: Pair Sum K

Given arr[N] and K, check if there exists a pair (i, j) such that arr[i] + arr[j] == K && i != j.

Example:

Let's say we have an array of 9 elements and K where K is our target sum.

Index: 0 1 2 3 4 5 6 7 8

Array: 8 9 1 -2 4 5 11 -6 4

K = 6: arr[2] + arr[5]: such a pair exists

K = 22: does not exist

K = 8: arr[4] + arr[8]: such a pair exists

Brute Force

Iterate on all pairs (i, j) check if their sum == K.

Time and Space Complexity:

TC: O(N^2) and SC: O(1)

Optimization with HashSet

Only insert elements into the set from [0, i - 1].

```
boolean targetSum(int arr[], int K) {
    int N = arr.length;
    HashSet<int> bs;

    for (int i = 0; i < N; i++) {
        if (bs.contains(K - arr[i])) { //check if pair exists
            return true;
        } else {
            bs.add(arr[i]); //add current ele for future use
        }
    }
    return false;
}
```

TC: O(N) and SC: O(N)

Practice:

Pair Sum (https://https://www.interviewbit.com/problems/2-sum/)

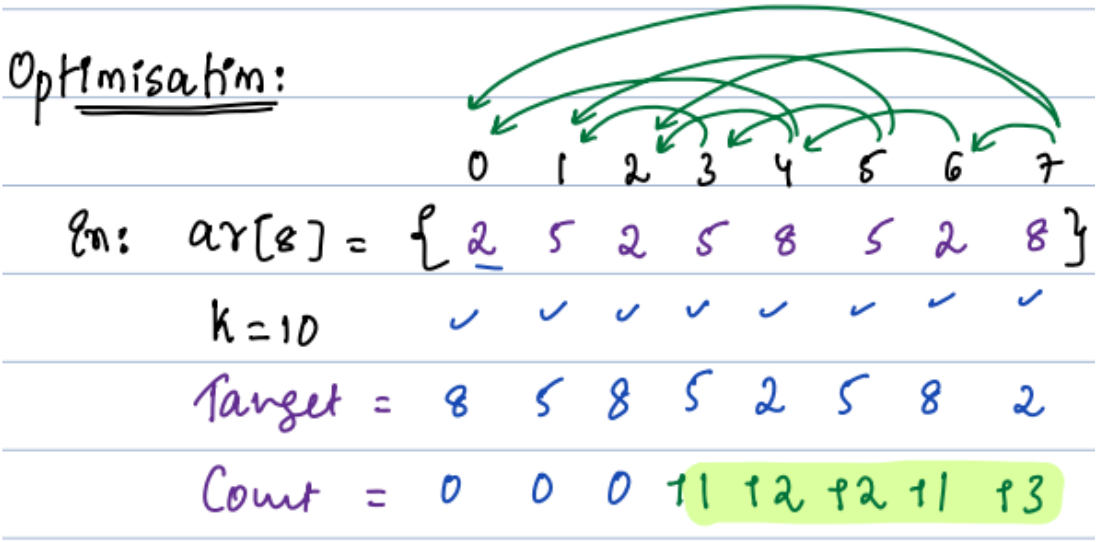
Pair Difference (https://https://www.interviewbit.com/problems/pair-with-given-difference/)

NOTE: In this, we will have to check two cases, A[j] = A[i] - K and A[j] = A[i] + K

Problem 2: Count no. of pairs with sum K

Given arr[n], count the number of pairs such that arr[i] + arr[j] = K && i != j.

In this, we shall be keeping track of how many times an element appears, so we'll use map.




```
int countTargetSum(int arr[], int K) {
    int N = arr.length;
    Hashmap<int, int> hm;

    int c = 0;

    for (int i = 0; i < n; i++) {
        if (hm.contains(K - arr[i])) {
            c = c + hm[K - arr[i]] //freq of target = pairs
        }

        if (hm.contains(arr[i])) {
            hm[arr[i]]++;
        } else {
            hm.put(arr[i], 1);
        }
    }
    return c;
}
```

Time and Space Complexity:
TC: O(N) and SC: O(N)

Practice:

Count Pair Difference (https://leetcode.com/problems/count-number-of-pairs-with-absolute-difference-k/description/)

Problem 3: Subarray with Sum K

Given an array arr[n] check if there exists a subarray with sum = K.

For subarray sum, we can use prefix array.
Equation: pf[a] - pf[b] = K

Say we fix pf[a], then pf[b] = pf[a]-K

Pseudocode:

```
boolean targetSubarray(int arr[], int K) {
    int N = arr.length;
    long a = 0;
    HashSet<long> hs;
    hs.add(0);
    for (int i = 0; i < N; i++) {
        a = a + arr[i];
        if (hs.contains(a - K)) {
            return true;
        } else {
            hs.add(a);
        }
    }
    return false;
}
```

Time and Space Complexity:
TC: O(N) and SC: O(N)

Problem 4: Distinct elements in every window of size K

Given an array of integers and a number, K. Find the count of distinct elements in every window of size K in the array.

Approach

- Initially, the algorithm populates the HashMap with the elements of the first subarray, counting distinct elements.
- Then, as the window slides one step at a time:
    - The algorithm updates the HashMap by decrementing frequency of the element leaving the window and increasing frequency of the new element entering the window.
    - The distinct element count for each subarray is determined by the size of the HashMap.

Pseudocode

```
void subfreq(int ar[], int k){
    int N = ar.length;

    HashMap < int, int > hm;

    // Step 1: Insert 1st subarray [0, k-1]
    for(int i = 0;i < k;i++){
        //Increase frequecy by 1
        if(hm.contains(ar[i])){
            int val = hm.get(ar[i]);
            hm.put(ar[i], val+1);
        }else{
            hm.put(ar[i], 1);
        }

        print(hm.size());
    }

    //step 2: Iterate all other subarrays
    int s = 1, e = k;

    while(e < N){
        //Remove ar[s-1]
        int val = hm[ar[s - 1]];
        hm[ar[s - 1]]--;

        if(hm.get[ar[s - 1]] == 0){
            hm.remove(ar[s - 1]);
        }

        //add ar[e]
        if(hm.contains(ar[e])){
            //Inc freq by 1
            int val = hm[ar[e]];
            hm[ar[e]]++;
        }else{
            hm.put(ar[e], 1);
        }

        print(hm.size());
        s++

    }
}
```

Time Complexity: `O(n)`

Space Complexity: `O(n)`

Revision - Sorting

Problem Statement

Find the smallest number that can be formed by rearranging the digits of the given number in an array. Return the smallest number in the form an array.

Example:

A[ ] = `{6, 3, 4, 2, 7, 2, 1}`

Answer: `{1, 2, 2, 3, 4, 6, 7}`

A[ ] = `{4, 2, 7, 3, 9, 0}`

Answer: `{0, 2, 3, 4, 7, 9}`

Observation / Hint

we can to construct a number using digits. The digits in a number can only `range from 0 to 9` , thus instead of sorting the number which takes `O(N log N)` time, one can leverage this fixed range to derive a faster solution.

Approach

- **Frequency Count:** Create an `array of size 10` to count the frequency of each digit in the given number.
- Using the frequency array, reconstruct the original array in ascending order.
- This method of sorting based on frequency counting is often called " **`Count Sort`** ".

Pseudocode

```
Frequency Array of size 10
F = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

for i -> 0 to (N - 1) {
   F[A[i]] ++
}

k = 0

for d -> 0 to 9 {   // For each digit
   for i -> 1 to F[d] {
      A[k] = d
      k++
   }
}

return A
```

Dry Run

- A = `[1, 3, 8, 2, 3, 5, 3, 8, 5, 2, 2, 3]` (Given Array)

- F = `[0, 1, 3, 4, 0, 2, 0, 0, 2, 0]` (Frequency Array)
- Reconstructing A using F:
  1 (once), 2 (three times), 3 (four times), 5 (two times), 8 (two times)
- Resulting A = `[1, 2, 2, 2, 3, 3, 3, 3, 5, 5, 8, 8]`

TC and SC

- **Time Complexity:** O(N)
- **Space Complexity:** O(1) (Since the size of the frequency array is constant, regardless of the size of N).

Will Count Sort work if the range of A[i] is more than 10^9?

- Count Sort isn't suitable for a range of 10^9 because a frequency array of this size would demand too much memory.
- Count Sort works well when the range of A[i] is ~ 10^6.

Each integer typically occupies `4 Bytes`.

Storing 10^9 integers requires 4GB, which is often impractical. An array up to 10^6 in length is more manageable, needing 4MB.

Problem Statement

Implement Count Sort for an array containing negative numbers.

Observation / Hint

Unlike conventional **Count Sort**, which operates on non-negative integers, this variation needs to account for negative numbers. The method involves adjusting indices in the frequency array based on the smallest element in the original array.

Approach

- **Find Range:** Determine the smallest and largest elements in the array to ascertain the range.
- **Adjust for Negative Numbers:** By adjusting indices in the frequency array based on the smallest element, negative numbers can be accounted for.

Example

- Given A = [-2, 3, 8, 3, -2, 3]
- Smallest = -2, Largest = 8
- Range = 11 (8 - (-2) + 1)

- Frequency array F = [2, 0, 0, 3, 0, 0, 0, 0, 1]
- 0th index frequency is mapped with -2, 1st index with -1, and so on.
- Reconstructed A using F: -2, -2, 3, 3, 3, 8

Pseudocode

```
//Find smallest and largest elements in A
//Create a frequency array F of size (largest - smallest + 1)

for i -> 0 to (N - 1) {
   F[A[i] - smallest_element] ++
}

//Reconstruct array A using F

for each index i in F {
   while F[i] > 0 {
      Append (i + smallest_element) to A
      F[i]--
   }
}
```

TC and SC

- **Time Complexity (TC):** O(N)
- **Space Complexity (SC):** O(N + Range)

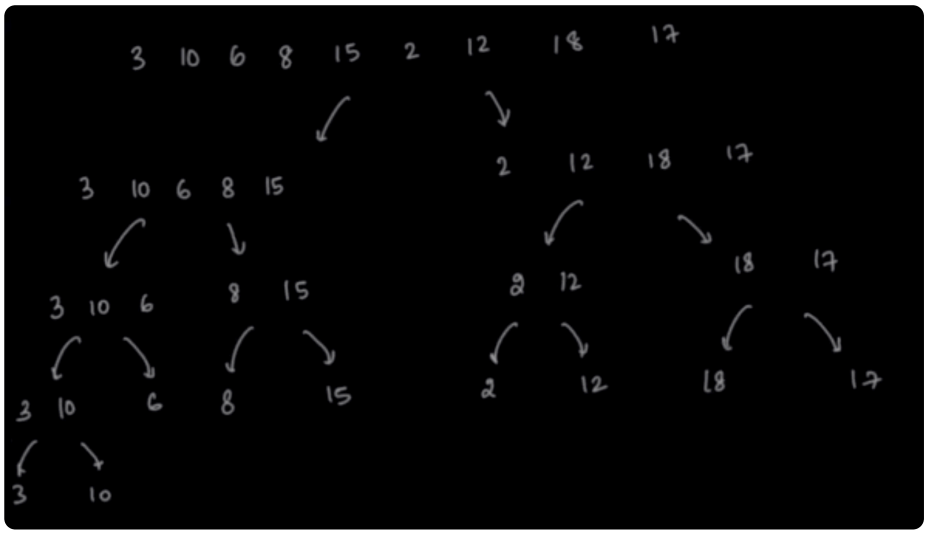
Merge Sort

Example: Sorting Numbers

Sort the array, A = {3, 10, 6, 8, 15, 2, 12, 18, 17}

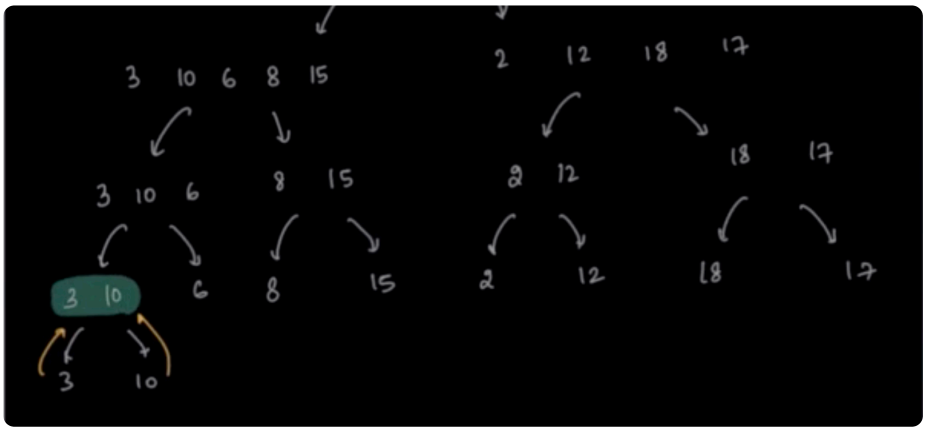
Divide

- The idea is to divide the numbers in two halves and then start merging the sorted arrays from bottom and pass above.

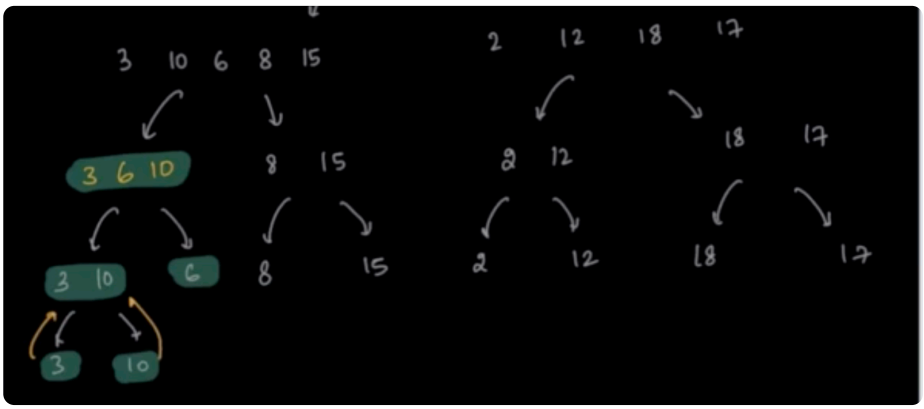


Merge

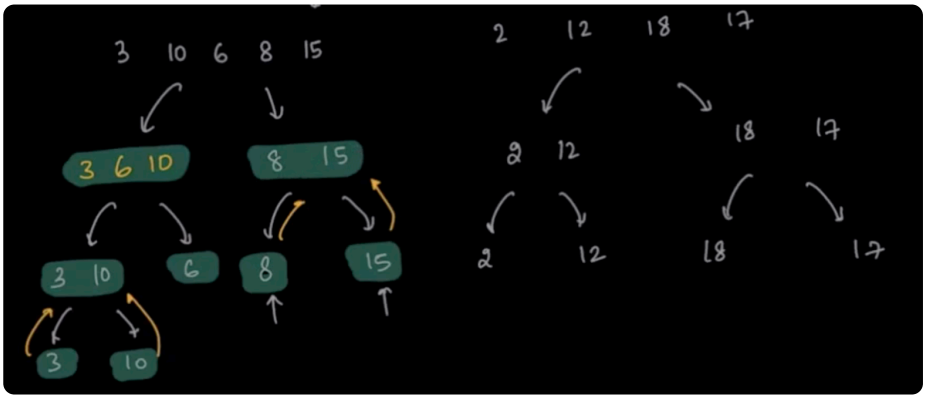
- Merging [3] and [10] as [3, 10]



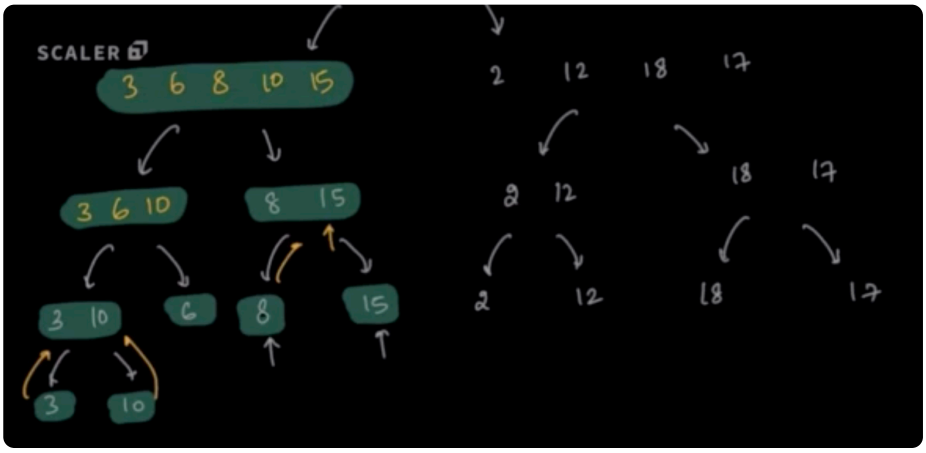
- Merging [3, 10] and [6] as [3, 6, 10]



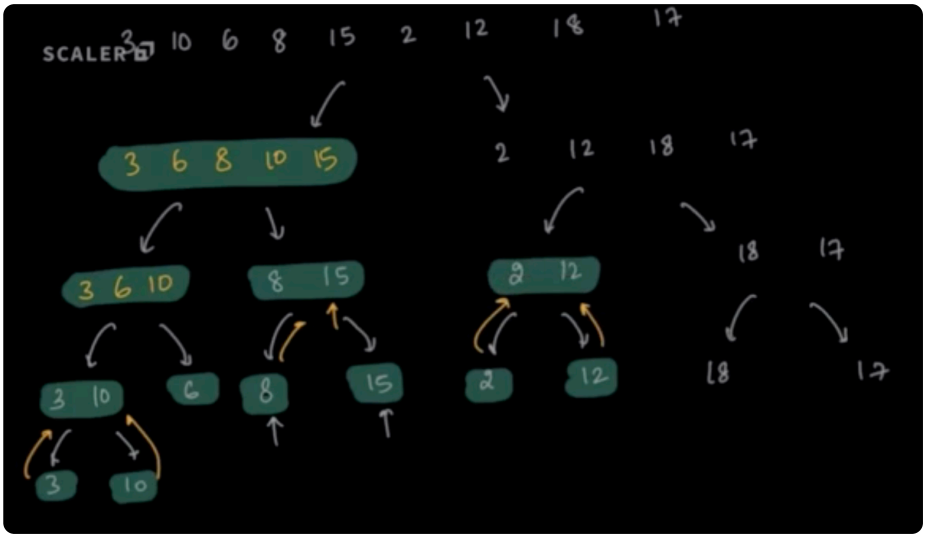
- Merging [8] and [15] as [8, 15]



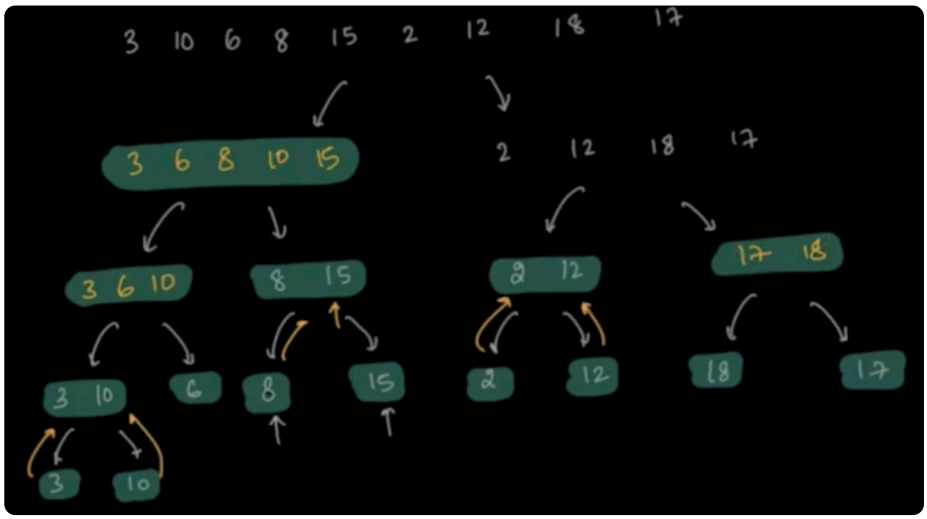
- Merging [3, 6, 10] and [8, 15] as [3, 6, 8, 10, 15]



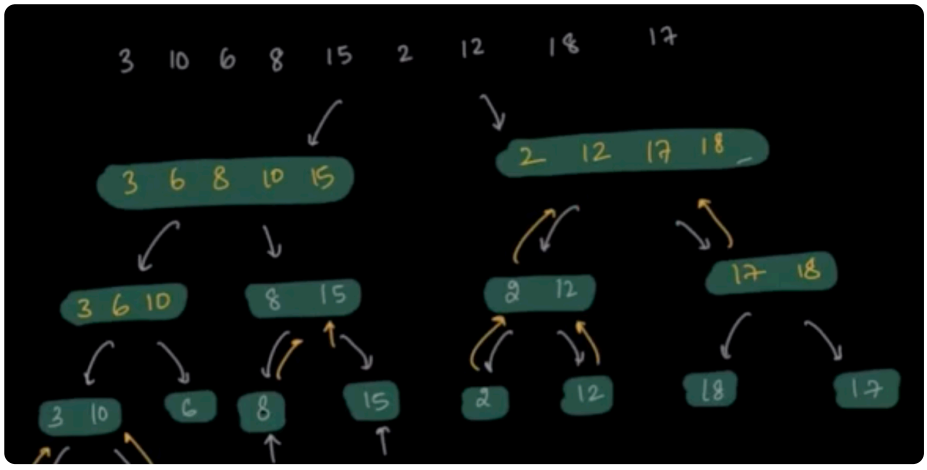
- Merging [2] and [12] as [2, 12]



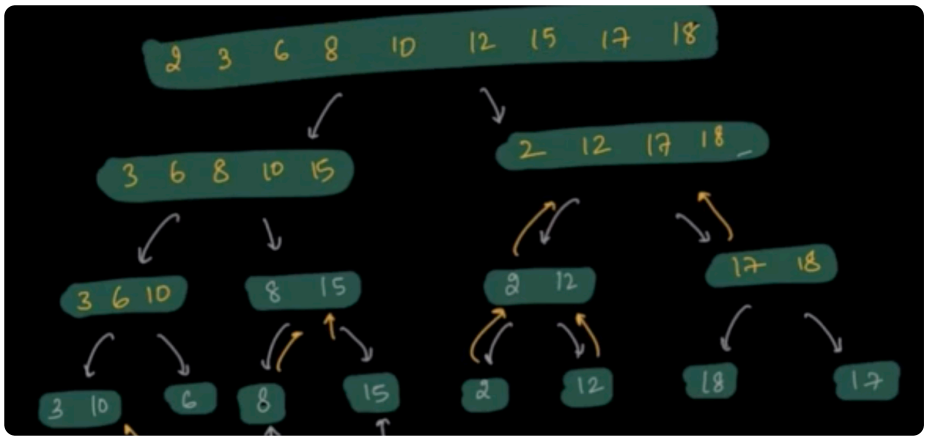
- Merging [18] and [17] as [17, 18]



- Merging [2, 12] and [17, 18] as [2, 12, 17, 18]



- Merging [3, 6, 8, 10, 15] and [2, 12, 17, 18] as [2, 3, 6, 8, 10, 12, 15, 17, 18]



In this way, we have finally sorted the array.

This algorithm of dividing the array into multiple subproblems and merging them one by one is called Merge Sort.

Since we are breaking down the array into multiple subproblems and applying the same idea to merge them, we are using the technique of Recursion.

Psuedocode

```
void merge(A[], l, mid, r) {
    int N = A.length();
    int n1 = mid-l+1;
    int n2 = r-mid;

    int B[n1], C[n2];

    int idx=0;
    for(int i=l; i<=mid; i++){
        B[idx] = A[i];
        idx++;
    }

    idx=0;
    for(int i=mid+1; i<=r; i++){
        C[idx] = A[i];
        idx++;
    }

    idx = l;
    i = 0; // moves over A
    j = 0; // moves over B

    while (i < n1 && j < n2) {
        if (B[i] <= C[j]) {
            A[idx] = B[i];
            i++;
        } else {
            A[idx] = C[j];
            j++;
        }
        idx++;
    }

    while (i < n1) {
        A[idx] = B[i];
        idx++;
        i++;
    }

    while (j < n2) {
        A[idx] = C[j];
        idx++;
        j++;
    }
}

void mergeSort(int A[], l, r){
    if(l == r) return; // base case

    int mid = (l + r) / 2;
    mergeSort (A, l, mid);
    mergeSort (A, mid + 1, r);
```

```
        merge(A, l, mid, r);
    }
```

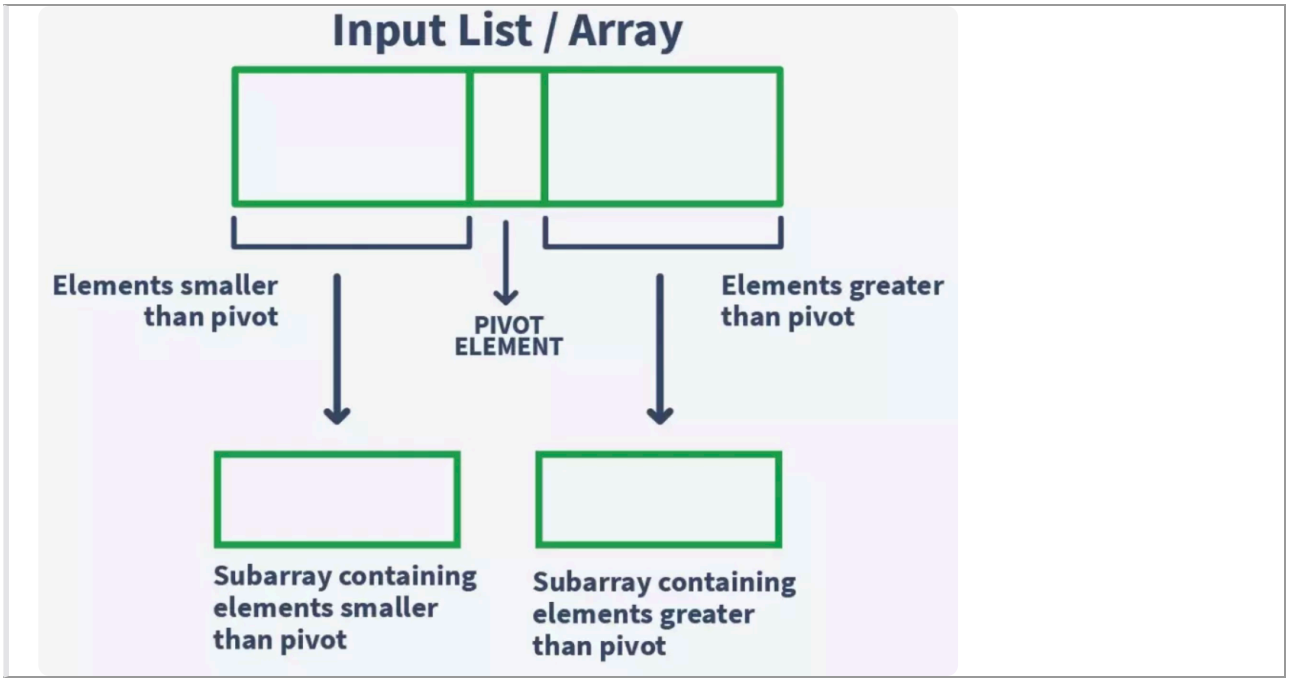
Sorting 2: Quick Sort & Comparator Problems

Problem Description

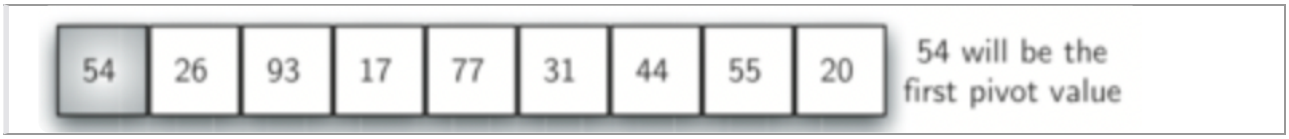
Given an integer array, consider first element as pivot, rearrange the elements such that for all `i` :

if `A[i] < p` then it should be present on left side
if `A[i] > p` then it should be present on right side

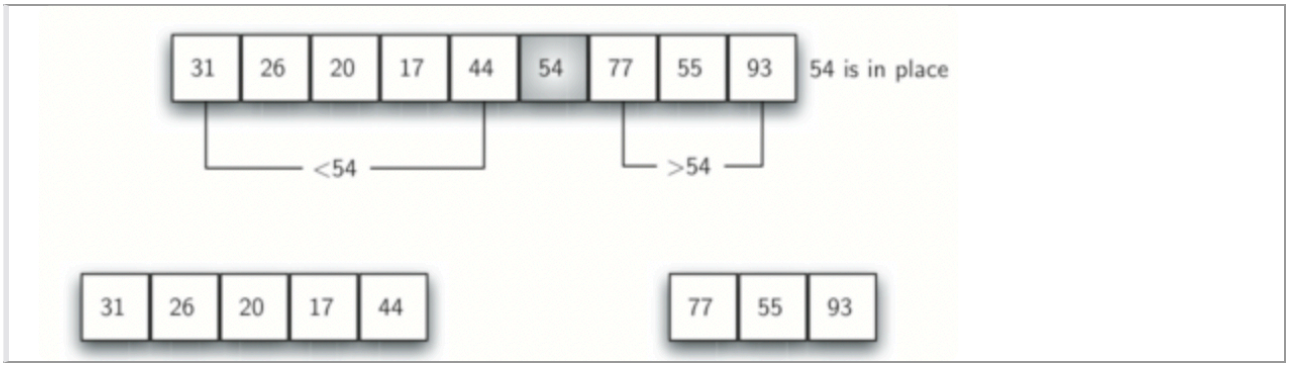
Note: All elements are distinct



Example:



The State of the array after Partitioning will be:



Question

Given an integer array, consider first element as pivot **p**, rearrange the elements such that for all `i` :

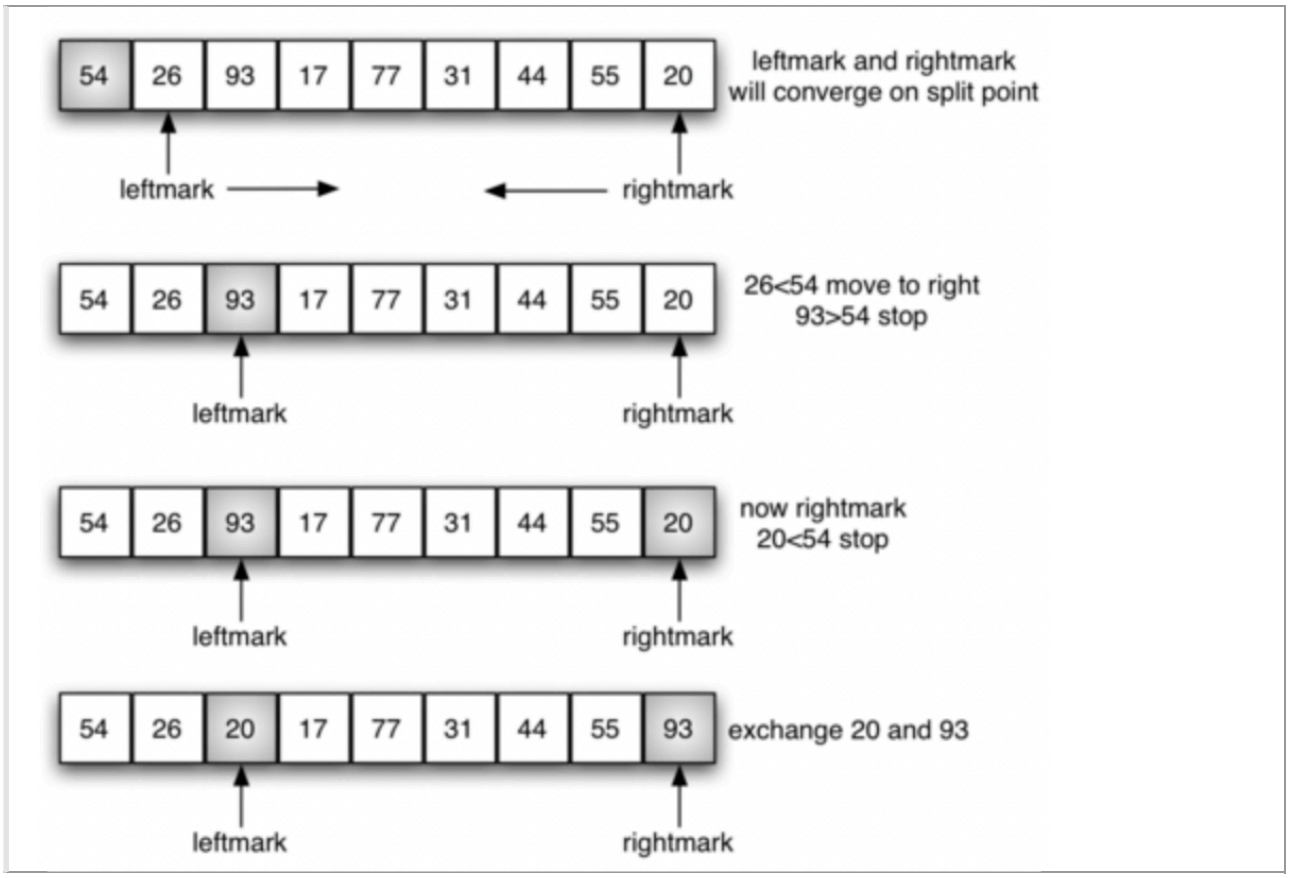
if `A[i] < p` then it should be present on left side
if `A[i] > p` then it should be present on right side

```
A = [10, 13, 7, 8, 25, 20, 23, 5]
```

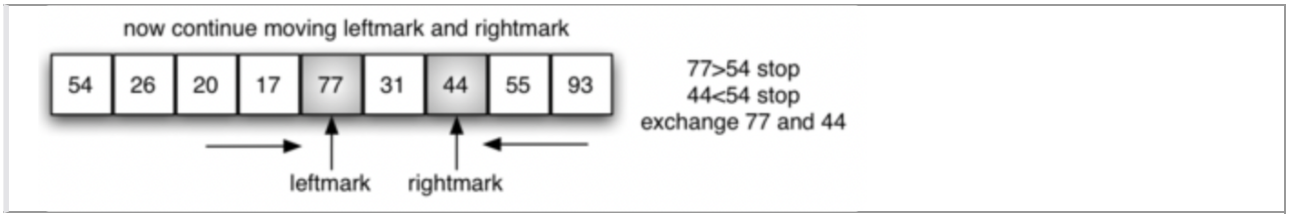
- Partitioning begins by locating two position markers—let's call them `leftmark` and `rightmark` —at the beginning and end of the remaining items in the list.
- The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point.

Process

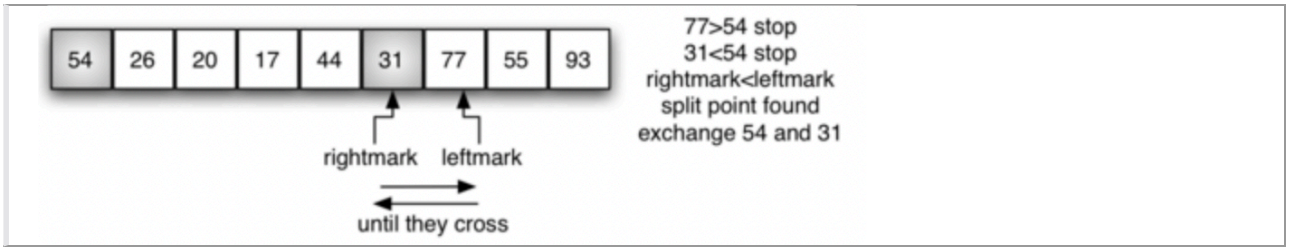
- We begin by incrementing leftmark until we locate a value that is greater than the pivot value.
- We then decrement rightmark until we find a value that is less than the pivot value.
- At this point we have discovered two items that are out of place with respect to the eventual split point. **For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again.**



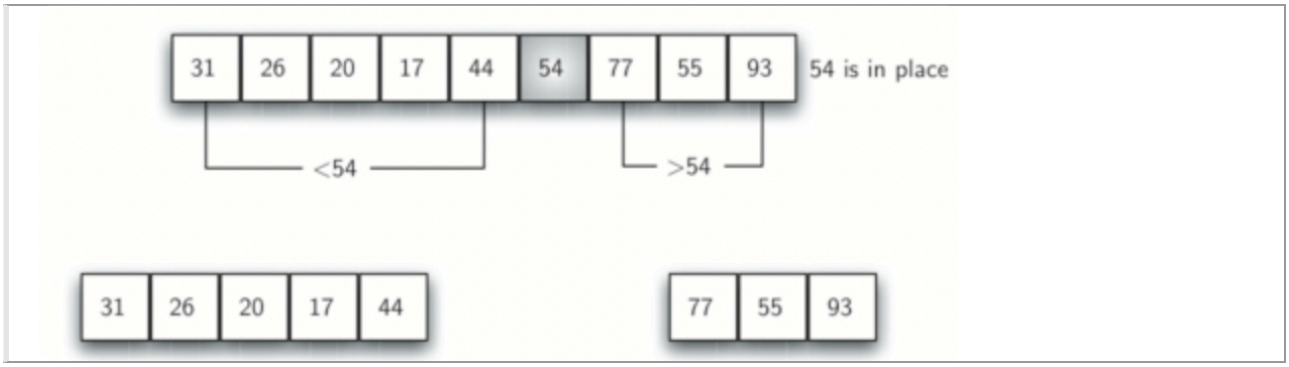
Continue:



- **At the point where rightmark becomes less than leftmark, we stop**. The position of rightmark is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place

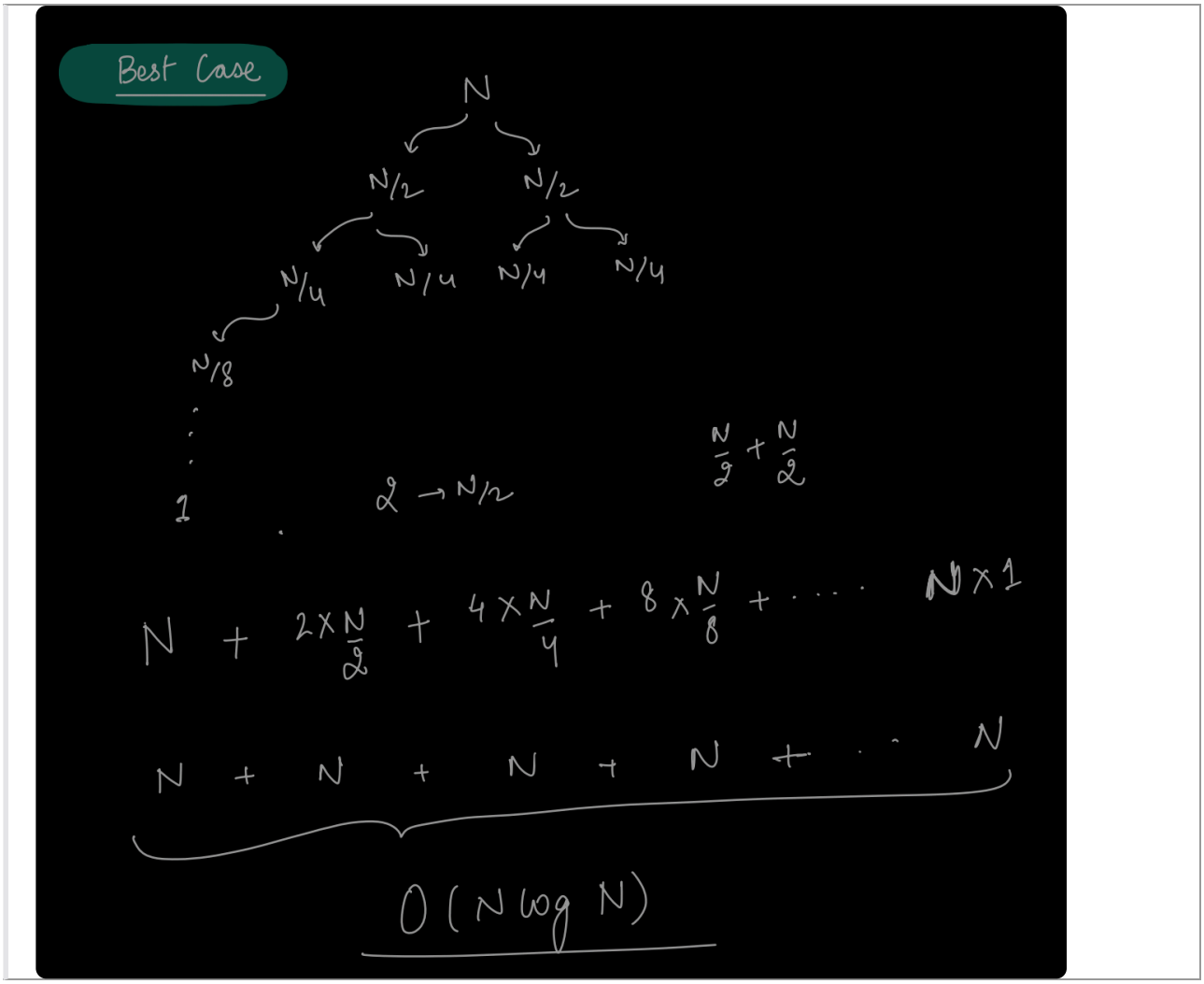


- **Now, we can exchange the 54(Pivot) with 31**



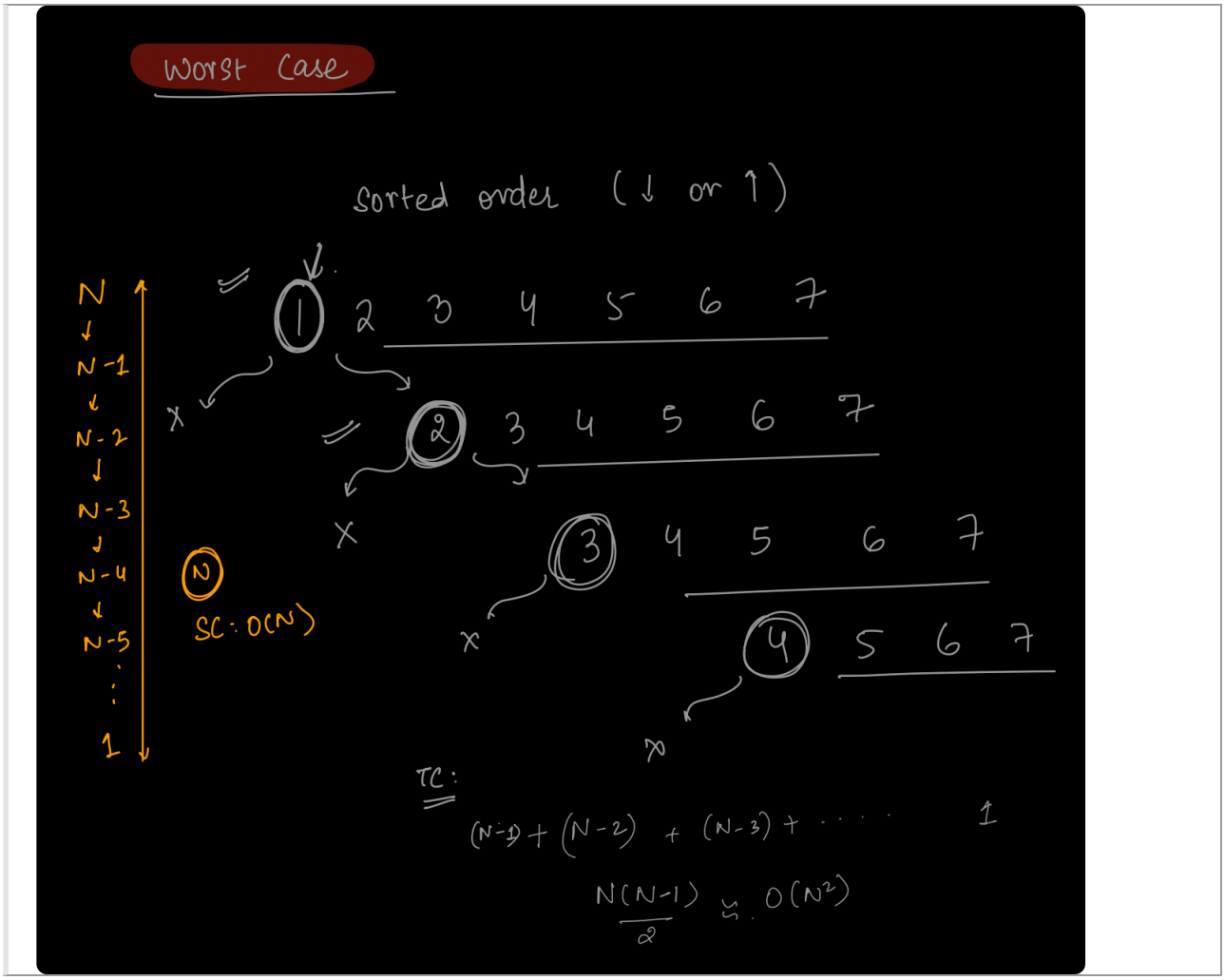
Best-Case Time Complexity:

The best-case scenario for QuickSort occurs when the pivot chosen at each step divides the input into approximately equal-sized subarrays.



Worst-Case Time Complexity:

The worst-case scenario for QuickSort occurs when the pivot chosen at each step is either the smallest or largest element in the remaining unsorted portion of the array. This leads to imbalanced partitions, and the algorithm performs poorly. The worst-case time complexity is $O(N^2)$, which occurs when the input is already sorted in ascending or descending order.



Average-Case Time Complexity

There are many ways to avoid the worst case of quicksort, like choosing the element from the middle of the array as pivot, randomly generating a pivot for each subarray, selecting the median of the first, last, and middle element as pivot, etc. By using these methods, we can ensure equal partitioning, on average. Thus, quick sort's average case time complexity is O(NlogN)

Space Complexity

The Space Complexity in quick sort will be because of recursion space. Partition function doesn't take any extra space.
So, space in Quick Sort is only because of Recursion Stack whereas in Merge Sort, the extra space is also taken by Merge Fucntion.

Comparator

- In programming, a **comparator** is a function that compares two values and returns a result indicating whether the values are equal, less than, or greater than each other.
- The **comparator** is typically used in sorting algorithms to compare elements in a data structure and arrange them in a specified order.

Comparator is a function that takes **two arguments**.

For languages - **Java, Python, JS, C#, Ruby**, the following logic is followed.

```
1. In sorted form, if first argument should come before second, -ve value is returned.
2. In sorted form, if second argument should come before first, +ve value is returned.
3. If both are same, 0 is returned.
```

For **C++**, following logic is followed.

```
1. In sorted form, if first argument should come before second, true is returned.
2. Otherwise, false is returned.
```

Problem Statement

Given an array of size n, sort the data in ascending order of count of factors, if count of factors are equal then sort the elements on the basis of their magnitude.

Example

```
A[ ] = { 9, 3, 10, 6, 4 }
Ans = { 3, 4, 9, 6, 10 }
```

Java

```
//please write the code for finding factors by yourself

public ArrayList<Integer> solve(ArrayList<Integer> A) {
    Collections.sort(A, new Comparator<Integer>(){
        @Override
        public int comp(Integer v1, Integer v2){
            if(factors(v1) == factors(v2)){
                if(v1<v2) return -1;
                else if(v2<v1) return 1;
                return 0;
            }
            else if(factors(v1)<factors(v2)){
                return -1;
            }
            return 1;
        }
    });
    return A;
}
```

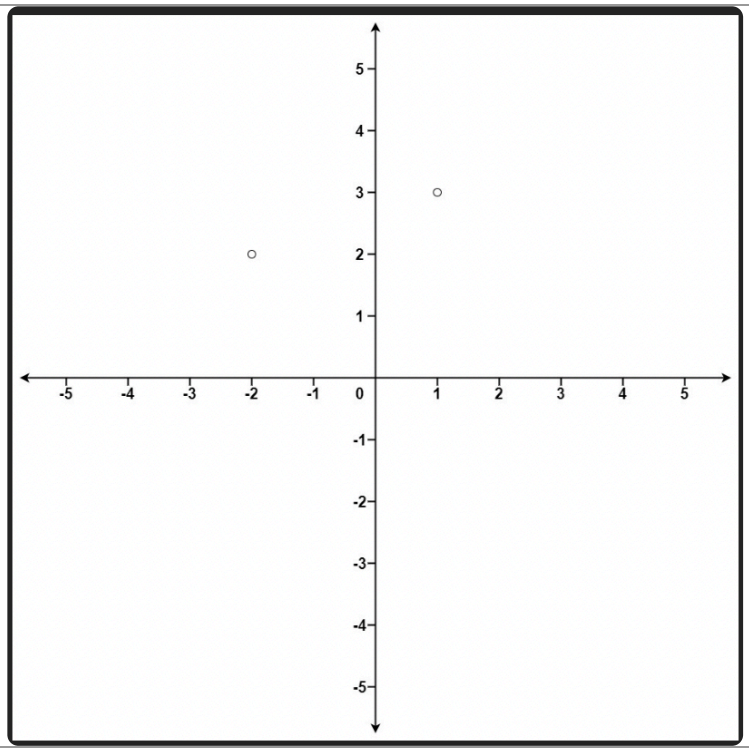
Problem Statement

Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane and an integer k, return the B closest points to the origin (0, 0).

The distance between two points on the X-Y plane is the Euclidean distance (i.e., $\sqrt{(x1 - x2)^2 + (y1 - y2)^2}$).

You may return the answer in any order.

Example 1:



Input: points = [[1,3],[-2,2]], B = 1
Output: [[-2,2]]
Explanation:
The distance between (1, 3) and the origin is sqrt(10).
The distance between (-2, 2) and the origin is sqrt(8).
Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.
We only want the closest B = 1 points from the origin, so the answer is just [[-2,2]].

Logic for Custom Sorting

Say there are two points, (x1, y1) and (x2, y2),
The distance of (x1, y1) from origin will be $sqrt((x1 - 0)^2 + (y1 - 0)^2)$
The distance of (x2, y2) from origin will be $sqrt((x2 - 0)^2 + (y2 - 0)^2)$

We can leave root part and just compare $(x1^2 + y1^2) and (x2^2 + y2^2)$

Below logic works for languages like - Java, Python, JS, …

```
// Need to arrange in ascending order based on distance

// If first argument needs to be placed before, negative gets returned
if((x1*x1 + y1*y1) < (x2*x2 + y2*y2))
    return -1;
// If second argument needs to be placed before, positive gets returned
else if ((x1*x1 + y1*y1) > (x2*x2 + y2*y2))
    return 1;
// If both are same, 0 is returned
else return 0
-------------------------------------------
// Instead of writing like above, we could have also written

return ((x1*x1 + y1*y1) - (x2*x2 + y2*y2))
```