**Intro:**

*1.Binary Number Conversion(Decimal to Binary and Binary to Decimal)*

*2.1s complement and 2s complement*

*3.Operators(AND,OR,XOR,NOT,SHIFT)*

*4.Overflow*

**Basics:**

**1.Swap 2 numbers using xor**

The XOR-based method for swapping two numbers without using a temporary variable is clever and efficient. Here's the intuition behind it:

**Key Concept:**

The XOR operation has a unique property: If you XOR a number with itself, the result is 0, and if you XOR a number with 0, it remains unchanged. This property is key to how this method works.

**Steps:**

Let the two numbers be a and b. Initially, they have some values:

1. a = a ^ b:
   * Here, a now holds the XOR of the two numbers. Think of it as a "combined" value.
2. b = a ^ b:
   * Since a now stores a ^ b, XORing it again with b retrieves the original value of a:
     + (a ^ b) ^ b = a (because b ^ b = 0 and a ^ 0 = a).
   * Now, b holds the original value of a.
3. a = a ^ b:
   * At this point, b holds the original value of a, and a holds the XOR of the original numbers (a ^ b). XORing a again with b retrieves the original value of b:
     + (a ^ b) ^ a = b (for similar reasons as above).

After these steps, the two numbers are successfully swapped!

static List<Integer> get(int a,int b)  
{  
 List<Integer> ans = new ArrayList<>();  
 a = a^b;  
 b = a^b;  
 a = a^b;  
 ans.add(a);  
 ans.add(b);  
 return ans;  
}

 **Time complexity:** O(1)

 **Space complexity:** O(1)

**2.Check if the ith bit is set or not.**

1. **Right Shift Operation (**n >> k**)**:
   * This operation shifts the bits of n to the right by k positions.
   * Effectively, this moves the kk-th bit of n to the least significant bit (LSB) position.
   * All other bits to the right of the kk-th bit are discarded.

Example:

* + If n = 10 (binary: 1010) and k = 1:
    - n >> k = 1010 >> 1 = 0101 (binary representation after the shift).

1. **Bitwise AND Operation (**ans & 1**)**:
   * The result of the right shift operation is stored in ans.
   * The bitwise AND operation (ans & 1) checks whether the least significant bit (LSB) of ans is 1 or 0.
   * If the LSB is 1, it means the kk-th bit of the original number n was set (i.e., it was 1).

Example:

* + Continuing with the example where n >> k resulted in 0101:
    - 0101 & 0001 = 0001 (result is 1, indicating that the kk-th bit was set).

1. **Conditional Check (**if ((ans & 1) == 1)**)**:
   * The result of the AND operation is compared to 1.
   * If it equals 1, the function returns true, indicating that the kk-th bit is set.
   * Otherwise, it returns false.Key Points:

* **Right shift (**>>**)** isolates the kk-th bit by discarding the lower kk bits.
* **Bitwise AND (**&**)** checks whether the isolated kk-th bit is 1 or 0.

static boolean checkKthBit(int n, int k) {   
 int ans = n >> k;   
 if((ans&1) == 1)  
 return true;   
 return false;   
}

static boolean checkKthBit(int n, int k) {  
 if((n & 1<<k) != 0)return true;  
 return false;  
}

 **Time complexity:** O(1)

 **Space complexity:** O(1)

**3.Check if a number is even or odd.**

Check the msb of the number , if it is 0 , then even,

If it is 1,then odd.

static boolean isEven(int n) {  
 if((n&1) == 1)return false;  
 return true;  
}

 **Time complexity:** O(1)

 **Space complexity:** O(1)

**4.Set and Unset the ith bit.**

Set – OR with 1.

Unset – AND with 0.

**5.Toggle the ith bit.**

Toggle – xor with 1.

**6.Set and Unset the rightmost bit.**

i) Set the rightmost unset bit

## **Set the Rightmost Unset Bit in Java**

### ****Problem Statement****

Given an integer n, we need to **set the rightmost unset bit** (change the lowest-order 0 to 1), while keeping the rest of the bits unchanged.

public static int setRightmostUnsetBit(int n) {  
 return n | (n + 1);  
}

## **Logic and Intuition**

The **rightmost unset bit** is the first 0 when scanning from the **right (least significant bit) to left** in the binary representation of n.

To set it, we use the **bitwise trick**:

n=n∣(n+1)n = n | (n + 1)

This works because:

1. **Adding 1 (n + 1) flips all bits after the rightmost 0 (including that 0 itself)**:
   * Example: 00101111 → 00110000
2. **Performing OR (|) sets that rightmost 0 to 1 while keeping the rest unchanged**.

## **Example Walkthrough**

Let’s take an example:  
n = 10 (Binary: 1010)

1. n + 1 = 11 → 1011
2. n | (n + 1) = 1010 | 1011
3. Result = 1011 (Decimal: 11)

Thus, the **rightmost unset bit is set**.

## **Code Implementation**

public class SetRightmostUnsetBit {

public static int setRightmostUnsetBit(int n) {

return n | (n + 1);

}

public static void main(String[] args) {

int n = 10; // Binary: 1010

int result = setRightmostUnsetBit(n);

System.out.println("Number after setting rightmost unset bit: " + result);

}

}

### ****Output****

Number after setting rightmost unset bit: 11

## **Why This Works?**

The binary representation of n and n + 1 ensures that only the **rightmost unset bit** is turned on:

### ****Mathematical Proof****

For any n:

1. n in binary: xxxx0111 1111 1111
2. n + 1 in binary: xxxx1000 0000 0000 (flips the rightmost 0 and everything after)
3. n | (n + 1) performs **bitwise OR**, ensuring the 0 turns into 1.

ii) Unset the rightmost set bit

## **Unset the Rightmost Set Bit in Java**

### ****Problem Statement****

Given an integer n, we need to **unset the rightmost set bit** (change it from 1 to 0), keeping the rest of the bits unchanged.

public static int unsetRightmostSetBit(int n) {  
 return n & (n - 1);  
}

## **Intuition and Logic**

A **set bit** is a 1 in the binary representation of a number. The **rightmost set bit** is the lowest-order 1.  
To unset it, we can use a **bitwise trick**:

n=n&(n−1)n = n \& (n - 1)

This operation works because:

1. n - 1 flips all bits **after the rightmost set bit**, including the rightmost set bit itself.
2. Performing n & (n - 1) **removes** only the rightmost 1, while keeping all other bits unchanged.

## **Example Walkthrough**

Let’s take an example:  
n = 10 (Binary: 1010)

1. n - 1 = 9 → 1001
2. n & (n - 1) → 1010 & 1001 = 1000
3. Result = 8 (Binary: 1000)

Thus, the **rightmost set bit is removed**.

### ****Output****

Number after unsetting rightmost set bit: 8

## **Why This Works?**

The binary representation of n and n - 1 ensures that only the **rightmost set bit** is turned off:

### ****Mathematical Proof****

For any n:

1. n in binary is like xxxx1000 (where xxxx represents any bits).
2. n - 1 in binary will be xxxx0111 (all bits after the rightmost set bit flip).
3. Performing n & (n - 1) keeps the xxxx part unchanged but clears the 1000 part.

**Time Complexity :** O(logN)

**Space Complexity:** O(1)

**7.Check if a number is power of 2 or not.**

Here’s a Java implementation to check if a number is a power of 2 using bit manipulation, along with the intuition behind it:

### Intuition

In binary representation, a power of 2 has exactly one bit set to 1. For example:

* 20=12^0 = 1 → 0001
* 21=22^1 = 2 → 0010
* 22=42^2 = 4 → 0100
* 23=82^3 = 8 → 1000

If we subtract 1 from a power of 2, all bits after the only 1 bit will become 1, and the original 1 bit will become 0. For example:

* 4(0100)−1=3(0011)4 (0100) - 1 = 3 (0011)
* 8(1000)−1=7(0111)8 (1000) - 1 = 7 (0111)

By performing a bitwise AND between the number and (number - 1), the result will be 0 for powers of 2.

public static boolean isPowerOfTwo(int num) {  
 // A number is a power of 2 if:  
 // 1. It is positive.  
 // 2. It has only one bit set in its binary representation.  
 return num > 0 && (num & (num - 1)) == 0;  
}

**8.Count the number of set bits.**

Here's the Java code to count the number of set bits (1s) in an integer using the **right shift** (>>) and **bitwise AND** (&) operator, along with an explanation:

### Intuition

To count the set bits using right shift:

1. Use the bitwise AND (&) operator to check the least significant bit (LSB). If the LSB is 1, increment the counter.
2. Use the right shift (>>) operator to move the bits one position to the right, effectively discarding the checked bit.
3. Repeat until the number becomes 0.

For example:

* 1313 → Binary: 1101
* Step 1: Check LSB using num & 1 → 1 (Count = 1), then right shift: 1101 >> 1 = 0110.
* Step 2: Check LSB using num & 1 → 0, then right shift: 0110 >> 1 = 0011.
* Step 3: Check LSB using num & 1 → 1 (Count = 2), then right shift: 0011 >> 1 = 0001.
* Step 4: Check LSB using num & 1 → 1 (Count = 3), then right shift: 0001 >> 1 = 0000.
* public static int countSetBits(int num) {  
   int count = 0;  
    
   while (num > 0) {  
   // Check the least significant bit using AND operator  
   if ((num & 1) == 1) {  
   count++;  
   }  
   // Right shift the number by 1 to discard the checked bit  
   num >>= 1;  
   }  
    
   return count;  
  }

**9.Divide the two integers without multiplication , division and mod operator.**

### ****Logic and Intuition Behind the Code****

This code efficiently **divides two integers** without using multiplication (\*), division (/), or modulo (%) operations. It uses **bitwise shifting** and **subtraction** to achieve division in **O(log N) time complexity**, which is much faster than subtracting one divisor at a time (**O(N)** complexity).

public int divide(int dividend, int divisor) {  
 if(dividend <= Integer.*MIN\_VALUE* && divisor == -1)return Integer.*MAX\_VALUE*;  
 if(divisor == 1)return dividend;  
 if(divisor == -1)return -dividend;  
 if(dividend == divisor)return 1;  
 boolean sign = true;  
 if(dividend>=0 && divisor<0)sign = false;  
 if(dividend<0 && divisor>0)sign = false;  
 long ans = 0;  
 long n = Math.*abs*((long)dividend);  
 long d = Math.*abs*((long)divisor);  
 while(n >= d)  
 {  
 int cnt = 0;  
 while(n >= (d<<(cnt+1)))  
 cnt++;  
 n = n - (d<<cnt);  
 ans += 1<<cnt;  
 }  
  
 if(ans >= Integer.*MAX\_VALUE* && sign == true)  
 {  
 return Integer.*MAX\_VALUE*;  
 }  
  
 if(ans >= Integer.*MAX\_VALUE* && sign == false)  
 {  
 return Integer.*MIN\_VALUE*;  
 }  
  
 if(sign == true)return (int)ans;  
 else return (int)-ans;  
}

## **Step-by-Step Explanation**

### ****1. Handling Edge Cases****

Before performing the actual division, the code first checks for special cases:

if(dividend <= Integer.MIN\_VALUE && divisor == -1) return Integer.MAX\_VALUE;

* If dividend = Integer.MIN\_VALUE (-2^31) and divisor = -1, the result would be 2^31, which is **out of bounds** for an int (Integer.MAX\_VALUE = 2^31 - 1).
* In this case, return Integer.MAX\_VALUE to prevent overflow.

if(divisor == 1) return dividend;

if(divisor == -1) return -dividend;

* If the divisor is 1, return dividend itself.
* If the divisor is -1, return -dividend (unless it's Integer.MIN\_VALUE, which is handled earlier).

if(dividend == divisor) return 1;

* If both numbers are equal, the result is always 1.

### ****2. Determining the Sign of the Result****

boolean sign = true;

if(dividend >= 0 && divisor < 0) sign = false;

if(dividend < 0 && divisor > 0) sign = false;

* If **one number is negative** and the other is **positive**, the result is **negative** (sign = false).
* Otherwise, the result is **positive** (sign = true).

### ****3. Converting to Positive Values****

Since negative numbers in Java have an **asymmetric range** (Integer.MIN\_VALUE has no positive counterpart), we **convert both numbers to their absolute values** using Math.abs((long) dividend).

long n = Math.abs((long) dividend);

long d = Math.abs((long) divisor);

* long is used to **prevent overflow** when converting Integer.MIN\_VALUE.

### ****4. Using Bitwise Left Shift for Efficient Subtraction****

Instead of subtracting d one by one (which would be slow), we **use bitwise left shift (<<) to speed up the subtraction process**.

while(n >= d) {

int cnt = 0;

while(n >= (d << (cnt + 1)))

cnt++;

n = n - (d << cnt);

ans += 1 << cnt;

}

#### **How It Works:**

* We repeatedly subtract the **largest multiple** of d that fits into n.
* We find the **largest power of 2** (2^cnt) such that d \* 2^cnt ≤ n.
* We subtract d \* 2^cnt from n and add 2^cnt to ans.

##### **Example:** 37 ÷ 3

1. d = 3, n = 37
2. Find largest power:
   * 3 × 2^3 = 24 (fits in 37) → cnt = 3
   * Subtract: 37 - 24 = 13
   * Add 2^3 = 8 to ans
3. Repeat with n = 13:
   * 3 × 2^2 = 12 (fits in 13) → cnt = 2
   * Subtract: 13 - 12 = 1
   * Add 2^2 = 4 to ans
4. Now n = 1, which is **less than d**, so we stop.
5. Final result: ans = 8 + 4 = 12.

### ****5. Handling Overflow****

After computing ans, we check for integer overflow:

if(ans >= Integer.MAX\_VALUE && sign == true) {

return Integer.MAX\_VALUE;

}

if(ans >= Integer.MAX\_VALUE && sign == false) {

return Integer.MIN\_VALUE;

}

* If ans exceeds Integer.MAX\_VALUE, we **return the max/min possible integer**.

### ****6. Returning the Final Result****

if(sign == true) return (int) ans;

else return (int) -ans;

* If sign is **true**, return ans as it is.
* If sign is **false**, return -ans.

**Time Complexity :** O(logN)

**Space Complexity:** O(1)

**1.Find xor of numbers from l to r.**

Great question! This is a classic bitwise logic problem that's often asked in coding interviews.

## 🎯 **Goal**

You want to find the **XOR of all integers from 1 to n**, and then from L to R — in **O(1)** time.

## 🔍 Part 1: XOR of all numbers from 1 to n

Let’s observe the pattern of XOR from 1 to n:

n XOR from 1 to n

1 1 -> 1

2 1^2 -> 3

3 1^2^3 -> 0

4 1^2^3^4 -> 4

5 1^2^3^4^5 -> 1

6 1^2^3^4^5^6 -> 7

7 ... -> 0

8 ... -> 8

There’s a pattern based on n % 4:

if n % 4 == 0 → result = n

if n % 4 == 1 → result = 1

if n % 4 == 2 → result = n + 1

if n % 4 == 3 → result = 0

### ✅ Function to compute XOR(1 to n) in O(1):

int xorUpto(int n) {

if (n % 4 == 0) return n;

if (n % 4 == 1) return 1;

if (n % 4 == 2) return n + 1;

return 0; // n % 4 == 3

}

## 🔁 Part 2: XOR of numbers from L to R

You can use this property:

XOR(L to R) = XOR(1 to R) ^ XOR(1 to L-1)

Because XOR is **reversible**:

a ^ a = 0

a ^ 0 = a

So if you have:

XOR(1 to R) = X

XOR(1 to L-1) = Y

Then XOR(L to R) = X ^ Y

## 💡 Final Implementation Logic:

int xorUpto(int n) {

if (n % 4 == 0) return n;

if (n % 4 == 1) return 1;

if (n % 4 == 2) return n + 1;

return 0;

}

int xorRange(int L, int R) {

return xorUpto(R) ^ xorUpto(L - 1);

}

public static int findXOR(int l, int r) {  
 return *xor*(l-1)^*xor*(r);  
}  
  
public static int xor(int n)  
{  
 if(n%4 == 0)return n;  
 else if(n%4 == 1)return 1;  
 else if(n%4 == 2)return n+1;  
 else return 0;  
}

## 🧠 Time Complexity

* Both xorUpto() and xorRange() run in **O(1)** time and **O(1)** space.

**2. Find the two numbers appearing odd number of times.**

## 🎯 **Problem**

You’re given an array where **all numbers occur an even number of times except two numbers**. Your task is to **find those two numbers** that appear an **odd number of times**, and return them in **decreasing order**.

## 🚀 Intuition using Bitwise XOR

### 🔍 Step-by-step logic:

### Step 1: XOR all elements of the array

* XOR of all elements gives you x ^ y, where x and y are the two odd occurring numbers.
* This is because:
  + XOR of a number with itself is 0: a ^ a = 0
  + XOR of a number with 0 is the number itself: a ^ 0 = a
* Since every other number occurs an **even** number of times, they will cancel each other out.
* So, the final XOR gives: xor = x ^ y

### Step 2: Find the rightmost set bit in xor

* xor will definitely be non-zero because x ≠ y.
* The rightmost set bit helps us **separate x and y**.
* Why? Because in the binary representation of x and y, there must be at least one bit where they differ — that’s why x ^ y is non-zero.
* Formula to get the rightmost set bit:
* int setBit = xor & (-xor); // isolates the rightmost set bit

### Step 3: Divide the array into two groups based on the set bit

* Now, we split all numbers in the array into **two groups**:
  + Group 1: Numbers with that particular bit set
  + Group 2: Numbers with that bit not set
* Why does this work?
  + One of x or y will be in group 1, and the other in group 2.
  + All other numbers that occur **even** number of times will still cancel out within their respective groups.

### Step 4: XOR within each group

* Now, do XOR within both groups.
* You'll end up with x in one group and y in another.

### Step 5: Return the two numbers in decreasing order

public int[] twoOddNum(int Arr[], int N)  
{  
 int xorr = 0;  
 for(int i=0;i<N;i++)  
 {  
 xorr = xorr^Arr[i];  
 }  
  
 int rsb = xorr & (-xorr);  
 int b1 = 0;  
 int b2 = 0;  
 for(int i=0;i<N;i++)  
 {  
 if((Arr[i] & rsb)!=0)  
 {  
 b1 = b1^Arr[i];  
 }  
 else  
 {  
 b2 = b2^Arr[i];  
 }  
 }  
  
 if(b1>b2)  
 {  
 return new int[]{b1,b2};  
 }  
 else  
 {  
 return new int[]{b2,b1};  
 }  
}

## 🧠 Time & Space Complexity

* **Time:** O(N)
* **Space:** O(1)

## 🧪 Example Dry Run

**Input:**

Arr = {4, 2, 4, 5, 2, 3, 3, 1}

1. XOR all = 4^2^4^5^2^3^3^1 = 5 ^ 1 = 4
2. Rightmost set bit of 4 = 100 → bit at position 3
3. Divide into two groups:
   * Group 1 (bit set): {4, 4, 5}
   * Group 2 (bit not set): {2, 2, 3, 3, 1}
4. XOR Group 1 = 4^4^5 = 5
5. XOR Group 2 = 2^2^3^3^1 = 1
6. Result = {5, 1}

int setBit = xor & (-xor);

This line **isolates the rightmost set bit** (1-bit) in a number. Here's the intuition:

## 🎯 Goal:

Find the **rightmost set bit** in the number xor — this helps you distinguish between the two unique numbers.

## 🔍 How -xor works

When you do -xor, you're computing the **two's complement** of xor.

**Two’s complement** is calculated as:

-xor = ~xor + 1

Where ~ is bitwise NOT (flip all bits), and +1 just adds 1 to it.

### 🔁 What happens when you do xor & -xor?

Let’s take a binary example:

Suppose:

xor = 12 → 1100 (binary)

Then:

-xor = -12 → two's complement of 1100 = 0100 (binary)

Now:

1100

&0100

-----

0100 → This isolates the \*\*rightmost set bit\*\*

This always works because:

* All bits **after** the rightmost set bit in xor are 0.
* When we do -xor, only the rightmost set bit remains in common.

## 🔥 Why we need the rightmost set bit?

Because it tells us **one bit where the two odd-occurring numbers differ**.

Once we get that bit, we can split the array into two groups:

* Group A: All numbers with that bit **set**
* Group B: All numbers with that bit **not set**

And since the two odd numbers differ at this bit:

* They will go into **separate groups**
* All other numbers (appearing even times) will cancel out in their respective groups

**3. Power Set**

## 🔍 What is a Power Set?

The **power set** of a set is the set of **all possible subsets**, including:

* The **empty set** {}
* The **full set** itself
* All combinations in between

### 🧠 Example:

For input: {a, b},  
The power set is:  
[{}, {a}, {b}, {a, b}]

## ✅ Total Subsets = 2n2^n

If the original set has n elements, the power set will contain 2^n subsets.

Why?  
For each element, we have **two choices**:

* Include it in the subset ✅
* Don’t include it ❌

So, for n elements:  
2 × 2 × ... × 2 = 2^n subsets

### ⚡ Approach 2: ****Bit Manipulation****

#### 🎯 Idea:

Use integers from 0 to 2^n - 1 to represent each subset using bits.

#### ✨ Intuition:

Each bit represents whether the element at that position is **included (1)** or **excluded (0)**.

For input: {a, b, c}  
Subsets represented by:

000 → {}

001 → {c}

010 → {b}

011 → {b, c}

...

111 → {a, b, c}

#### ✅ Code idea:

public List<List<Integer>> subsets(int[] nums) {  
 int subset = (int)Math.*pow*(2,nums.length);  
  
 List<List<Integer>> res = new ArrayList<>();  
 for(int num = 0;num<subset;num++)  
 {  
 List<Integer> ans = new ArrayList<>();  
 for(int i=0;i<nums.length;i++)  
 {  
 if((num & (1<<i)) > 0)  
 {  
 ans.add(nums[i]);  
 }  
 }  
 res.add(ans);  
 }  
 return res;  
}

|  |  |  |  |
| --- | --- | --- | --- |
| Bit Manipulation | Use binary to represent inclusion/exclusion | O(2^n) | O(2^n) |

**Maths**

**1. Print all prime factors of a number**

## 🧠 **Core Intuition:**

Every number can be broken down into **prime factors**. This code finds **each prime factor only once**, no matter how many times it divides the number.

## 🔍 **Line-by-Line Explanation:**

### ```java

List ans = new ArrayList<>(); int n = 780;

- Creates a list `ans` to store the \*\*distinct prime factors\*\*.

- We start with `n = 780`.

---

### ```java

for(int i = 2; i \* i <= n; i++) {

* Loop from 2 up to √n.  
  ✅ Because any factor larger than √n would have a smaller corresponding factor already checked.
* We use i \* i <= n instead of i <= Math.sqrt(n) for performance (no square root calculation).

### ```java

if(n % i == 0) { ans.add(i);

- If `i` divides `n`, then `i` is a factor — and the \*\*first time\*\* we find it, it's a \*\*prime factor\*\*.

- We \*\*add `i` to the list\*\* as a prime factor.

---

### ```java

while(n % i == 0) {

n = n / i;

}

* This **removes all occurrences** of the current factor from n.
* Example: if n = 780 and i = 2, we divide n by 2 repeatedly until it's no longer divisible by 2.
* This step **ensures uniqueness** — we don't add 2 again.

### After the loop:

if(n != 1) ans.add(n);

* After the loop, if n is still greater than 1, it means:
  + The remaining number is a **prime number greater than √original\_n**
  + So we **add it to the list**

✅ This covers cases like:

n = 97 → √97 ≈ 9.8 → loop doesn’t find anything → 97 is prime → added here

### Finally:

for(int i = 0; i < ans.size(); i++) {

System.out.println(ans.get(i));

}

* Just prints all the **distinct prime factors** found.

## 📌 Example with n = 780:

### Prime Factorization:

780 = 2² × 3 × 5 × 13

### Code output:

2

3

5

13

public static void main(String[] args) {  
 List<Integer> ans = new ArrayList<>();  
 int n = 780;  
 for(int i=2;i\*i<=n;i++)  
 {  
 if(n%i == 0)  
 {  
 ans.add(i);  
 while(n%i == 0)  
 {  
 n = n/i;  
 }  
 }  
 }  
 if(n!=1)ans.add(n);  
 for(int i=0;i<ans.size();i++)  
 {  
 System.*out*.println(ans.get(i));  
 }  
}

## 🧠 Time & Space Complexity

* **Time:** O(sqrt(N)\*logN)
* **Space:** O(1)

**2.Power Exponentiation**

public double myPow(double x, int n) {  
 if(x==0)return 0.0;  
 boolean flag = false;  
 if(n<0)  
 {  
 flag = true;  
 n = -n;  
 }  
 double ans = 1.0;  
 while(n>0)  
 {  
 if(n%2 == 1)  
 {  
 ans = ans \* x;  
 n = n - 1;  
 }  
 else  
 {  
 x = x\*x;  
 n = n/2;  
 }  
 }  
 if(flag)  
 {  
 return 1.0/ans;  
 }  
 return ans;  
}

# 🧠 Intuition + Logic:

### Step 1: ****Handle special case:****

if(x == 0) return 0.0;

* If the base x=0x = 0, any positive power of 0 is 0.
* (Technically 000^0 is debated, but here they just return 0.0.)

### Step 2: ****Handle negative powers manually:****

boolean flag = false;

if(n < 0) {

flag = true;

n = -n;

}

* If exponent n is **negative**, remember it using a flag.
* Flip n to **positive** because it’s easier to work with positive numbers.
* We'll later invert the final answer if the flag is true.

**⚠️ Problem:**  
Here, you **directly do** n = -n without handling **Integer.MIN\_VALUE** overflow!

* n = Integer.MIN\_VALUE = -2147483648
* -n = 2147483648 ➔ overflow for int!

✅ **This works fine only if input guarantees** n ≠ Integer.MIN\_VALUE, else you need a long conversion like we discussed earlier.

### Step 3: ****Binary Exponentiation (Fast Power):****

while(n > 0) {

if(n % 2 == 1) {

ans = ans \* x;

n = n - 1;

} else {

x = x \* x;

n = n / 2;

}

}

🔵 **Main idea:**

* **When n is odd:**
  + Use one x immediately: ans = ans \* x
  + Reduce n by 1 to make it even.
* **When n is even:**
  + Square x: x = x \* x
  + Halve n: n = n/2

This keeps reducing n much faster than normal multiplication.

### Step 4: ****Handle negative flag (if needed):****

if(flag) {

return 1.0 / ans;

}

return ans;

* If original n was **negative**, take reciprocal: 1/ans.
* Otherwise, just return the answer.

# 🎯 Time and Space Complexity:

### ⏳ Time Complexity:

**O(log n)**

* Every time, n becomes half (n/2).
* So number of steps ≈ log⁡2n\log\_2 n.

✅ Much faster than naive O(n)O(n) approach.

### 🧹 Space Complexity:

**O(1)**

* No extra space is used (only a few variables).
* Iterative approach → no recursion stack.

✅ Very efficient memory usage.

**3.Sieve of Eratosthenes.**

public int countPrimes(int n) {  
 int[] primes = new int[n];  
 for(int i=2;i\*i<=n;i++)  
 {  
 if(primes[i] == 0)  
 {  
 for(int j=i\*i;j<n;j+=i)  
 {  
 primes[j] = 1;  
 }  
 }  
 }  
 int cnt = 0;  
 for(int i=2;i<n;i++)  
 {  
 if(primes[i] == 0)cnt++;  
 }  
 return cnt;  
}

[**https://www.geeksforgeeks.org/sieve-of-eratosthenes**](https://www.geeksforgeeks.org/sieve-of-eratosthenes)

**4.Count Primes in the range from L to R.**

int countPrimes(int L, int R) {  
 int[] primes = new int[R+1];  
 Sieve(primes,R);  
 int cnt = 0;  
 for(int i=2;i<=R;i++)  
 {  
 if(primes[i] == 0)cnt = cnt+1;  
 primes[i] = cnt;  
 }  
 return primes[R] - primes[L-1];  
}  
  
void Sieve(int[] primes,int N)  
{  
 for(int i=2;i\*i<=N;i++)  
 {  
 if(primes[i] == 0)  
 {  
 for(int j=i\*i;j<=N;j+=i)  
 {  
 primes[j] = 1;  
 }  
 }  
 }  
}

# ✨ Code Explanation

int countPrimes(int L, int R) {

int[] primes = new int[R+1]; // Create a prime array from 0 to R

Sieve(primes, R); // Fill the primes array using Sieve

* Create an array primes of size R+1.
* Then call **Sieve()** to mark all non-primes (1 for non-prime, 0 for prime).

### Inside the Sieve function:

void Sieve(int[] primes,int N)

{

for(int i=2;i\*i<=N;i++)

{

if(primes[i] == 0) // if prime

{

for(int j=i\*i;j<=N;j+=i)

{

primes[j] = 1; // mark multiples as non-prime

}

}

}

}

* Standard **Sieve of Eratosthenes**:
  + For each prime number i, mark its multiples as non-prime.

### Back to countPrimes:

int cnt = 0;

for(int i=2;i<=R;i++)

{

if(primes[i] == 0) cnt = cnt+1; // Count primes up to i

primes[i] = cnt; // Store prime count upto index i

}

* This loop **builds prefix sums**:
  + primes[i] now becomes the **number of primes from 1 to i**.
* Example:
  + primes[10] = number of primes ≤ 10

Finally:

return primes[R] - primes[L-1];

* To count primes between L and R:
  + **Number of primes ≤ R** - **Number of primes ≤ (L-1)**

# ⏳ Time Complexity

| **Step** | **Time Complexity** |
| --- | --- |
| Sieve of Eratosthenes | **O(N log log N)** |
| Building prefix sum array | **O(N)** |
| Query (countPrimes) | **O(1)** |

✅ So **total preprocessing** = **O(N log log N)**  
✅ Then **each query** (countPrimes(L, R)) = **O(1)** !

# 🧠 Space Complexity

* primes[] array of size **O(N)**

**5.Prime Factorisation using Sieve.**

class Solution {  
 static int *MAX* = 200000;  
 static int[] *primes* = new int[*MAX*+1];  
 static void sieve() {  
 for(int i=2;i<=*MAX*;i++)  
 {  
 *primes*[i] = i;  
 }  
 for(int i=2;i\*i<=*MAX*;i++)  
 {  
 if(*primes*[i] == i)  
 {  
 for(int j=i\*i;j<=*MAX*;j+=i)  
 {  
 if(*primes*[j] == j)  
 {  
 *primes*[j] = i;  
 }  
 }  
 }  
 }  
 }  
  
 static List<Integer> findPrimeFactors(int N) {  
 List<Integer> ans = new ArrayList<>();  
 while(N>1)  
 {  
 int val = *primes*[N];  
 ans.add(val);  
 N /= val;  
 }  
 return ans;  
 }  
}

# ✨ Code Intuition and Logic

## 1. **Initialization**

for(int i=2;i<=MAX;i++)

{

primes[i] = i;

}

* You set every number's prime factor initially as itself.
* **Assumption:** Every number i is prime.
* Later, if i is found to be **composite**, you will update it.

## 2. **Modified Sieve (Smallest Prime Factor marking)**

for(int i=2;i\*i<=MAX;i++)

{

if(primes[i] == i)

{

for(int j=i\*i;j<=MAX;j+=i)

{

if(primes[j] == j)

{

primes[j] = i;

}

}

}

}

* You only process numbers **which are still marked prime** (primes[i] == i).
* For every prime i,
  + you go through its multiples starting from i\*i.
  + If a multiple j is **still unmarked** (primes[j] == j), you mark it with its **smallest prime factor** i.

✅ So, each composite number is marked **only once** with its **smallest prime divisor**.

## 3. **Finding Prime Factors**

while(N > 1)

{

int val = primes[N];

ans.add(val);

N /= val;

}

* You keep picking the smallest prime factor of N (which is stored in primes[N]).
* You divide N by that prime and continue.
* When N == 1, you stop.

🔵 **Example for N = 780:**

* primes[780] = 2 → add 2, N = 780/2 = 390
* primes[390] = 2 → add 2, N = 390/2 = 195
* primes[195] = 3 → add 3, N = 195/3 = 65
* primes[65] = 5 → add 5, N = 65/5 = 13
* primes[13] = 13 → add 13, N = 13/13 = 1

Thus, prime factors are [2, 2, 3, 5, 13].

# 📈 Time and Space Complexity

| **Part** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| **Sieve Building** | **O(N log log N)** | **O(N)** |
| **Prime Factorization** | **O(log N)** | **O(log N)** (for output list) |

# 🔥 Bonus Tip:

This technique is often called **SPF (Smallest Prime Factor Sieve)**,  
and it is **heavily used in competitive programming** for fast prime factorization!