VIDEO1 → Prefix Sum :- 1D prefix sums.

Prefix sum is a powerful technique that can be used to preprocess an array to facilitate fast subarray sum queries without modifying original array.

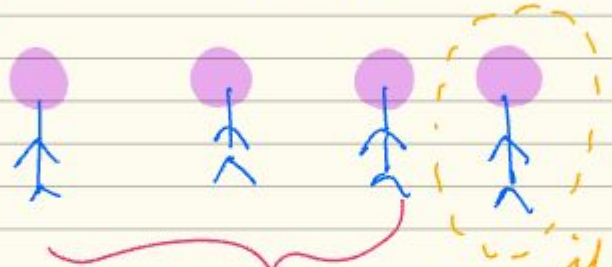
Ex:- Prefix sum :-

arr = [1, 2, 9, -1, -2, -3]

prefixArr: [1, 3, 12, 11, 9, 12].

arr → [2, 6, 9, 1, 4]

prefix sum → [2, 2+6, 2+6+9, 2+6+9+1, 2+6+9+1+4]



if we are standing here then.

→ we are talking about these persons.

arr → [4, 6, 9, 12, 1]

Q. queries :- find the sum b/w l and r :-

(l, r) :- (i) (0, 4)
(ii) (1, 3)
(iii). (2, 6)

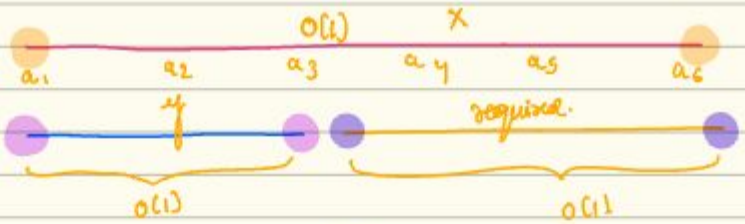
Q. Sum of elements from index l to r.

BRUTE FORCE :-

(i) Input the array, l, r

(ii) Loop through l to r and find the sum.

```
{
    sum=0;
    for(int i = l ; i<=r; i++){
        sum += arr[i];
    }
    return sum;
}
```



∴ require = x - y

If I have prefix sum vector with me, I can answer every query in O(1).

[l, r]

a_0	a_1	a_2	a_3	a_4	a_5
a_0	a_0+a_1	$a_0+a_1+a_2$	$a_0+a_1+a_2+a_3$	$a_0+a_1+a_2+a_3+a_4$	$a_0+a_1+a_2+a_3+a_4$

$P[2] + a_3$ → $P[3]$ → $P[3] + a_4$.

∵ P[i] = P[i-1] + a[i].

∴ prefix[r] - prefix[l-1] (0-based indexing).

→ Code with prefix sum

```
while(q--){
    int l, r;
    cin >> (pr[r] - ((l==0) ? 0 : pr[l-1]));
}
```

runs for q times.

∴ l=0 is an edge case, if l is zero then we cannot subtract it will give me a runtime error.

→ prefix[0] = a[0]

```
for(int i=1; i<n; i++){
    prefix[i] = prefix[i-1] + v[i];
}
```

runs for n time.

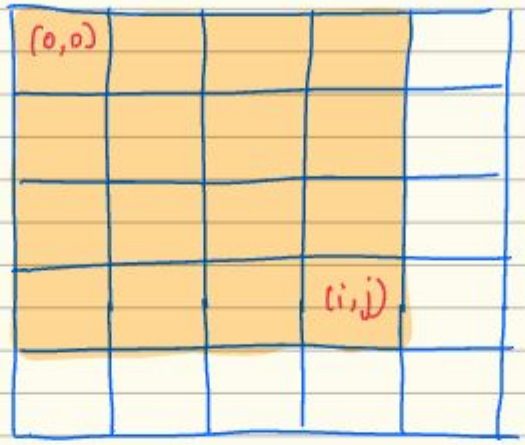
Time Complexity:- O(n+q)

Space Complexity:- O(N)

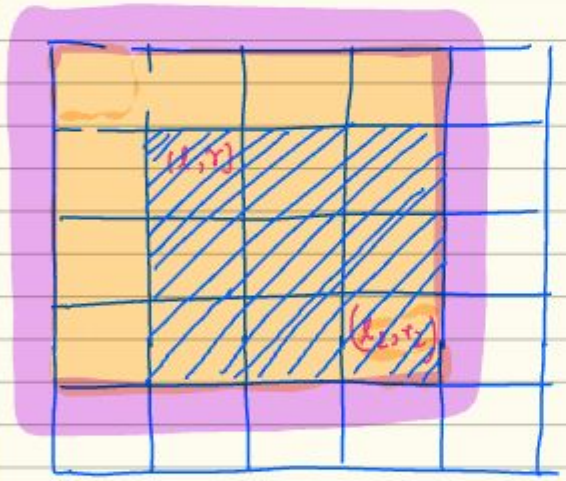
2D prefix sum :-

Similar to 1D prefix sums but extended to 2D arrays

Find the sum of elements in coloured rectangle in O(1).



p[i][j] = sum of all elements of the form grid[x][y] such that $x \le i$ and $y \le i$.



O(1)

$\therefore P[l_2][r_2] - P[l_1-1][r_2] - P[l_2][r_1-1] + P[l_1-1][r_1-1]$.

Code :- 2D prefix sum :-

```
int[][] v = new int[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            v[i][j] = scanner.nextInt();
        }
    }

    // Compute prefix sum matrix
    int[][] pre = new int[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            pre[i][j] = v[i][j];
            if (j > 0) {
                pre[i][j] += pre[i][j - 1];
            }
            if (i > 0) {
                pre[i][j] += pre[i - 1][j];
            }
            if (i > 0 && j > 0) {
                pre[i][j] -= pre[i - 1][j - 1]; // Avoid double-counting
            }
        }
    }

    // Process queries
    int q = scanner.nextInt();
    while (q-- > 0) {
        int l1 = scanner.nextInt();
        int r1 = scanner.nextInt();
        int l2 = scanner.nextInt();
        int r2 = scanner.nextInt();

        int ans = pre[l2][r2];
        if (l1 > 0) {
            ans -= pre[l1 - 1][r2];
        }
        if (r1 > 0) {
            ans -= pre[l2][r1 - 1];
        }
        if (l1 > 0 && r1 > 0) {
            ans += pre[l1 - 1][r1 - 1]; // Add back the overlapping region
        }
        System.out.print(ans + " ");
    }
```

VIDEO 2 :- **Difference Arrays :-**

Q1. Given an array of N elements, answer the following queries.

- QUERY :- L, R
- Result :- $A[L] + 2 * A[L+1] + 3 * [L+2] + \dots [R-L+1] * A[R]$.

$A = [2 \quad 4 \quad 3 \quad 6 \quad 9]$

$Q = [2 \quad 4]$

↳ $A[2] + 2 \times A[3] + 3 \times A[4]$

$Q = [1, 4]$

↳ $A[1] + 2 * A[2] + 3 \times A[3] + 4 \times A[4]$.

Solution :-

for every element we are multiplying the current element with :-

$\because i - L + 1$

$A_L + 2 \times A_{L+1} + 3 \times A_{L+2} + \dots$

So for any query :-

$Q :- (L, R)$

$$\sum_{i=L}^{R} A_i \times (i - L + 1)$$

$\because$ Whenever these type of questions come try to write them in terms of summation.

$\therefore$ General Term

- Expanding

$$\sum_{i=L}^{R} i \times A_i - (L-1) \times A_i$$

$\Rightarrow (\Sigma a + b)$

↳ $\Sigma a + \Sigma b$

↳ Expanding.

$\therefore \sum_{i=L}^{R} i \times A_i - \sum_{i=L}^{R} (L-1) \times A_i$

↳ Can we calculate them in O(1) ?

in normal prefix sums :- (we calculate) :- A_1 A_2 A_3 A_4.

Here, we calculate :- $1 \times A_1$ $2 \times A_2$ $3 \times A_3$ $4 \times A_4$.

Now,

$(L-1) \sum_{i=L}^{R} A_i \longrightarrow O(1)$

A_1 A_2 A_3 A_4 A_5 A_6 A_7

$1 \times A_1$ $2 \times A_2$ $3 \times A_3$ $4 \times A_4$ $5 \times A_5$ $6 \times A_6$ $7 \times A_7$

Calculate prefix Sum :-

$L = 3 \quad R = 5$

$\therefore P[5] - P[2] \longrightarrow i \times A_i$

```
prefix1[i] = prefix1[i-1] + (i * v[i]);
prefix2[i] = prefix2[i-1] + v[i];
while(q--){
    int l, r
    input(l, r);
cout<<(prefix1[r] - prefix1[l-1]) - (l-1) * (prefix2[r] - prefix2[l-1]);
}
```

Problem Overview :-

Given an array v of size n, we need to answer q queries. Each query asks for the sum of (i - l + 1) * v[i] for all elements from index l to r (inclusive).

In other words, for a subarray from l to r, we want:

∴ sum = (1)*v[l] + (2)*v[l+1] + (3)*v[l+2] + ... + (r-l+1)*v[r]

For each query (l, r):

pre1[r] - pre1[l-1] gives the sum of i*v[i] from l to r

pre2[r] - pre2[l-1] gives the sum of v[i] from l to r

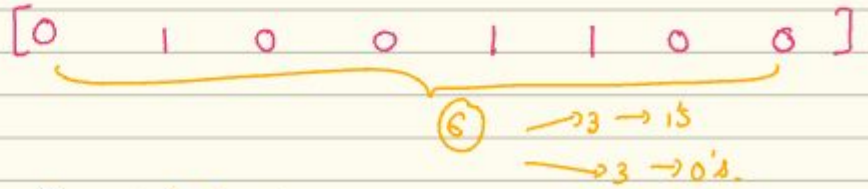
The result is (sum of i*v[i]) - (l-1)*(sum of v[i])

∴ TC: O(n+q)

```java
public class DifferenceArray {
    Run | Debug
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int arr[] = { 1, 2, 3, 4, 5 };
        int n = arr.length;
        int pref1[] = new int[n + 1];
        int pref2[] = new int[n + 1];
        for (int i = 0; i <= n; i++) {
            pref1[i + 1] = pref1[i] + (i * arr[i]);
            pref2[i + 1] = pref2[i] + arr[n - i - 1];
        }
        int queries = sc.nextInt();
        while (queries-- > 0) {
            int r = sc.nextInt();
            int l = sc.nextInt();

            int sum = (pref1[r] - pref1[l - 1]) - (l - 1) * (pref2[r] - pref2[l - 1]);
            System.out.println(sum);
        }
        sc.close();

    }
}
```

Q1. Contiguous Array :-

Given a binary array nums, return the maximum length of a contiguous subarray with an equal number of 0 and 1.



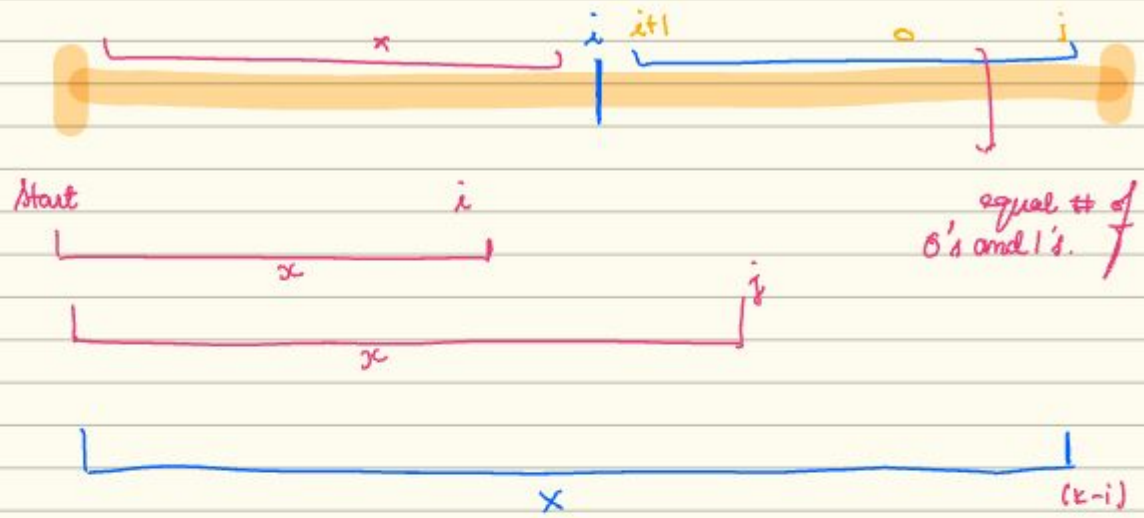
map with prefix sum :-

0 → -1
1 → +1

∴ we can calculate summation of prefixes.

summation = 0

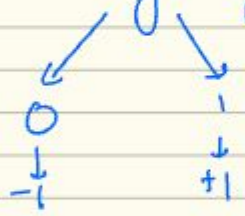
i → 1's and 0's equal



∴ (k-i+1).

Arr :- 0 1 0 1 0 0 1

Binary Arr -1 +1 -1 +1 -1 -1 +1



Prefix Arr -1 0 -1 0 -1 -2 -1

∴ So maximum sum length will be from the first -1 that has occured till the last -1.

Prefix Arr -1 0 -1 0 -1 -2 -1

Key Insight
The problem is transformed into finding the longest subarray where the count of 0s and 1s is equal. This is equivalent to finding the longest subarray where the sum is zero when we treat:

0s as -1

1s as +1

Algorithm Steps
Initialization:

sum keeps track of the running total (starting at 0)

mp (a HashMap) stores the first occurrence of each sum value

subArrayLength stores the maximum length found

Iterate through the array:

For each element:

If it's 0, add -1 to sum

If it's 1, add +1 to sum

Check three conditions:

If sum == 0: This means from index 0 to current index, we have equal 0s and 1s

Update subArrayLength to current index + 1

If sum is in the map: This means the subarray between the first occurrence of this sum and current index has equal 0s and 1s

Calculate this length (current index - first occurrence index)

Update subArrayLength if this is longer than previous maximum

Else: Store this sum with its first occurrence index in the map

```
Map <Integer, Integer> hm = new HashMap<>();
int sum=0;
int subarraylength= 0;
for(int i=0; i<n; i++){
    sum += nums[i]==0 ? -1 : 1;

    if(sum ==0){
        subarraylength = i+1;
    } else if (hm.containsKey(sum )){
        subarraylength = Math.max(subarraylength, i-map.get(sum));
    } else{
        hm.put(sum, i);
    }return subarraylength;
}
```

⇝ Difference Arrays

On difference array. can be used to perform multiple range updates where we need the final state of the arrays only after performing all the queries

Difference arrays helps us achieve above O(N) time total time & space

We can process every range update in O(1).
When we need to print our final answer we perform an O(N) computation.

⇝ Ex:- Given an array with all 0's initially perform the following Q queries on it.

In the i^{th} query you will be given 3 integers. L_i, R_i, x_i. You need to add x_i to all the values in the array from index L_i to R_i.

After performing all the queries, print the final state of the array.

↳

Q = (i) L_i R_i x_i
1 4 3
(∵ L_i, R_i → 0 based indexing).

(ii) 2 5 6
⋮
multiple such queries.

∴ So, after processing all the queries give me the final update array.

Brute force code:-

```
Void solve() {

    int n = sc.nextInt();
    int[] arr = new int[n];

    for(int i=0; i<n; i++){
        arr[i] = sc.nextInt();
    }

    int q;

    while(q--)0){
        int l, r, x = sc.nextInt();

        for(int i=l; i<=r; i++){
            v[i] += x;
        }
    }
```

query.

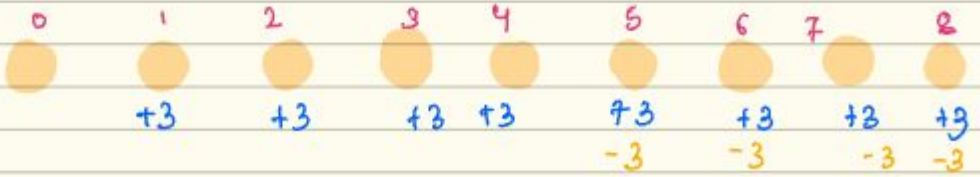
ranges.

q :-

1, 4, 3

2, 5, 2

Generically in prefix sums:- we get:-



If we add it like normal prefix sums

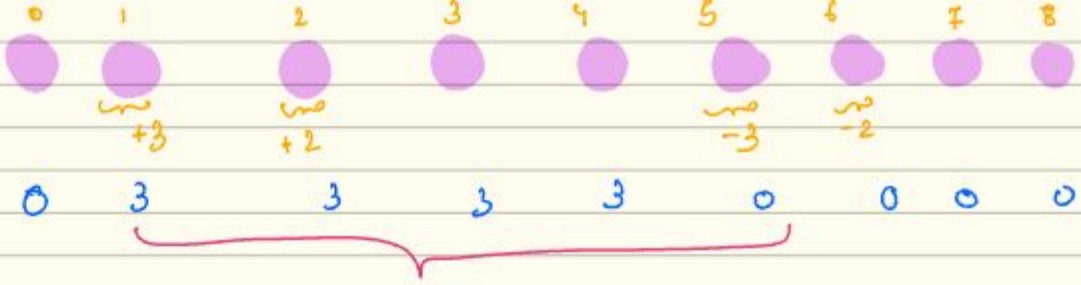
∴ Difference Array :-

↳ By how much every index value will change.

Queries:-

q1:- 1, 4, 3

q2:- 2, 5, 2



So final prefix sum :- 0 3 5 5 5 2 0 0 0
will look like this

∴ finally we take prefix sum :-

[l , r]

∴ diff [l] += x

∴ diff [r+1] -= x

↳ code :-

```
Void solve () {

    int n = sc.nextInt();
    int [] arr = new int [n];

    for (int i = 0; i < n; i++) {
        arr[i] = sc.nextInt();
    }

    int q;

    while (q-- > 0) {

        int l, r, x = sc.nextInt();

        diff [l] += x;          // from where we want to start
        diff [r+1] -= x;        // this will where we want to stop + 1
    }

    for ( int i = 1; i < n; i++) {
        diff [i] += diff [i-1];
    }                           } prefix sum.

    for (int i = 0; i < n; i++) {
        sysо ( v[i] + diff [i] + " ");
    }
```

query.

range.

TC :- O (n+q)
SC :- O(n)

Working example:-

	0	1	2	3	4	5	6	7
ORIGINAL ARRAY :-	0	+3	+3	+3	+3			
APPLYING +X & -X :-	0	+3	0	0	0	-3	0	0
Prefix Array :-	0	3	3	3	3	0	0	0

→ Problem :- Grey and Array :-

LECTURE 3 :- Coordinate Compression :-

Problem 1 :- Given an array a_n, containing integers from 1 to 10^{12}, compress them to numbers ranging from 1 to 10^6.

n numbers → $n \leq 2e5$

$a_i \leq 1e^{12}$

$1 \longrightarrow 1e^6$

$a_i < a_2$

∴ even after conversion $a_i < a_j$ must hold true.

initially $1 \longrightarrow 10^{12}$

$1 \longrightarrow 10^6$

[2, 1e11, 1e12, 1000, 40]

∴ sorting the array :-

[2, 40, 1000, 1e11, 1e12]

↓ for now on 2 is not 2, it is 0

40 → 1

1000 → 2

Coordinate compression.

Relative order is also maintained

[0, 1, 2, 3, 4]

$2e^5$ $[0 \rightarrow 1e6]$ $\leq 2e^5 - 1$

∴ so, now all the numbers in the array will be from

Ex :- 2, 40, 20, 60

↓ ↓ ↓ ↓

0 1 1 2.

D - Snuke Prime (Atcoder)

Snuke Inc. offers various kinds of services.

A payment plan called Snuke Prime is available.

In this plan, by paying C yen (the currency of Japan) per day, you can use all services offered by the company without additional fees.

You can start your subscription to this plan at the beginning of any day and cancel your subscription at the end of any day.

Takahashi is going to use N of the services offered by the company.

He will use the i-th of those services from the beginning of the a_i-th day until the end of the b_i-th day, where today is the first day.

Without a subscription to Snuke Prime, he has to pay c_i yen per day to use the i-th service.

Find the minimum total amount of money Takahashi has to pay to use the services.

Constraints

- $1 \leq N \leq 2 \times 10^5$
- $1 \leq C \leq 10^9$
- $1 \leq a_i \leq b_i \leq 10^9$
- $1 \leq c_i \leq 10^9$
- All values in input are integers.

Input

Input is given from Standard Input in the following format:

$$
\begin{array}{lll}
N & C & \\
a_1 & b_1 & c_1 \\
\vdots & & \\
a_N & b_N & c_N
\end{array}
$$

N services

i^{th} service a_i b_i c_i

a_i ↑ start date

b_i ↑ end date

c_i ↑ amount of money to be paid every day.

Ex:-

a_i	b_i	c_i
2	4	6

So we need to pay:-

6 amount on → day 2

day 3

day 4

Ex:-

queries:-

1) 2 4 6

2) 4 5 5

C = 10

(Subscription price only for that particular day).

Day 2	day 3	day 4	day 5	day 6.
6	~~6~~	~~6~~		
	10	~~5~~ 10	5	

· When A, B <= 1e 6

	ai	bi	ci
Service 1 →	2	4	6
Service 2 →	4	6	5

c = 10. → if you pay amount 10 then you can avail service for that whole day.

	day 2	day 3	day 4	day 5	day 6
	6	6	6		
			5	5	5
Total Cost:-	6	6	~~11~~	5	5

avail = 10 subscription

∴ So, it minimize final service cost.

APPROACH - 1:-

Difference Array.

Assume days as index.

ai _ _ _ _ _ _ bi

ci ci ci ci ci → ∵ We cant add ci to whole query as that might give TLE.

Hence, we can apply difference array concept.

day 2, 4, 5

4, 5, 5

$10^6 + 1$ day 1 → day 10^6 → every index represents a day.

2	3	4	5	6	7	8	9
+6		+5	-6				

0	0	6	6	11	5	5	0	0	0

Example 2:-

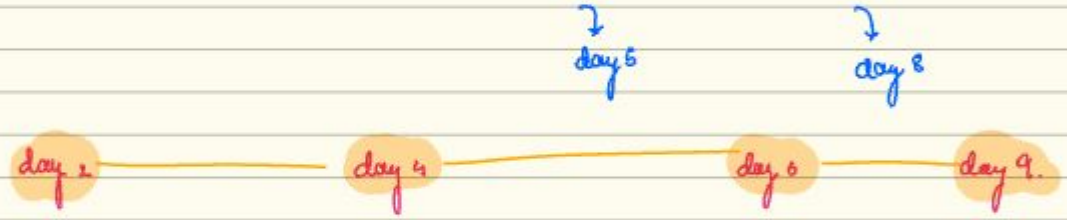
(i) 2 5 c

(ii). 4 8 9

Info 1 :→ day 7 & day 8 cost will be equal to day 6.

Info 2:→ Cost on day 6.

So, if day 5 and day 8 are given & we need to predict their costs we can doit by looking at the changes in values.



So we can declare ai, bi+1 → as important points.

∴ So if we form the difference array and map the important points we can make it a comparision array out of it.

∴ So if I know the answers for important points, we will also know the answers for all days.

∴ ai bi+1 → important points.

So, from the previous problem:-

We can map all the important points.

```
Set < Integer > st                    → stores unique elements in ascending order.

for(int i=0; i<n; i++){
        st.insert(start[i]);
        st.insert(end[i]+1);
    }

int idx = 0;                          → started with 0, the first point gets mapped with 0, the 1, third 2, ..... and so on.
Map<Integer, Integer> mp;             → map for mapping points.

    for(int i: st){
        mp[i] = idx;
        idx ++;
    }
```

∴ Size of set = st.size() - 1 or idx + 1

continued.

```
int [ ] diff = new int [idx+1, 0];

    for(int i=0; i<n; i++){
        diff(mp[start[i]]) += cost[i];
        diff(mp[end[i]+1]) -= cost[i];
    }                                 } difference array, getting values from map.

    for(int i = 1; i<= idx; i++){
        diff[i] += diff[i-1];
    }                                 } prefix sum
```

day 2	day 4	day 6	day 9
↓	↓	↓	↓
0	1	2	3
6	9	-6	-9
6	15	9	0

∴ Important points

∴ So for all the important points we have the answer.

~> Now, how to get answer for all other points if we have answer for all important points?

Code:-

→ Set will now look like:-

SET	DIFFERENCE ARRAY
2	6
4	15
6	9
9	0

If we subtract 4-2 → we will get no. of days having cost 6.

6 - 4 → no. of days having cost as 15.

SET

4 - 2 → 6

6 - 4 → 15

9 - 6 → 9

difference b/w nos:-

{

span = a[i] - a[i-1].

ans = max.min(diff[i-1], i)* span;

}

```java
public class SnukePrime {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt();
        long c = sc.nextLong();

        int[] start = new int[n];
        int[] end = new int[n];
        long[] cost = new long[n];

        for (int i = 0; i < n; i++) {
            start[i] = sc.nextInt();
            end[i] = sc.nextInt();
            cost[i] = sc.nextLong();
        }

        // Create a set to store all important time points
        Set<Integer> timeSet = new HashSet<>();
        for (int i = 0; i < n; i++) {
            timeSet.add(start[i]);
            timeSet.add(end[i] + 1);
        }

        // Convert to sorted list
        List<Integer> times = new ArrayList<>(timeSet);
        Collections.sort(times);

        int idx = 0;
        Map<Integer, Integer> timeToIndex = new HashMap<>();
        for (int time : times) {
            timeToIndex.put(time, idx++);
        }

        // Create difference array for costs
        long[] diff = new long[idx + 1];

        for (int i = 0; i < n; i++) {
            int startIdx = timeToIndex.get(start[i]);
            int endIdx = timeToIndex.get(end[i] + 1);
            diff[startIdx] += cost[i];
            diff[endIdx] -= cost[i];
        }

        // Convert difference array to actual costs
        for (int i = 1; i <= idx; i++) {
            diff[i] += diff[i - 1];
        }

        long totalCost = 0;

        for (int i = 0; i < times.size() - 1; i++) {
            int duration = times.get(i + 1) - times.get(i);
            long dailyCost = diff[i];
            long subscriptionCost = (long) duration * c;

            totalCost += Math.min(dailyCost, subscriptionCost);
        }

        System.out.println(totalCost);
        sc.close();
```

// n log
n log n
n
n

Overall TC :- $O(n \log n)$.

~ Steps for using coordinate compression :-

(i). Figure out the important points.

(ii) Map out the important points.

(iii). Solve questions with only those important points.

(iv). Figure out the answer for all remaining points.

MODULE - 2, BIT MANIPULATION BEGGINNER.

→ Video 1 :- Bit Manipulation.

~ Number System :- The systems we use runs through a decimal system with base 10. However, computers are not programmed through base 10 systems.

Early computers were programmed with base 2 (binary), base 8 (octal) and base 16 (hexadecimal) systems.

$(9)_{10} = (1001)_2$

$(9)_{10} = (11)_8$

$(10)_{10} = (A)_{16}$

10 → A
11 → B
12 → C
13 → D
14 → E
15 → F

→ Conversion of decimal number to a different base number.

(i) $(55)_{10} = (?)_2$

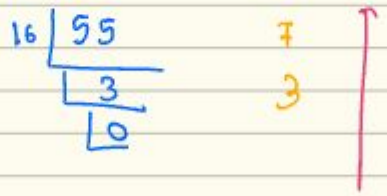
(ii) $(55)_{10} = (?)_8$

(iii) $(55)_{10} = (?)_{16}$

Answer ~ Conversion :-

(i) $(55)_{10} = (\quad)_2$

(ii). $(55)_{10} = (?)_{16}$



↝ Conversion of a number with any base to decimal number :-

(i). $(1101111)_2 = (?)_{10}$

(ii). $(67)_8 = (?)_{10}$

(iii). $(37)_{16} = (?)_{10}$.

$967 \rightarrow 9\times100 + 6\times10 + 7\times1$

$10^2 \quad 10^1 \quad 10^0$

So this means, we actually write the break down of 967. i.e. 7 has a 10 power of 0, 6 has a 10 power of 1 and so on....

$9\times10^2 + 6\times10^1 + 7\times10^0$

(i). $(1\ 1\ 0\ 1\ 1\ 1\ 1)_2 = (?)_{10}$.

$= (1\times2^0) + (1\times2^1) + (1\times2^2) + (0\times2^3) + (1\times2^4) + (1\times2^5)$

$= 55$

(ii) $(37)_{16} = (?)_{10}$

$= 7\times16^0 + 3\times16^1$

$= 55$

↝ Conversion of a number from one base to any other base.

BASE 1 → BASE 2

BASE 1 → BASE 10 → BASE 2

∵ Direct conversion from one base to another is not possible.

↝ Bitwise operator.

↝ OR :- (|)

The OR operator ("|") takes every corresponding bit of the numbers and checks whether at least one of them is set.

0	0	0
0	1	1
1	0	1
1	1	1

A ~ 1 0 1 1 0 1 0

B ~ 0 1 0 0 0 1 1

Result (A|B) :- 1 1 1 1 0 1 1

∵ 1 → Set Bit

∴ 0 → Unset Bit.

↝ AND :- (&)

Checks whether all of them are set.

0	0	0
0	1	0
1	0	0
1	1	1

⇝ XOR (^)

Checks the parity of number of set bits.

0	0	0
0	1	1
1	0	1
1	1	0

A ⇝ 1 0 1 0 1 1 0

B ⇝ 0 1 1 1 0 1 0

C ⇝ 1 1 1 1 1 1 1

0 0 1 0 0 1 1

∴ Parity of number of set bits.

↳ ODD :- resultant bit => 1

↳ even :- resultant bit => 0

⇝ Bitwise left shift operator :-

A<<B shifts the A to the left by B bits, adding B zeros at the end.

The result is same as $A * 2^B$.

$(5)_{10} \longrightarrow (101)_2$

$(10)_{10} \longrightarrow (1010)_2$

$5 \times 2^1 = 10$

$5 << 1 = 10$

⇒ a << b

⇒ $a \times 2^b$

→ 5 => 101

5 <<1 => 1010 (<<1)

→ 5 <<2

1 0 1 0 0 → 5×2^2

= 20

∴ Same (A<<B)

↝ Bitwise Right shift operator :-

∴ The result is same as $A/2^B$.

A >>B Shifts the A to the right by B bits, deleting B bits from the end.

A >>B → $A / 2^B$

integer division.

↝ Properties of Bitwise Operator:-

(i). OR, AND, XOR are associative and commutative.
(ii). A^0 = A
(iii). A^A = A
(iv). If A^B = C, then A^C = B and B^C = A
(v). A&B <= MIN(A,B)
(vi). A|B >= MAX(A,B)
(vii). (A&1) is 1 if A → odd, else 0
(viii). A & (A-1) is 0 if A is power of 2.

(i) OR, AND, XOR are associative and commutative.

Associative property:- $a + (b+c) = (a+b) + c$. ⊕

OR, AND, XOR

A → 1 0 1 0 1 A|B|C

B → 1 1 0 1 1 A&B&C

C → 1 0 0 0 0 A^(B^C)

(A^B)^C

∴ Order placement of bits/numbers do not matter.

Commutative Property:- $a+b=b+a$

AND, OR, XOR $xy=yx$.

A & (B | C) = (A & B) | C ↝ No.

A × (B + C) = (A × B) + C

↓

No.

(A × B) + (A × C) → works.

(ii). $A \wedge 0 = A$ ↝ If we XOR a number with zero we get the same number.

A → 1 0 1 1 0 1 0

∧ 0 0 0 0 0 0 0

1 0 1 1 0 1 0

∴ Hence, $A \wedge 0 = A$

↳ ∴ XOR with 0 does not affect a bit.

↳ ∴ XOR with 1 toggles a bit.

$A \wedge 1$ = will toggle a bit

1 0

1 1

0 1

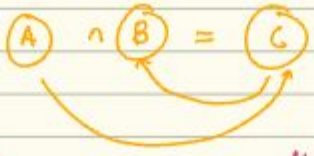
(iii). $A \wedge A = A$ ↝ XOR with 2 same numbers will be 0.

A → 1 0 1 0

A → 1 0 1 0

∧ 0 0 0 0

(iv) If $A \wedge B = C$, then $A \wedge C = B$ and $B \wedge C = A$



$A \wedge C = B$

$B \wedge C = A$.

∴ Shuffling ∧

Ex:-

$(010) \wedge (101) = (111)$

$(010) \wedge (111) = 101$

(V). $A \& B <= MIN(A,B)$ ⇝ Only if both the bits are set we get a 1 on performing AND (&) operation.

A ⟶	1	0	1	1	0
B ⟶	1	0	0	1	0
A&B ⟶	1	0	0	1	0

$A \rightarrow (1\times2^4) + (0\times2^3) + (1\times2^2) + (1\times2^1) + (0\times2^0)$

$B \rightarrow (1\times2^4) + (0\times2^3) + (0\times2^2) + (1\times2^1) + (0\times2^0)$

$(A \& B) \Rightarrow 1\times2^4 + 0\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$

∴ Hence, $A \& B <= Min(A,B)$

(VI). $A | B >= MAX(A,B)$

∵ Similar to previous question

(vii). $(A \& 1)$ is 1 if A ⟶ odd, else 0.

A ⟶ 1 0 0 1 0 1

$(1\times2^0) + (0\times2^1) + (1\times2^2) + (0\times2^3) + (0\times2^4) + (1\times2^5)$.

(0×2¹), (1×2²), (0×2³), (0×2⁴) — even.

↓

odd.

$(lsb \times 2^0) + even.$

(lsb × 2⁰) — odd

∵ (lsb = 1)

A → 1 0 1 1 1

& 0 0 0 0 1

0 0 0 0 1

∴ if (A & 1) then number is odd else A → is even.

A → 1 0 1 1 0

& 0 0 0 0 1

0 0 0 0 0

(viii). A & (A-1) is 0 if A is power of 2.

A number in power of 2 will look like ~ 1 0 0 0 0

$2^0 \rightarrow 1$
$2^1 \rightarrow 1\ 0$
$2^2 \rightarrow 1\ 0\ 0$
$2^3 \rightarrow 1\ 0\ 0\ 0$
$2^4 \rightarrow 1\ 0\ 0\ 0\ 0$
$2^5 \rightarrow 1\ 0\ 0\ 0\ 0\ 0$
$2^6 \rightarrow 1\ 0\ 0\ 0\ 0\ 0\ 0$

1 0 0 0

0 1 1 1

0 0 0 0

1 0 0 0 → 1 0 0 1 → 1 0 1 0 → 1 0 1 1

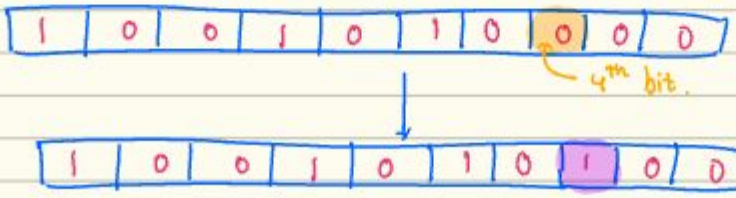
0 1 1 1

~ Video 2:- Bit Manipulation 2.

We will discuss few very important properties that must be remembered, if we solve questions in CP.

(i). SET THE k^{th} BIT

Set the 4^{th} bit in the number.



(∵ 1 based indexing).

We can apply (|) or operation with the same number, but construction of the number to operate is difficult.

```
  1  0  0  1  0  1  0  0  0  0
  0  0  0  1  0  1  1  0  0  0
-------------------------------
  1  0  0  1  0  1  1  0  0  0
-------------------------------
```

∴ we can achieve the number 0 0 0 0 0 0 1 0 0 0 by `1 << (k-1)`

```
  1  0  0  1  0  1  0  0  0  0
  0  0  0  0  0  0  1  0  0  0
-------------------------------
  1  0  0  1  0  1  1  0  0  0
-------------------------------
```

∴ To, set the k^{th} bit `A |(1 << (k-1))`

(ii) CHECK IF k^{th} BIT IS SET:-

- check if 5^{th} bit in the number 1 0 01 01 0000 is set or not?

if we apply AND operator to the binary's number k^{th} bit with 1 and if we get 1 then it is set. & vice versa.

```
  1  0  0  1  0  1  0  0  0  0
AND)&)
  0  0  0  0  0  1  0  0  0  0
-------------------------------
  1  0  0  0  0  0  0  0  0  0
-------------------------------
```

∴ `if(A & (1 << (k-1))) {`

k^{th} bit is set

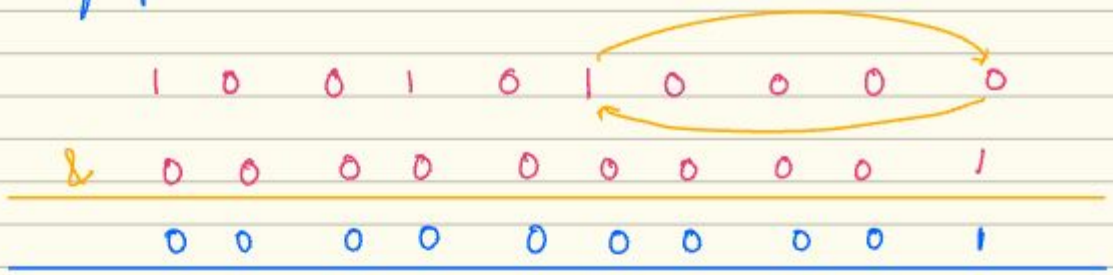
`} else{`

k^{th} bit is not set

`}`

Another approach:-

Shifting the bit to the LSB and & it.



$\therefore (A >> K-1) \& 1$

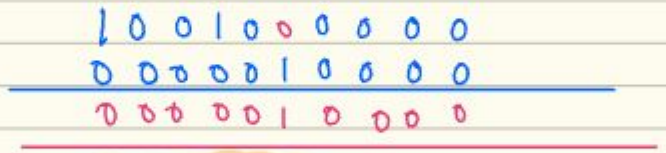
(iii) TOGGLE THE K^{th} BIT:-

Toggle the 4^{th} bit in the number 1 0 0 1 0 1 0 0 0 0

1 0 0 1 0 1 0 0 0 0

Toggle:- 0 → 1, 1 → 0 } flipping bits.

∴ Apply XOR 1 with the K^{th} bit you want to toggle.



∴ A ^ (1 << (K-1))

↳ 1 Based indexing.

(iv) UNSET THE K^{th} BIT:-

Unset the 5^{th} bit in the number 1 0 0 1 0 1 0 0 0 0

1 0 0 1 0 1 0 0 0 0

if the bit is already 0:- then no change.

A very simple approach would be:-

If the bit is set:-

```
if ( A & (1<<(K-1)) ) {
```

A = A ^ (1<<(K-1))

}

∴ If the bit is set
↳ toggle the bit.

~> Some tricks related to Bitwise Operator:-

(i). Parity of numbers of set bits in A^B.

$A+B = (A \hat{} B) + 2*(A \& B)$

```
A = | 0 | 1 | 0 | 1 | 1 | 0 |   → x
B = | 0 | 1 | 0 | 1 | 1 | 0 |   → y
    -----------------------
    | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
    -----------------------
```

∵ (X → no. of set bits in A)
∴ (Y → no. of set bits in B)

then,
what tells the parity of number of set bits in A ^B.

X+Y → Even
↳ Even
↳ Odd. → odd

(ii). $A+B = (A \hat{} B) + 2*(A \& B)$.

```
A   0 1 0 1
B   0 0 1 1
    -------
    0 1 1 0
```

But due to
XOR (1^1)
give me zero.

```
  1
+ 1
-----
  2 → (10).
```

∴ So, now if i add this thing twice, we will be adding 2 ones together.

```
  0 1 1 0
  0 0 0 1
```

∴ Hence, they equal one.

performing A+B

Ex:-

A → 1 0 1 0

B → 1 0 0 1

1 0 0 1 1

addition of binary 1+1=2 that is represented as 10 in binary.

now, performing A^B.

A → 1 0 1 0

B → 1 0 0 1

0 0 1 1

There a difference between these 2 bit numbers. (∵ Hence we add this number twice). So that we are adding it twice).

(iii). A+B = (A|B)+(A&B) :-

Practically apply the similar logic as above.

↳ Here, the AND resolves the query.

(iv). Swap two numbers without using swap function and temporary variable.

Apply this to swap a number

A = A^B

B = A^B

A = A^B

A

↓

A^B

B ⟵ (A^B)^B

B ⟵ A

A ⟵ (A^B)^A

= B

∴ Hence, swapped!!

~> Problem:-

B. Make Almost Equal With Mod

time limit per test: 1 second
memory limit per test: 256 megabytes

xi - Solar Storm

You are given an array $a_1, a_2, \ldots, a_n$ of distinct positive integers. You have to do the following operation **exactly once**:

- choose a positive integer k;
- for each i from 1 to n, replace a_i with $a_i \bmod k$[1].

Find a value of k such that $1 \le k \le 10^{18}$ and the array $a_1, a_2, \ldots, a_n$ contains **exactly** 2 distinct values at the end of the operation. It can be shown that, under the constraints of the problem, at least one such k always exists. If there are multiple solutions, you can print any of them.

[1] $a \bmod b$ denotes the remainder after dividing a by b. For example:

- $7 \bmod 3 = 1$ since $7 = 3 \cdot 2 + 1$
- $15 \bmod 4 = 3$ since $15 = 4 \cdot 3 + 3$
- $21 \bmod 1 = 0$ since $21 = 21 \cdot 1 + 0$

Input

Each test contains multiple test cases. The first line contains the number of test cases t ($1 \le t \le 500$). The description of the test cases follows.

The first line of each test case contains a single integer n ($2 \le n \le 100$) — the length of the array a.

The second line of each test case contains n integers $a_1, a_2, \ldots, a_n$ ($1 \le a_i \le 10^{17}$) — the initial state of the array. It is guaranteed that all the a_i are distinct.

Note that there are no constraints on the sum of n over all test cases.

Output

For each test case, output a single integer: a value of k ($1 \le k \le 10^{18}$) such that the array $a_1, a_2, \ldots, a_n$ contains exactly 2 distinct values at the end of the operation.

Example

input

```
5
4
8 15 22 30
5
60 90 98 128 388
6
328 769 541 986 215 734
5
1000 2000 7000 11000 16000
2
2 1
```

output

```
7
30
3
5000
1000000000000000000
```

Note

In the first test case, you can choose $k = 7$. The array becomes $[8 \bmod 7, 15 \bmod 7, 22 \bmod 7, 30 \bmod 7] = [1, 1, 1, 2]$, which contains exactly 2 distinct values ($\{1, 2\}$).

In the second test case, you can choose $k = 30$. The array becomes $[0, 0, 8, 0, 8]$, which contains exactly 2 distinct values ($\{0, 8\}$). Note that choosing $k = 10$ would also be a valid solution.

In the last test case, you can choose $k = 10^{18}$. The array becomes $[2, 1]$, which contains exactly 2 distinct values ($\{1, 2\}$). Note that choosing $k = 10^{18} + 1$ would not be valid, because $1 \le k \le 10^{18}$ must be true.

a = 2 4 9 6 3 ← distinct.

ai %k : 0 0 1 0 1

ai = ai % k

k = 2.

exactly 2 distinct values.

ai = ai % 2.

You have to choose some k

~> Solution:-

even odd ⟶ 2

we will generalize the things.

⟶ 4 , 8 , 10 , 12 , 11 , 13

% 8 → 4 , 0 , 2 , 4 , 3 , 5.

Let's write the numbers in binary notation :-

4 → 0 0 1 0 0 → 4

8 → 0 1 0 0 0 → 0

10 → 0 1 0 1 0 → 2

12 → 0 1 1 0 0 → 4

11 → 0 1 0 1 1 → 3

13 → 0 1 1 0 1 → 5

8
1
2^3

if you did for 4
4
1
2^2
then last 2 bits will be considered.

∴ **Observation :-** The last 3 bits are the the remainder for applying mod to a number n.

∴ If we divide a number with 2^k → then I get last k bits in binary representation of number as my remainder.

Approach :-

A1 — — — — — 1 1 0

A2 0 1 0

A3 0 1 0

A4 1 1 0

2^1
2^2
2^3

∴ So, if we keep dividing from 2^1 to 2^{13}, we would definately find a k with 2 distinct numbers.

K = some power of 2 (1 → 0)

if the numbers is odd, the you can get your first distinct number from 2^1

but for even LSB is always zero, ∧0 in order to make your first distinct element you need 2^2.

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        solve();
    }

    static void solve() {
        Scanner scanner = new
Scanner(System.in);
        long n = scanner.nextLong();
        long[] v = new long[(int)n];
        for (int i = 0; i < n; i++) {
            v[i] = scanner.nextLong();
        }

        for (int i = 1; i <= 61; i++) {
            long num = (1L << i);
            Set<Long> st = new HashSet<>();
            for (int j = 0; j < n; j++) {
                st.add(v[j] % num);
            }
            if (st.size() == 2) {
                System.out.println(num);
                return;
            }
        }
```

Key Observations:

1. **Modulo Operation (`%`):**
   - When we compute `x % num`, the result is the remainder when `x` is divided by `num`.
   - Since `num` is a power of 2 (`2^i`), the possible remainders range from `0` to `num - 1`.
2. **Why Two Distinct Remainders?**
   - If all elements modulo `num` produce exactly **two distinct values**, it means the array can be split into two groups based on their remainders.
   - This is useful in problems where we need to classify numbers into two categories (e.g., binary classification, XOR-based partitioning, etc.).
3. **Why Check Powers of 2 (`2^i`)?**
   - Powers of 2 have unique properties in binary representation (they have a single `1` followed by zeros).
   - Checking remainders with `2^i` helps in identifying a bit position where numbers differ.

Step-by-Step Explanation of the Code

1. **Input Reading:**
   - The code reads `n` (size of the array) and then the array elements.
2. **Loop Over Powers of 2 (`2^1` to `2^61`):**
   - For each `i` from `1` to `61`, it computes `num = 2^i`.
3. **Compute Remainders:**
   - For each element in the array, compute `v[j] % num` and store the result in a **set** (to track unique remainders).
4. **Check for Exactly Two Unique Remainders:**
   - If the set size is exactly `2`, it means the array can be divided into two distinct groups when taken modulo `num`.
   - The smallest such `num` is printed, and the program exits.

Video 3 :- Problem solving 1

→ Bitwise Equation :- Codechef - difficulty 1679.

Difficulty: 1679

Statement | Submissions | Solution | Hints | AI Help

Bitwise Equation

Given an integer N, find four **positive distinct** integers a, b, c and d such that:

- $1 \leq a, b, c, d \leq 10^{18}$
- $((a \,\&\, b) \mid c) \oplus d = N$

Here $\&$, $\mid$, and $\oplus$ represent bitwise AND, OR and XOR, respectively.

If there are multiple answers, print **any** of them. If no answer exists, print -1.

Input Format

- The first line of input will contain a single integer T, denoting the number of test cases.
- Each test case consists of one line of input, containing a single integer N.

Output Format

For each test case, output -1 if there is no way to find four distinct integers to satisfy the equation.

Otherwise, print on a new line **any** four space-separated integers a, b, c and d that satisfy the conditions.

Constraints

- $1 \leq T \leq 10^4$
- $0 \leq N < 2^{32}$

Sample 1:

Given :-

$((a \,\&\, b) \mid c) \oplus d = N$

$a, b, c, d \leq 10^{18}$

$N < 2^{32}$

N (any number can be represented in 32 bits). $N < 2^{32}$.

---

a, b, c, d

$((a \,\&\, b) \mid c) \wedge d = n$

∵ The final operation that we doing is the XOR ops.

For ex:-

$n = 10110$

and d contains n (i.e. all those set in n are also set in D. (now in D, there's one more set bit).

$((a \,\&\, b) \mid c) \wedge d = n$ → 10110.

$(a \,\&\, b \mid c)$:- 0 0 0 1 0 0 0 0 0 0 0 0 0

d → 0 0 0 1 0 0 0 1 0 1 1 0

d ↗ n

n

∴ $d = (1L << 35) + n$

∴ In my d, all bits that were set in n will be set and additionaly bit no. 35 will be set.

no, we have to somehow make :-

∴ $(a \,\&\, b) \mid c = 1LL << 35$

$c = (1LL << 35)$ ∴ as we must take a true number.

$a \,\&\, b = 0$

$d = (1LL << 35) + (1LL << 36) + n$

$(a \& b = 1LL << 36)$

$a = 1LL << 37 + 1LL << 36$
$b = 1LL << 36.$

$(a \& b) \mid c = \underbrace{(1LL << 35)}_{\text{coming from } c} + \underbrace{(1LL << 36)}_{\text{coming from } a \& b.}$

⇝ Problem 2:- Given an array N elements, answer the following Q queries.

(i) Queries:- L, R

(ii). Result:- $A[L] \wedge A[L+1] \wedge A[L+2] \wedge \ldots \wedge A[R]$.

$Q \approx 10^5$
$N \approx 10^5$

Solution:-

$a \wedge a = 0$

Property:- $a \wedge a = a$.

$a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7$

$pxf[r] \leftarrow (a_1 \wedge a_2 \wedge a_3 \wedge a_4 \wedge a_5 \wedge a_6) \wedge (a_1 \wedge a_2) \rightarrow pxf[l-1]$

$(a_1 \wedge a_2 \wedge a_3 \wedge a_4 \wedge a_5 \wedge a_6)$ → XOR of prefix till idx 6

$(a_1 \wedge a_2)$ → XOR of prefix till idx 1.

$\downarrow$

$a_3 \wedge a_4 \wedge a_5 \wedge a_6$

TC: $O(N+Q)$.

→ Problem 3 = Fortune Telling.

B. Fortune Telling

time limit per test: 1 second
memory limit per test: 256 megabytes

Haha, try to solve this, SelectorUnlimited!
— antontrygubO_o

Your friends Alice and Bob practice fortune telling.

Fortune telling is performed as follows. There is a well-known array a of n non-negative integers indexed from 1 to n. The tellee starts with some non-negative number d and performs one of the two operations for each $i = 1, 2, \ldots, n$, **in the increasing order of** i. The possible operations are:

- replace their current number d with $d + a_i$
- replace their current number d with $d \oplus a_i$ (hereinafter $\oplus$ denotes the bitwise XOR operation)

Notice that the chosen operation may be different for different i and for different tellees.

One time, Alice decided to start with $d = x$ and Bob started with $d = x + 3$. Each of them performed fortune telling and got a particular number in the end. Notice that the friends chose operations independently of each other, that is, they could apply different operations for the same i.

You learnt that either Alice or Bob ended up with number y in the end, but you don't know whose of the two it was. Given the numbers Alice and Bob started with and y, find out who (Alice or Bob) could get the number y after performing the operations. It is guaranteed that on the jury tests, **exactly one** of your friends could have actually gotten that number.

Hacks

You cannot make hacks in this problem.

Input

On the first line of the input, you are given one number t ($1 \le t \le 10^4$) — the number of test cases. The following $2 \cdot t$ lines contain test cases.

The first line of each test case contains three numbers n, x, y ($1 \le n \le 10^5$, $0 \le x \le 10^9$, $0 \le y \le 10^{15}$) — the length of array a, Alice's initial number (Bob's initial number is therefore $x + 3$), and the number that one of the two friends got in the end.

The second line of each test case contains n numbers — the array a ($0 \le a_i \le 10^9$).

It is guaranteed that the sum of n over all test cases does not exceed 10^5.

Output

For each test case, print the name of the friend who could get the number y: "Alice" or "Bob".

Alice
d

Bob
d+x

Who will end by y.

a_1 a_2 a_3 ..... a_n.

$d = ((d + a_1) \oplus a_2) + a_3) + a_4)$

→ Test case :-

Alice :- 1 Bob :- 7 Fortune :- 9.
(Who will reach this first).

$a_1 \quad a_2 \quad a_3 \quad \cdots \quad a_n$

$\oplus$ $\quad$ $\oplus x$

OBSERVATION:- Parity of numbers that Alice and Bob are starting with are different.

∵ So by checking this, we get to know is it possible to reach y.

Ex:- changes

$3+5 = 8$

$3 \oplus 5 = 6$ → odd.

} if we do + or ⊕ with an odd number the parity of the number changes.

$3+4 = 7$ → even — remains same.

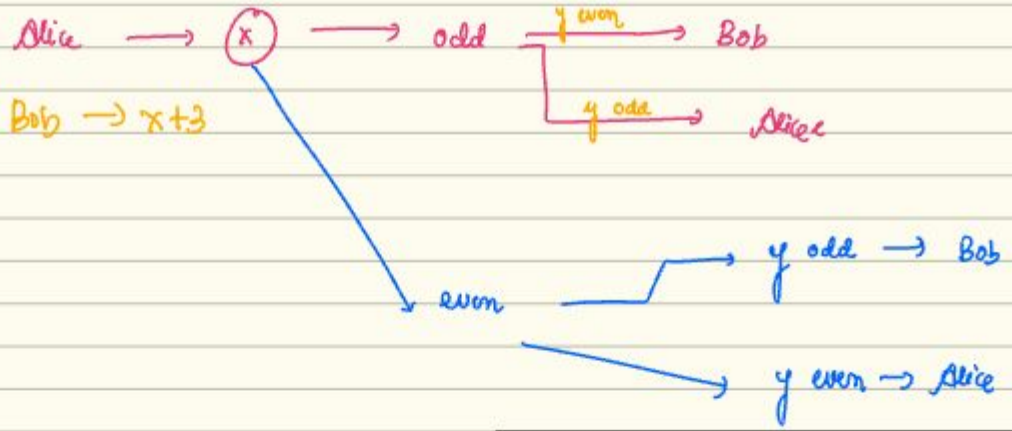
$3 \oplus 4 = 7$ → even

#) + and ⊕ has same effect on parity.

* If number is odd → parity changes
* If number is even → parity remains same.

ODD → odd

EVEN → even




```java
import java.util.*;

public class FortuneTelling {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int t = sc.nextInt();  // Number of test cases

        while (t-- > 0) {
            long n = sc.nextLong();  // length of array
            long x = sc.nextLong();  // Alice's starting nu
            long y = sc.nextLong();  // Final number

            long[] v = new long[(int)n];
            for (int i = 0; i < n; i++) {
                v[i] = sc.nextLong();
            }

            int f = 1; // parity toggle: 1 = even, 0 = odd
            for (int i = 0; i < n; i++) {
                if ((v[i] & 1) == 1) {
                    f = 1 - f;
                }
            }

            if ((x & 1) == 1) {
                if (((f ^ y) & 1) == 0) {
                    System.out.println("Alice");
                } else {
                    System.out.println("Bob");
                }
            } else {
                if (((f ^ y) & 1) == 0) {
                    System.out.println("Bob");
                } else {
                    System.out.println("Alice");
                }
            }
        }

        sc.close();
    }
}
```

1. What do addition and XOR affect?

- Both `+` and `^` affect the **parity** (whether the number is even or odd).
- The **parity of the result** after all operations depends on the **number of odd elements** in the array.
- **Why?**
  - Even + Even = Even
  - Even + Odd = Odd
  - Even ^ Even = Even
  - Even ^ Odd = Odd
- So, adding or XORing an **odd number** flips the parity.
- Therefore, **the total number of odd elements** in array `a` determines how many times the parity c flip.

2. Final parity formula

Let's say we start with number `d`. If the number of odd values in array `a` is odd, the parity will flip. So:

- If count of odd numbers is even: final parity = parity of starting number
- If count is odd: final parity = opposite parity of starting number

So, if we **track only parity**:

- Alice: starts with parity of `x`
- Bob: starts with parity of `x + 3`
  - Note: `+3` doesn't always make the parity opposite. E.g., `0 + 3 = 3 (odd)`; so we check pari of `x + 3`

✅ **Goal:**

We determine whether `y` is **reachable** by comparing:

- The final parity after applying all ops starting from Alice or Bob
- With the parity of `y`

```java
int f = 1; // f = 1 means even parity
for (int i = 0; i < n; i++) {
    if ((v[i] & 1) == 1) {  // if v[i] is odd
        f = 1 - f;  // toggle f (even <-> odd)
    }
}
```

💡 **What is this doing?**

- Counts how many odd numbers are in the array.
- If count is odd, `f` toggles an odd number of times → final parity flips
- If count is even, `f` toggles an even number of times → parity stays the same
- So, `f` = 1 means **final parity = same as starting parity**
- `f` = 0 means **final parity = flipped from starting parity**

✅ **Now comes the decision logic:**

```java
if ((x & 1) == 1) {
```

- Alice's starting number is **odd**.

```java
if (((f ^ y) & 1) == 0) {
    System.out.println("Alice");
```

- `f ^ y` toggles parity if `f == 0`
- `& 1` isolates the parity bit (even/odd)
- If parity of `y` matches expected final parity → Alice is valid

```java
} else {
    System.out.println("Bob");
```

Module 3:- Bit Manipulation Intermediate.

↳ Video 1:- Bitsets & Bit Masking.

in C++:-

(i). `__builtin_popcount()` OR `__builtin_popcountll()`

⟶ ll here represents long numbers.

Ex:- `__builtin_popcount(1010)=2`

(ii). `__builtin_clz()` OR `__builtin_clzll()`

Ex:- `__builtin_clz(1010)=28`

(iii). `__builtin_ctz()` OR `__builtin_ctzll()`

Ex:- `__builtin_clz(10 010)=1`

NOTE:- Time Complexity of all these functions is $\log_2 n$

This can also be done manually via for loop:-

```
int n;
int cnt=0
for(int i=0; i<=31; i++){
    if(n & (1<<i)){
        cnt++;
    }
}
cout<<cnt;
}
```

TC:- $O(\log_2 n)$

Because can represent a number in $\log n$ bits.

Built in functions in JAVA:-

(i)

`__builtin_popcount()` or `__builtin_popcountll()`

C++:

```cpp
__builtin_popcount(x);      // For int
__builtin_popcountll(x);    // For long long
```

Java Equivalent:

```java
Integer.bitCount(x);        // For int
Long.bitCount(x);           // For long
```

Example in Java:

```java
System.out.println(Integer.bitCount(100)); // Output: 3
```

(ii)

`__builtin_clz()` or `__builtin_clzll()`

(Count leading zeros)

C++:

```cpp
__builtin_clz(x);          // For int
__builtin_clzll(x);        // For long long
```

Java Equivalent:

```java
Integer.numberOfLeadingZeros(x);   // For int
Long.numberOfLeadingZeros(x);      // For long
```

Example in Java:

```java
System.out.println(Integer.numberOfLeadingZeros(101)); // Output: 25
```

(iii).

`__builtin_ctz()` or `__builtin_ctzll()`

(Count trailing zeros)

C++:

```cpp
__builtin_ctz(x);          // For int
__builtin_ctzll(x);        // For long long
```

Java Equivalent:

```java
Integer.numberOfTrailingZeros(x);   // For int
Long.numberOfTrailingZeros(x);      // For long
```

Example in Java:

```java
System.out.println(Integer.numberOfTrailingZeros(10)); // Output: 1
```

0 0 1 0 1 0 0 1 0 0 0 0

Trailing zeros :- 4

Leading zeros :- 2.

∵ Every integer can be represented in 32 bit.

int a = 4 ⟶ (100)

↳ 32.

32 bits.

0 0 0 0 0 0 0 0 . . . . 0 1 0 0

29 zeros first from the bit no.

⇝ Bitset

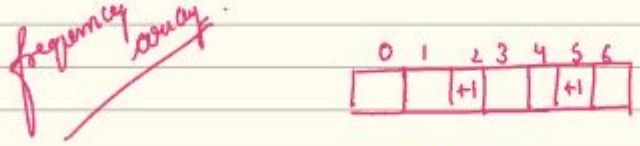
A bitset is an array of bools where each boolean value is not stored in a separate byte. Instead, bitset optimizes the space such that each boolean value takes 1-bit space only, so space taken by bitset is less than that of an array of bool or vector of bool.

Ex:-

I have an array → (0 → 4)

∴ $0 \le a[i] \le 40$

Tell me the no. of unique elements in my array.



instead we can make a boolean array.

bool arr[41] → uses 41 bytes

in C++, vector<bool> arr(41). → uses only [41/8] bytes.

1 bit → 0/1

8 bits → 1 byte.

Helps improve the trade off b/w TC & SC:-

Runtime speed.

⇝ Bitset:-

Space of bitset < space of vector<bool>

∵ In bitset, we do not have dynamic size. Size is fixed at compile time.

bitset<N> b.

N → fixed.

Overhead of bitset is less because of no metadata.

↝ Space and Runtime Comparision.

	bitset<N>	vector<bool> bit(N)	bool arr[N]
Storage space	[N/8] bytes	[N/8] bytes + metadata	N bytes + metadata

Runtime speed:- Bitset > bool array[] > vector<bool> array()

↝ Ways of declaring Bitset :-

✅ **Overview of `BitSet`:**

- A `BitSet` is a vector of bits that grows as needed.
- Bits are indexed by non-negative integers.
- It is more memory-efficient than using a `boolean[]` for large bit arrays.

✅ **Common Usage:**

java — Copy — Edit

```java
import java.util.BitSet;

public class Main {
    public static void main(String[] args) {
        BitSet bitset = new BitSet();

        bitset.set(2);  // Set bit at index 2
        bitset.set(4);  // Set bit at index 4

        System.out.println(bitset);          // prints: {2, 4}
        System.out.println(bitset.get(2));   // true
        System.out.println(bitset.get(3));   // false

        bitset.clear(2);                     // Clear bit at index 2
        System.out.println(bitset);          // prints: {4}
    }
}
```

✅ **Key Methods:**

Method	Description
`set(int index)`	Sets the bit at the specified index to `true`.
`clear(int index)`	Sets the bit at the specified index to `false`.
`get(int index)`	Returns the value of the bit at the index.
`flip(int index)`	Toggles the bit at the specified index.
`and(BitSet set)`	Logical AND with another `BitSet`.
`or(BitSet set)`	Logical OR with another `BitSet`.
`xor(BitSet set)`	Logical XOR with another `BitSet`.
`cardinality()`	Number of bits set to `true`.
`length()`	Index of the highest set bit + 1.
`toString()`	Prints indices of bits set to `true`.

✅ Example:

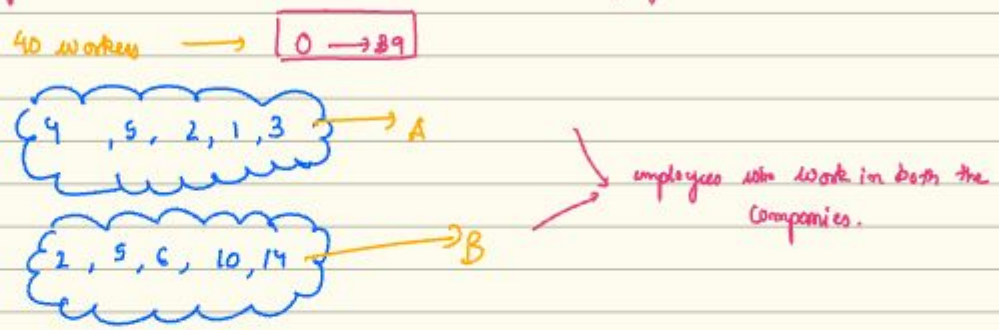
```java
BitSet a = new BitSet();
BitSet b = new BitSet();
a.set(1); a.set(2); a.set(3);
b.set(2); b.set(3); b.set(4);

a.and(b);
System.out.println(a);  // Output: {2, 3}
```

Ex:-

40 workers :- 0 ⟶ 39.

Company A

Company B.




```java
public class Main {
    public static void solve() {
        BitSet b1 = new BitSet(40);   // b1[i]=0 → ith worker doesn't work there the bit
        BitSet b2 = new BitSet(40);

        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int m = sc.nextInt();

        for (int i = 0; i < n; i++) {
            int x = sc.nextInt();
            b1.set(x);
        }

        for (int i = 0; i < m; i++) {
            int x = sc.nextInt();
            b2.set(x);
        }

        BitSet b3 = (BitSet) b1.clone(); // Clone b1
        b3.and(b2);                      // AND with b2

        System.out.println(b3.cardinality()); // Count of common set bits
    }

    public static void main(String[] args) {
        solve();
    }
}
```

of the ith bit is Zero or else it is 1.

→ Bitset function :-

(i). Count :-

1. Count (number of set bits)

Equivalent to `.count()` in other languages.

java

```java
BitSet bitset = new BitSet();
bitset.set(1);
bitset.set(3);
bitset.set(5);

int count = bitset.cardinality(); // Returns the number of set bits (1s)
System.out.println(count); // Output: 3
```

(ii). Flip (invert bits) :-

Flips bits in a given range.

java

```java
BitSet bitset = new BitSet(6);
bitset.set(1);
bitset.set(3);

bitset.flip(0, 6); // Flip bits from index 0 to 5 inclusive

System.out.println(bitset); // Shows flipped bits
```

(iii). Any (Check if any bit is set).

Equivalent to `.any()`.

java

```java
boolean any = !bitset.isEmpty(); // True if any bit is set
System.out.println(any);
```

(iv). None (Check if no bits are set).

Equivalent to `.none()`.

java

```java
boolean none = bitset.isEmpty(); // True if no bits are set
System.out.println(none);
```

(v). All (Check if all bits in a range are set).

Java doesn't have a direct `all()` function, but you can implement it like this:

java

```java
public static boolean allSet(BitSet bs, int from, int to) {
    return bs.get(from, to).cardinality() == (to - from);
}

// Usage
BitSet bitset = new BitSet();
bitset.set(0, 5); // set bits 0 to 4 (inclusive of 0, exclusive of 5)

boolean all = allSet(bitset, 0, 5);
System.out.println(all); // true
```

(vi). set () :- Set the bit at given index 1.

(vii) reset() :- Set the bit at given index to 0.

(viii). size() :- Returns the size bitset.

Time Complexity :- $O(N/32)$ (where N is size of bitset).

⇝ Bit Masking :- Bit mask a.k.a lightweight is an integer stored in a computer's memory as a sequence / string of bits. Thus, we can use integers to represent a lightweight small set of boolean values.

⇝ Problem 1 :- Given an array A of N elements, print all possible subsets of the array in any order.

Ex :-

arr = [1, 2, 3]

Answer :- [ ], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3].

∵ This done by recursion in 2^N (generating sub sets).

	i	i	i		
	0	0	0		
0	0	0	0	→	[ ]
1	0	0	1	→	[1]
2	0	1	0	→	[2]
3	0	1	1	→	[1, 2]
4	1	0	0	→	[3]
5	1	0	1	→	[1, 3]
6	1	1	0	→	[2, 3]
7	1	1	1	→	[1, 2, 3]

} 8 subsets.

2^{n-1}.

Array of size n :-

every number from 0 — $2^n - 1$ will represent a unique subset.

Code :-

```java
import java.util.Scanner;

public class printSubsets {
    Run | Debug
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt();                   // Read size of array
        int[] arr = new int[n];                 // Create array of size n

        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();              // Read array elements
        }

        for (int mask = 0; mask < (1 << n); mask++) { // Corrected loop variable (was `i` instea
            for (int i = 0; i < n; i++) {
                if ((mask & (1 << i)) != 0) {   // Corrected condition (Java doesn't allow `if(m
                    System.out.print(arr[i] + " "); // Use print instead of println to print on
                }
            }
            System.out.println();               // New line after each subset
        }

        sc.close(); // Always good practice to close Scanner
    }
}
```

Problem :- LC :- Maximum Good people based on Statements.

2151. Maximum Good People Based on Statements

Discuss Approach ›

Hard | Topics | Companies | Hint

There are two types of persons:

- The **good person**: The person who always tells the truth.
- The **bad person**: The person who might tell the truth and might lie.

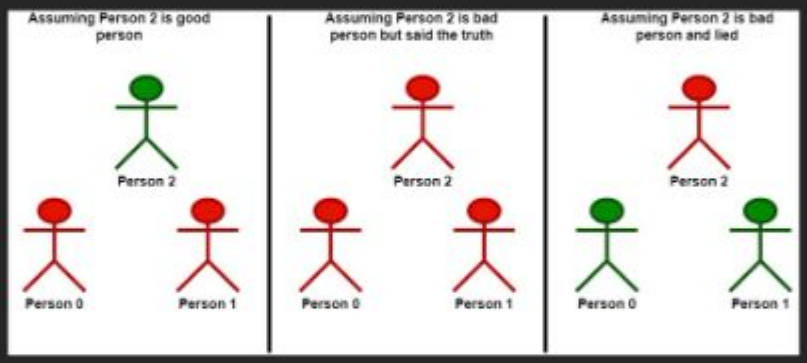
You are given a **0-indexed** 2D integer array `statements` of size `n x n` that represents the statements made by `n` people about each other. More specifically, `statements[i][j]` could be one of the following:

- `0` which represents a statement made by person `i` that person `j` is a **bad** person.
- `1` which represents a statement made by person `i` that person `j` is a **good** person.
- `2` represents that **no statement** is made by person `i` about person `j`.

Additionally, no person ever makes a statement about themselves. Formally, we have that `statements[i][i] = 2` for all `0 <= i < n`.

Return *the **maximum** number of people who can be **good** based on the statements made by the* `n` *people*.

Example 1:



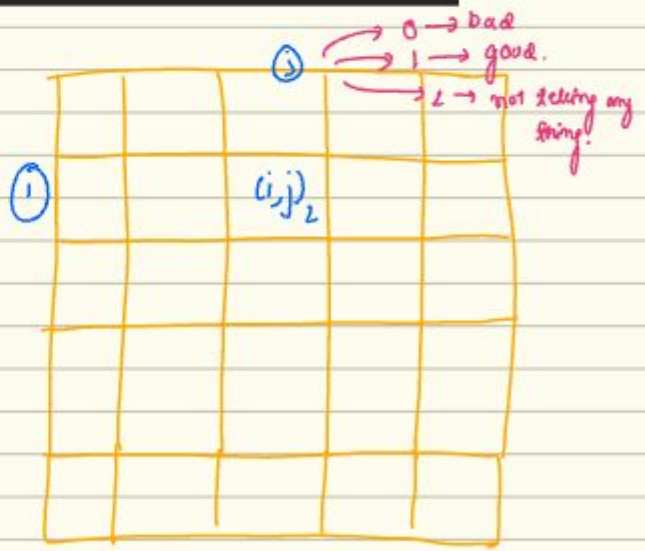

```
Input: statements = [[2,1,2],[1,2,2],[2,0,2]]
Output: 2
```

n people

good → truth

bad → may lie.

This says that what i says about j.



Video 8:- Problem Solving 2.

Problem 1:- COOKCHEF :- Maximum AND.

Chef's good friend Shubham has an assignment to complete, but he is too lazy to do it, so he asked Chef to help him. On the other hand, Chef was busy in the kitchen, so he in turn asked you to help Shubham.

You are given a sequence of positive integers $A_1, A_2, \ldots, A_N$ and an integer K. For any positive integer X such that exactly K bits in the binary representation of X are equal to 1, consider the sum $S = \sum_{i=1}^{N} (X \wedge A_i)$; here, $\wedge$ denotes bitwise AND. You should choose X in such a way that S is maximum possible. If there is more than one such value of X, you should find the smallest one.

Input

- The first line of the input contains a single integer T denoting the number of test cases. The description of T test cases follows.
- The first line of each test case contains two space-separated integers N and K.
- The second line contains N space-separated integers $A_1, A_2, \ldots, A_N$.

Output

For each test case, print a single line containing one integer — the smallest possible value of X.

Constraints

- $1 \le T \le 1,000$
- $1 \le N \le 10^5$
- $1 \le K \le 30$
- $1 \le A_i \le 10^9$ for each valid i
- the sum of N over all test cases does not exceed 10^5

Subtasks

Subtask #1 (50 points): $K \le 2$

Subtask #2 (50 points): original constraints

Sample 1:

Input	Output
1 4 1 3 4 5 1	4

Explanation:

Example case 1: Exactly one bit in X should be 1. Our options are:

- If $X = 1, S = 1 + 0 + 1 + 1 = 3$.
- If $X = 2, S = 2 + 0 + 0 + 0 = 2$.
- If $X = 4, S = 0 + 4 + 4 + 0 = 8$.
- For any greater valid X, $S = 0$.

The maximum value of S is 8, so the answer is the only value of X where we get $S = 8$.

N

A_1 A_2 . . . . . A_N

X ⟶ Exactly k set bits.

Sum = $(A_1 \& X) + (A_2 \& X) \ldots\ldots\ldots + (A_N \& X)$.

Aim:- maximize value of S.

If multiple X — choose the smallest value of X.

⇝ MODULE - 4

↳ Video 1:- Ad-hoc and patterns

You are given an array, A, of integers of length n. Next you are given q queries. For each q_j, your task is to add $2^{2i}-1$ to all numbers A_j in the array such that A_j is divisible by 2^{2i} and $1 \le j \le n$. Print the array, A, after you have processed all queries.

Sample input:-

n = 5 q = 3

A_i: 1 2 3 4 4

+3 +3

q_i: 2

3

4.

Solution:-

(i). 1st query:- $2^2 = 4$

$2^2 - 1 = 4 - 1 = 3$

$\Rightarrow$ 1 2 3 7 7

(ii). 2nd query:- $2^3 = 8$

$2^3 - 1 = 8 - 1 = 7$

Nothing will change here because nothing is divisible by $2^3 = 16$.

(iii) 3rd query: Similarly not divisible by 2^2. Hence the whole array remains as:-

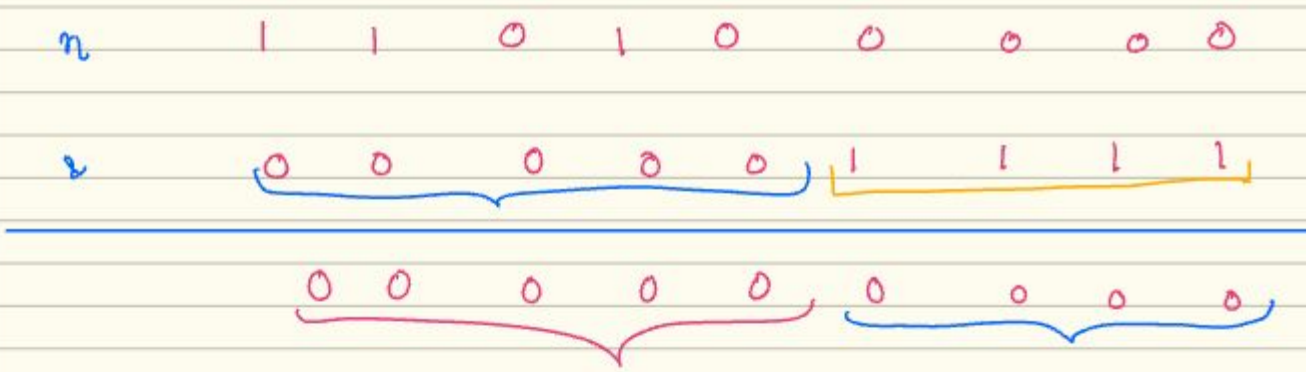
1 2 3 7 7

2^x = 1 1 0 1 0 0 0 0 0

$2^4 = 16$

if you want this number to be divisible by 16 then it must have atleast 4 zeros in the end.

∴ So, from LSB you must have atleast 4 zeros or continuous bits.



∴ (if a[i] & $(2^x - 1) == 0$)

if (a[i] is divisible by 2^x

∵ Let's say I maintain a max variable → max_x if I have a query ≥ max_x, I will ignore that else, I proceed & update max_x??

29 30 31 4 6 2

Pending

~> Problem 2 :- Stick lengths :-

Stick Lengths

TASK

Time limit: 1.00 s **Memory limit:** 512 MB

There are n sticks with some lengths. Your task is to modify the sticks so that each stick has the same length.

You can either lengthen and shorten each stick. Both operations cost x where x is the difference between the new and original length.

What is the minimum total cost?

Input

The first input line contains an integer n: the number of sticks.

Then there are n integers: $p_1, p_2, \ldots, p_n$: the lengths of the sticks.

Output

Print one integer: the minimum total cost.

Constraints

- $1 \le n \le 2 \cdot 10^5$
- $1 \le p_i \le 10^9$

Example

Input:
5
2 3 1 5 2

Output:
5

n length

P = { 2 3 1 5 2 }

So if new array looks like = 2 2 2 2 2

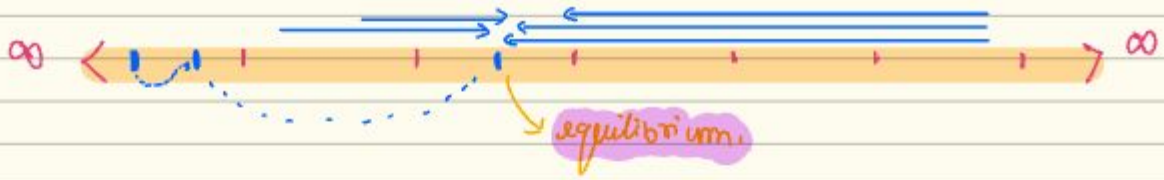
↳ then difference =

Cost: 0 1 1 3 0 = 5

} We need to minimize this cost as much as possible and 5 is the lowest value

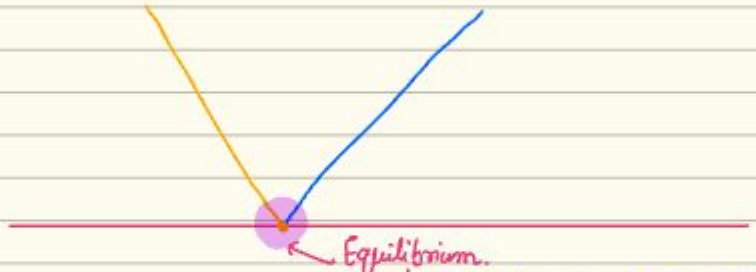
TC :-
O(n)
O(n log₂n)
∴ Sort BS

yellow = represent numbers, blue → represents the value you are wanting to make.



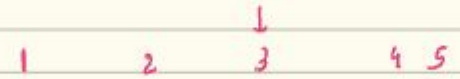
we will start moving the brick to the right
↳ keep noticing the change in costs.

Imp:- While we keep moving the stick it first decreases the cost but after moving it upto a point it again starts to increase. This point is called the **Equilibrium!**

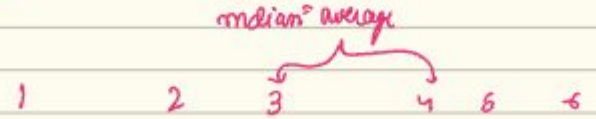


∴ So, the best place to point this equilibrium is the middle of the array after sorting the array.

n = odd



n = even



Code :-

Problem 3:- B. Odd Grasshopper.

B. Odd Grasshopper

time limit per test: 1 second
memory limit per test: 256 megabytes

The grasshopper is located on the numeric axis at the point with coordinate x_0.

Having nothing else to do he starts jumping between integer points on the axis. Making a jump from a point with coordinate x with a distance d to the left moves the grasshopper to a point with a coordinate $x - d$, while jumping to the right moves him to a point with a coordinate $x + d$.

The grasshopper is very fond of positive integers, so for each integer i starting with 1 the following holds: exactly i minutes after the start he makes a jump with a distance of exactly i. So, in the first minutes he jumps by 1, then by 2, and so on.

The direction of a jump is determined as follows: if the point where the grasshopper was before the jump has an **even** coordinate, the grasshopper jumps to the **left**, **otherwise** he jumps to the **right**.

For example, if after 18 consecutive jumps he arrives at the point with a coordinate 7, he will jump by a distance of 19 to the right, since 7 is an odd number, and will end up at a point $7 + 19 = 26$. Since 26 is an even number, the next jump the grasshopper will make to the left by a distance of 20, and it will move him to the point $26 - 20 = 6$.

Find exactly which point the grasshopper will be at after exactly n jumps.

Input
The first line of input contains an integer t ($1 \le t \le 10^4$) — the number of test cases.

Each of the following t lines contains two integers x_0 ($-10^{14} \le x_0 \le 10^{14}$) and n ($0 \le n \le 10^{14}$) — the coordinate of the grasshopper's initial position and the number of jumps.

Output
Print exactly t lines. On the i-th line print one integer — the answer to the i-th test case — the coordinate of the point the grasshopper will be at after making n jumps from the point x_0.

Output
Print exactly t lines. On the i-th line print one integer — the answer to the i-th test case — the coordinate of the point the grasshopper will be at after making n jumps from the point x_0.

Example

input

```
9
0 1
0 2
10 10
10 99
177 13
10000000000 987654321
-433494437 87178291199
1 0
-1 1
```

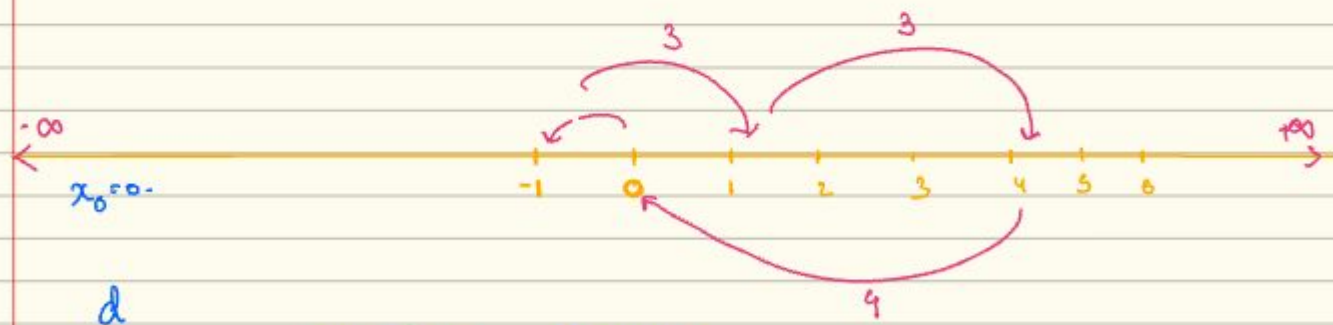
output

```
-1
1
11
118
190
9812345679
-87611785637
1
0
```

Note
The first two test cases in the example correspond to the first two jumps from the point $x_0 = 0$.

Since 0 is an even number, the first jump of length 1 is made to the left, and the grasshopper ends up at the point $0 - 1 = -1$.

Then, since -1 is an odd number, a jump of length 2 is made to the right, bringing the grasshopper to the point with coordinate $-1 + 2 = 1$.



$x_0 = 0$.

d

dir

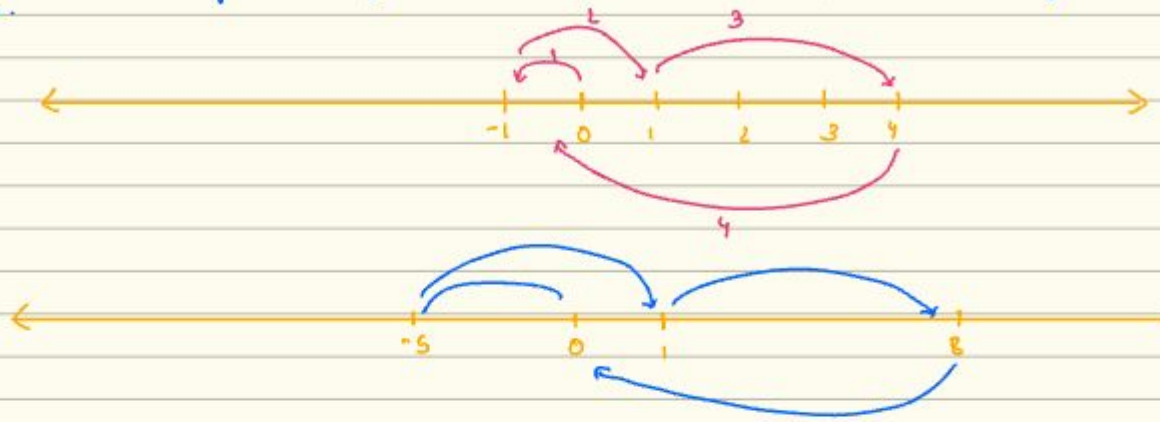
even = left , odd = right.

∴ We need to find the final coordinate value.

$x_0 = 0$

↳ end coordinate

Lets observe, mostly adhoc problems are non standard problems and require observation.



1 2 3 4 | 5 6 7 8 | 9 10 11 12

k=0 k=1 k=2

∴ So after observing after every 4 jump, we are coming back to zero.

$0 \longrightarrow (-4k+1) \longrightarrow 1 \longrightarrow (4k+4)$

$\longrightarrow 0$

$k = 0, 1, 2 \ldots\ldots\ldots$

∴ So,

```
if (n % 4 == 1){
    -1
}
if (n % 4 == 2){
    1
}
```

Ans = final_pos

Wrt $x_0 = 0$

if I find x_0 to be odd, then

Ans = final_pos - x_0

else

Ans = final_pos + x_0.

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int t = sc.nextInt();
        while (t-- > 0) {
            long start = sc.nextLong();
            long jumps = sc.nextLong();

            long finalPos;
            if (jumps % 4 == 1)
                finalPos = -jumps;
            else if (jumps % 4 == 2)
                finalPos = 1;
            else if (jumps % 4 == 3)
                finalPos = jumps + 1;
            else // jumps % 4 == 0
                finalPos = 0;

            if (start % 2 == 0)
                finalPos = start + finalPos;
            else
                finalPos = start - finalPos;

            System.out.println(finalPos);
        }

        sc.close();
    }
}
```

~ Video 2 :- Problem Solving 1.

C - Modulo Summation

Time Limit: 2 sec / Memory Limit: 1024 MB

Score : 300 points

Problem Statement

You are given N positive integers $a_1, a_2, \ldots, a_N$.

For a non-negative integer m, let $f(m) = (m \bmod a_1) + (m \bmod a_2) + \ldots + (m \bmod a_N)$.

Here, $X \bmod Y$ denotes the remainder of the division of X by Y.

Find the maximum value of f.

Constraints

- All values in input are integers.
- $2 \leq N \leq 3000$
- $2 \leq a_i \leq 10^5$

Input

Input is given from Standard Input in the following format:

N
a_1 a_2 $\ldots$ a_N

Output

Print the maximum value of f.

Sample Input 1 Copy

```
3
3 4 6
```

Sample Output 1 Copy

```
10
```

$f(11) = (11 \bmod 3) + (11 \bmod 4) + (11 \bmod 6) = 10$ is the maximum value of f.

Sample Input 2 Copy

```
5
7 46 11 20 11
```

Sample Output 2 Copy

```
90
```

$N = a_1 \quad a_2 \quad a_3 \quad a_4 \ldots\ldots a_n$

$f(m) = (m \,\%\, a_1) + \ldots\ldots\ldots (m \,\%\, a_n)$

maximize each & every term.

∴ So to maximize the overall value :- we need some 'm' value for which every term is the largest.

$(m \,\%\, a_i)$.

↓ a

$5 \,\%\, 3 = 2$
$6 \,\%\, 3 = 0$
$7 \,\%\, 3 = 1$
$8 \,\%\, 3 = 2$
$9 \,\%\, 3 = 0$

∴ The pattern observed is :-

the values repeat and fall in the range b/w (0 to a-1).

The desired sum will be :- $(a_1 - 1) + (a_2 - 1) + (a_3 - 1) + \ldots\ldots (a_n - 1)$.

overall

if $(a_i \,\%\, m == 0)$, the sum will be 0.

But if we add -1 to it, we add an extra non divisibility to increase our sum.

$m = (a_1 \times a_2 \times a_3 \times a_4 \times \ldots\ldots\ldots a_n) - 1$

$m \,\%\, a_i = 0$.

(∵ Extra $a_i - 1$ will be remaining).

∴ $m \,\%\, a_i = (a_i - 1)$

∴ Sum of array - n

Code :-

B. Rectangle Filling :-

PROBLEMS SUBMIT STATUS STANDINGS CUSTOM TEST

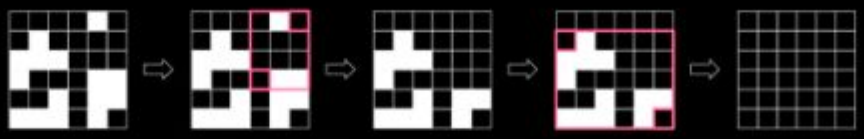
B. Rectangle Filling

time limit per test: 1 second
memory limit per test: 256 megabytes

There is an $n \times m$ grid of white and black squares. In one operation, you can select any two squares of the same color, and color all squares in the subrectangle between them that color.

Formally, if you select positions (x_1, y_1) and (x_2, y_2), both of which are currently the same color c, set the color of all (x, y) where $\min(x_1, x_2) \le x \le \max(x_1, x_2)$ and $\min(y_1, y_2) \le y \le \max(y_1, y_2)$ to c.

This diagram shows a sequence of two possible operations on a grid:



Is it possible for all squares in the grid to be the same color, after performing any number of operations (possibly zero)?

Input

The first line of the input contains a single integer t ($1 \le t \le 10^4$) — the number of test cases. The description of the test cases follows.

The first line of each test case contains two integers n and m ($1 \le n, m \le 500$) — the number of rows and columns in the grid, respectively.

Each of the next n lines contains m characters '`W`' and '`B`' — the initial colors of the squares of the grid.

It is guaranteed that the sum of $n \cdot m$ over all test cases does not exceed $3 \cdot 10^5$.

Output

For each test case, print "`YES`" if it is possible to make all squares in the grid the same color, and "`NO`" otherwise.

You can output the answer in any case (upper or lower). For example, the strings "`yEs`", "`yes`", "`Yes`", and "`YES`" will be recognized as positive responses.

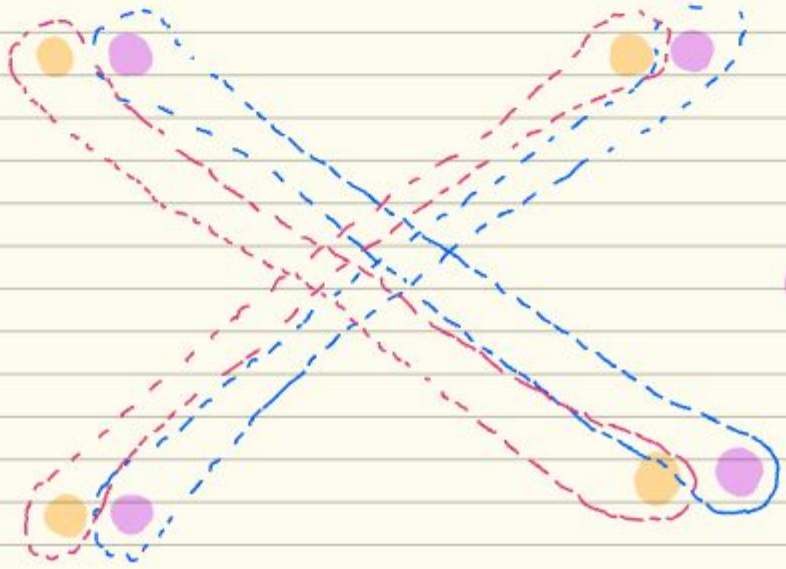
Example

input

```
8
2 1
W
B
6 6
WWWBBW
WBWWWW
BBBWWW
BWWBBB
WWBWBB
BBBWBW
1 1
W
2 2
BB
BB
3 4
BWBW
WBWB
BWBW
4 2
BB
BB
WW
WW
4 4
WWBW
BBWB
WWBB
BBBB
1 5
WBBWB
```

output

```
NO
YES
YES
YES
YES
NO
YES
NO
```



1 operation.

Extreme corners with same colour.

we can represent it as:-

CASE 1:-

a[0][0] == a[n-1][m-1] || a[0][m-1] == a[n-1][0]

CASE 2:- What if the diagonal don't match:-

(OR)

(OR)

(OR)

∴ If the first row has all same & last row has all same & they don't match.

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int t = sc.nextInt();
        while (t-- > 0) {
            int n = sc.nextInt();
            int m = sc.nextInt();
            sc.nextLine(); // consume newline

            String[] grid = new String[n];
            for (int i = 0; i < n; i++) {
                grid[i] = sc.nextLine();
            }

            String ans = "YES";

            if (grid[0].charAt(0) != grid[n - 1].charAt(m - 1)) {
                boolean impossible = true;

                // Check top and bottom row
                for (int j = 0; j < m - 1; j++) {
                    if (grid[0].charAt(j) != grid[0].charAt(j + 1) ||
                        grid[n - 1].charAt(j) != grid[n - 1].charAt(j + 1)) {
                        impossible = false;
                    }
                }
                if (impossible) {
                    ans = "NO";
                }

                impossible = true;
                // Check left and right column
                for (int i = 0; i < n - 1; i++) {
                    if (grid[i].charAt(0) != grid[i + 1].charAt(0) ||
                        grid[i].charAt(m - 1) != grid[i + 1].charAt(m - 1)) {
                        impossible = false;
                    }
                }
                if (impossible) {
                    ans = "NO";
                }
            }

            System.out.println(ans);
        }

        sc.close();
    }
}
```

~ Missing Coin Sum :-

CSES

Login

Algoritmit ongelmanratkaisussa

Missing Coin Sum

TASK

Time limit: 1.00 s **Memory limit:** 512 MB

You have n coins with positive integer values. What is the smallest sum you cannot create using a subset of the coins?

Input

The first line has an integer n: the number of coins.

The second line has n integers $x_1, x_2, \ldots, x_n$: the value of each coin.

Output

Print one integer: the smallest coin sum.

Constraints

- $1 \le n \le 2 \cdot 10^5$
- $1 \le x_i \le 10^9$

Example

Input:
5
2 9 1 2 7

Output:
6

Part 1

...
Collecting Numbers
Creating Strings
Movie Festival
Missing Coin Sum
Josephus Problem I
Concert Tickets
Room Allocation
Distinct Values Subarrays II
...

N

c_1 c_2 . . . . . c_n

[1 2 3 4 5 6 . . . . . X] (X+1)

base case :-

1 should be there else, answer is 1

~ Number Theory Beggines.

↳ Video 1.

GOAL:-

(i). Prime factorization using Trial Division.

(ii). Sieve's Algorithm.

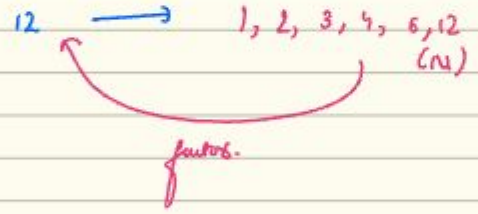
(iii). Smallest Prime factor (SPF) of a number

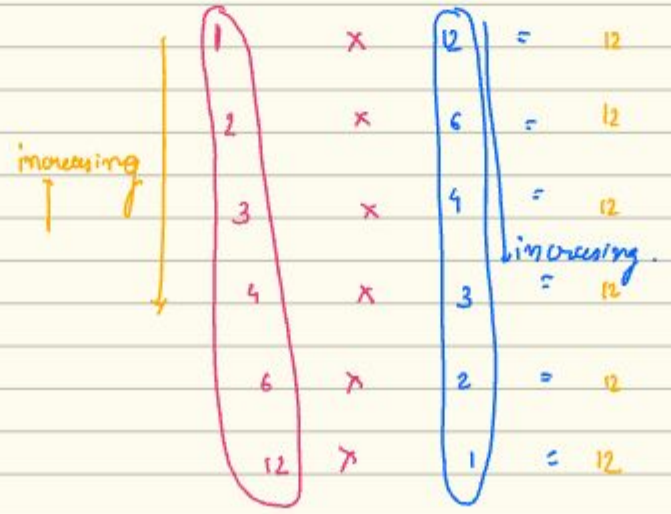
(iv). Fast prime factorization using SPF.

(i). **FACTORIZATION:-** is when you break a number down into smaller numbers that, multiplied together, given you that original number.

Example:-

Factorization of 12: $3*4$, $2*6$, $2*2*3$





∵ So for finding prime factors, we don't need to traverse the whole till N.

There's a fixed point where the factors first increase then decrease.

its $\sqrt{N}$.

∵ $1 \longrightarrow \sqrt{N}$

code:-

```
public class PrimeFactors {

    public static void main(String[] args) {
        Scanner sc= new Scanner(System.in);
        ArrayList<Integer> factors = new ArrayList<>();
        int n = sc.nextInt();
        for(int i =1 ; i*i<=n ; i++){
            if(n%i==0){
                factors.add(i);
            }
            if(i !=n/i){
                factors.add(n/i);
            }
        }
        for(int i =0 ; i< factors.size(); i++){
            System.out.println(factors.get(i));
        }
    }
}
```

(ii). Prime factorization using Trial division.

- Trial Division:- is the most basic method of Prime Factorization.
- The above method works with the observation that the maximum factor for any number N other than the number is itself is always less than or equal to the sqrt(N).
- TC :- Sqrt(N).

PROOF :-

smallest prime no. => 2.

```
for (i=2 ; i*i <=N; i++){
    if(N%i==0){
        while(N%i==0){ N/=i;
        }
    }
}
```

i → definately a prime number.

How is it calculated.

36 → (2) → i = 3

4 × 9 ≤ 36

36 / 2 (Completely gets divided)

↓ 18 ↓ 9 ↓ 3

102 →

2 | 102 (2)

3 | 51 (3)

17 → So the number we get is itself a prime number.

5 × 5 ≤ 17.

(iii). Sieve of Eratosthenes :-

- The sieve of Eratosthenes is one of the most efficient ways to find all primes smaller than or equal to the number N.
- Through this method we can precompute all prime numbers less than or equal to 10^7.
- Time Complexity :- $N^* \log(\log N)$

Example :- N = 16

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

(i) first mark all the factors / divisible numbers / factors of N → False.

N = 16

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F	F	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F

∴ So all the True marked numbers are prime factors.

↳ 2, 3, 5, 7, 11, 13.

∴ BASIC METHOD :- for every prime, we are marking its multiple as non prime

Optimization 1 :->

∴ If a number is non-prime number and do to mark it as non prime number we need to iterate till sq. root of that number.

So, we will be able to mark that number as non-prime no.

↳ ∵ After this the indices having value TRUE we get the prime numbers.

Code:-

```
class SieveOfEratosthenes {
    public List<Integer> sieve(int n) {
        boolean[] isPrime = new boolean[n + 1];
        Arrays.fill(isPrime, true);
        isPrime[0] = false;
        isPrime[1] = false;

        for (int i = 2; i * i <= n; i++) {
            if (isPrime[i]) {
                for (int j = i * i; j <= n; j += i) {
                    isPrime[j] = false;
                }
            }
        }

        List<Integer> primes = new ArrayList<>();
        for (int i = 2; i <= n; i++) {
            if (isPrime[i]) {
                primes.add(i);
            }
        }
        return primes;
    }
}
```

Time Complexity:- $i \to N/i$

$\text{sqrt}(N)$

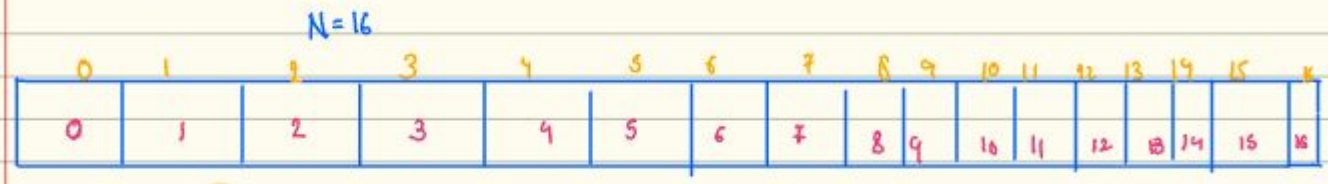
$$\frac{N}{1} + \frac{N}{2} + \frac{N}{3} \cdots\cdots + \frac{N}{\text{sqrt}(N)}$$

$$N\left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \cdots \frac{1}{N}\right) \approx \log N.$$

∴ TC of Sieve's Algorithm = $O(N \log N)$.

(iv). Smallest Prime factor (SPF) of a number.

- SPF of a number N is the smallest prime number that divides that number N.
- SPF of a prime number N is the number itself.
- By using Sieve of Eratosthenes we can precompute SPF of numbers upto N.
- Time Complexity:- $N * \log(\log N)$.



∴ spf[i] == i → i is a prime number.

spf[i] → Smallest prime factor of number i.

Code:-

Core Logic

1. **What is SPF?**
   `spf[x]` stores the *smallest prime* that divides `x`.
   - If `x` is prime, then `spf[x] = x`.
   - If `x` is composite, `spf[x]` is its smallest prime divisor.
2. **How SPF is computed (preprocessing):**
   - Initially, `spf[i] = i` for all `i`.
   - Then we run a modified sieve (like Sieve of Eratosthenes):
     - For every prime `i` (where `spf[i] == i`), we set `spf[j] = i` for all multiples `j` of `i` starting from i^2.
     - This ensures that every number gets marked with its smallest prime divisor.

   Complexity:
   - **Time:** $O(N \log \log N)$ — same as the sieve.
   - **Space:** $O(N)$.
3. **Prime factorization using SPF:**
   - Given `n`, repeatedly divide `n` by `spf[n]` and store the factors.
   - This is efficient because each division removes at least one prime factor and uses the precomputed smallest prime.

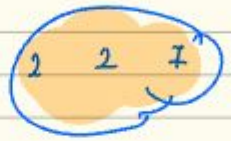
```java
import java.util.*;

public class SPFPrimeFactorization {
    static final int N = (int) 1e7;
    static int[] spf = new int[N];

    // Precompute SPF for every number up to N
    static void computeSPF() {
        for (int i = 0; i < N; i++) {
            spf[i] = i;
        }
        for (int i = 2; i * i < N; i++) {
            if (spf[i] == i) { // i is prime
                for (int j = i * i; j < N; j += i) {
                    if (spf[j] == j) { // unmarked yet
                        spf[j] = i;
                    }
                }
            }
        }
    }

    // Factorize n using precomputed SPF
    static List<Integer> getPrimeFactors(int n) {
        List<Integer> primeFactors = new ArrayList<>();
        while (n > 1) {
            primeFactors.add(spf[n]);
            n /= spf[n];
        }
        return primeFactors;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        computeSPF(); // O(N log log N)

        int t = sc.nextInt();
        while (t-- > 0) {
            int n = sc.nextInt();
            List<Integer> factors = getPrimeFactors(n);
            System.out.println(factors);
        }
    }
}
```

(iv). Using SPF for Prime factorization :-

- SPF of number can be used to calculate prime factorization in more efficient way as compared to trial division method.

- Time Complexity :-
  - Precomputation time :- $N * \log(\log(N))$
  - Query time :- $O(\log N)$

while (n != 1)




```
public int solve(){
    int n;
    input(n)
        int [ ] primes = new int (n);
        while (n > 1){
            primes.add (spf[n]);
            n /= spf[n];
        }
    }
```

↳ Video 2
↳ Problem Solving.

↳ A. Amost prime :-

A. Almost Prime

time limit per test: 2 seconds
memory limit per test: 256 megabytes

A number is called almost prime if it has exactly two distinct prime divisors. For example, numbers 6, 18, 24 are almost prime, while 4, 8, 9, 42 are not. Find the amount of almost prime numbers which are between 1 and n, inclusive.

Input
Input contains one integer number n ($1 \le n \le 3000$).

Output
Output the amount of almost prime numbers between 1 and n, inclusive.

Examples

input	Copy
10	

output	Copy
2	

input	Copy
21	

output	Copy
8	

```java
import java.util.*;

public class AlmostPrime {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        boolean[] isPrime = new boolean[n + 1];
        Arrays.fill(isPrime, true);
        isPrime[0] = false;
        isPrime[1] = false;

        // Sieve of Eratosthenes
        for (int i = 2; i * i <= n; i++) {
            if (isPrime[i]) {
                for (int j = i * i; j <= n; j += i) {
                    isPrime[j] = false;
                }
            }
        }

        // Count numbers with exactly 2 distinct prime factors
        int count = 0;
        for (int num = 2; num <= n; num++) {
            int primeFactors = 0;
            int temp = num;
            for (int p = 2; p <= n && p * p <= temp; p++) {
                if (isPrime[p] && temp % p == 0) {
                    primeFactors++;
                    while (temp % p == 0) {
                        temp /= p;
                    }
                }
            }
            if (temp > 1) { // Remaining prime factor
                primeFactors++;
            }
            if (primeFactors == 2) {
                count++;
            }
        }

        System.out.println(count);
    }
}
```

$n \le 3000$

$1 \rightarrow n$

↳ Check how many primes the numbers has.

TRIAL DIVISION
↳ $\sqrt{n}$

SPF
↳ $\log(n)$
↳ $n \log n$

Problem 2

Square difference.

B. Square Difference

time limit per test: 2 seconds
memory limit per test: 256 megabytes

Alice has a lovely piece of cloth. It has the shape of a **square** with a side of length a centimeters. Bob also wants such piece of cloth. He would prefer a **square** with a side of length b centimeters (where $b < a$). Alice wanted to make Bob happy, so she cut the needed square out of the corner of her piece and gave it to Bob. Now she is left with an ugly L shaped cloth (see pictures below).

Alice would like to know whether the area of her cloth expressed in square centimeters is prime. Could you help her to determine it?

Input
The first line contains a number t ($1 \le t \le 5$) — the number of test cases.

Each of the next t lines describes the i-th test case. It contains two integers a and b ($1 \le b < a \le 10^{11}$) — the side length of Alice's square and the side length of the square that Bob wants.

Output
Print t lines, where the i-th line is the answer to the i-th test case. Print "**YES**" (without quotes) if the area of the remaining piece of cloth is prime, otherwise print "**NO**".

You can print each letter in an arbitrary case (upper or lower).

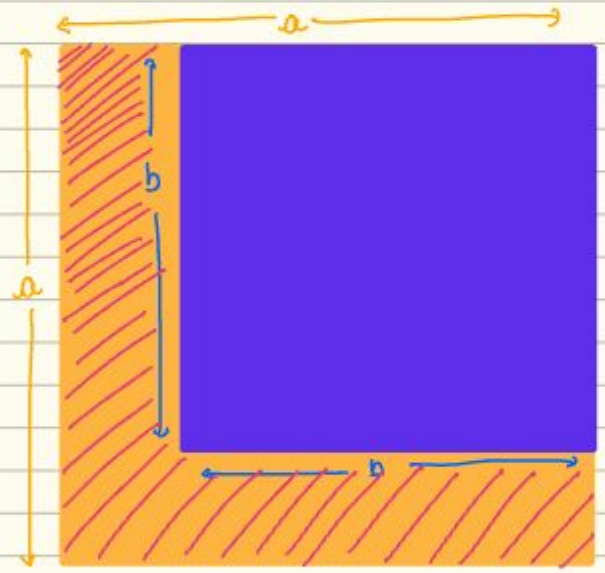
Example

input

```
4
6 5
16 13
61698858361 24777622638
34 33
```

output

```
YES
NO
NO
YES
```



Area = $a \cdot b$

$a^2 - b^2$

we need to check whether $a^2 - b^2$ is prime or not.

$1 \le b < a \le 10^{11}$

$a^2 - b^2 = (a-b)(a+b)$

prime prime

$a - b = 1$ $(a > b)$

& $a + b \longrightarrow$ prime.

$a + b \longrightarrow 2 \times 10^{11} \longrightarrow$ TRIAL DIVISION.

code :-

```
import java.util.*;

public class SquareDifference {

    static boolean isPrime(long n) {
        if (n <= 1) return false;
        for (long i = 2; i * i <= n; i++) {
            if (n % i == 0) return false;
        }
        return true;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int t = sc.nextInt();

        while (t-- > 0) {
            long a = sc.nextLong();
            long b = sc.nextLong();

            if (Math.abs(a - b) == 1 && isPrime(a + b)) {
                System.out.println("YES");
            } else {
                System.out.println("NO");
            }
        }
    }
}
```

→ Problem :- Digit Space → Codechef.

Digit Space

For an integer x, define its *digit space* D_x to be the set of all positive integers whose decimal representation is a rearrangement of the decimal representation of x, when written without leading zeros.
For example, if $x = 2024$, then $D_x = \{4022, 4202, 4220, 2042, 2024, 2402, 2420, 2204, 2240\}$.

For two integers x and y, define $f(x, y)$ to be the maximum **prime** number that divides both x and y.
If no prime divides both x and y, we say $f(x, y) = 1$ instead.
For instance, $f(35, 105) = 7$ and $f(2, 3) = 1$.

You are given two integers X and Y.
Compute the maximum value of $f(a, b)$ across all (a, b) such that $a \in D_X$ and $b \in D_Y$.
That is, compute the maximum possible common prime factor of two numbers, such that one is a rearrangement of X and the other is a rearrangement of Y.

Input Format

- The first line of input will contain a single integer T, denoting the number of test cases.
- Each test case consists of single line of input, containing two space-separated integers X and Y.

Output Format

For each test case, output on a new line the maximum possible value of $f(a, b)$ where $a \in D_X$ and $b \in D_Y$.

Constraints

- $1 \leq T \leq 200$
- $1 \leq X, Y < 10^7$

Sample 1:

Input	Output
4 51 5 433 24 55 100 2 3	5 7 5 1

Explanation:

Test case 1: We have $D_{51} = \{15, 51\}$ and $D_5 = \{5\}$.
It's best to choose $a = 15$ and $b = 5$, which gives $f(15, 5) = 5$.

Test case 2: We have $D_{433} = \{433, 343, 334\}$ and $D_{24} = \{24, 42\}$.
It's optimal to choose $a = 343$ and $b = 42$, giving $f(a, b) = 7$.

Test case 3: We have $D_{55} = \{55\}$ and $D_{100} = \{100\}$.
Our only choice is $f(55, 100) = 5$.

Digit space:-

X = 2024

D_X = {4202, 4220, ... }

$f(\vec{a}, \vec{b}) \longrightarrow$ largest prime that divides both a and b.

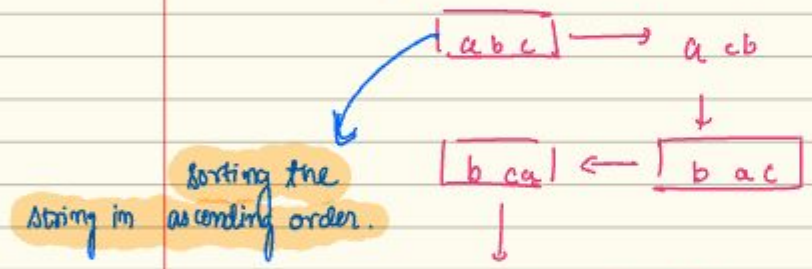
D_X = { } D_Y = { }

$f(a,b)$ → compute maximum value of a, b such that the digit a comes from a and b comes from digit b space.

$a \in D_X$ and $b \in D_Y$

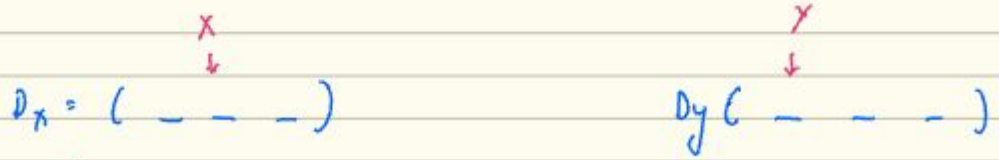
Built-in C++ function:- next_permutation.

↓ String.

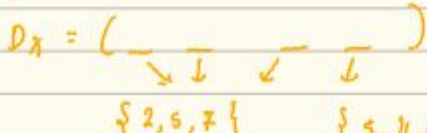


∴ We must start with lexographically smallest value then apply next_permutation fn. to get all permutations

So now,



∴ Ans = largest prime that divides some number in D_X and some number in D_Y.



AIM :- for a digit spell, we will try to get all prime factors.

$x \rightarrow x < 10^7$ $\boxed{10^7 - 1}$

string.

7

↓

all the permutations.

↓

just ignoring the numbers having leading zeros.

7

↓

7 × 7!

↓

prime factorization.

MODULE 8 → Number Theory Intermediate

↳ Exponentiation, GCD, LCM, Pigeonhole.

GOAL :-

- ↳ Modular Operations
- ↳ Binary Exponentiation.
- ↳ GCD and LCM
- ↳ Pigeon hole principle.

(i) MODULAR OPERATIONS - Addition.

$(A+B) \% MOD = (A \% MOD + B \% MOD) \% MOD.$

int A ⟶ INT_MAX

int B ⟶ INT_MAX

int C = A+B , this is not allowed due to overflow.

↳ this may also overflow with long long data type.

long range.

$2^{64}-1$ → LLONG_MAX

Modular operation → A big prime number

Ex:- give me the results modulo (5)

↳ MOD.

7, 6

$(7+6) \% 5 = 3$

$((7 \% 5) + (60 \% 5)) \% 5 = 3$

$(2+1) \% 5 = 3$

$(LLONG\_MAX + LLONG\_MAX) \% MOD.$

Modular operation - subtraction.

↳ Mod 5

$(A - B) \% MOD = (A \% MOD - B \% MOD + MOD) \% MOD.$

11, 8

↓

$(11 - 8) \% 5 \rightarrow 3$

$[11 \% 5 - 8 \% 5) \rightarrow -2$

$(-2 + MOD) \% MOD$

↓

$(-2 + 5) \% MOD \rightarrow 3$

Modular operation - Multiplication :-

$(A * B) \% MOD = ((A \% MOD) * (B \% MOD)) \% MOD.$

$(7 \times 8) \% 5 = (56 \% 5) \rightarrow (1)$

$= ((7 \% 5) \times (8 \% 5)) \% 5$

$= (2 \times 3) \% 5$

$= 1$

Modular operation - Division.

$(A / B) \% MOD = (A \% MOD) * (B^{-1} \% MOD) \% MOD.$

$(B^{-1}) \% MOD = (B^{MOD-2}) \% MOD$

Fermat's little theorem :- $a^{m-1} \equiv 1 \mod m \rightarrow a^{m-2} \equiv a^{-1} \mod m$

↳ used to calculate $B^{-1} \% mod$

if m is prime and a is not divisible by m.

$a = 7$

$m = 3$

$7^2 \equiv 1 \ (3)$

$a^{m-1} =$

$a^{m-2} = 1 \pmod m$

$a^{m-2} = \frac{1}{a} \pmod m$

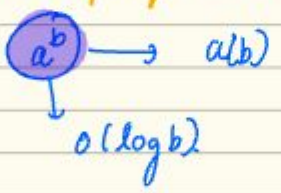
$a^{-1} = (a^{m-2})$ where m is the mod

```
(a^(m-2) %m)
int x=1
for(int i=0; i<m-2; i++){
    x *= m;
    x %= m;
}
```

O (m-2).

(ii) **BINARY EXPONENTIATION:-** Binary Exponentiation or Exponentiation by squaring is the process is the of calculating a number raised to the power another number (A^B) in logarithmic Time of the exponent or power.

Brute force time complexity :- O(b)



CORE CONCEPT:-

$$a^n = \begin{cases} 1 & \text{if } (n == 0) \\ \left(a^{\frac{n}{2}}\right)^2 & \text{if } n > 0 \text{ and } n \text{ even} \\ \left(a^{\frac{n-1}{2}}\right)^2 & \text{if } n > 0 \text{ and } n \text{ odd} \end{cases}$$

if (n is even) =>

$(a^{n/2}) \longrightarrow$ temp.

$a^n \longrightarrow$ temp * temp.

Ex :- we need to calculate 3^6.

ans = 3 x 3 x 3 x 3 x 3 x 3

temp $\longrightarrow 3^3$

ans $\longrightarrow$ temp * temp.

for Even (n == even) cases.

$\rightarrow$ Calculating for 3^7.

ans = 3 x 3 x 3 x 3 x 3 x 3 x 3

ans = temp x temp x 3

for Odd (n == odd) cases.

This reduces TC to logarithmic TC :-

O (N)

↓

b/2

↓

b/4

↓

.

.

.

O

Code:-

a^b → we need to find.

```
public findexponent(int a, int b){
    if(b==0){
        return 1;
    }
    int temp= findexponent(a,b/2);
    temp *= temp;
    if(b&1){
        temp*=a;
    }
    return temp;
}
}
```

MODULO MOD + EXPONENTION

```
public findexponent(int a, int b){
    if(b==0){
        return 1;
    }
    int temp= findexponent(a,b/2
    temp *= temp;
    temp%= mod;
    if(b&1){
        temp*=a;
        temp%= mod;
    }
    return temp;
}
}
```

~> GCD ~> Great Common Divisor / Highest Common factor (HCF).

- GCD of a and B (inbuilt function) = ___gcd(a,b)

Euclidian Theorem can also be used.

- Time Complexity :- O(log(max(a,b)))
- If m is any integer, then gcd(a,b) = gcd(a+m*b,b)
- GCD of a,b,c = GCD(a, GCD(b,c)).

- Co-prime:- also known as relatively prime or mutually prime numbers, are two or more integers that have a greatest common divisor (GCD) or highest common factor (HCF) of 1

HCF(3,5) → (1)
↘ ↙
Co-prime.

∴ GCD(a,b) ≤ min(a,b).

∴ GCD(a,0) ⟶ a

Ex:- gcd(3,0) ⟶ 3

∴ gcd(a,b) ⟶ gcd(a + m×b, b)

a %% c = 0
b %% c = 0

a + m×b → 0 %% c
a %% c = 0

Ex:- gcd(a + mb, b)

b %% c_1 = = 0 & (a + mb) %% c_1 = = 0

(m×b) %% c_1 = = 0

(a %% c_1 = = 0)

c_1 - gcd(a,b)

a %% c_1 = = 0

b %% c_1 = = 0

(iii) · LCM ⟶ least Common Multiple.

- LCM(a,b) * GCD(a,b) = a*b.
- LCM of a,b,c not equal to $\frac{(a * b * c)}{gcd(a,b,c)}$

Code:-

(iv). **Pigeonhole Principle:-** states that if there are N containers and M items with $M > N$ then there is atleast one container that contains more than one item.

Ex:-

2 containers 3balls.

∴ then there must be atleast 1 container that will have more than 1 ball.

⇒PROBLEM:-

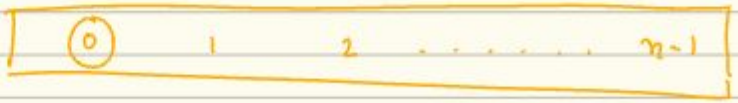
There is an array containing N elements. Determine whether there exists a non-empty such that sum of elements in the subarray is divisible by N.

↳ return YES/NO.

Prefix sum ⟶ YES (always)

$(a_3 + a_4) \% n == 0$

% n ⟶ repeats.



Video 2

↳ Problem Solving 1 :-

PROBLEM 1:- C. Mere Array

C. Mere Array

time limit per test: 2 seconds
memory limit per test: 256 megabytes

You are given an array $a_1, a_2, \ldots, a_n$ where all a_i are integers and greater than 0.

In one operation, you can choose two different indices i and j ($1 \le i, j \le n$). If $gcd(a_i, a_j)$ is equal to the minimum element of the **whole array** a, you can swap a_i and a_j. $gcd(x, y)$ denotes the greatest common divisor (GCD) of integers x and y.

Now you'd like to make a non-decreasing using the operation any number of times (possibly zero). Determine if you can do this.

An array a is non-decreasing if and only if $a_1 \le a_2 \le \ldots \le a_n$.

Input
The first line contains one integer t ($1 \le t \le 10^4$) — the number of test cases.

The first line of each test case contains one integer n ($1 \le n \le 10^5$) — the length of array a.

The second line of each test case contains n positive integers $a_1, a_2, \ldots a_n$ ($1 \le a_i \le 10^9$) — the array itself.

It is guaranteed that the sum of n over all test cases doesn't exceed 10^5.

Output
For each test case, output "YES" if it is possible to make the array a non-decreasing using the described operation, or "NO" if it is impossible to do so.

Example

input

```
4
1
8
6
4 3 6 6 2 9
4
4 5 6 7
5
7 5 2 2 4
```

output

```
YES
YES
YES
NO
```

Note
In the first and third sample, the array is already non-decreasing.

In the second sample, we can swap a_1 and a_3 first, and swap a_1 and a_5 second to make the array non-decreasing.

In the forth sample, we cannot the array non-decreasing using the operation.

a → n (Non decreasing). → increasing order. (Sorted in ascending order).

a_1 a_2 a_3 . . . . . a_n

∴ To make this array non decreasing, we need to swap the array.

We can only swap if, gcd(ai, aj) = min(a).

arr[] = | 2 | 3 | 8 | 6 |

gcd

2 → Hence, we can swap

final array = 2, 3, 6, 8

a_1 a_2 a_3 a_4 a_5 a_6

a_2 → min^m.

∴ The elements that have minimum elements as divisor can also be swapped.

2 8 4 $min^m \rightarrow 2$.

Y Y Y

All the elements that are swappable can arranged in any order I want.

2 8 4

4 8 2

4 2 8

2 4 8

STEPS :-

(i). Take out all the swappable elements, sort them and place them in their order.

(ii). Check whether that array is sorted or not.

Ex:-

4 3 6 6 2 9

Y Y Y Y

2 4 6 6

→ ∴ sorted

∴ 2 3 4 6 6 9

Code:-

```
public class Main {

    public static void solve(Scanner sc) {
        int n = sc.nextInt();
        int[] v = new int[n];
        for (int i = 0; i < n; i++) {
            v[i] = sc.nextInt();
        }

        int mini = Arrays.stream(v).min().getAsInt();

        // Collect all elements divisible by mini
        List<Integer> divisible = new ArrayList<>();
        for (int num : v) {
            if (num % mini == 0) {
                divisible.add(num);
            }
        }

        // Sort them
        Collections.sort(divisible);

        // Replace in original positions
        int idx = 0;
        for (int i = 0; i < n; i++) {
            if (v[i] % mini == 0) {
                v[i] = divisible.get(idx++);
            }
        }

        // Check if sorted
        for (int i = 1; i < n; i++) {
            if (v[i] < v[i - 1]) {
                System.out.println("NO");
                return;
            }
        }
        System.out.println("YES");
    }
```

Explaination :-

∴ All numbers divisible by m can be moved freely among each other.

If a number `x` is not divisible by `m`,
then:

- `gcd(m, x)` will be less than m, so the operation is not allowed.
- That means non-divisible numbers cannot move at all from their original positions.

So:

- Divisible numbers: fully movable among themselves.
- Non-divisible numbers: stuck in place.

∴ Put them back in their original divisible positions — because we can imagine we've swapped them to be sorted.

Check if the whole array is now sorted.

If yes "YES", because it means we could have achieved it by allowed swaps.

If no "NO", because the stuck non-divisible numbers prevent sorting.

Problem 2 :- Strong Elements (codechef)

Strong Elements

Chef has an array A of length N.

An index i is called *strong* if we can change the gcd of the whole array just by changing the value of A_i.

Determine the number of *strong* indices in the array.

Input Format

- First line will contain T, number of test cases. Then the test cases follow.
- First line of each test case contains an integer N denoting the size of the array A.
- Second line contains N space separated integers $A_1, A_2, \ldots, A_N$ - denoting the array A.

Output Format

For each test case, output the number of strong indices in the array.

Constraints

- $1 \leq T \leq 5 \cdot 10^4$
- $2 \leq N \leq 3 \cdot 10^5$
- $1 \leq A_i \leq 10^9$
- Sum of N over all test cases do not exceed $3 \cdot 10^5$.

Sample 1:

Input	Output
3 3 5 10 20 4 3 5 7 11 4 2 2 2 2	3 0 4

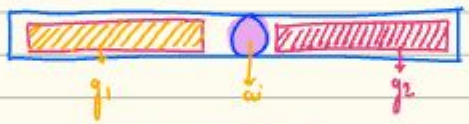
Explanation:

Test Case 1: All the indices are strong.

- For index 1 you can change the element to 10 which changes the gcd of the array to 10.
- For index 2 you can change the element to 12 which changes the gcd of the array to 1.
- For index 3 you can change the element to 12 which changes the gcd of the array to 1.

Test Case 2: No index is strong. If you change any single element, gcd still remains the same.

Test Case 3: All the indices are strong. You can change any element to 3. This changes the gcd of the array to 1.



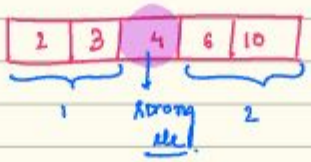
∴ gcd of entire array = $\gcd(q_1, a_i, q_2)$

∴ $\gcd(a_i, \gcd(q_1, q_2))$

$\gcd(a_i, \gcd(q_1, q_2))$

Strong q_1 → ∴ this number can only be strong if $\gcd(\text{left}(a_i), \text{right}(q_2)) \neq 1$

Ex



∴ Even if we change the value of strong element, it doesn't matter because the left GCD is 1. The entire GCD's values doesn't depend on the

strong element.

∴ We need to know the GCD (left element) and GCD (right Elements).

$O(n^2)$ → can't be done.

∴ we need to maintain prefix GCD & Suffix GCD

TC:- $O(n)$.

Code :-

```
public class StrongElements {

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

    public static void solve(Scanner sc) {
        int n = sc.nextInt();
        int[] v = new int[n];
        for (int i = 0; i < n; i++) {
            v[i] = sc.nextInt();
        }

        if (n == 1) {
            System.out.println(1);
            return;
        }

        int[] prefix_gcd = new int[n];
        int[] suffix_gcd = new int[n];

        prefix_gcd[0] = v[0];
        suffix_gcd[n - 1] = v[n - 1];

        for (int i = 1; i < n; i++) {
            prefix_gcd[i] = gcd(prefix_gcd[i - 1], v[i]);
        }

        for (int i = n - 2; i >= 0; i--) {
            suffix_gcd[i] = gcd(suffix_gcd[i + 1], v[i]);
        }

        int ans = 0;
        for (int i = 1; i < n - 1; i++) {
            if (gcd(prefix_gcd[i - 1], suffix_gcd[i + 1]) != 1) {
                ans++;
            }
        }

        // checking for first element
        if (suffix_gcd[1] != 1) {
            ans++;
        }

        // checking for the last element
        if (prefix_gcd[n - 2] != 1) {
            ans++;
        }

        System.out.println(ans);
    }
```

23 Problem 3: XOR Pyramid Peak

Xor Pyramid Peak

TASK

Time limit: 1.00 s **Memory limit:** 512 MB

Consider a xor pyramid where each number is the xor of lower-left and lower-right numbers. Here is an example pyramid:

			9				
			15	6			
		4	11	13			
		11	15	4	9		
		1	10	5	1	8	
		7	6	12	9	8	0
	8	15	9	5	12	4	4
2	10	5	12	9	5	1	5

Given the bottom row of the pyramid, your task is to find the topmost number.

Input

The first line has an integer n: the size of the pyramid.

The next line has n integers $a_1, a_2, \ldots, a_n$: the bottom row of the pyramid.

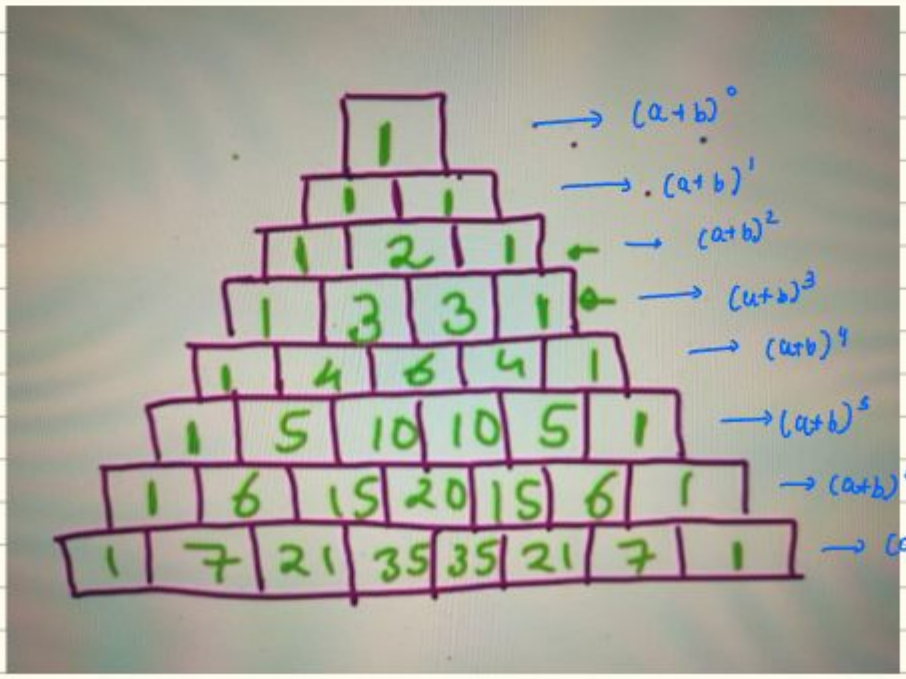
Constraints

- $1 \le n \le 2 \cdot 10^5$
- $1 \le a_i \le 10^9$

Example

Input:
```
8
2 10 5 12 9 5 1 5
```

Output:
```
9
```



$(a+b)^2 = a^2 + 2ab + b^2$

$(a+b)^3 = a^3 + 2a^2b + 3ab^2 + b^3$

∴ If we do XOR of a number even no. of times it will return → 0.

∴ If we do XOR of a number odd no. of times it will return → number itself.

Video 2:- Problem Solving 2.

Problem 1:- A. Prime Subtraction.

$x - p \times z = y$

$x - y$

Number that will be subtracted in total from x to get y.

if $(x - y \geq 1)$ then it is always possible to get y from x.

if $((x - y) == 1)$ ⟶ No

↳ else ⟶ Yes.

Problem 2:- Divisor Analysis :-

Divisor Analysis

TASK | STATISTICS

Time limit: 1.00 s **Memory limit:** 512 MB

Given an integer, your task is to find the number, sum and product of its divisors. As an example, let us consider the number 12:

- the number of divisors is 6 (they are 1, 2, 3, 4, 6, 12)
- the sum of divisors is $1+2+3+4+6+12=28$
- the product of divisors is $1 \cdot 2 \cdot 3 \cdot 4 \cdot 6 \cdot 12 = 1728$

Since the input number may be large, it is given as a prime factorization.

Input

The first line has an integer n: the number of parts in the prime factorization.

After this, there are n lines that describe the factorization. Each line has two numbers x and k where x is a prime and k is its power.

Output

Print three integers modulo 10^9+7: the number, sum and product of the divisors.

Constraints

- $1 \leq n \leq 10^5$
- $2 \leq x \leq 10^6$
- each x is a distinct prime
- $1 \leq k \leq 10^9$

Example

Input:
```
2
2 2
3 1
```

Output:
```
6 28 1728
```

$N = P_1^{k_1} \times P_2^{k_2} \times P_3^{k_3} \times P_4^{k_4} \ldots\ldots$

P_i's → Prime factors of N

k_i's → Power of every prime factor.

Number of divisors = $P_1^{k_1} \times P_2^{k_2} \times \ldots\ldots P_i^{k_i}$ $\left(12 = 2^2 \times 3\right) = 4 \times 3$.

$0, 1, 2, \ldots\ldots k_1$

$(k_1+1)\ (k_2+1) \ldots\ldots (k_i+1)$ (occurrence).

$12 \longrightarrow 2^2$

$N = P_1^{k_1} \times P_2^{k_2} \times P_3^{k_3} \times \ldots\ldots\ldots$

Sum of divisors:

$(P_1^0 + P_1^1 + \ldots\ldots P_1^{k_1})\ (P_2^0 + P_2^1 + \ldots\ldots P_2^{k_2})$

Sum of G.P.

$$\left(\frac{P_1^{k_1+1} - 1}{P_1 - 1}\right)\left(\frac{P_2^{k_1+1} - 1}{P_2 - 1}\right) \ldots\ldots$$

The problem deals with **prime factorization of a number** and then calculating:

1. **Number of divisors**
   - If a number is factorized as:

     $$N = p_1^{a_1} \cdot p_2^{a_2} \cdot \cdots \cdot p_n^{a_n}$$

     Then,

     $$\text{Number of divisors} = (a_1 + 1)(a_2 + 1)\ldots(a_n + 1)$$
2. **Sum of divisors**
   - Formula:

     $$\text{Sum of divisors} = \prod_{i=1}^{n} \frac{p_i^{a_i+1} - 1}{p_i - 1}$$
   - Each prime's contribution is a **geometric series**.
3. **Product of divisors**
   - Formula:

     $$\text{Product of divisors} = N^{d/2}$$

     where $d = \text{Number of divisors}$.
   - Equivalent implementation: compute carefully using modular exponentiation.

Why two mods (`mod` and `mod2`)?

- Because the final exponents can get **too large**.
- We compute powers modulo `mod-1` (i.e., `1e9+6`) when applying Fermat's little theorem in exponentiation.

```
public class DivisorsCalculator {
    static final long MOD = 1000000007L;
    static final long MOD2 = 1000000006L; // (MOD - 1)

    // Fast exponentiation (a^b % mod)
    static long expo(long a, long b, long mod) {
        long res = 1;
        a %= mod;
        while (b > 0) {
            if ((b & 1) == 1) res = (res * a) % mod;
            a = (a * a) % mod;
            b >>= 1;
        }
        return res;
    }

    // Modular inverse (works only if mod is prime)
    static long modInverse(long a, long mod) {
        return expo(a, mod - 2, mod);
    }

    static long modMul(long a, long b, long mod) {
        return ((a % mod) * (b % mod)) % mod;
    }

    static long modSub(long a, long b, long mod) {
        return ((a % mod - b % mod + mod) % mod);
    }

    static long modDiv(long a, long b, long mod) {
        return modMul(a, modInverse(b, mod), mod);
    }

    public static void solve(Scanner sc) {
        int n = sc.nextInt();
        long[] prime = new long[n];
        long[] power = new long[n];

        for (int i = 0; i < n; i++) {
            prime[i] = sc.nextLong();
            power[i] = sc.nextLong();
        }

        // Number of divisors
        long ans1 = 1;
        for (int i = 0; i < n; i++) {
            ans1 = (ans1 * (power[i] + 1)) % MOD;
        }
        System.out.print(ans1 + " ");

        // Sum of divisors
        long ans2 = 1;
        for (int i = 0; i < n; i++) {
            long num = expo(prime[i], power[i] + 1, MOD);
            num = (num - 1 + MOD) % MOD;
            long deno = prime[i] - 1;
            ans2 = (ans2 * modDiv(num, deno, MOD)) % MOD;
        }
        System.out.print(ans2 + " ");
```

```
        long num = 1;
        long pro = 1;
        for (int i = 0; i < n; i++) {
            pro = expo(pro, (power[i] + 1), MOD);
            long temp = (power[i] * (power[i] + 1)) / 2;
            long base = expo(prime[i], temp, MOD);
            long val = expo(base, num, MOD);
            pro = (pro * val) % MOD;
            num = (num * (power[i] + 1)) % MOD2;
        }
        System.out.println(pro);
    }

    public static void main(String[] args) throws Exception {
        Scanner sc = new Scanner(System.in);
        int t = 1;
        // t = sc.nextInt(); // if multiple test cases
        while (t-- > 0) {
            solve(sc);
        }
        sc.close();
    }
}
```

Problem Solving 3

↳ Video 3.

C. No Prime Differences

time limit per test: 2 seconds
memory limit per test: 256 megabytes

You are given integers n and m. Fill an n by m grid with the integers 1 through $n \cdot m$, in such a way that for any two adjacent cells in the grid, the absolute difference of the values in those cells is not a prime number. Two cells in the grid are considered adjacent if they share a side.

16	7	1	9
12	8	2	3
13	4	10	11
14	5	6	15

It can be shown that under the given constraints, there is always a solution.

Input

The first line of the input contains a single integer t ($1 \le t \le 1000$) — the number of test cases. The description of the test cases follows.

The first and only line of each test case contains two integers n and m ($4 \le n, m \le 1000$) — the dimensions of the grid.

It is guaranteed that the sum of $n \cdot m$ over all test cases does not exceed 10^6.

Output

For each test case, output n lines of m integers each, representing the final grid. Every number from 1 to $n \cdot m$ should appear exactly once in the grid.

The extra spaces and blank lines in the sample output below are only present to make the output easier to read, and are **not** required.

If there are multiple solutions, print any of them.

Example

input

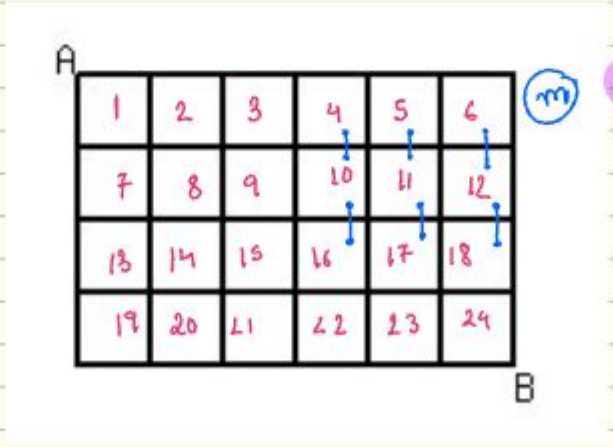
```
3
4 4
5 7
6 4
```

output

```
16  7  1  9
12  8  2  3
13  4 10 11
14  5  6 15

29 23 17  9  5  6  2
33 27 21 15 11  7  1
32 31 25 19 20 16 10
26 30 24 18 14  8  4
35 34 28 22 13 12  3

 2  3  7 11
 8  9  1 18
17 13  5  4
18 14  6 12
19 23 15 21
20 24 16 22
```

Note

The first sample case corresponds to the picture above. The only absolute differences between adjacent elements in this grid are 1, 4, 6, 8, and 9, none of which are prime.

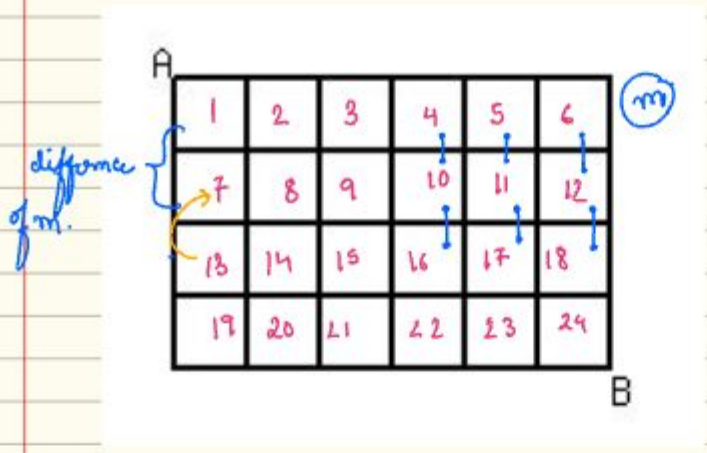


difference between adjacent is m - if we write the number from 1 . . . . . continously

m → not a prime

(m x m)

(√m).



∴ let us put 2nd in the next row. then.

the the number placed then will be $2^{*}m$.
& hence it will have 1, 2, m as its factors and hence it will never be a prime number.

∴ Hence for n block (n rows) we can place $[n/2]$ floor values at even positions.

$[n/2]$ (odd)	11	12	13	14	15
$[n/2]$ (even)	1	2	3	4	5
$[n/2]$ (odd)	16	17	18	19	20
$[n/2]$ (even)	6	7	8	9	10.

∴ if the n is atleast 4, the we won't get prime as any of its neighbour.

Ex: 1

$n=2$ $m=5$

6	7	8	9	10
1	2	3	4	5

5 prime.

$n=3$ $m=5$

6	7	8	9	10
1	2	3	4	5
11	12	13	14	15

5 prime.

Code :-

Problem 2 → A - Division.

A. Division

time limit per test: 1 second
memory limit per test: 512 megabytes

Oleg's favorite subjects are History and Math, and his favorite branch of mathematics is division.

To improve his division skills, Oleg came up with t pairs of integers p_i and q_i and for each pair decided to find the **greatest** integer x_i, such that:

- p_i is divisible by x_i;
- x_i is not divisible by q_i.

Oleg is really good at division and managed to find all the answers quickly, how about you?

Input

The first line contains an integer t ($1 \le t \le 50$) — the number of pairs.

Each of the following t lines contains two integers p_i and q_i ($1 \le p_i \le 10^{18}$; $2 \le q_i \le 10^9$) — the i-th pair of integers.

Output

Print t integers: the i-th integer is the largest x_i such that p_i is divisible by x_i, but x_i is not divisible by q_i.

One can show that there is always at least one value of x_i satisfying the divisibility conditions for the given constraints.

Example

input

```
3
10 4
12 6
179 822
```

output

```
10
4
179
```

Note

For the first pair, where $p_1 = 10$ and $q_1 = 4$, the answer is $x_1 = 10$, since it is the greatest divisor of 10 and 10 is not divisible by 4.

For the second pair, where $p_2 = 12$ and $q_2 = 6$, note that

- 12 is not a valid x_2, since 12 is divisible by $q_2 = 6$;
- 6 is not valid x_2 as well: 6 is also divisible by $q_2 = 6$.

The next available divisor of $p_2 = 12$ is 4, which is the answer, since 4 is not divisible by 6.

I/P :- p. q.

O/P :- largest X.

p % x == 0 — necessary

x % q ≠ 0

∴ Approach :-

```
if (p % q != 0){
    x = p
}
```

→ ∴ p can be the largest divisor if p % q != 0

& if p % q == 0

Every prime factor of q is a prime factor of p as well.

∴ The power of prime number in p is always greater than power of prime in q.

$p \rightarrow 2^3 \times 3^3 \times 5^1 \times 7^2$

$q \rightarrow 2^2 \times 3^3 \times 5^3 \times 7^0$

$\therefore q$ will divide p.

largest

$x \rightarrow 2^1 \times 3^2 \times 5^1 \times 7^2$

$p \rightarrow 2^6 \times 3^2 \times 5$

$q \rightarrow 2^2 \times 3^2$.

$\therefore$ Take minimum of all, and the lower down the value.

$X \rightarrow p/32$ $\quad$ $x \rightarrow p/3$

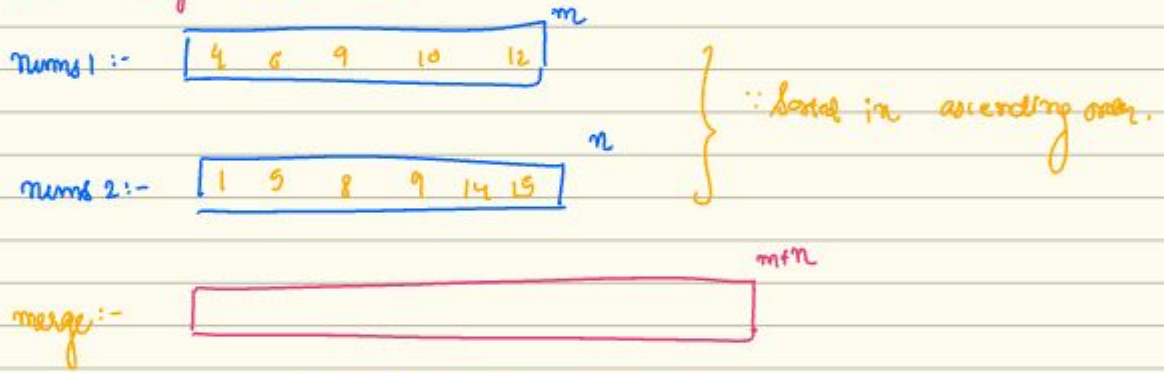
MODULE 10 :- Advanced Searching and Sorting.

↳ Video 1 :- Advanced Sorting.

→ Merge Sort :-

- Merge sort follows divide and conquer strategy.
- Recursively splits the array into halves until each contains a single element.
- Merges sorted halves back together in a sorted manner.
- TC :- $O(n \log n)$
- SC :- $O(n)$

Merge sorted array :- (Leet code)

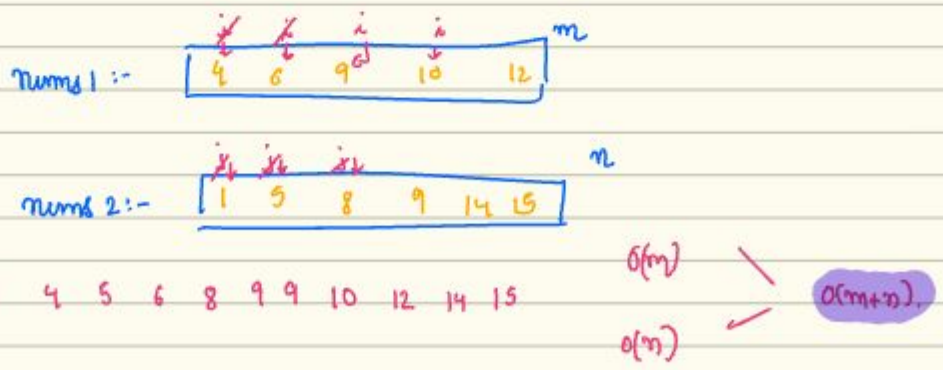


$(m+n) \log (m+n)$

∴ Can we sort this in $O(m+n)$?

↳ Yes,

∴ The arrays are already sorted



1 4 5 6 8 9 9 10 12 14 15

$O(m)$, $O(n)$ → $O(m+n)$.

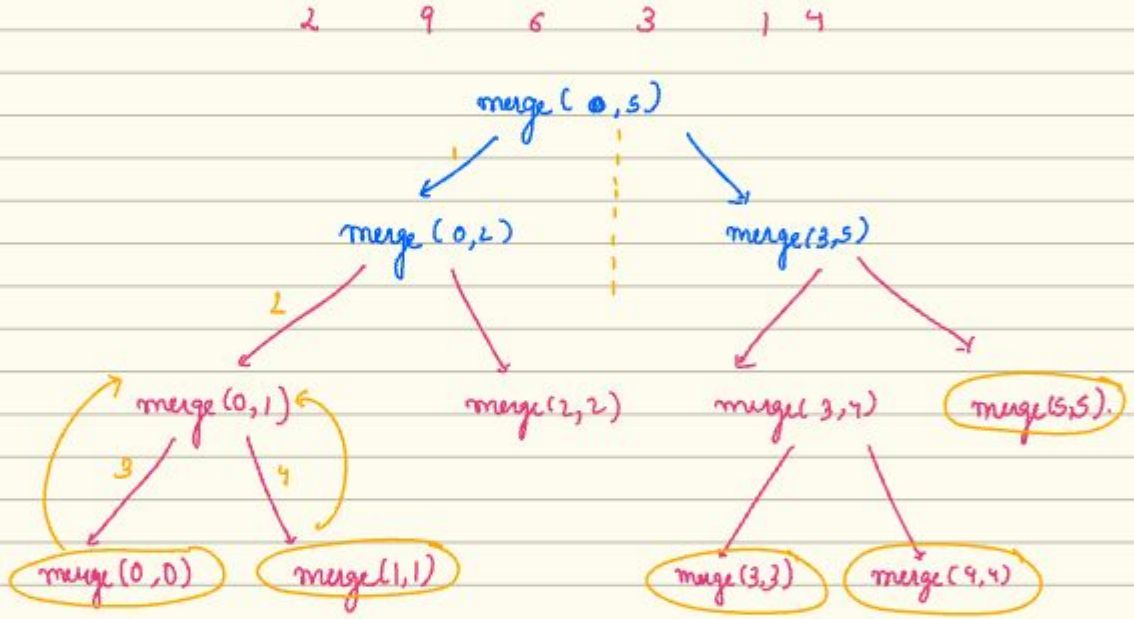
Two halves :- $mid = si + \left(\frac{ei - si}{2}\right)$

$mid = \left(\frac{0+5}{2}\right) = 2.$

(si ⟶ mid) (mid+1 ⟶ ei)

H_1 H_2.

3 3



CODE:-

```java
public static void merge(int[] arr, int si, int ei) {
    int mid = (si + ei) / 2;
    int i = si, j = mid + 1;
    int k = 0;

    while (i <= mid && j <= ei) {
        if (arr[i] <= arr[j]) {
            temp[k] = arr[i];
            i++;
        } else {
            temp[k] = arr[j];
            j++;
        }
        k++;
    }

    while (i <= mid) {
        temp[k++] = arr[i++];
    }

    while (j <= ei) {
        temp[k++] = arr[j++];
    }

    int t = si;
    for (int idx = 0; idx <= ei - si; idx++) {
        arr[t++] = temp[idx];
    }
}
```

```java
public static void mergeSort(int[] arr, int si, int ei) {
    if (si >= ei) return;

    int mid = (si + ei) / 2;
    mergeSort(arr, si, mid);
    mergeSort(arr, mid + 1, ei);
    merge(arr, si, ei);
}

public static void solve(Scanner sc) {
    int n = sc.nextInt();
    int[] arr = new int[n];
    for (int i = 0; i < n; i++) {
        arr[i] = sc.nextInt();
    }

    mergeSort(arr, si:0, n - 1);

    for (int val : arr) {
        System.out.print(val + " ");
    }
    System.out.println();
}
```

$T(n) = T(n/2) + T(n/2) + O(n)$

$T(n) = 2 \times T(n/2) + O(n)$.

~~$T(n/2) = 4 \times T(n/4) + O(n) \times 2$~~

~~$T(n/2) = 8 \times T(n/8) + O(n) \times 4$~~

$\therefore T(n) = O(n) + O(n) + O(n) + \ldots$ x times.

$x \longrightarrow \log_2 n$

n
$\downarrow$
$n/2$
$\downarrow$
$n/4$
$\downarrow$
$n/8$

$O(n) \times \log_2 n$

$\downarrow$

$2^x = n$

$\therefore x = \log_2 n$

→ Inversion Count.

- For an array, inversion count indicates how far or close the array is from being sorted. If the array is from being sorted.
- If the array is already sorted the inversion count is 0.
- If an array is sorted in reverse order then the inversion count is the more.
- Two array elements arr[i] and arr[j] form an inversion if arr(i) > arr(j) & i < j.

Inversion Count: 2

2	9	6	4	5	3
0	0	1	2	2	3

↳ Inversion count = 9.

$\therefore$ Best D/S for getting inversion count => ordered set.

$\therefore I = I_1 + I_2$ + inversions between left half and right half.

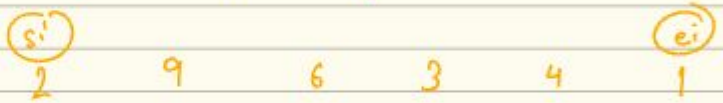
$1 + 2 + 6 = 9$.

```java
public class InversionCountMergeSort {

    static int[] temp = new int[100000];
    static long inversionCount; // To store total inversions

    public static void merge(int[] arr, int si, int ei) {
        int mid = (si + ei) / 2;
        int i = si, j = mid + 1;
        int k = 0;

        while (i <= mid && j <= ei) {
            if (arr[i] <= arr[j]) {
                temp[k++] = arr[i++];
            } else {
                temp[k++] = arr[j++];
                inversionCount += (mid - i + 1); // key step
            }
        }

        while (i <= mid) {
            temp[k++] = arr[i++];
        }

        while (j <= ei) {
            temp[k++] = arr[j++];
        }

        int t = si;
        for (int idx = 0; idx <= ei - si; idx++) {
            arr[t++] = temp[idx];
        }
    }
```

```java
    public static void mergeSort(int[] arr, int si, int ei) {
        if (si >= ei) return;

        int mid = (si + ei) / 2;
        mergeSort(arr, si, mid);
        mergeSort(arr, mid + 1, ei);
        merge(arr, si, ei);
    }

    public static void solve(Scanner sc) {
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }

        inversionCount = 0; // reset for each test case
        mergeSort(arr, si:0, n - 1);

        System.out.println(inversionCount);
    }

    Run | Debug
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int t = sc.nextInt();

        while (t-- > 0) {
            solve(sc);
        }

        sc.close();
    }
}
```

→ Quick Sort →

- Quick sort is also a Divide and Conquer algorithm.
- Selects a pivot element, partitions the array, and sorts each partion recursively.
- Efficient for largest dataset.
- Time Complexity:-
  - Best & avg. Case:- $O(n \log n)$.
  - Worst case :- $O(n^2)$ (when pivot selection is bad)
- Space Complexity:- $O(\log n)$ (in-place sorting).



Quick sort.

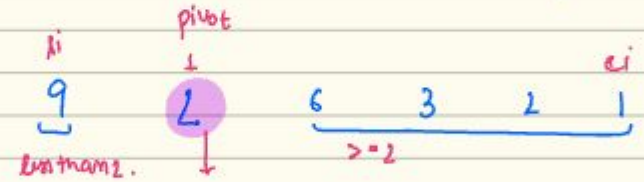
1

count=1

No. of elements less than 2.

∴ No. of elements < 2. = 1

then the correct pos. for 2 will be si+1.



2 is placed in its correct position.

∴ If I place all elements towards left of 2 as < 2, then automatically. all elements towards right of 2 >= 2.

∴ si - p - 1 (p) p+1 → ei

↓

Partition Index

```java
public static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;        // index of smaller element

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            // swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // place pivot in correct position
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
}

public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

public static void solve(Scanner sc) {
    int n = sc.nextInt();
    int[] arr = new int[n];
    for (int i = 0; i < n; i++) {
        arr[i] = sc.nextInt();
    }

    quickSort(arr, low:0, n - 1);

    for (int val : arr) {
        System.out.print(val + " ");
    }
    System.out.println();
}
```

↳ Video 2

↳ Binary search.

→ Binary search:-

- Binary search is an efficient searching algorithm used in sorted array.
- It works by repeatedly dividing the search space in half.
- Time Complexity:- O(log N)

Linear Search:- O(N) complexity.

→ Binary search:-

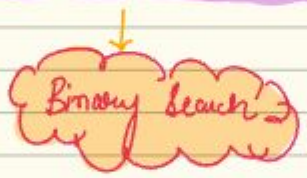
si
2 3 4 6 9 ei
12

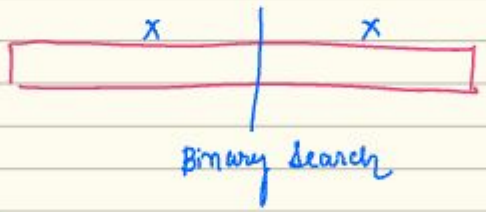
int si = INT_MAX - 1
int ei = INT_MAX

si = 0, ei = 5 } mid → (si+ei)/2

si + (ei - si)/2

∴ At every step we reduce our search space by half





Binary search code:-

```
public static int binarySearch(int[] arr, int target) {
  int low = 0;
  int high = arr.length - 1;

  while (low <= high) {
    int mid = low + (high - low) / 2;

    if (arr[mid] == target) {
      return mid;
    } else if (arr[mid] < target) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }

  return -1; // target not found
}
```

→ **Binary search on answer :-**

- Used when search for a minimum or maximum satisfying condition.
- Example:- Finding the smallest k that satisfies a function.
- Typical form:-

→ TTTTTTFFFFF, where T = true and F = false

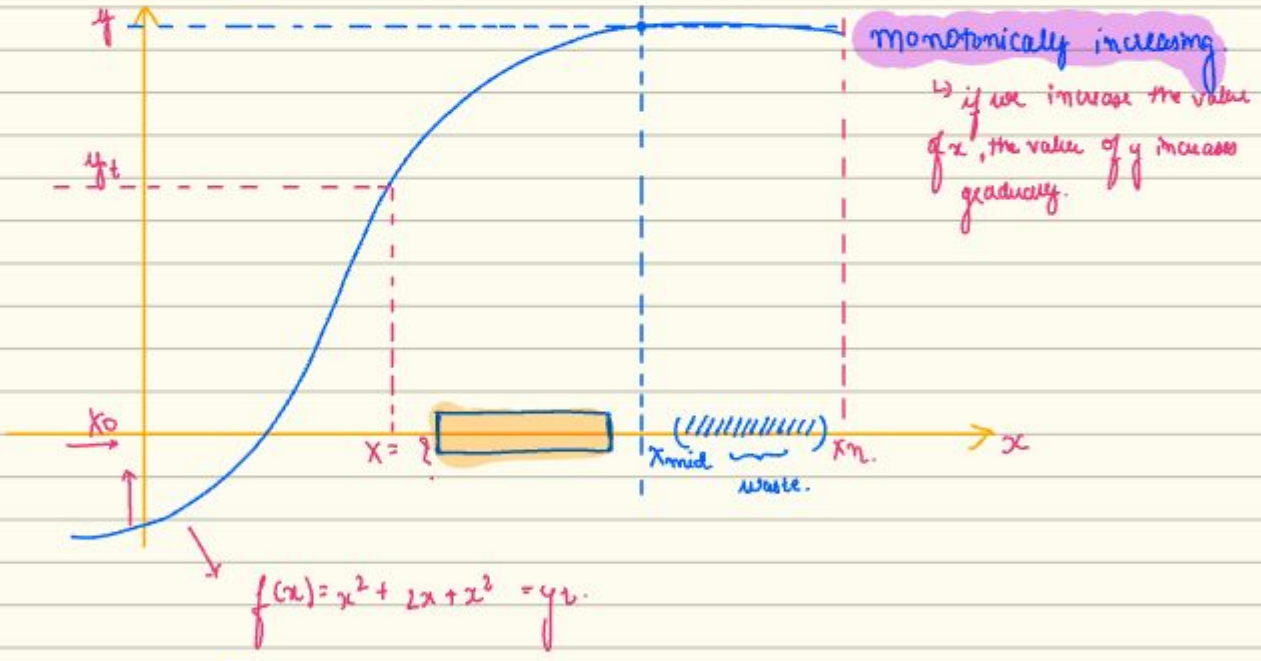
∴ Binary search for the last True or first false.

→ FF FFFF TTTTT, where T = true and F = false

∴ Binary search for the first True or last false.

Binary search ans :-

→ minimize the maximum.
→ maximize the minimum.



bool function → monotonically increasing.
↓ decreasing.

T → f(x) > Y → returns false.

f(x) < Y → returns true.

→ Problem :-

Given a number N, return the square root of N rounded down to an integer.

Ex :- N = 14, Output = 3.

↳ Video 3

↳ Problem Solving

→ Problem 1:- E. Building An aquarium. (Codeforces).

E. Building an Aquarium

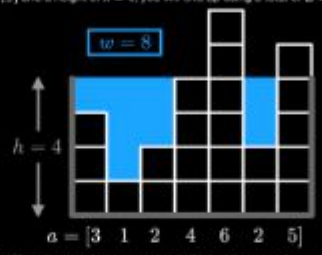
time limit per test: 2 seconds

memory limit per test: 256 megabytes

You love fish, that's why you have decided to build an aquarium. You have a piece of coral made of n columns, the i-th of which is a_i units tall. Afterwards, you will build a tank around the coral as follows:

- Pick an integer $k \geq 1$ — the height of the tank. Build walls of height k on either side of the tank.
- Then, fill the tank up with water so that the height of each column is k, unless the coral is taller than k; then no water should be added to this column.

For example, with $a = [3, 1, 2, 4, 6, 2, 5]$ and a height of $k = 4$, you will end up using a total of $w = 8$ units of water, as shown.



You can use at most x units of water to fill up the tank, but you want to build the biggest tank possible. What is the largest value of k you can select?

Input

The first line contains a single integer t ($1 \leq t \leq 10^4$) — the number of test cases.

The first line of each test case contains two positive integers n and x ($1 \leq n \leq 2 \cdot 10^5$; $1 \leq x \leq 10^9$) — the number of columns of the coral and the maximum amount of water you can use.

The second line of each test case contains n space-separated integers a_i ($1 \leq a_i \leq 10^9$) — the heights of the coral.

The sum of n over all test cases doesn't exceed $2 \cdot 10^5$.

Output

For each test case, output a single positive integer k ($k \geq 1$) — the maximum height the tank can have, so you need at most x units of water to fill up the tank.

We have a proof that under these constraints, such a value of k always exists.

Example

input

```
5
7 9
3 1 2 4 6 2 5
3 10
1 1 1
4 1
1 4 3 4
6 1984
2 6 5 9 1 8
1 1000000000
1
```

output

```
4
4
2
115
1000000001
```

Note

The first test case is pictured in the statement. With $k = 4$ we need 8 units of water, but if k is increased to 5 we need 13 units of water, which is more than $x = 9$. So $k = 4$ is optimal.

Code:-

```java
import java.util.Scanner;

public class BuildingAnAquarium {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int t = sc.nextInt();
        while (t-- > 0) {
            int n = sc.nextInt();
            int x = sc.nextInt();
            int[] a = new int[n];
            for (int i = 0; i < n; i++) {
                a[i] = sc.nextInt();
            }

            long start = 1;  // Changed to long to handle large numbers
            long end = (long) 1e12;  // Changed to long
            long ans = -1;

            while (start <= end) {
                long mid = start + (end - start) / 2;
                if (check(a, x, mid)) {
                    ans = mid;
                    start = mid + 1;
                } else {
                    end = mid - 1;
                }
            }
            System.out.println(ans);
        }
    }

    public static boolean check(int[] a, long x, long mid) {  // Changed parameters
        long units = 0;  // Changed to long
        int n = a.length;
        for (int i = 0; i < n; i++) {
            if (a[i] < mid) {
                units += (mid - a[i]);
            }
        }
        return units <= x;  // Compare with x, not mid
    }
}
```

Problem 2:- Number of flowers in full bloom.

2251. Number of Flowers in Full Bloom

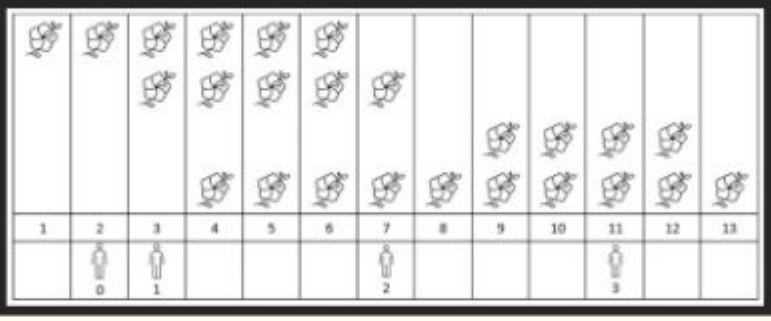
Discuss Approach ›

Hard | Topics | Companies | Hint

You are given a **0-indexed** 2D integer array `flowers`, where `flowers[i] = [start_i, end_i]` means the `i^th` flower will be in **full bloom** from `start_i` to `end_i` (**inclusive**). You are also given a **0-indexed** integer array `people` of size `n`, where `people[i]` is the time that the `i^th` person will arrive to see the flowers.

Return *an integer array* `answer` *of size* `n`*, where* `answer[i]` *is the* ***number*** *of flowers that are in full bloom when the* `i^th` *person arrives.*

Example 1:




```
Input: flowers = [[1,6],[3,7],[9,12],[4,13]], people = [2,3,7,11]
Output: [1,2,2,2]
Explanation: The figure above shows the times when the flowers are in full
bloom and when the people arrive.
For each person, we return the number of flowers in full bloom during their
arrival.
```

code :-

→ MODULE 5 :- Recursion

↳ Video 1 :- Recursion.

⇝ Recap on functions :- Functions are reusable blocks of code that can be run whenever called

They can take in parameters (input) and return a value (output)

Syntax :- `return_type (d1 param1, d2 param2 ....) {`

`// result must be same as return_type`

`return result;`

`}`

⇝ What is Recursion :-

- Recursion happens when a function calls itself on different set of input parameters.
- Used when the solution for current problem involves first solving a smaller sub-problem.

Example - Sum of First N natural numbers.

(i) If we want to find sum of first N natural numbers, we can easily get that if we know the sum of first (N-1) natural numbers.

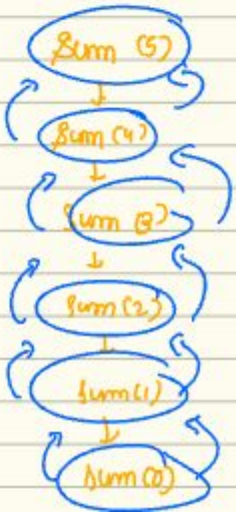
(ii) So finding sum of first (N-1) natural numbers become a smaller subtask for us.

(iii). Similar thing happens with recursion. By using the answer from smaller subproblem we can solve bigger problems.

Sum of first 40 Natural numbers.

$N \rightarrow (N)(N+1)/2.$

$1+2+3+4+5+\ldots\ldots\ldots+40.$



Recursion :-

40 → 39 th

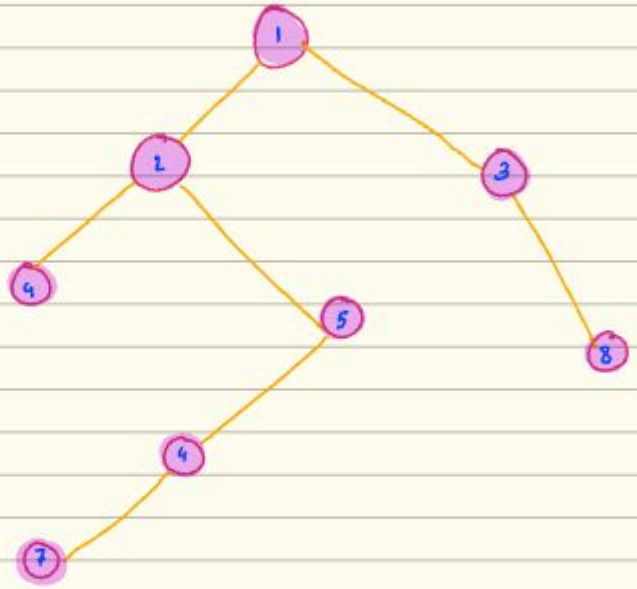
(X) + 40

∴ It helps us to solve bigger problems by solving smaller subproblems.

⇝ Recursion Tree :-

- A recursive tree is similar to a "mind map" of the function call.
- Recursive trees are useful to help us understand how the function acts.

Ex :-



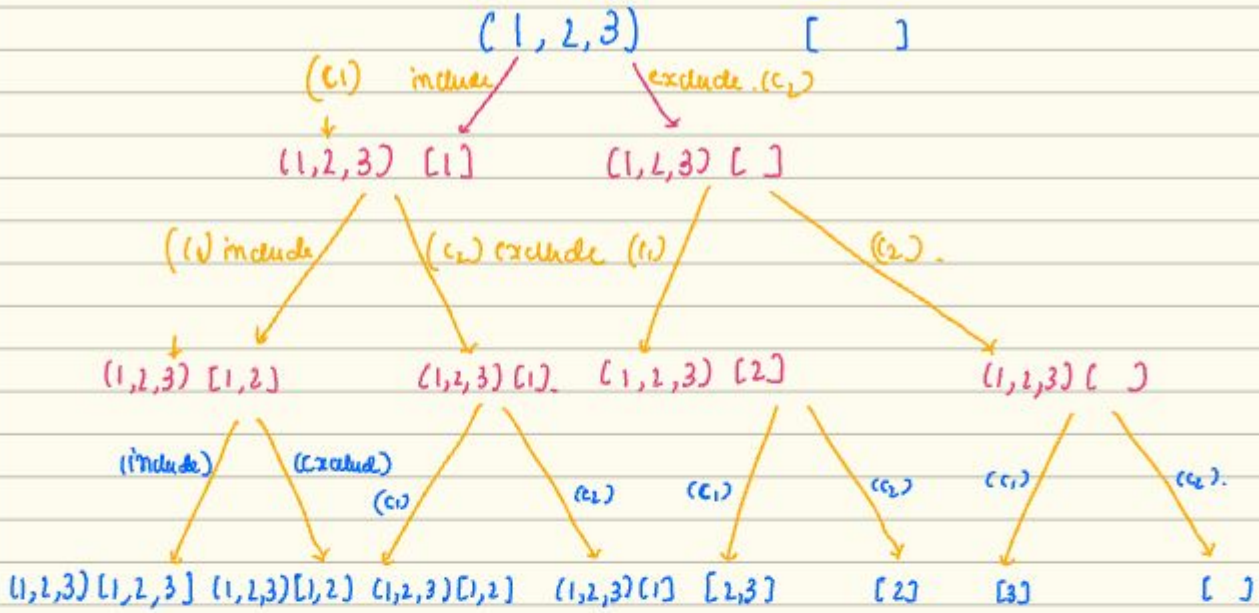
Example :- Recursion Tree

Solution :- For all the elements in the array, I have two choices. Either I include them in the subset (Choice (1) or I don't include them (Choice (2)

Question :- Generate all the subsets of an array [1, 2, 3] →

[ ], [1], [2], [3], [1,2], [2, 3], [1,3], [1,2,3].

Recursion Tree for generating all the subsets.



⇝ Basic structure of Recursive function.

(i). Parameters to start the function.

(ii). Appropriate base case(s) to end the recursion (why?).

(iii). Recursively solve the sub problems.

(iv). Process the result and return value.

→ Example:- N^{th} fibonacci number

The fibonacci sequence is the series of numbers where each number is the sum of the two preceding numbers.

For ex:-

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

Mathematically, we can describe this as:-

$$F_n = F_{n-1} + F_{n-2}, \text{ for all } n >= 3$$

Ex:- N^{th} Fibonacci Number:-

```
int fib(int n){
    if(n==0) return 0;
    if(n==1) return 1;
    return fib(n-1) + fib(n-2);
}
```

BASE CASE.

5th Fib no:-

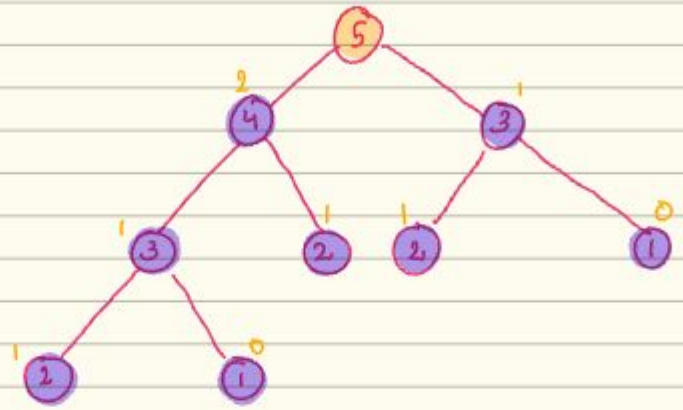
1st fib → 0

2nd fib → 1

5th ← 4th + 3rd

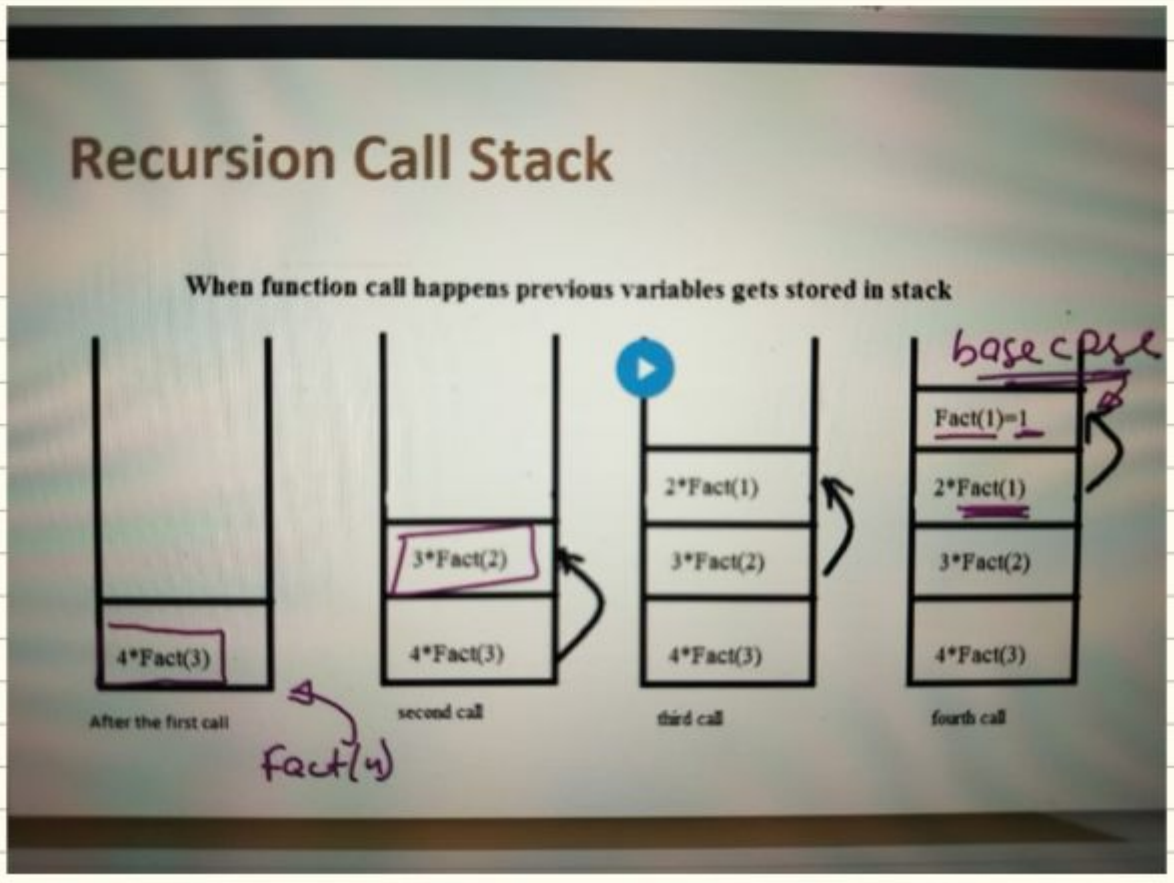
4th ← 3rd + 2nd

3rd ← 1st + 2nd



→ Recursion Call Stack:-

- When a recursion is executed, the function calls are placed in a stack.
- The first call is at the bottom of the stack, and the last call is at the top.
- After a call, it is popped from the stack, and the returned value is passed to the current top element.



→ Video 2

Time Complexity in Recursion

→ Substitution Method:- In this method we make a guess for the solution and that we use mathematical induction to prove if the guess is correct or incorrect.

Ex:-

Consider a recurrence relation:-

$$T(n) = 2T(n/2) + n.$$

We guess the solution as $T(n) = O(n \log n)$. Now we use induction to prove our guess.

We need to prove that $T(n) <= cn \log n$ (c is a constant). We can assume that it is true for values smaller than n.

→ Big O notation

$$T(n) = 2T(n/2) + n$$
$$<= 2cn/2 \log(n/2) + n$$

∵ From module:-

$$= cn \log n - cn \log 2 + n$$
$$= cn \log n - cn + n$$
$$<= cn \log n$$

Derivation.

- $T(n) = 2T\left(\frac{n}{2}\right) + n.$
- Assume, the TC:- $T(n) = O(n \log n)$

manually deriving:-

- ∵ $T(n) \leq cn \log n$
- $T(n) \leq \cancel{x} \times \frac{cn}{\cancel{x}} \log\left(\frac{n}{2}\right) + n$
- $T(n) \leq cn \log\left(\frac{n}{2}\right) + n$
- $\log\left(\frac{a}{b}\right) = \log a - \log b.$
- $T(n) \leq cn \log n - cn \log + n$

$$\leq cn \log n$$

- $T(n) \leq cn \log n$

∴ $T(n) = O(n \log n)$

$T(n) \leq (c+1)n$

↓

$O(n)$ ✗ $T(n) \leq cn + n$ — contradiction.

$T(n) \leq cn$

$T(n): O(n^2) \Rightarrow T(n) \leq n^2$

for some some constant c.

$T(n): 2T(n/2) + n$

$\Rightarrow T(n) \leq 2 \times \frac{cn^2}{4} + n$

$\therefore \quad Tn \leq \frac{cn^2}{2} + n$

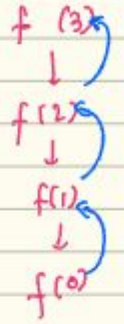
Video 3

Problem Solving.

→ Problem 1 :-

Given two number a and b, calculate a^b.

```java
public static int findPow(int a, int b) {
    if (b == 0) {
        return 1;
    }
    if (b == 1) {
        return a;
    }
    return findPow(a, b - 1) * a;
}
```



Time Complexity :- O(b).

→ Problem 2 :-

Generate Binary String for number (n).

Ex :-

n = 5 , return (101).

genrateBinaryString.java > ...

```java
public class genrateBinaryString {    "genrate": Unknown word.
    public static String toBinary(int n) {
        if (n == 0) {
            return "";
        }
        return toBinary(n / 2) + (n % 2);
    }

    Run | Debug
    public static void main(String[] args) {
        int num = 5;
        String binary = toBinary(num);


        if (binary.equals(anObject:"")) {
            binary = "0";
        }

        System.out.println("Binary of " + num + " is: " + binary);
    }
}

```

→ Problem 3 :-

↳ Generate Binary String without Adjacent Zeros.

3211. Generate Binary Strings Without Adjacent Zeros

Discuss Approach ›

Medium | Topics | Companies | Hint

You are given a positive integer `n`.

A binary string `x` is **valid** if all substrings of `x` of length 2 contain **at least** one `"1"`.

Return all **valid** strings with length `n`, in *any* order.

Example 1:

> **Input:** `n = 3`
>
> **Output:** `["010","011","101","110","111"]`
>
> **Explanation:**
>
> The valid strings of length 3 are: `"010"`, `"011"`, `"101"`, `"110"`, and `"111"`.

Example 2:

> **Input:** `n = 1`
>
> **Output:** `["0","1"]`
>
> **Explanation:**
>
> The valid strings of length 1 are: `"0"` and `"1"`.

</> Code

Java | Auto | Format

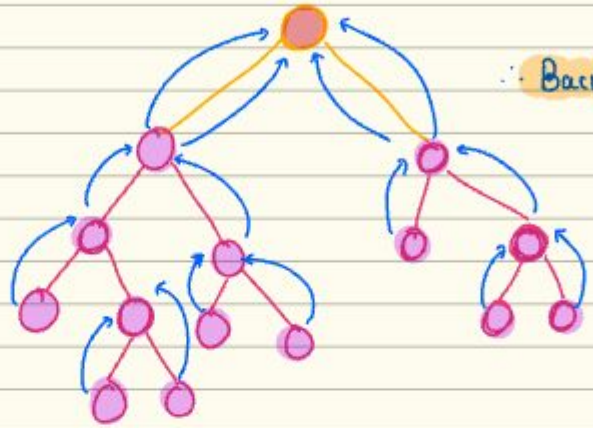
```java
class Solution {
    static void func(int n,String str,List<String> arr,int last){
        if(n==0){
            arr.add(str);
            return;
        }
        func(n-1,str+"1",arr,1);
        if(last==1){
            func(n-1,str+"0",arr,0);
        }
    }
    public List<String> validStrings(int n) {
        List<String> arr=new ArrayList<>();
        func(n,"",arr,1);
        return arr;
    }
}
```

Help in coding › Saved Ln 17, Col 2

MODULE 6:- Backtracking.

Video 1 → Backtracking problems.

→ Backtracking:- Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, that incrementally builds candidates to the solutions. As soon as it determines that a candidate cannot possibly lead to a valid complete solution, it abandons this partial candidate and "backtracks" (return to the upper level) and reset to the upper level's state so that the search process can continue to explore the next branch.



∴ Backtracking.

→ Problem 1

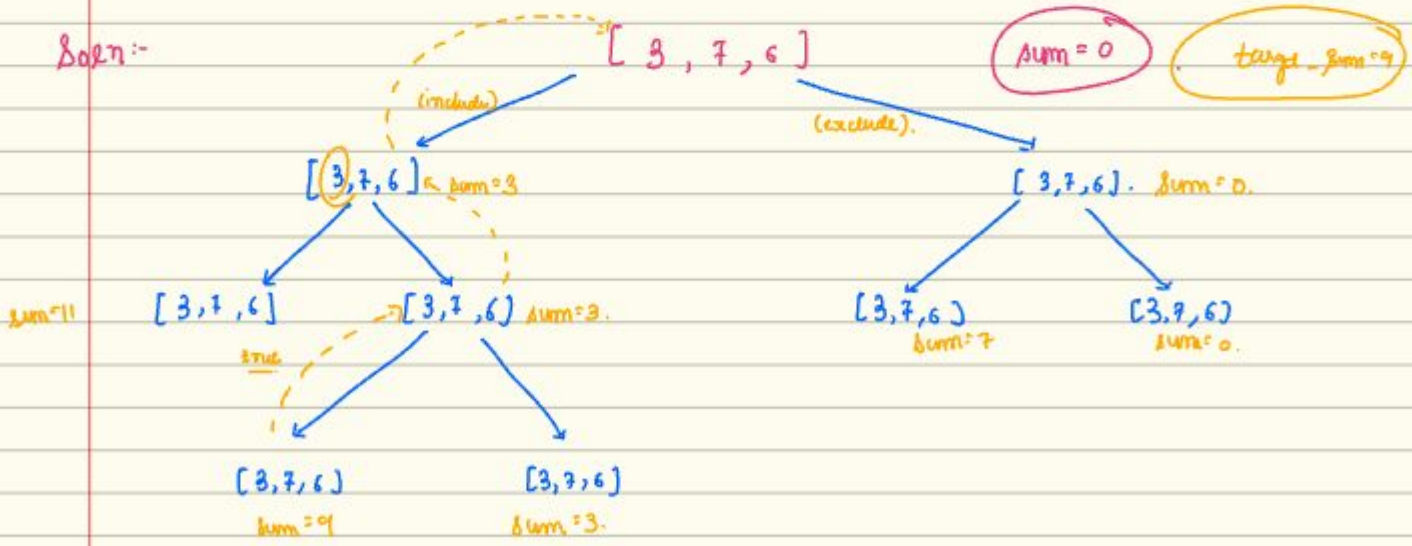
Given an array arr[] of non-negative integers and a value sum, the task is to check if there is a subset of the given array whose sum is equal to given target_sum. If such subset return that subset else return an empty set.

Ex:-

Input:- arr[] = {3, 34, 4, 12, 5, 2}, target_sum = 9

Output:- {4, 5}

Soln:-



∴ Time Complexity ↓

Similar T.C. in case of simple recursion.

→ Problem 2:-

4) Word Search

79. Word Search

Solved

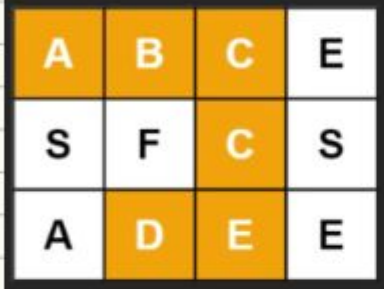
Discuss Approach ›

Medium | Topics | Companies

Given an `n x n` grid of characters `board` and a string `word`, return `true` *if* `word` *exists in the grid.*

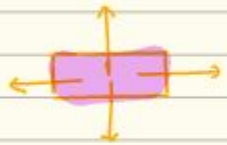
The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:



Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
Output: true

directions :-



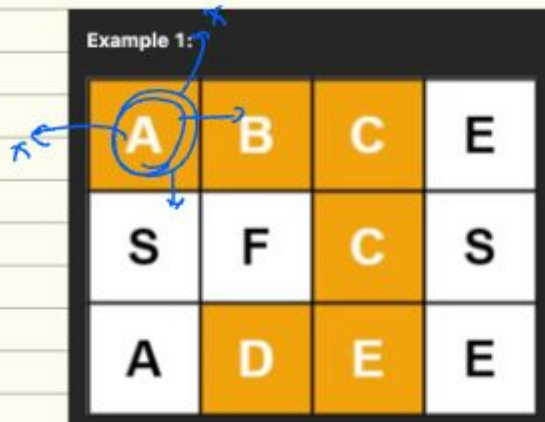
```java

    // Recursive function to search for the word starting from a given position in the maze
    public static boolean search(char[][] maze, String word, int row, int col, int idx) {
        // Base case: If the entire word has been found, return true
        if (idx == word.length()) {
            return true;
        }

        // Check for out-of-bounds or mismatched characters
        if (row < 0 || col < 0 || row >= maze.length || col >= maze[0].length || maze[row][col] != word.charAt(idx)) {
            return false;
        }

        // Mark the current cell as visited
        maze[row][col] = '*';

        // Define the possible directions to move in the maze
        int[] r = { -1, 1, 0, 0 };
        int[] c = { 0, 0, -1, 1 };

        // Recursively search in all four directions from the current cell
        for (int i = 0; i < c.length; i++) {
            boolean ans = search(maze, word, row + r[i], col + c[i], idx + 1);
            if (ans == true) {
                return ans; // If the word is found, return true
            }
        }

        // Backtrack: Restore the original character in the maze
        maze[row][col] = word.charAt(idx);
        return false; // If the word is not found starting from the current cell, return false
    }
}
```

At "A" we are writing in the current place and the word starts with letter "A" we can the start exploring in.

↓

∴ Start letter on the board is == starting letter of the word.

Example 1:



Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
Output: true

Things required:- → Current cell (r,c)
→ Current character in the word.

→ Video 2:-

↳ Backtracking Problems 2.

→ Sudoku Solver :-

37. Sudoku Solver

Solved

Discuss Approach ›

Hard | Topics | Companies

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy **all of the following rules**:

1. Each of the digits `1-9` must occur exactly once in each row.
2. Each of the digits `1-9` must occur exactly once in each column.
3. Each of the digits `1-9` must occur exactly once in each of the 9 `3x3` sub-boxes of the grid.

The `'.'` character indicates empty cells.

Example 1:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Input: board = [["5","3",".",".","7",".",".",".","."],
["6",".",".","1","9","5",".",".","."],[".","9","8",".",".",".",".",".","6","."],
["8",".",".",".",".","6",".",".",".","3"],["4",".",".",".","8",".",".","3",".",".",".","1"],
["7",".",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".",".",".","2","8","."],
[".",".",".",".","4","1","9",".",".",".","5"],[".",".",".",".",".",".",".","8",".",".",".","7","9"]]
Output: [["5","3","4","6","7","8","9","1","2"],
["6","7","2","1","9","5","3","4","8"],["1","9","8","3","4","2","5","6","7"],
["8","5","9","7","6","1","4","2","3"],["4","2","6","8","5","3","7","9","1"],
["7","1","3","9","2","4","8","5","6"],["9","6","1","5","3","7","2","8","4"],
["2","8","7","4","1","9","6","3","5"],["3","4","5","2","8","6","1","7","9"]]
Explanation: The input board is shown above and the only valid solution is shown below:

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

COLOUMN ↓

ROW →

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

∴ Every coloumn must contain digits 1-9 only one time (they must not repeat).

∴ Every row must contain digits 1-9 exactly one.

∴ Each of the digits 1-9 must occur exactly once of the 9 boxes (inner-grids).

∵ Filling the rows in order or sequence will be more convinient from randomly filling them.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

→ first blank space to fill the values from 1-9.

↓

Let's try all possible variations.

can I place) :-

1 2 3(×) 4(✓) 5(×) 6(×)

from the current row no. and coloumn no. we can do the following :-

if the current row no. is → 7

then

7 - 7 % 3 = 7(6,0)

∴ Hence, checking for 3*3 grid :-

```
int c1 = 8 - (r%3);
int c2 = c - (c%3);
```

→ Code :-

→ N - Queens :-

51. N-Queens

Solved

Hard · Topics · Companies

The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *all distinct solutions to the* ***n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.

```
class Solution {
    List<List<String>> ans=new ArrayList<>();
    public List<List<String>> solveNQueens(int n) {
        List<String> board = new ArrayList<>();
        for(int i=0;i<n;i++){
            StringBuilder sb=new StringBuilder("");
            for(int j=0;j<n;j++){
                sb.append(".");
            }
            board.add(sb.toString());

        }
        solve(board,0);
        return ans;
    }
```

```
    public void solve(List<String> board,int row){
        if(row==board.size()){
            ans.add(new ArrayList<>(board));
            return;
        }
        for(int i=0;i<board.size();i++){
            if(isValid(board,row,i)){
                StringBuilder newRow=new StringBuilder(board.get(row));
                newRow.setCharAt(i,'Q');
                board.set(row,newRow.toString());
                solve(board,row+1);
                newRow.setCharAt(i,'.');
                board.set(row,newRow.toString());

            }
        }
    }

    public boolean isValid(List<String> board,int row,int col){
        for(int i=row;i>=0;i--){
            if(board.get(i).charAt(col)=='Q'){
                return false;
            }
        }

        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board.get(i).charAt(j) == 'Q')
                return false;
        }
```

```
        for (int i = row, j = col; i >= 0 && j < board.size(); i--, j++) {
            if (board.get(i).charAt(j) == 'Q')
                return false;
        }
        return true;
    }
}
```

→ Video 3 : 1

Backtracking Problems 3

→ Tower of Hanoi

Tower Of Hanoi

Discuss Approach ›

Difficulty: **Medium** Accuracy: **35.23%** Submissions: **176K+** Points: **4**

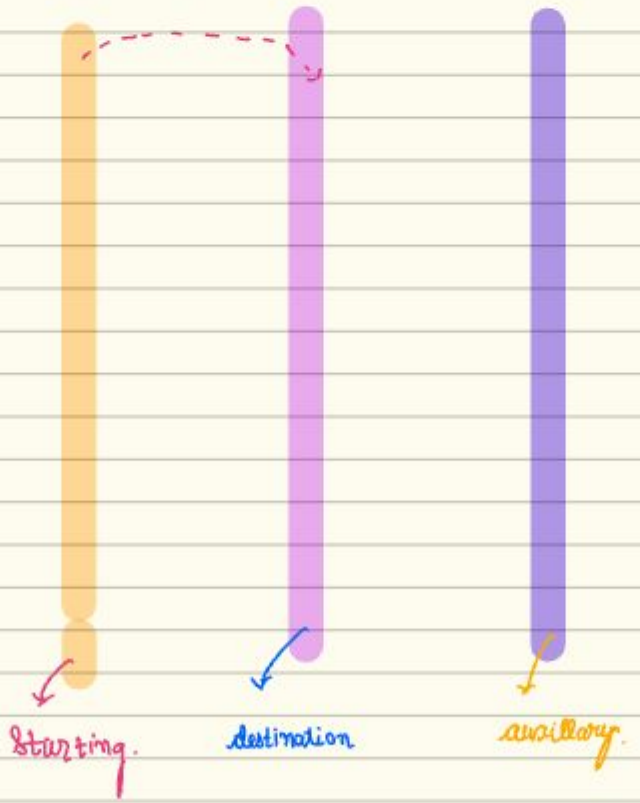
You are given **n** disks placed on a starting rod (from), with the smallest disk on top and the largest at the bottom. There are three rods: the **starting** rod(from), the **target** rod (to), and an **auxiliary** rod (aux).
You have to calculate the total number of **moves** required to transfer all **n** disks from the starting rod to the target rod, following these rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk or on an empty rod.

Return the number of moves needed to complete the task.

Examples:

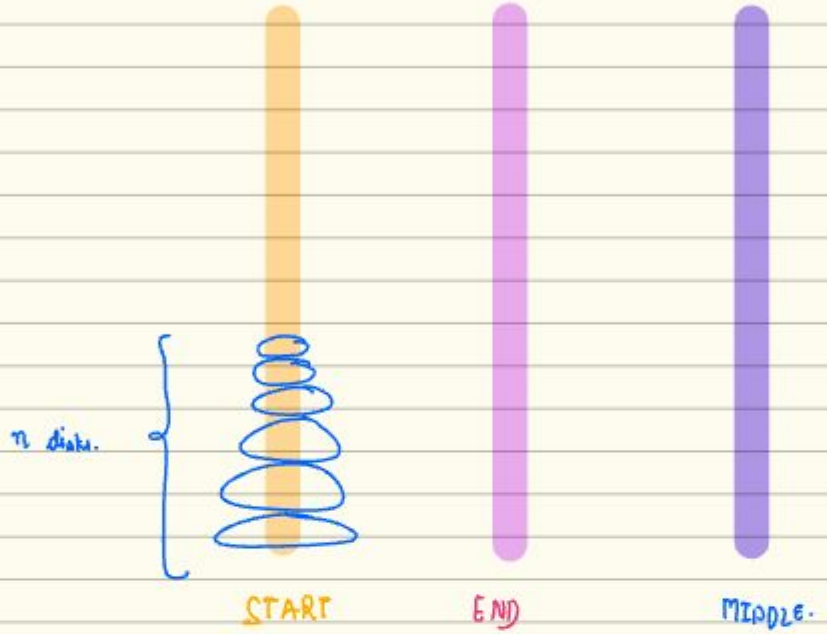
Input: n = 2
Output: 3
Explanation: For n =2 , steps will be as follows in the example and total 3 steps will be taken.
move disk 1 from rod 1 to rod 2
move disk 2 from rod 1 to rod 3
move disk 1 from rod 2 to rod 3

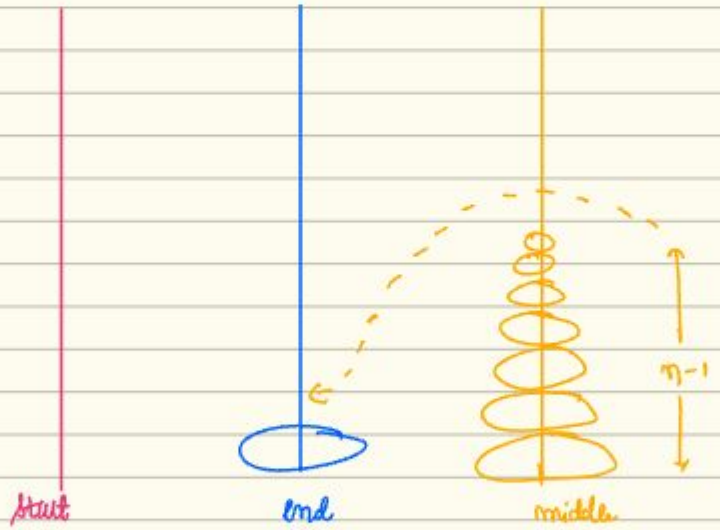


∴ move all disk to the third rod.

→ RULES FOR PLACING DISK:-

- Only 1 disk can be moved at a time.
- A disk can only be placed at the top of a large disk.





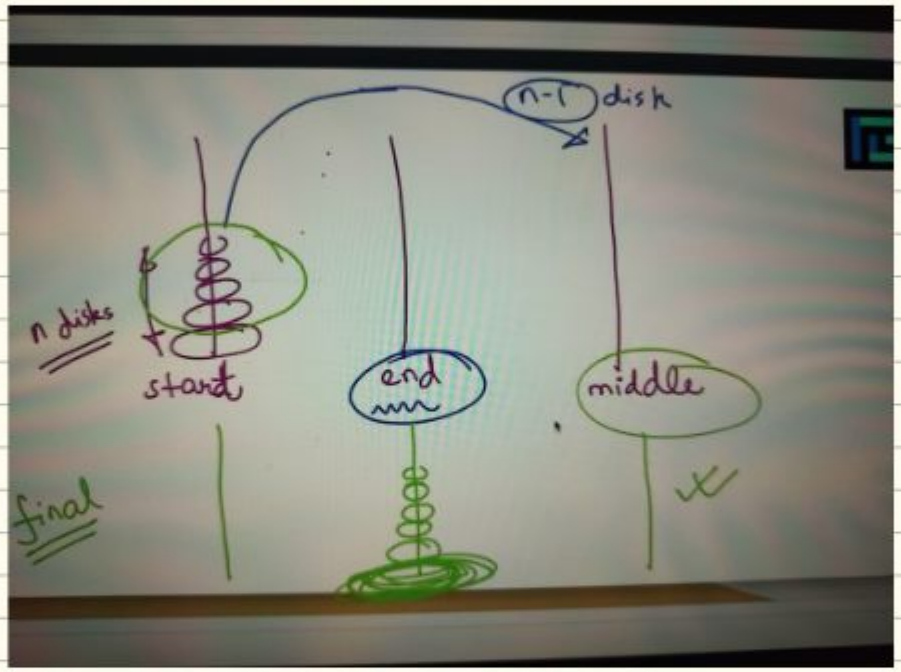

```
Tower of Hanoi(int n, int start, int end, int middle){

    Number of moves required to transfer n disks from start rod to end
    rod using the middle rod.

}
```

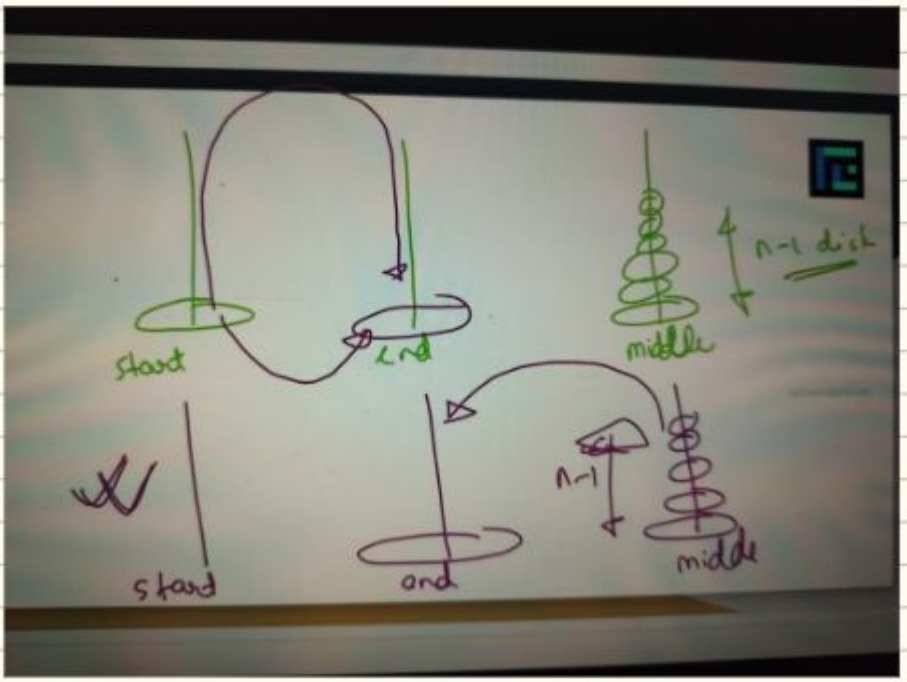
① A $\xrightarrow[\text{using C}]{\text{n-1 disk}}$ B

② 1 disk from :- A → C

③ B $\xrightarrow[\text{using A}]{\text{n-1 disk}}$ C

```
int moves += f(n-1, A, B, C)
moves += 1;
moves += f(n-1, B, C, A)
return moves;
```





Code:-

```java
public class TowerofHanoi {    "Towerof": Unknown word.
    Run | Debug
    public static void main(String[] args) {
        Scanner sc= new Scanner(System.in);
        // main - base function
    }    public int towerOfHanoi(int n, int from, int to, int aux) {

        if(n<=1){
            return n;
        }

        int moves = towerOfHanoi(n-1, from , aux , to);
        moves++;
        moves+= towerOfHanoi(n-1, aux , to, from);
        return moves;


    }
}
```

→ Unique Paths III

980. Unique Paths III

Discuss Approach ›

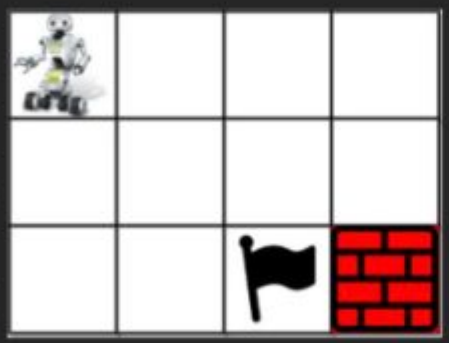
Hard | Topics | Companies

You are given an `m x n` integer array `grid` where `grid[i][j]` could be:

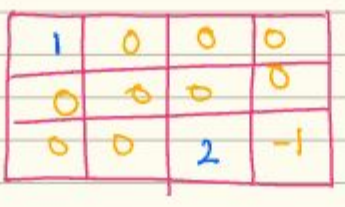
- `1` representing the starting square. There is exactly one starting square.
- `2` representing the ending square. There is exactly one ending square.
- `0` representing empty squares we can walk over.
- `-1` representing obstacles that we cannot walk over.

Return *the number of 4-directional walks from the starting square to the ending square, that walk over every non-obstacle square exactly once.*

Example 1:



Input: grid = [[1,0,0,0],[0,0,0,0],[0,0,2,-1]]
Output: 2
Explanation: We have the following two paths:
1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2)
2. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2)



1 → Start step
2 → End step.
-1 → End/0 obstacle.

→ Code :—

```java
class Solution {
    int emptyCells = 0;  // total non-obstacle cells
    int paths = 0;       // number of valid paths

    public int uniquePathsIII(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;

        int st_row = 0, st_col = 0;

        // Count empty cells and find start position
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] != -1) {
                    emptyCells++;
                }
                if (grid[i][j] == 1) {
                    st_row = i;
                    st_col = j;
                }
            }
        }

        boolean[][] vis = new boolean[m][n];
        helper(grid, st_row, st_col, m, n, vis, 1);

        return paths;
    }
```

```java
    public void helper(int[][] grid, int i, int j, int m, int n, boolean[][] vis, int
count) {
        // Base case: out of bounds or obstacle or already visited
        if (i < 0 || j < 0 || i >= m || j >= n || grid[i][j] == -1 || vis[i][j]) {
            return;
        }

        // If we reach the destination (grid[i][j] == 2)
        if (grid[i][j] == 2) {
            if (count == emptyCells) {
                paths++;
            }
            return;
        }

        // Mark as visited
        vis[i][j] = true;

        // Explore in all 4 directions
        helper(grid, i - 1, j, m, n, vis, count + 1); // up
        helper(grid, i + 1, j, m, n, vis, count + 1); // down
        helper(grid, i, j - 1, m, n, vis, count + 1); // left
        helper(grid, i, j + 1, m, n, vis, count + 1); // right

        // Backtrack (unmark visited)
        vis[i][j] = false;
    }
}
```

→ Video 5

↳ Backtracking Problems 4.

→ Problem 1:-

Print all possible decodings of a given digit sequence

Print all Possible Decodings of a given Digit Sequence

Last Updated : 21 Aug, 2025

Given the numeric string **str**, where 1 represents **'a'**, 2 represents **'b'**, ..., 26 represents **'z'**, the task is to print all possible alphabetical strings that can be obtained from **str**.

Examples:

Input: *str = "1123"*
Output:
aabc
kbc
alc
aaw
kw
Explanation:
The given string can be splitted as:
1) "1123" = "1" + "1" + "2" + "3" = aabc
2) "1123" = "11" + "2" + "3" = kbc
3) "1123" = "1" + "12" + "3" = alc
4) "1123" = "1" + "1" + "23" = aaw
5) "1123" = "11" + "23" = kw

Input: *str = "56"*
Output:
ef
Explanation:
The given string can be splitted as:
1) "56" = "5" + "6" = ef

Approach: It can be observed that every single character represents an alphabet apart from **0**. This problem is recursive and can be broken into sub-problems. The terminating condition will be when the passed string is empty. Below are the steps to solve the problem:

1. Create a helper function **getChar()** that returns the corresponding alphabet of the given numeric character.
2. Create a recursive function that takes the input as a string and returns an array of the desired answer of every character extracted.
3. The base case is when the **input string is empty**. Return an array of length one containing an empty string for this case.
4. Extract every single character by using the helper function and append it to the empty string first and store it in an array, say **output1**.
5. At the same time check if the length of the characters is greater than or equal to two and also check if the two characters extracted lies in the range of alphabets. Now, store the corresponding character in an array, say **output2**.
6. Then create a final output array whose length will be the sum of the length of **output1** and **output2** and store their answers and return it.
7. Repeat the above steps for every single or pair of adjacent characters extracted. Now, return the final array obtained.
8. Traverse the array and for each string, generate corresponding strings of lowercase alphabets and print them.

→ Approach :-

Ex :- "1 1 2 3"

a a b c
k b c
a l c
k w
a a w

∴ We have 26 letters in english alphabet. So at most we can consider 2 letters.

Ex :- 1 0 ∴ We must take these both characters, because zero won't return anything.

1 1 2 3

f(idx, string)
idx = 1, string = "a"

Recursion Tree :-

"1 1 2" "" "" (idx=0, "")

(idx=1, "a")
(idx=2, "k")
(idx=1, "aa") (idx=3, "aj")
(idx=3, "kb")
idx=3, "aab"

∴ final formations :-

aab, aj, kb.

SINGLE CHOICE	TWO CHOICES
⇝ idx = n-1 (Single char)	∴ When no single choice, then we have 2 choices.
⇝ curr[idx] + curr[idx+1] > 26 (Single)	Single char / double.
Ex :- 279	
⇝ curr[idx+1] = 0 (Two Char)	

→ Code:-

```java
public class PrintAllDecodings {

    // helper to map numbers to letters
    public static char getCorresponding(int n) {
        return (char)(n + 'a' - 1);  // 1 -> 'a', 2 -> 'b', ..., 26 -> 'z'
    }

    // recursive function
    public static void decodings(String str, int idx, String current) {
        // base case: reached end of string
        if (idx == str.length()) {
            System.out.println(current);
            return;
        }

        // take one digit
        int oneDigit = str.charAt(idx) - '0';
        if (oneDigit >= 1 && oneDigit <= 9) {
            decodings(str, idx + 1, current + getCorresponding(oneDigit));
        }

        // take two digits (if possible)
        if (idx + 1 < str.length()) {
            int twoDigits = (str.charAt(idx) - '0') * 10 + (str.charAt(idx + 1) - '0');
            if (twoDigits >= 10 && twoDigits <= 26) {
                decodings(str, idx + 2, current + getCorresponding(twoDigits));
            }
        }
    }

    Run | Debug
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);    Resource leak: 'sc' is never closed
        String str = sc.next();
        decodings(str, idx:0, current:"");
    }
}
```

→ Problem 2

↳ Count possible decodings of a given digit sequence Set - II

Ex:- "1*"

1 — 9

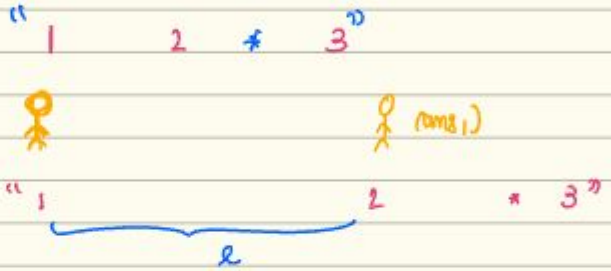
(9) a → 9.
b
c
d
e
f
:

∴ 9+9 = 18

Ex:- Possible strings from 1*

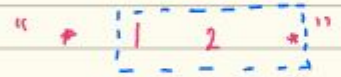
1	1	→	"aa", "k" , 2 possible string outputs
1	2	→	2
1	3	→	2
1	4	→	2
1	5	→	2
1	6	→	2
1	7	→	2
1	8	→	2
1	9	→	2

∴ TOTAL STRING := 18



idx = 0

$\Sigma ans = \because ans_1 + ans_2$



∴ ans += 9 x ans1

∴ ans += 16 x ans2

→ Code:-

```cpp
int helper(int idx, string &num) {
    int n = num.size();
    if (idx == n) {
        return 1;
    }
    //single choice
    if (idx == n - 1) {
        if (num[idx] == '*') {
            return 9;
        }
        else {
            return 1;
        }
    }
    if (num[idx] == '*') {
        //**
        if (num[idx + 1] == '*') {
            //single char
            int ans = 0;
            ans += helper(idx + 1, num) * 9;
            //double char
            ans += helper(idx + 2, num) * 16;
            return ans;
        }
        else {
            //single char
            int ans = 0;
            ans += helper(idx + 1, num) * 9;
            //double char
            int temp = (num[idx + 1] - '0');
            if (temp <= 6) {
                ans += helper(idx + 2, num) * 2;
            }
            else {
                ans += helper(idx + 2, num) * 1;
            }
            return ans;
        }
    }
    else {
        // []*
        if (num[idx + 1] == '*') {
            int ans = 0;
            //single char
            if (num[idx] != '0') {
                ans += helper(idx + 1, num) * 1;
            }
            //double char
            int temp = (num[idx] - '0');
            if (temp <= 1) {
                ans += helper(idx + 2, num) * 9;
            }
            else if (temp == 2) {
                ans += helper(idx + 2, num) * 6;
            }
            return ans;
        }
        // [][]
        else {
            //single char
            int ans = 0;
            if (num[idx] != '0') {
                ans += helper(idx + 1, num) * 1;
            }
            // double char
            int temp = (num[idx] - '0') * 10 + (num[idx + 1] - '0');
            if (temp <= 26) {
                ans += helper(idx + 2, num) * 1;
            }
            return ans;
        }
    }
}
int solve()
{
    string s;
    cin >> s;
    int ans = helper(0, s);
    cout << ans << endl;
}
```

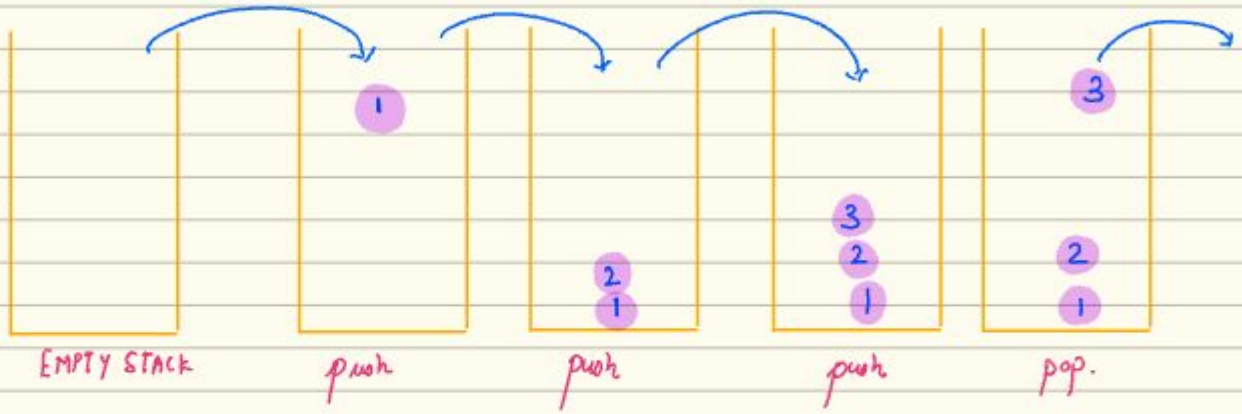
→ MODULE 9:- Stacks, Queues, Dequeue.

↳ Video 1:-

↳ Stacks, Queues, Dequeue.

→ Stacks:-

A stack is a container that stores elements in a last-in first-out (LIFO) order.



⇝ STACKS - Important built-in functions.

(i). push() :- pushes value inside stack.

(ii). pop() :- deletes the top-most value from the stack.

(iii). peek() :- returns top most value from stack.

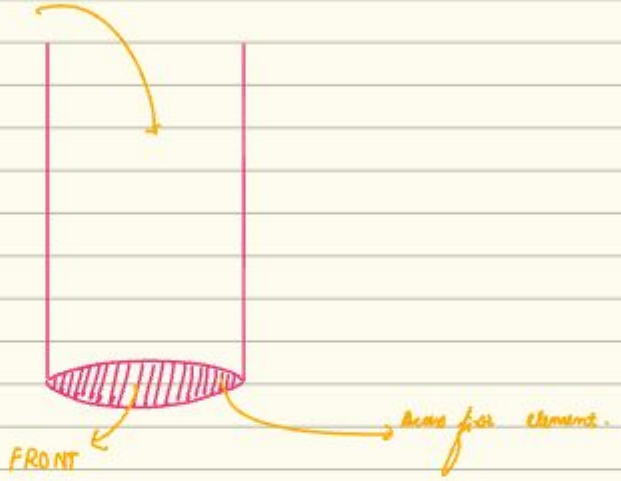
(iv). isEmpty() :- returns true if stack is empty, else returns false.

(v) size() :- returns size of the stack.

NOTE:-

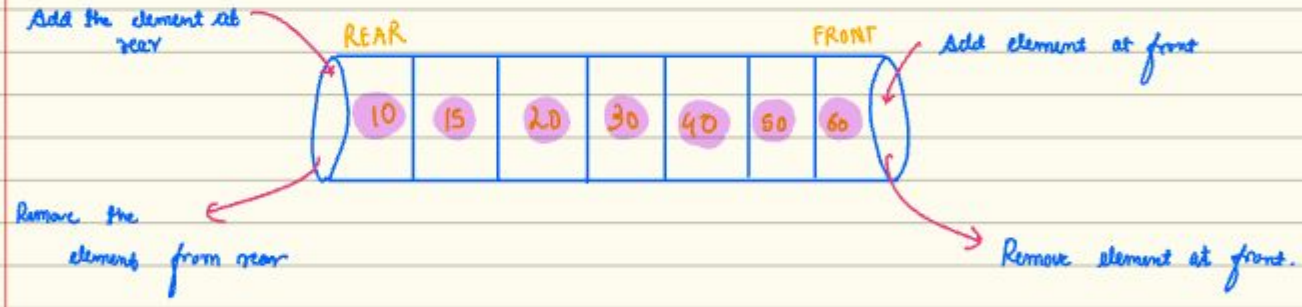
- Time Complexity of all above functions is O(1).
- pop() and peek() will throw exception if stack is empty which leads to run time error.

→ **Queues :-** A queue is a container that push and pops elements in first-in first-out (FIFO) order.



→ **Dequeues :-**

A dequeue is a data structure that allows you to add or remove elements from both ends.



↝ **Built in Deque functions :-**

Syntax :- `Deque<Integer> dq = new ArrayDeque<>();`

↳ ∵ Some are handled by array or linked list.

- **Adding Elements :-**

- `addFirst(E e)` → Inserts element at front
- `addLast(E e)` → Inserts element at end
- `offerFirst(E e)` → Inserts at front (returns `false` if fails instead of throwing exception)
- `offerLast(E e)` → Inserts at end (safe version)

```java
Deque<Integer> dq = new ArrayDeque<>();
dq.addFirst(10);
dq.addLast(20);
dq.offerFirst(5);
dq.offerLast(25);
```

- Removing Elements:-

Removing Elements

- `removeFirst()` → Removes and returns first element
- `removeLast()` → Removes and returns last element
- `pollFirst()` → Removes and returns first element (or `null` if empty)
- `pollLast()` → Removes and returns last element (or `null` if empty)

java — Copy code

```
dq.removeFirst();  // exception if empty
dq.removeLast();
dq.pollFirst();    // null if empty
dq.pollLast();
```

- Accessing Elements (Peek).

Accessing Elements (Peek)

- `getFirst()` → Returns first element (exception if empty)
- `getLast()` → Returns last element (exception if empty)
- `peekFirst()` → Returns first element (null if empty)
- `peekLast()` → Returns last element (null if empty)

java — Copy code

```
System.out.println(dq.getFirst());
System.out.println(dq.peekLast());
```

- Other useful methods:-

Other Useful Methods

- `size()` → Number of elements
- `isEmpty()` → Check if empty
- `contains(Object o)` → Check if element exists
- `iterator()` → Forward iterator
- `descendingIterator()` → Reverse iterator
- `clear()` → Remove all elements

java — Copy code

```
System.out.println(dq.size());
System.out.println(dq.isEmpty());
System.out.println(dq.contains(20));
```

→ Time complexity for above functions is O(1).

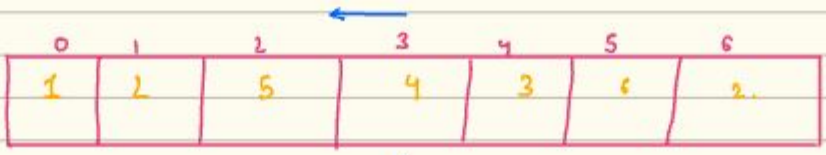
⇝ For example:-

you have:-

Array → $n \approx 10^5$

$q \approx 10^5$

In every, what is the index of the first element in the left of index i that has the value smaller the value present at index i.



By Index 1

By Value -1 1 2 2 2 3 1

∵ Generally, for q queries:-
the TC. would be:-

$O(n+q)$

∴ But using monotonic stack, we can answer this query in O(1) TC. and forming the stack in O(N)

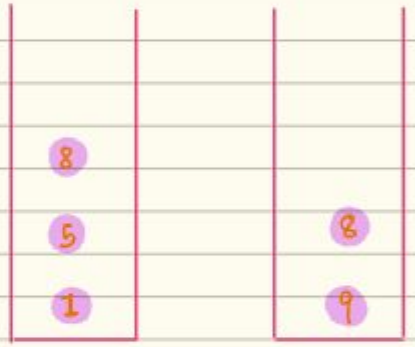
→ Monotonic Stack:-

A monotonic stack is a common data structure in computer science that maintains its elements in specific order. Unlike traditional stacks, are arranged in a specific order.

Unlike traditional stacks, monotonic stacks ensure that elements inside the stacks are arranged in an increasing or decreasing order based on their arrival time.

ARRAY:- 1 7 9 5 8

MONOTONIC STACK :-



Monotonic Increasing stack.

Monotonic decreasing stack.

It is specially used to find the following things in O(1) time Complexity after precomputation time of O(N).

(i). Next smaller Element towards Left/Right.
(ii). Next greater Element towards Left/Right.

Left → Right.

ARRAY:- 2 5 4 3 9 6 2

ANS:- -1 2 2 2 3 3 -1

(NSL)

This answer array will store next smaller element towards the left.

∴ Topmost element of stack stores the answer.

- Topmost element of the stack stores the ans.

(i). As 2 might be the answer for some of the elements, we will push 2 inside the stack.

(ii). Adding 5.

(iii). 5 is of no use to us, pop() 5 out of the queue., because if we start travelling from right to left, the nearest smaller answer will be 4.

- Keep popping the useless elements.

Code:-

```
class NextGreaterElement {
    public int[] secondGreaterElement(int[] nums) {    "nums": Unknown
        int n = nums.length;    "nums": Unknown word.
        int[] sngr = new int[n];    "sngr": Unknown word.
        Arrays.fill(sngr, -1);    "sngr": Unknown word.

        Stack<Integer> s1 = new Stack<>();
        Stack<Integer> s2 = new Stack<>();

        for (int i = 0; i < n; i++) {
            while (!s2.isEmpty() && nums[i] > nums[s2.peek()]) {    "nu
                sngr[s2.peek()] = nums[i];    "sngr": Unknown word.
                s2.pop();
            }

            Stack<Integer> temp = new Stack<>();
            while (!s1.isEmpty() && nums[i] > nums[s1.peek()]) {    "nu
                temp.push(s1.pop());
            }

            while (!temp.isEmpty()) {
                s2.push(temp.pop());
            }

            s1.push(i);
        }

        return sngr;    "sngr": Unknown word.
    }
}
```

→ Video 2

↳ Heaps.

Heap :-

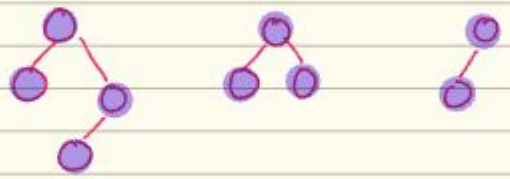
(i). A heap a complete binary tree data structure that satisfies the heap order property.

(ii) Heap Order :-

(i) Max Heap :- Parent node is greater than or equal to its children.

(ii). Min Heap :- Parent node is smaller than or equal to its children.

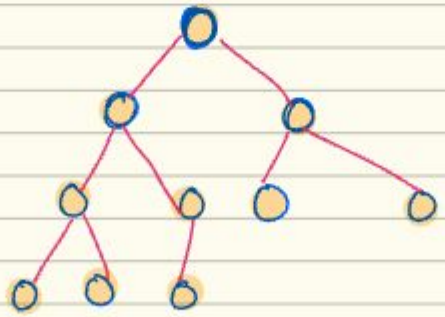
Binary Tree :- If every parent node has at most 2 children



Complete Binary Tree :-

All levels except possibly the last are completely filled.
That means every level has the maximum number of nodes it can hold.

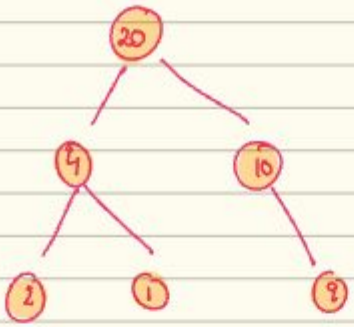
In the last level, all nodes are as far left as possible.
So, if the last level is not completely filled, the missing nodes appear only on the right side.



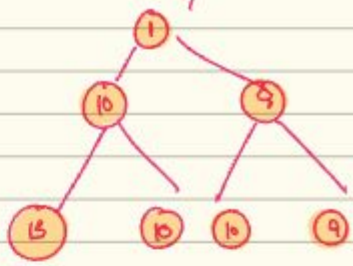
Complete Binary Tree :- Heap order property

↙ max ↘ min

max - Heap:-



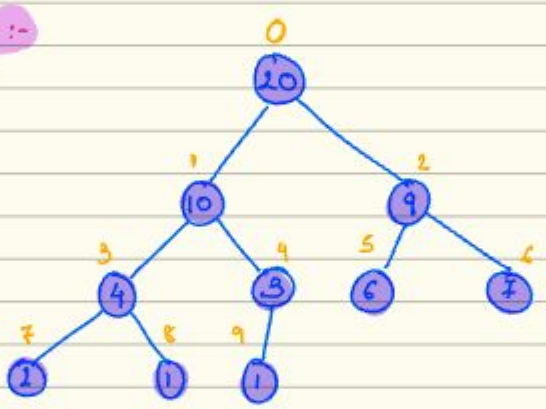
min - Heap.



→ Heap Representation :-

- Stored as an array to optimize space.
- Indexing :-
  - Parent at index i
  - Left child at $2*i+1$
  - Right child at $2*i+2$

MAX HEAP :-

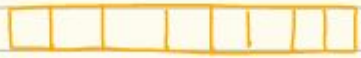


index :-
parent → i
left → $2*i+1$
right → $2*i+2$.

∵ i → index of parent
= $(i-1)/2$.

Array :- 20 10 9 4 3 6 7 2 1 1

Given an array :-



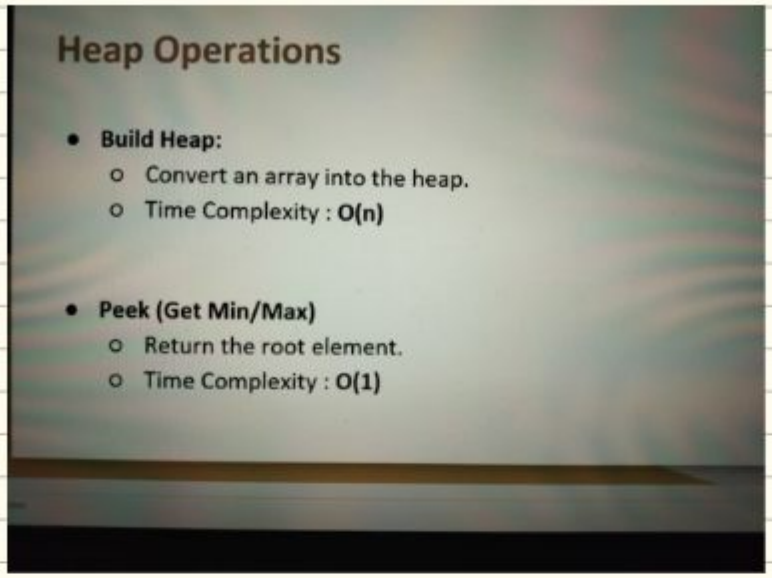
(i). From the given array, build a max heap.

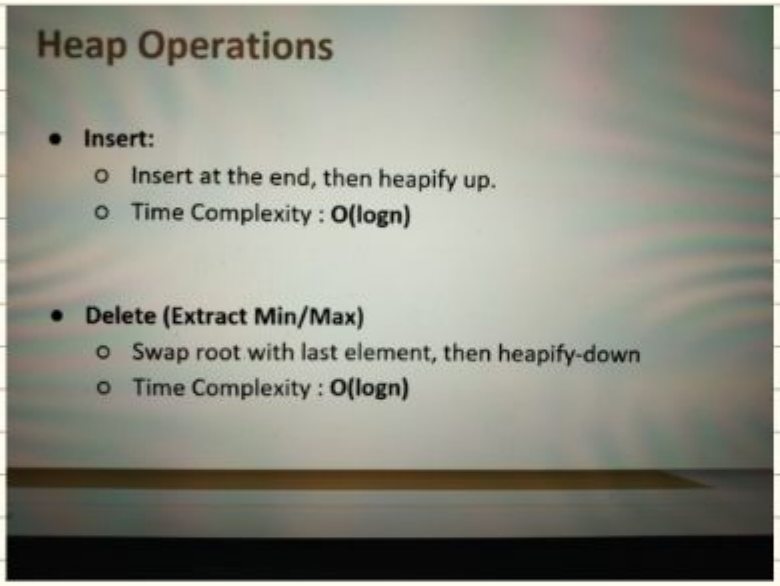
(ii). Give a function to give the maximum Element

(iii). fn. Insert element in heap.

(iv). Delete the top most element from the heap.

→ Heap Operations :-

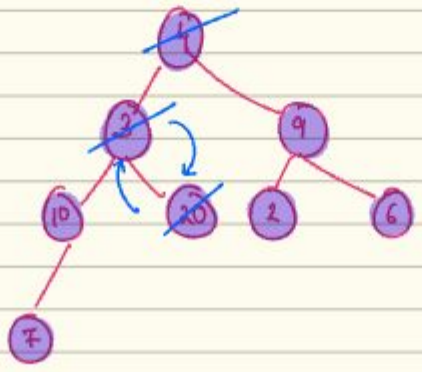




Array :- 4 3 9 10 20 2 6 7

Operation :-

(i). Build max Heap / Rearrange elements of array such that this array symbolizes max Heap.



functions :-

Heapify - up :- function that swaps the elements up until it reaches its correct position

TC :- O(log N)

Heapify - down :- function that swaps the elements down until it reaches its correct position

TC :- O(log N).

max Heap.

Heapify-up → 20.

∵ swapping incorrect elements.

1	2^0
2	2^1
4	2^2
8	2^3
h = 16	2^4
	⋮
	2^{h-1}

$1 + 2^1 + 2^2 + \dots\dots\dots + 2^{h-1} = n.$

$2^h = n+1$

$h = \log_2 (n+1)$

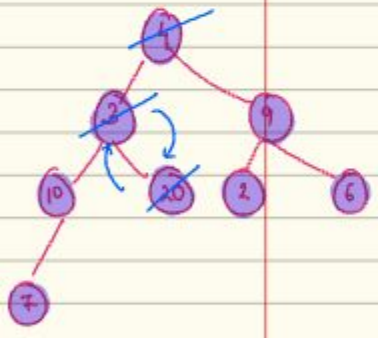
$h = O(\log N).$

$1(2^h - 1) = n$

∵ Leaf nodes are nodes that don't have any children.

20 4 3 9 10 2 20 2 6 7 8

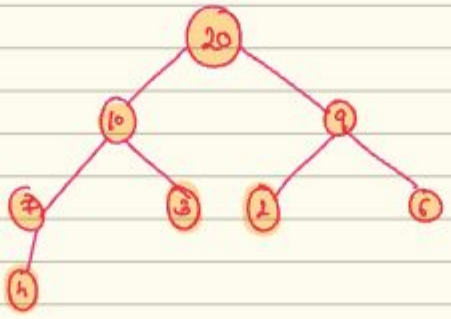
These are the leaf nodes.



STEPS:-

- Traverse from last non-leaf node → 0, and heapify every node.

∴ Hence now the final structure will look like



∴ Index of non leaf node → $\boxed{n/2 - 1}$

n is the size of the array.

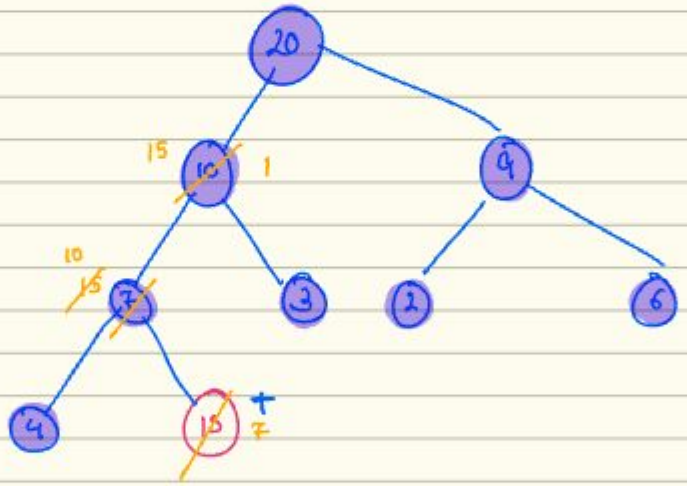
∴ Index of non leaf node = Parent of last node = $\left(\frac{n-1-1}{2}\right) = \left(\frac{n-2}{2}\right) = \left(\frac{n}{2} - 1\right)$

↝ MAX ELEMENT from array, MAX-HEAP → arr[0].

Maximum element → arr[0].

↝ INSERT AN ELEMENT ~ MAX HEAP.

∴ we add the element in the last of tree and then we heapify up it and the make it place in its correct position.



↝ DELETE AN ELEMENT ~ we can only delete Top-most element.

Algorithm that we follow:-

- Swap this top most element with the last element.

4 20 10 9 7 3 2 6 4 20

- and the erase the last element.

- Problem:- the only problem is that 4 is not in it's correct position.

- So, we will heapify down (4), and hence, the order is maintained.

→ Code :-

```
Void heapify_down (int idx){

    int n = heap.size();
    int left = 2*idx+1;
    int right = 2*idx+2;
    int largest = idx;
    int val = heap[idx];

    if (left < n && heap[left] > value]{
            value = heap[left]
            largest = left;
    }

    if (right<n && heap[right] > value){
            value = heap[right];
            largest = right;
        }
    if (largest == idx) return;
     swap( heap[idx], heap[largest]);
    heapify_down (largest);
    }
```

```
void heapify_down (int idx){

        if ( idx == 0) return;
        int parent = (idx -1) /2;

    if (heap[idx] > heap [parent]){

        swap( heap[idx] , heap[ parent]);
            heapify_up= (parent);
        }
    }
```

(∵ Compare value of the elements with its parents, if it is greater than swap).

→ Reaching the idx of top most element.

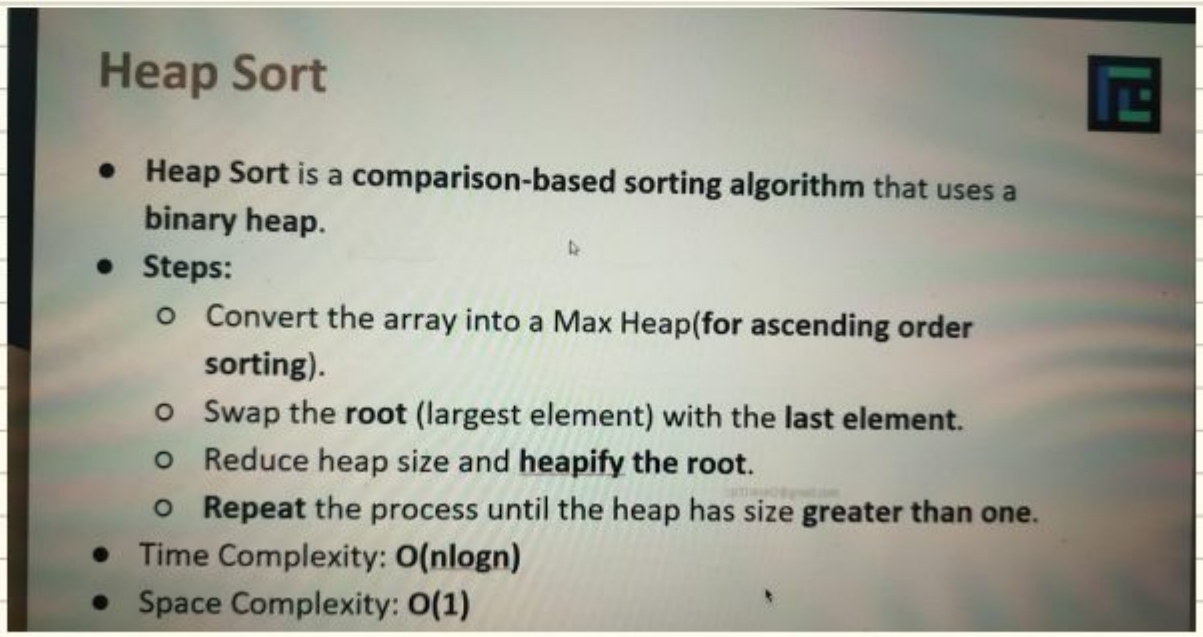
→ Getting parent's idx from our idx

```
void Build (){
        int n = heap.size();

    for(int i= n/2-1 ; i>=0, i--){
            heapify_down(i);
        }
    }
```

→ Traversing from last non leaf node to first node

```
Void get_max(){
    if(heap.size()==0){
        return -1;
    }else{
        return heap[0];
    }
}

Void insert(int val){
    heap.add(val);
    heapify_up(heap.size()-1);
}

void delete(){
    int n= heap.size();
    swap(heap[0], heap[n-1]);
    heap.remove(arr.size()-1);
    if(heap.size()==0) return;
    heapify_down(0);
}
```

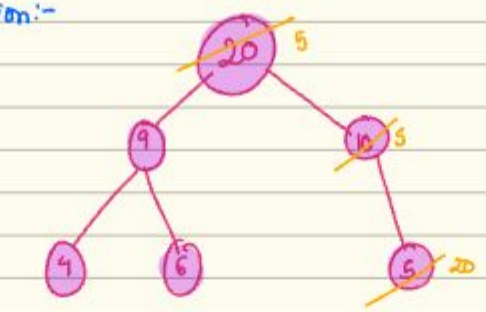
~ Heap Sort :-



→ Example:-

10 20 9 10 5 4 6 5 20

After 1st operation:-



heapify - down.

10 9 9 6 5 10 9 6 20

→ Video 3

↳ Priority Queue & Problem Solving :-

→ Inbuilt Priority Queue :-

(i). A priority queue is a type of container adaptor, specifically designed such that the first element of the queue is either the greatest or the smallest of all elements in the queue.

MAX - HEAP

↓

top → max^m element

3 6 9

↓

returns 9

MIN - HEAP (default → min in JAVA)

↓

top → min^m element

↓

3 6 9

↓

returns 3.

→ Syntax for Priority Queue :-

(i). MIN - PRIORITY QUEUE :-

```
PriorityQueue<Integer> minPQ = new PriorityQueue<>();
```

(ii). MAX - PRIORITY QUEUE :-

```
PriorityQueue<Integer> maxPq = new PriorityQueue<>(Collections.reverseOrder());
```

→ Inbuilt functions of pq. in JAVA:-

1. Add elements

java

```java
pq.add(x);      // inserts element, throws exception if fails
pq.offer(x);    // inserts element, returns false if fails
```

2. Access top element (smallest/largest depending on heap)

java

```java
pq.peek();      // returns head (but does NOT remove it), null if empty
```

3. Remove top element

java

```java
pq.poll();      // returns and removes head, null if empty
pq.remove();    // returns and removes head, throws exception if empty
```

Size & Check

java

```java
pq.size();      // number of elements
pq.isEmpty();   // true if empty
pq.contains(x); // check if an element exists
```

Remove Specific Element

java

```java
pq.remove(x);   // removes a specific element (not just top)
```

Iterate (not sorted order!)

java

```java
for (int val : pq) {
    System.out.print(val + " ");
}
```

Clear

java

```java
pq.clear();     // removes all elements
```

→ Problem:-

↳ k^{th} largest element in an array:-

215. Kth Largest Element in an Array

Solved

Discuss Approach

Medium · Topics · Companies

Given an integer array `nums` and an integer `k`, return the k^{th} largest element in the array.

Note that it is the k^{th} largest element in the sorted order, not the k^{th} distinct element.

Can you solve it without sorting?

Example 1:

```
Input: nums = [3,2,1,5,6,4], k = 2
Output: 5
```

Example 2:

```
Input: nums = [3,2,3,1,2,4,5,5,6], k = 4
Output: 4
```

→ Code :-

```java
class Solution {
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());

        for (int num : nums) {
            pq.add(num);
        }

        // Poll k-1 times
        for (int i = 0; i < k - 1; i++) {
            pq.poll();
        }

        return pq.peek();
    }
}
```

→ Problem 2

↳ Longest Valid Parenthesis IV :-

2454. Next Greater Element IV

Discuss Approach ›

Hard | Topics | Companies | Hint

You are given a **0-indexed** array of non-negative integers `nums`. For each integer in `nums`, you must find its respective **second greater** integer.

The **second greater** integer of `nums[i]` is `nums[j]` such that:

- `j > i`
- `nums[j] > nums[i]`
- There exists **exactly one** index `k` such that `nums[k] > nums[i]` and `i < k < j`.

If there is no such `nums[j]`, the second greater integer is considered to be `-1`.

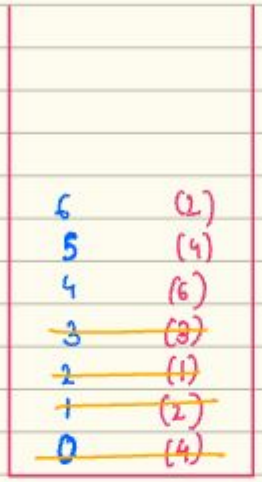
- For example, in the array `[1, 2, 4, 3]`, the second greater integer of `1` is `4`, `2` is `3`, and that of `3` and `4` is `-1`.

Return *an integer array* `answer`, *where* `answer[i]` *is the second greater integer of* `nums[i]`.

Example 1:

```
Input: nums = [2,4,0,9,6]
Output: [9,6,6,-1,-1]
Explanation:
0th index: 4 is the first integer greater than 2, and 9 is the second
integer greater than 2, to the right of 2.
1st index: 9 is the first, and 6 is the second integer greater than 4, to
the right of 4.
2nd index: 9 is the first, and 6 is the second integer greater than 0, to
the right of 0.
3rd index: There is no integer greater than 9 to its right, so the second
greater integer is considered to be -1.
4th index: There is no integer greater than 6 to its right, so the second
greater integer is considered to be -1.
Thus, we return [9,6,6,-1,-1].
```

Index	0	1	2	3	4	5	6
Array	4	2	1	3	6	4	2
Next Greater Element. (filling the array with -1)	~~-1~~ 6	~~-1~~ 3	~~-1~~ 3	~~-1~~ 6	-1	-1	-1



Elements in the stack are in decreasing order.

Instead of storing values, lets store the index

(i) Pushing the first index into the stack.

(ii) As 2 cannot be the answer for 4 (4th index), pushing 2 into the stack.

(iii) The next element greater element to 1 is 3, so we are popping out the previous smaller elements.

(Second next greater element on Right)

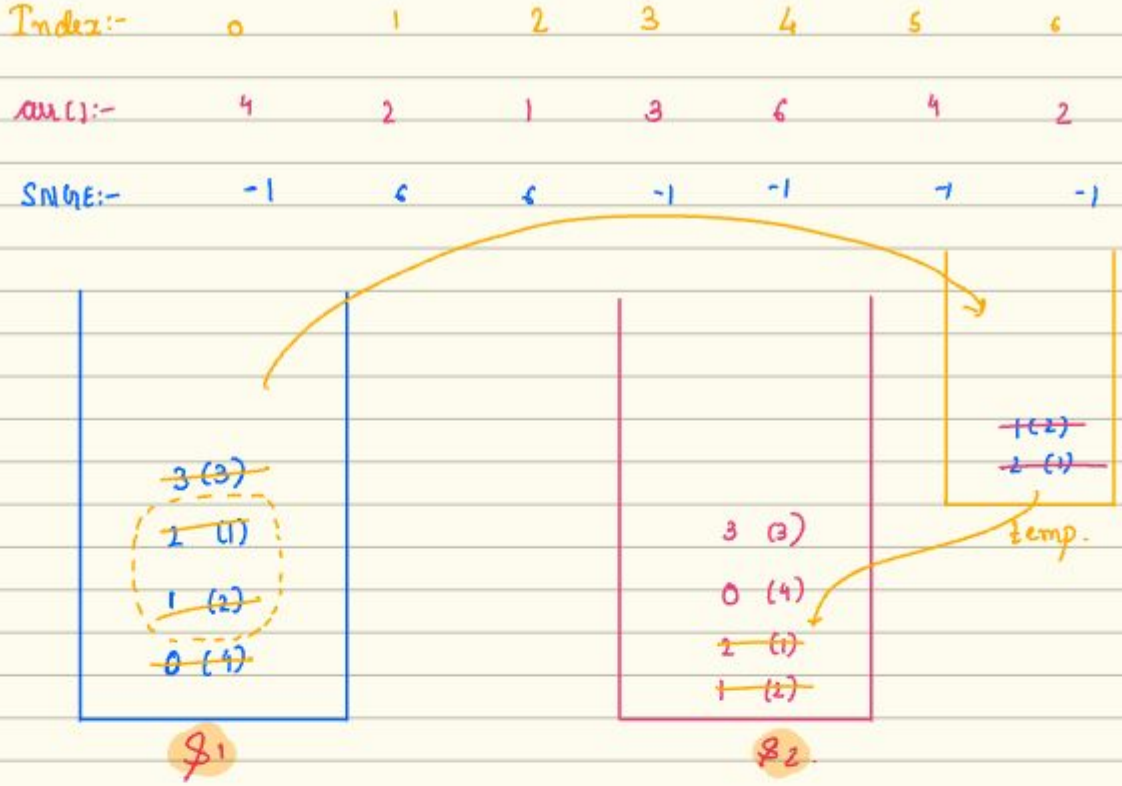
SNGR

S_1

Store the index of elements that has not got its first greater towards right.

S_2

Index of those elements that has got its greater elements towards right and are looking for SNGR.



∵ We are maintaining another temp. stack to maintain the order of the elements.

→ Code :-

```java
class Solution {
    public int[] secondGreaterElement(int[] nums) {
        int n = nums.length;
        int[] nextGreater = new int[n];

        Arrays.fill(nextGreater,-1);
        Stack<Integer> s1 = new Stack<>(),s2 = new Stack<>(),temp = new Stack<>();

        for(int i=0; i<n ;i++){
            while(!s2.isEmpty() && nums[s2.peek()] < nums[i]){
                nextGreater[s2.pop()] = nums[i];
            }
            while(!s1.isEmpty() && nums[s1.peek()] < nums[i]){
                temp.push(s1.pop());
            }
            while(!temp.isEmpty()){
                s2.push(temp.pop());
            }
            s1.push(i);


        }

        return nextGreater;
    }
}
```

→ Problem 2 :-

↳ Maximal Rectangle :-

85. Maximal Rectangle

Solved

Discuss Approach ›

Hard | Topics | Companies

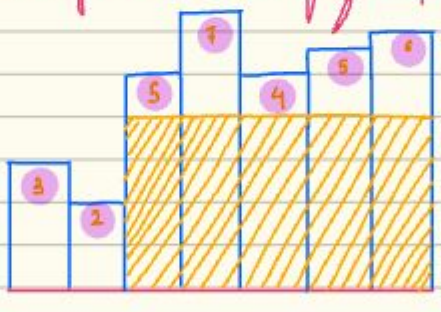
Given a `rows x cols` binary `matrix` filled with `0`'s and `1`'s, find the largest rectangle containing only `1`'s and return *its area*.

Example 1:

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

```
Input: matrix = [["1","0","1","0","0"],["1","0","1","1","1"],
["1","1","1","1","1"],["1","0","0","1","0"]]
Output: 6
Explanation: The maximal rectangle is shown in the above picture.
```

What is the largest rectangle that can be performed?



∴ Rectangle is going to contain this tower completely