**1.Find Prime Numbers in a range**

Given two integers M and N, generate all primes between M and N including M and N.

**Example 1:**

**Input:**

M=1,N=10

**Output:**

2 3 5 7

**Explanation:**

The prime numbers between 1 and 10

are 2,3,5 and 7.

**Example 2:**

**Input:**

M=2, N=5

**Output:**

2,3,5

**Explanation:**

The prime numbers between 2 and 5 are

2,3 and 5.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function primeRange() which takes 2 integer inputs M and N and returns a list of all primes between M and N including N and M.

**Expected Time Complexity:**O(N\*sqrt(N))  
**Expected Auxillary Space:**O(1)

**Constraints:**  
1<=M<=N<=106

Program:

//{ Driver Code Starts

// Initial Template for C++

#include <bits/stdc++.h>

using namespace std;

// } Driver Code Ends

// User function Template for C++

class Solution {

public:

vector<int> primeRange(int a, int b) {

//Sieve of Eratosthenes

vector<bool> prime(b+1,true);

vector<int> ans;

prime[0]=prime[1]=false;

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

{

if(prime[i])

{

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

{

prime[j]=false;

}

}

}

for(int i=max(2,a);i<=b;i++)

{

if(prime[i]) ans.push\_back(i);

}

return ans;

}

};

//{ Driver Code Starts.

int main() {

int t;

cin >> t;

while (t--) {

int N, M, K;

cin >> M >> N;

Solution ob;

vector<int> ans = ob.primeRange(M, N);

for (auto u : ans) cout << u << " ";

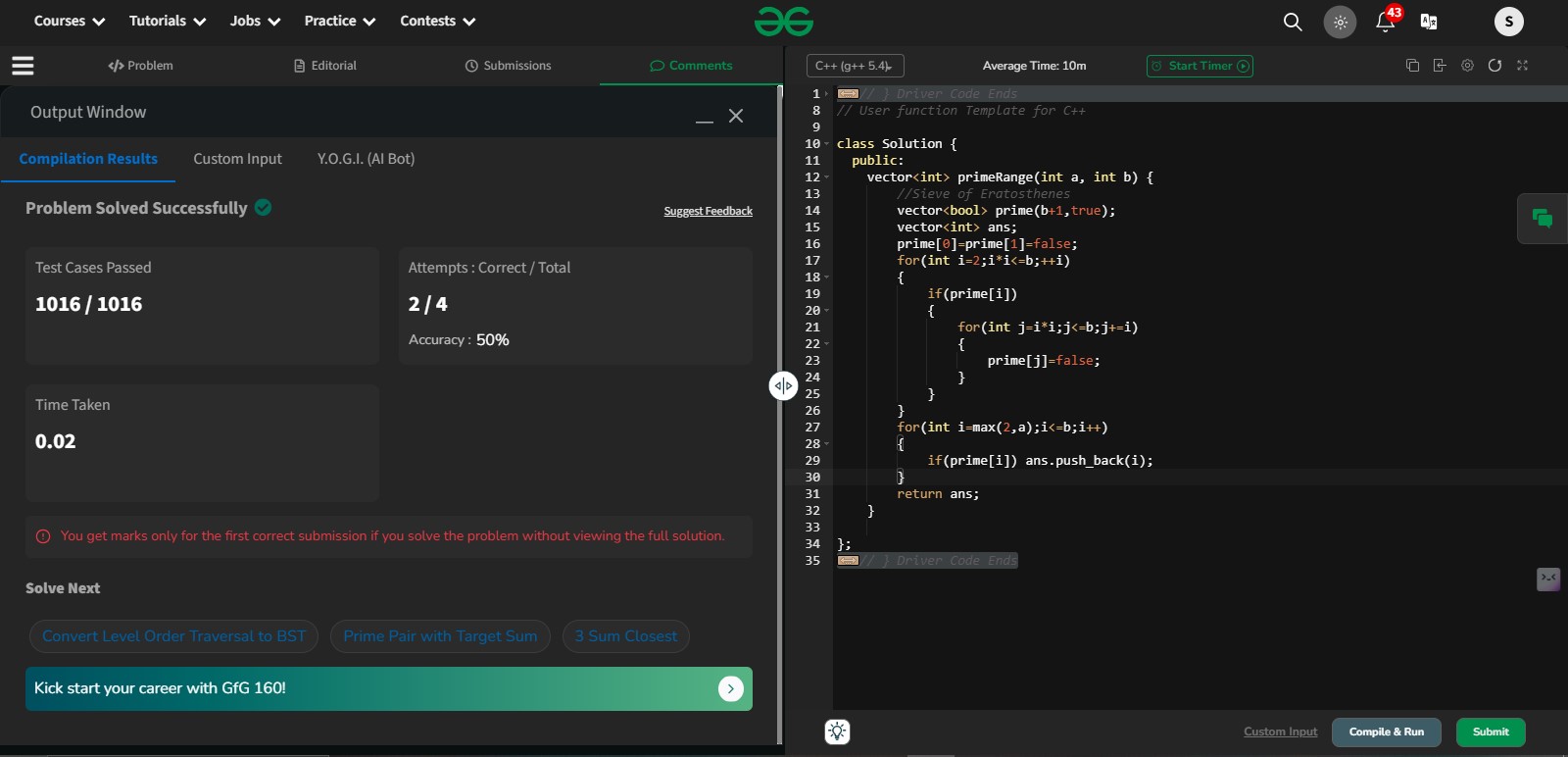
cout << "\n";

cout << "~" << "\n";

}

}

// } Driver Code Ends  
output:



Time complexity:

 Sieve generation:O(n log log n)

 Collecting primes: O(b−a)   
Space complexity: O(b)

**2. Armstrong numbers between the intervals**

#include <iostream>

#include <cmath>

#include <vector>

using namespace std;

// Precompute powers of digits (0-9) for a given max number of digits

vector<vector<int>> precompute\_powers(int max\_digits) {

vector<vector<int>> powers(10, vector<int>(max\_digits + 1, 1));

for (int digit = 0; digit <= 9; digit++) {

for (int power = 1; power <= max\_digits; power++) {

powers[digit][power] = pow(digit, power);

}

}

return powers;

}

int main() {

int st, en;

cout << "Enter the start: ";

cin >> st;

cout << "Enter the end: ";

cin >> en;

int max\_digits = to\_string(en).size();

vector<vector<int>> powers = precompute\_powers(max\_digits);

for (int i = st; i <= en; i++) {

int temp = i;

int sum = 0;

int n = to\_string(temp).size();

while (temp != 0) {

int digit = temp % 10;

sum += powers[digit][n];

temp /= 10;

}

if (sum == i) {

cout << i << " ";

}

}

return 0;

}

Explanation :

Yes, the Armstrong number checking code can be optimized by addressing the following aspects:

1. **Avoid Repeated Digit Count Calculation:**  
   You repeatedly calculate the number of digits (nnn) for each number in the range. This can be done only once when the number is converted to a string.
2. **Avoid Recomputing Powers in the Loop:**  
   Instead of recalculating digitn\text{digit}^ndigitn repeatedly, precompute the powers of digits from 0 to 9 for each nnn (the maximum number of digits in the range). Use these precomputed values during the Armstrong check.

**Optimization Details:**

1. **Precomputing Powers:**
   * Use a 2D array to store digitn\text{digit}^ndigitn for digits 0-9 and powers up to the maximum number of digits.
   * Lookups in this table are O(1)O(1)O(1), avoiding recalculations in every loop.
2. **Reducing Redundant Calculations:**
   * The number of digits (nnn) is calculated once per number using to\_string().
3. **Efficient Loops:**
   * Each number's digits are processed in a straightforward way with no nested loops.

**Time Complexity:**

1. **Precomputation:**  
   O(10×max\_digits)O(10 \times \text{max\\_digits})O(10×max\_digits), which is negligible for typical ranges.
2. **Armstrong Check:**  
   Each number takes O(digits\_in\_number)O(\text{digits\\_in\\_number})O(digits\_in\_number), resulting in overall complexity O(N×max\_digits)O(N \times \text{max\\_digits})O(N×max\_digits), where NNN is the range size.

### Example Input/Output:

#### Input:

mathematica

Copy code

Enter the start: 1

Enter the end: 10000

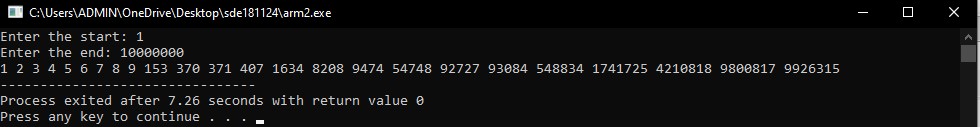
#### Output:

yaml

Copy code

1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474

output:



**3.Can a number be expressed as a sum of two prime numbers?**

Program:

#include <iostream>

#include <vector>

using namespace std;

int main() {

int n;

cout << "Enter your value: ";

cin >> n;

// Step 1: Create a sieve to mark prime numbers

vector<bool> prime(n + 1, true);

prime[0] = prime[1] = false;

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

if (prime[i]) {

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

prime[j] = false;

}

}

}

// Step 2: Check for two primes that sum up to n

int x = 2; // Start from the smallest prime

int y = n - 1; // Start from the largest possible number less than n

bool found = false;

while (x <= y) {

if (prime[x] && prime[y] && (x + y == n)) {

cout << "Yes, " << n << " can be expressed as the sum of " << x << " and " << y << endl;

found = true;

break; // Found a valid pair, exit the loop

}

else if (x + y > n) {

--y; // Decrease y if the sum is too large

} else {

++x; // Increase x if the sum is too small

}

}

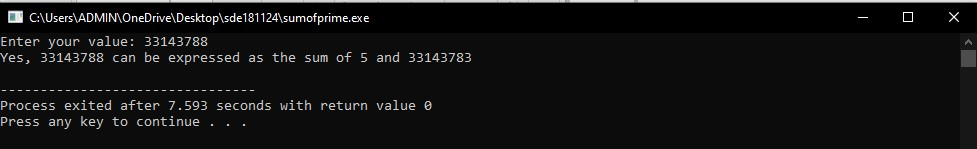
if (!found) {

cout << "No, " << n << " cannot be expressed as the sum of two prime numbers." << endl;

}

return 0;

}  
  
Output:



Time Complexity: O(n log log n)

Space Complexity: O(n)

Program: simplified approach – avoids infinite loops

#include <iostream>

#include <vector>

using namespace std;

int main() {

int n;

cout << "Enter your value: ";

cin >> n;

// Step 1: Generate all prime numbers using Sieve of Eratosthenes

vector<bool> prime(n + 1, true);

prime[0] = prime[1] = false;

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

if (prime[i]) {

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

prime[j] = false;

}

}

}

// Step 2: Store all prime numbers in a list

vector<int> primes;

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

if (prime[i]) {

primes.push\_back(i);

}

}

// Step 3: Use two-pointer technique to find a pair

int x = 0, y = primes.size() - 1;

bool found = false;

while (x <= y) {

int sum = primes[x] + primes[y];

if (sum == n) {

cout << "Yes, " << n << " can be expressed as the sum of " << primes[x] << " and " << primes[y] << endl;

found = true;

break;

} else if (sum > n) {

--y; // Decrease the larger prime

} else {

++x; // Increase the smaller prime

}

}

if (!found) {

cout << "No, " << n << " cannot be expressed as the sum of two prime numbers." << endl;

}

return 0;

}  
  
**4. Replace all 0's with 5 in a given integer**

Given a number **N.** The task is to complete the function **convertFive()** which replaces all zeros in the number with 5 and returns the number.

**Example:**

**Input**

2

1004

121

**Output**

1554

121

**Explanation:**

**Testcase 1:** At index 1 and 2 there is 0 so we replace it with 5.

**Testcase 2:** There is no,0 so output will remain the same.

**Input:**  
The first line of input contains an integer **T**, denoting the number of test cases. Then T test cases follow.  
Each test case contains a single integer N denoting the number.

**Output:**  
The function will return an integer where all zero's are converted to 5.

**Your Task:**  
Since this is a functional problem you don't have to worry about input, you just have to complete the function **convertFive().**

**Constraints:**  
1 <= T <= 103  
1 <= N <= 104

Program:

//{ Driver Code Starts

#include<bits/stdc++.h>

using namespace std;

// Driver program to test above function

// } Driver Code Ends

class Solution{

public:

/\*you are required to complete this method\*/

int convertFive(int n)

{

string me=to\_string(n);

string ans="";

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

{

if(me[i]=='0')

{

ans+="5";

}

else

{

ans+=me[i];

}

}

return stoi(ans);

}

};

//{ Driver Code Starts.

int main()

{

int T;

cin>>T;

while(T--)

{

int n;

cin>>n;

Solution obj;

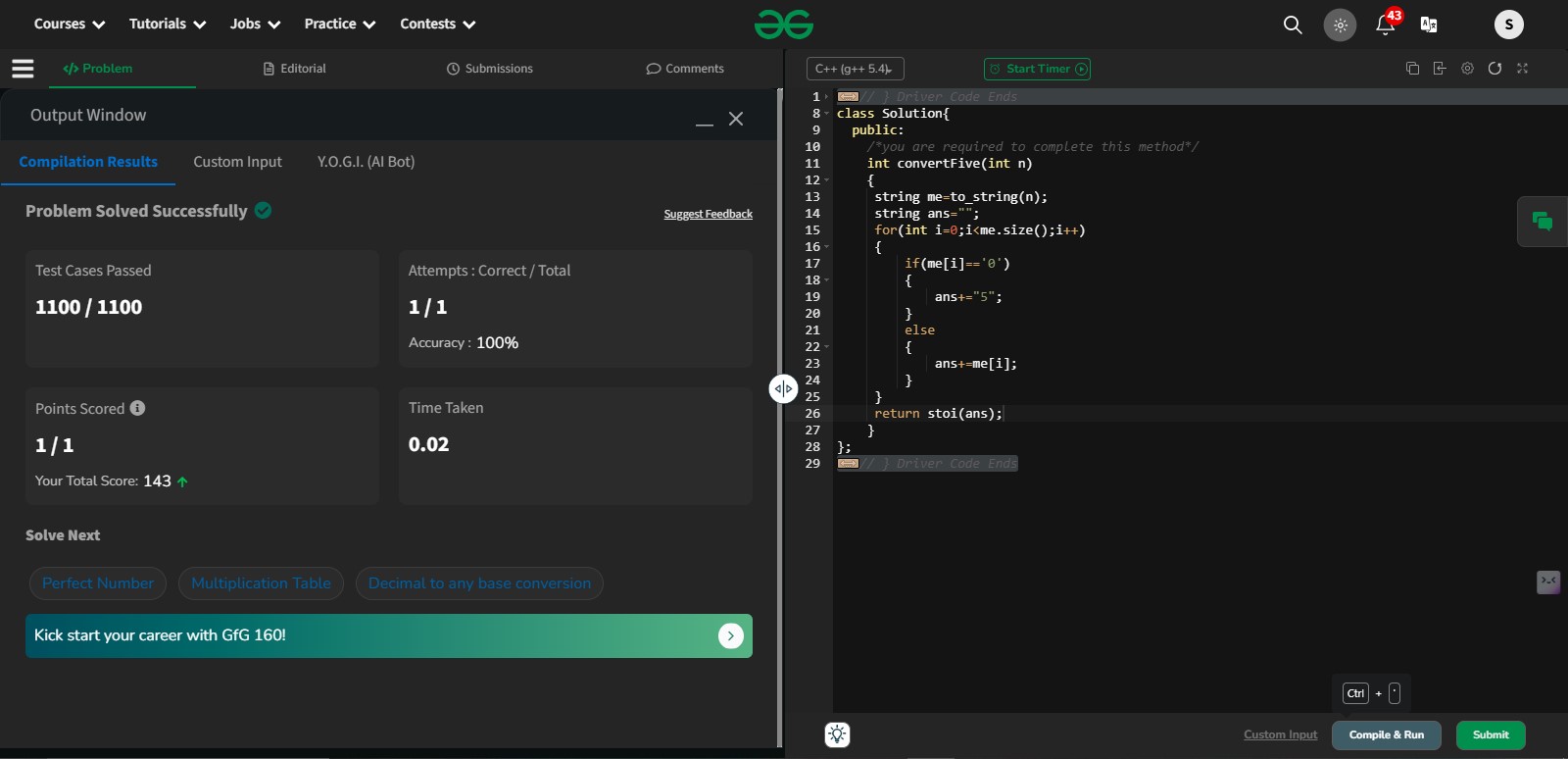
cout<<obj.convertFive(n)<<endl;

cout << "~" << "\n";

}

}

// } Driver Code Ends  
Output:



Time Complexity: O(n) where n is the length of the integer

**5.Binary to decimal**

Given a Binary Number **b**, find its decimal equivalent.

**Examples:**

**Input:** b = 10001000

**Output:** 136

**Input:** b = 101100

**Output:** 44

**Constraints:**  
1 <= number of bits in binary number  <= 16

Program:

//{ Driver Code Starts

#include <bits/stdc++.h>

using namespace std;

// } Driver Code Ends

class Solution {

public:

int binary\_to\_decimal(string &b) {

int n=b.size();

int sum=0;

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

{

if(b[i]=='1')

{

sum+=(1<<(n-i-1));

}

}

return sum;

}

};

//{ Driver Code Starts.

int main() {

int T;

cin >> T;

while (T--) {

string str;

cin >> str;

Solution ob;

int ans = ob.binary\_to\_decimal(str);

cout << ans << "\n";

cout << "~"

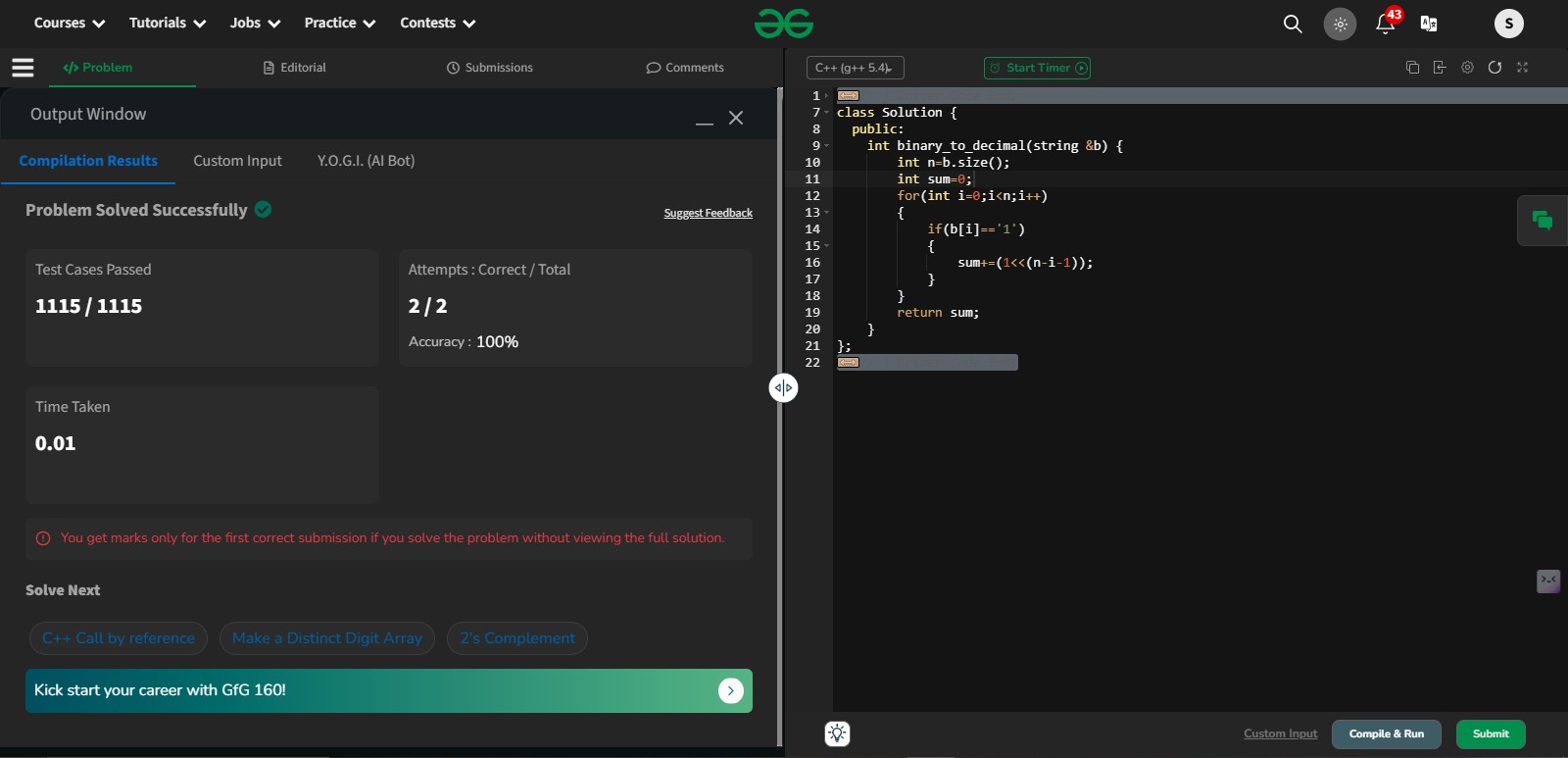
<< "\n";

}

return 0;

}

// } Driver Code Ends  
  
Output:



Time Complexity: O(n)

**6.Decimal to binary**

Given a decimal number N, compute its binary equivalent.

**Example 1:**

**Input:** N = 7

**Output:** 111

**Example 2:**

**Input:** N = 33

**Output:** 100001

**Your Task:**  
You don't need to read input. Complete the function **toBinary()** which takes the decimal number N as the input parameter and **prints its binary equivalent**.  
**Note:**Print the output in a single line, the next line character is printed by the Driver Code.

Program:

//{ Driver Code Starts

//Initial Template for C++

#include <bits/stdc++.h>

using namespace std;

// } Driver Code Ends

//User function Template for C++

void toBinary(int n)

{

string s="";

while(n!=0)

{

s=to\_string(n%2)+s;

n/=2;

}

cout<<s;

}

//{ Driver Code Starts.

int main() {

//code

int t;

cin>>t;

while(t--)

{

int n;

cin>>n;

toBinary(n);

cout<<endl;

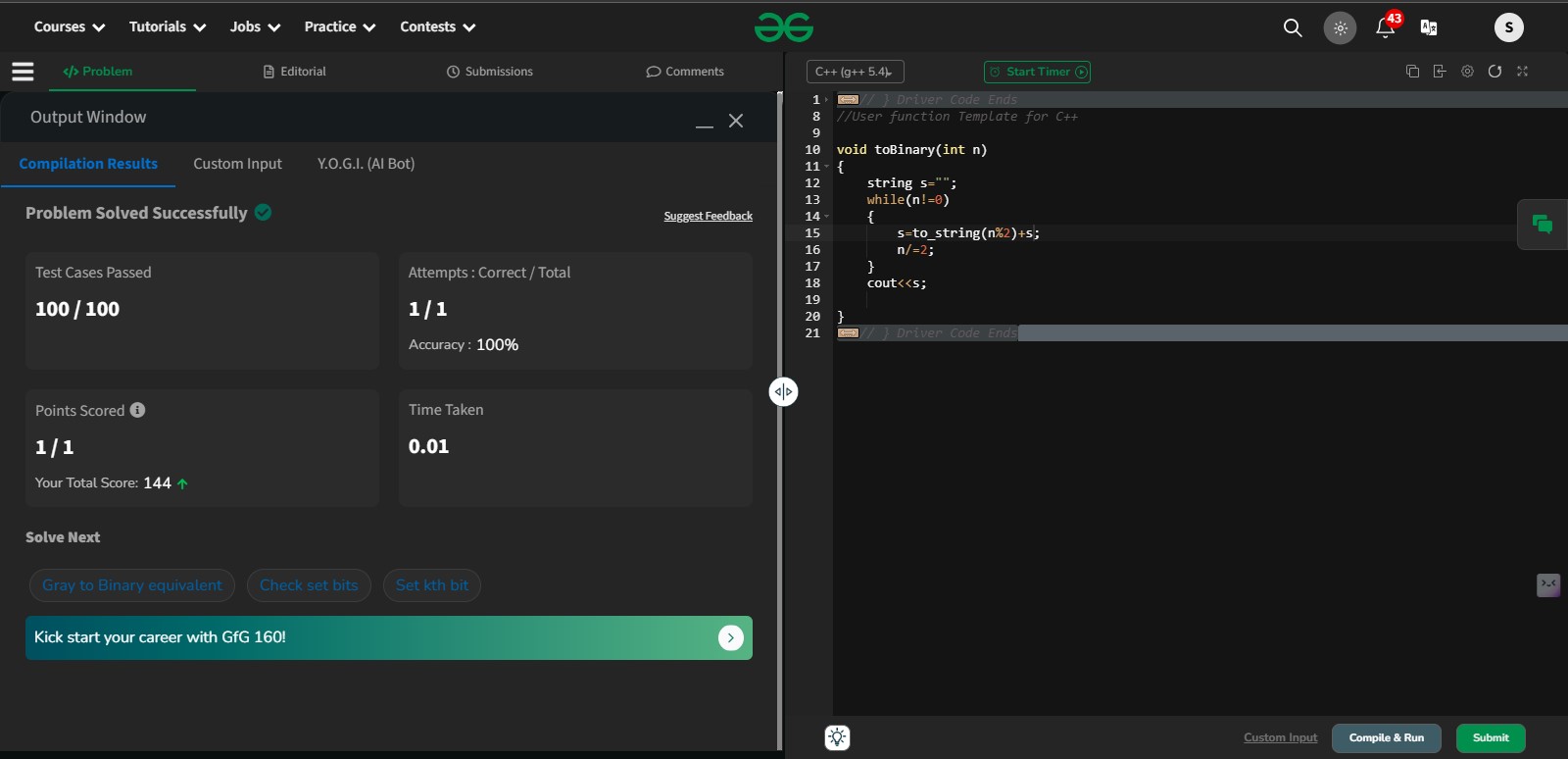
cout << "~" << "\n";

}

return 0;

}

// } Driver Code Ends  
Output:



Time Complexity:O(n)

**7.Decimal to octal**  
Program:

#include <iostream>

#include <vector>

#include<algorithm>

#include<cmath>

using namespace std;

int main()

{

int num;

cin>>num;

string s="";

while(num!=0)

{

s=to\_string(num%8)+s;

num/=8;

}

cout<<s;

}



Time Complexity: O(n)  
**8.octal to decimal**

Program:

#include<iostream>

#include<cmath>

#include<string>

using namespace std;

int main()

{

int x;

cout<<"enter an octal value";

cin>>x;

int ans=0;

int base=0;

while(x!=0)

{

int temp=x%10;

ans+=(temp\* pow(8,base));

x/=10;

base++;

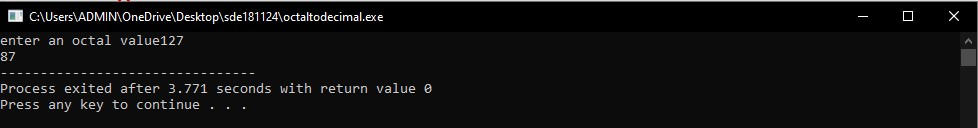
}

cout<<ans;

return 0;

}

Output:



9.Binary to Octal

Program:

#include<iostream>

#include<cmath>

using namespace std;

int main()

{

int bin;

cin>>bin;

string a=to\_string(bin);

int decimal=0;

int n=a.size();

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

{

if(bin%10==1)

{

decimal+=pow(2,i);

}

bin/=10;

}

cout<<"Decimal value is:"<<decimal<<endl;

string x="";

while(decimal!=0)

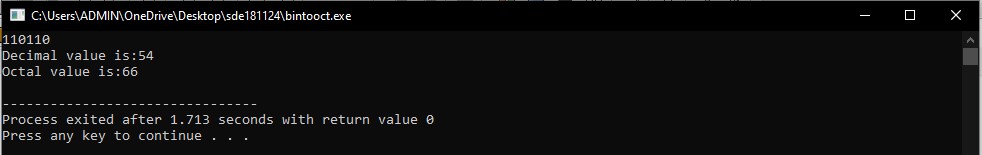
{

x=to\_string(decimal%8)+x;

decimal/=8;

}

cout<<"Octal value is:"<<stoi(x)<<endl;

}  
Output:

Time Complexity: O(n)

10.Octal to bonary

Program:

#include<iostream>

#include<cmath>

using namespace std;

int main()

{

int oct;

cin>>oct;

int i=0;

int deci=0;

while(oct!=0)

{

deci+=(oct%10)\*pow(8,i);

oct/=10;

++i;

}

cout<<"Decimal value is:"<<deci<<endl;

string binary="";

while(deci!=0)

{

binary=to\_string(deci%2)+binary;

deci/=2;

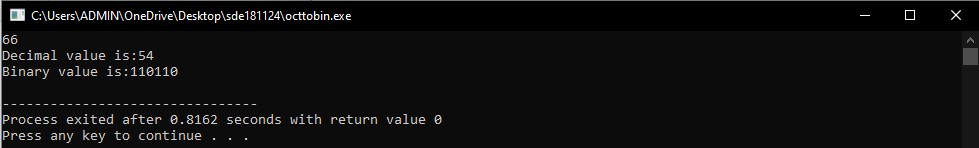
}

cout<<"Binary value is:"<<binary<<endl;

return 0;

}

Output:



Time Complexity: O(n)

11.Maximum number of Handshakes

There are **N** people in a room. If two persons shake hands exactly once, find the maximum number of handshakes possible.

**Example 1:**

**Input:** N = 2

**Output:** 1

**Explaination:** There are two people and they

can shake hands maximum one time.

**Example 2:**

**Input:** N = 3

**Output:** 3

**Explaination:** Let the people be person 1, 2 and 3.

So the possible handshakes are (1, 2), (2, 3) and

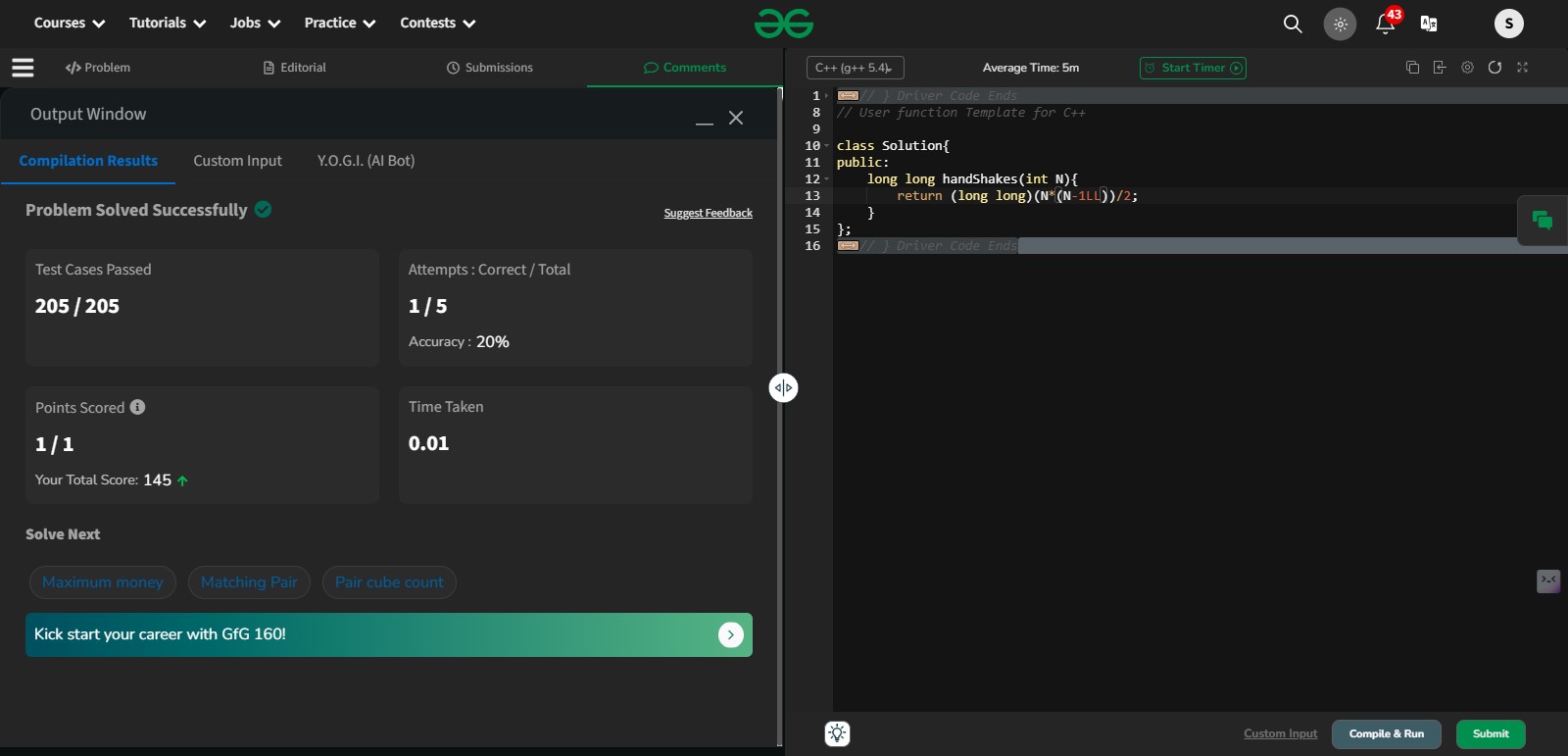
(1, 3).

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **handShakes()** which takes N as input parameter and returns the maximum number of handshakes possible.

**Expected Time Complexity:** O(1)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
2 ≤ N ≤ 109

Output:



12. Quadrants in which coordinates lie

Program:

#include <iostream>

using namespace std;

int main()

{

int x, y;

cout << "Enter the x value: ";

cin >> x;

cout << "Enter the y value: ";

cin >> y;

if (x > 0 && y > 0)

cout << "Quadrant I" << endl;

else if (x < 0 && y > 0)

cout << "Quadrant II" << endl;

else if (x < 0 && y < 0)

cout << "Quadrant III" << endl;

else if (x > 0 && y < 0)

cout << "Quadrant IV" << endl;

else if (x == 0 && y != 0)

cout << "Point is on the Y-axis" << endl;

else if (y == 0 && x != 0)

cout << "Point is on the X-axis" << endl;

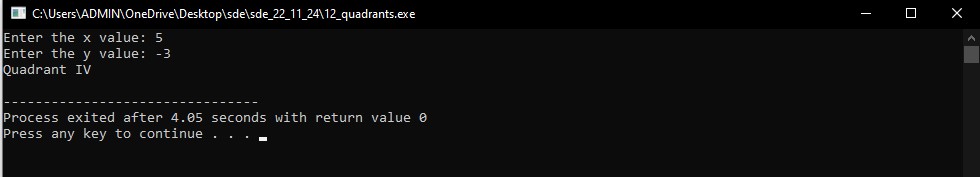
else

cout << "Point is at the origin" << endl;

return 0;

}

Output:



Time Complexity: O(1)

13.Convert digit/number to words

Write a function to convert a given number **n** into words.

*The idea is to break down the number into* ***International Number System****, i.e., smaller groups of three digits* ***(hundreds, tens, and ones),*** *and convert each group into words.*

**Examples :**

**Input:** n = 0

**Output:** "Zero"

**Input:** n = 123

**Output:** "One Hundred Twenty Three"

**Input:** n = 10245

**Output:** "Ten Thousand Two Hundred Forty Five"

**Input:** n = 2147483647  
**Output:** "Two Billion One Hundred Forty Seven Million Four Hundred Eighty Three Thousand Six Hundred Forty Seven"

**Constraints:**  
0 <= n <= 231-1

Program:

class Solution {

public:

string convertToWords(int n) {

if (n == 0)

return "Zero";

vector<string> units = {

"", "One", "Two", "Three",

"Four", "Five", "Six", "Seven",

"Eight", "Nine", "Ten", "Eleven",

"Twelve", "Thirteen", "Fourteen", "Fifteen",

"Sixteen", "Seventeen", "Eighteen", "Nineteen"

};

vector<string> tens = {

"", "", "Twenty", "Thirty", "Forty",

"Fifty", "Sixty", "Seventy", "Eighty", "Ninety"

};

vector<string> multiplier =

{"", "Thousand", "Million", "Billion"};

string res = "";

int group = 0;

while (n > 0) {

if (n % 1000 != 0) {

int value = n % 1000;

string temp = "";

if (value >= 100) {

temp = units[value / 100] + " Hundred ";

value %= 100;

}

if (value >= 20) {

temp += tens[value / 10] + " ";

value %= 10;

}

if (value > 0) {

temp += units[value] + " ";

}

temp += multiplier[group] + " ";

res = temp + res;

}

n /= 1000;

group++;

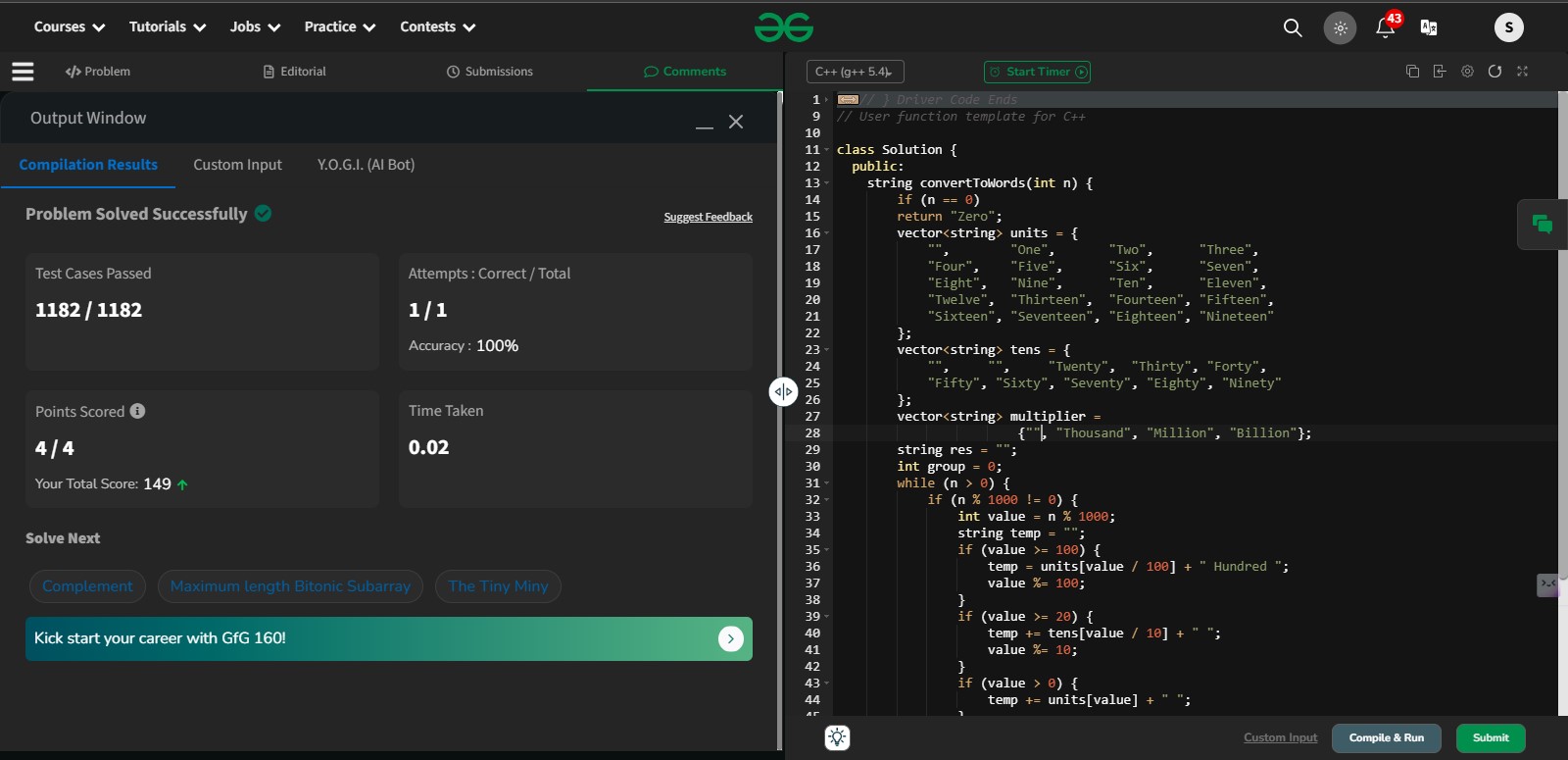
}

return res.substr(0, res.find\_last\_not\_of(" ") + 1);

}

};

Output:



Time Complexity: O(log n) n->length of the number

14. Number of days in a given month of a given year

Program:

#include<iostream>

using namespace std;

int main()

{

string month;

cin>>month;

if(month=="february")

{

cout<<28<<endl;

}

else if(month=="april" || month=="june" || month=="september" || month=="november")

{

cout<<30<<endl;

}

else

{

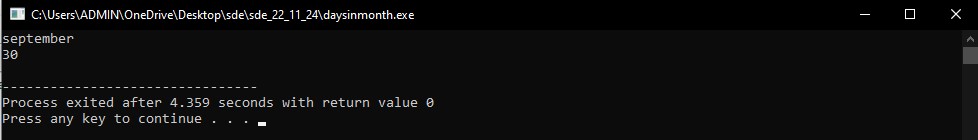
cout<<31<<endl;

}

return 0;

}

Output:



Time Complexity: O(n)

15. Permutations in which n people can occupy r seats in a theatre

Program:

#include<iostream>

using namespace std;

long long factorial(int n)

{

long long fact=1;

for(int i=1;i<=n;i++)

{

fact\*=i;

}

return fact;

}

long long permutations(int n,int r)

{

if(r>n) return 0;

else if(n==r) return 1;

else

{

return factorial(n)/factorial(n-r);

}

}

int main()

{

int n,r;

cout<<"Enter the n value(no of seats):";

cin>>n;

cout<<"Enter the r calue(no of people):";

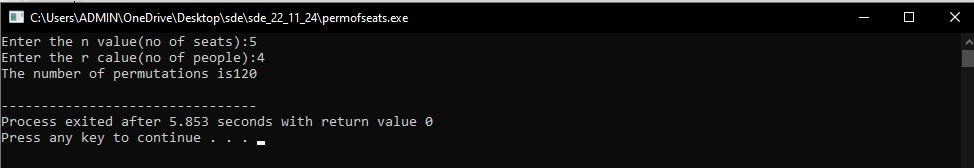
cin>>r;

cout<<"The number of permutations is"<<permutations(n,r)<<endl;

return 0;

}

Output:



16. Number of times digit 3 occurs in each and every number from 0 to n

Program:

#include<iostream>

using namespace std;

int countthree(int n)

{

int count=0;

for(long long place=1;place<=n;place\*=10)

{

long long higher=n/(place\*10);

long long current=(n/place)%10;

long long lower=n%place;

if(current<3) count+=higher\*place;

else if(current==3) count+=higher\*place+lower+1;

else count+=(higher+1)\*place;

}

return count;

}

int main()

{

int n;

cout<<"Enter an value of n:";

cin>>n;

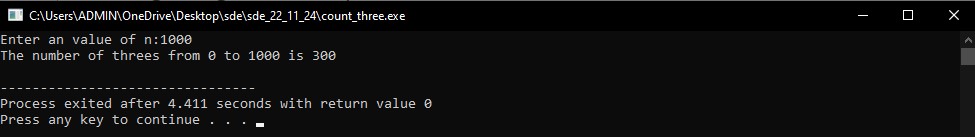
int result=countthree(n);

cout<<"The number of threes from 0 to "<<n<<" is "<<result<<endl;

return 0;

}

Output:



Time Complexity:O(log n)

**Working:**

### Optimized Approach:

We can calculate the number of times the digit 3 appears in each place (ones, tens, hundreds, etc.) for the numbers from 0 to n. This approach uses **digit dynamic programming** or **digit-based counting**.

#### Strategy:

1. **Digit-by-digit analysis**: For each digit position (ones, tens, hundreds, etc.), count how often 3 appears.
2. For each digit place, we count how many numbers have a 3 in that position by analyzing the digits to the left and right of it.

### Efficient Algorithm Explanation:

* For each place value (ones, tens, hundreds, etc.), we check how many times the digit 3 appears.
* We split the number into three parts:
  + The **higher digits** (left of the current digit position).
  + The **current digit** (the digit at the current position).
  + The **lower digits** (right of the current digit position).

By considering the current digit's value and the higher and lower parts, we can compute the count of 3s in that position efficiently.

**Explanation:**

1. **Digit Place Iteration**: The for loop iterates through each digit place, starting from 1 (ones place), then 10 (tens place), and so on. It does this by multiplying place by 10 at each step.
2. **Higher, Current, Lower Digits**:
   * **Higher**: The digits to the left of the current digit.
   * **Current**: The digit at the current place.
   * **Lower**: The digits to the right of the current digit.
3. **Counting '3's**:
   * If the current digit is less than 3, the number of occurrences of 3 in this place is simply the number of times the higher digits can vary (multiplied by the place value).
   * If the current digit is exactly 3, we add the number of occurrences from the higher digits and the lower digits (which can range from 0 to the value of lower).
   * If the current digit is greater than 3, we add one more full set of place for all the higher digits.

**Example:**

For n = 30:

* The digit 3 occurs 1 time in 3. For n = 100:
* The digit 3 occurs 20 times (in 3, 13, 23, 30-39).

**Time Complexity:**

* The loop runs for the number of digit places in n, which is O(log⁡n)O(\log n)O(logn), where nnn is the input number.
* For each digit place, the operations inside the loop take constant time.
* Therefore, the overall time complexity is O(log⁡n)O(\log n)O(logn), which is much more efficient than the previous approach.

**Conclusion:**

This optimized approach counts the occurrences of the digit 3 in O(log⁡n)O(\log n)O(logn) time, making it feasible even for large values of nnn (up to 2312^{31}231).

17. Number of integers which has exactly 9 divisors

Program:

//Number of integers which has exactly 9 divisors

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

// Function to check if a number is prime

bool isPrime(int n) {

if (n <= 1) return false;

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

if (n % i == 0) return false;

}

return true;

}

// Function to find all numbers with exactly 9 divisors up to a given limit

vector<int> findNumbersWith9Divisors(int limit) {

vector<int> result;

// Case 1: Numbers of the form p1^8

for (int i = 2; i <= sqrt(limit); ++i) {

if (isPrime(i)) {

long long p1\_8 = pow(i, 8); // Using long long to handle larger numbers

if (p1\_8 <= limit) {

result.push\_back(p1\_8);

}

}

}

// Case 2: Numbers of the form p1^2 \* p2^2

for (int i = 2; i <= sqrt(limit); ++i) {

if (isPrime(i)) {

for (int j = i + 1; j <= sqrt(limit); ++j) {

if (isPrime(j)) {

long long p1\_2\_p2\_2 = pow(i, 2) \* pow(j, 2); // Using long long here as well

if (p1\_2\_p2\_2 <= limit) {

result.push\_back(p1\_2\_p2\_2);

}

}

}

}

}

return result;

}

int main() {

int limit;

cout << "Enter the limit: ";

cin >> limit;

// Find numbers with exactly 9 divisors

vector<int> numbers = findNumbersWith9Divisors(limit);

// Display the result

cout << "Numbers with exactly 9 divisors up to " << limit << " are: ";

for (int num : numbers) {

cout << num << " ";

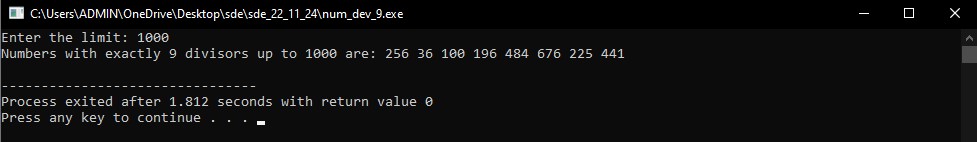
}

cout << endl;

return 0;

}

Output:



Time complexity: O(sqrt(limit) log log limit)