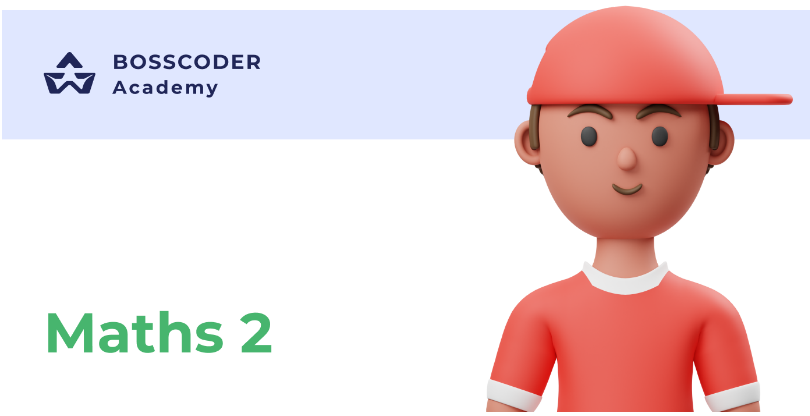
Maths 2

Rajat Garg

17th February, 2023



Agenda

1. Factors and their properties
2. Prime Number and its properties
3. Modular Arithmetics and its properties

Problems

1. [Find all factors of a prime number]

Given a natural number n, print all distinct divisors of it.

Example

Input: n=10

Output: 1 2 5 10

Naive Approach

A Naive Solution would be to iterate all the

```
#include <bits/stdc++.h>
using namespace std;

void printDivisors(int n)
{
    for (int i = 1; i <= n; i++)
        if (n % i == 0)
            cout << " " << i;
}

int main()
{
    int n;
    cin >> n;
    printDivisors(n);
    return 0;
}
```


Optimised Approach

If we look carefully, all the divisors are present in pairs. For example if n = 100, then the various pairs of divisors are: (1,100), (2,50), (4,25), (5,20), (10,10)
Using this fact we could speed up our program significantly.
We, however, have to be careful if there are two equal divisors as in the case of (10, 10). In such case, we'd print only one of them.

Implementation

```
#include <bits/stdc++.h>
using namespace std;
```

```
    for (int i = 1; i <= sqrt(n); i++)
    {
        if (n % i == 0)
        {
            if (n / i == i)
                cout << " " << i;

            else
                cout << " " << i << " " <<
        }
    }
}

int main()
{
    int n;
    cin >> n;
    printDivisors(n);
    return 0;
}
```

Time Complexity: O(sqrt(n))
Space Complexity: O(1)

2. **Number of factors of a number are odd or even**

Given a number find if the number of factors are odd or even
Example
Input: N=100
Output: Odd

<u>Brute force Method</u>

Count the number of divisors if it is odd then

```
#include <bits/stdc++.h>
using namespace std;

int Divisors(int n)
{
    int cnt = 0;
    for (int i = 1; i <= sqrt(n); i++)
    {
        if (n % i == 0)
        {
            if (n / i == i)
                cnt++;

            else
                cnt += 2;
        }
    }
    return cnt;
}

int main()
{
    int n;
    cin >> n;
    if (Divisors(n) & 1)
        cout << "Odd" << endl;
    else
        cout << "Even" << endl;
    return 0;
}
```

Optimised Approach

If we observe carefully all the factors of any divisors will be odd.

Implementation

```
#include <bits/stdc++.h>
using namespace std;

bool isSquare(int x)
{
    int y = sqrt(x);
    return y * y == x;
}

int main()
{
    int n;
    cin >> n;
    if (isSquare(n))
        cout << "Odd" << endl;
    else
        cout << "Even" << endl;
    return 0;
}
```

Time Complexity:O(1)
Space Complexity:O(1)

3. **Prime Number**

For a given number **N** check if it is prime or not. A prime number is a number which is only **divisible by 1 and itself**.

Naive Approach

A naive solution is to iterate through all numbers from 2 to sqrt(n) and for every number check if it divides n. If we find any

BOSSCODER ACADEMY UPSKILL WITH US

```
#include <bits/stdc++.h>
using namespace std;

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;
}

int main()
{
    isPrime(11) ? cout << " true\n" : cout
    return 0;
}
```

Efficient Approach

In the previous approach given if the size of the given number is too large then its square root will be also very large, so to deal with large size input we will deal with a few numbers such as 1, 2, 3, and the numbers which are divisible by 2 and 3 in separate cases and for remaining numbers, we will iterate our loop from 5 to sqrt(n) and check for each iteration whether that (iteration) or (that

BOSSCODER ACADEMY UPSKILL WITH US

```
#include <bits/stdc++.h>
using namespace std;

bool isPrime(int n)
{

    if (n <= 1)
        return false;

    if (n == 2 || n == 3)
        return true;

    if (n % 2 == 0 || n % 3 == 0)
        return false;

    for (int i = 5; i <= sqrt(n); i = i + 6
        if (n % i == 0 || n % (i + 2) == 0)
            return false;

    return true;
}

int main()
{
    int n;
    cin>>n;
    isPrime(n) ? cout << " true\n" : cout <
    return 0;
}
```

Time Complexity: O(N)
Space Complexity: O(N)

prime numbers from 1 to N.

Naive Approach

Iterate in a loop if a number is a prime print it else continues

Implementation

```
#include <bits/stdc++.h>
using namespace std;

bool isPrime(int n)
{

    if (n == 1 || n == 0)
        return false;

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

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

    return true;
}

int main()
{
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
        if (isPrime(i))
            cout << i << " ";
    }
```

Better Approach

If a number 'n' is not divided by any number less than or equal to the square root of n then, it will not be divided by any other number greater than the square root of n. So, we only need to check up to the square root of n.

Implementation

```
#include <bits/stdc++.h>
using namespace std;

bool isPrime(int n)
{

    if (n == 1 || n == 0)
        return false;

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

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

    return true;
}

int main()
{
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
```

```
    }
    return 0;
}
```

Optimised Approach

Sieve of Eratosthenes When the algorithm terminates, all the numbers in the list that are not marked are prime.

Implementation

```
#include <bits/stdc++.h>
using namespace std;

void SieveOfEratosthenes(int n)
{

    bool prime[n + 1];
    memset(prime, true, sizeof(prime));

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

        if (prime[p] == true)
        {

            for (int i = p * p; i <= n; i +
                prime[i] = false;
        }
    }

    for (int p = 2; p <= n; p++)
        if (prime[p])
            cout << p << " ";
}
```

```
{
    int n;
    cin >> n;
    SieveOfEratosthenes(n);
    return 0;
}
```

Time Complexity: O(n*log(log(n)))
Space Complexity: O(n)

Modular Arithmetic

Modular Arithmetic deals with the computation of mod of the result after certain operations such as addition, subtraction, etc. Finding a mod(m) is the same as finding the remainder when we divide a by m, i.e., a % m.

Modular Addition

Modular addition is adding two numbers and taking the combined result's modulus.
(a + b) mod(m) = (a mod(m) + b mod(m)) mod(m)
(a + b) % m = (a % m + b % m) % m

Modular Subtraction

Modular Subtraction is subtracting two numbers and taking the combined result's modulus.
(a - b) mod(m) = (a mod(m) - b mod(m) + m) mod(m)
(a - b) % m = (a % m - b % m + m) % m

Modular Multiplication

Modular Multiplication is multiplying two

(a * b) mod(m) = (a mod(m) * b mod(m)) mod(m)
(a * b) % m = (a % m * b % m) % m

Modular Division

Unlike earlier, we can't directly just do the below for dividing two numbers and taking the combined result's modulus.
(a / b) mod(m) ≠ (a mod(m) / b mod(m)) mod(m)
(a / b) % m ≠ (a % m / b % m) % m
We instead do the below-described operation if the modular inverse of b exists.
(a / b) mod(m) = (a mod(m) * modular_inverse(b, m)) mod(m)
Here, modular_inverse(b, m) refers to the modular inverse of b under m.

Problem

5. [Count pairs in array whose sum is divisible by K]
Given an array **A[]** and positive integer **K**, the task is to count the total number of pairs in the array whose sum is divisible by **K**.
Example
Input: N=6 , a[]={2,2,1,7,5,3},k=4
Output: 5

Naive Approach

The simplest approach is to iterate through every pair of the array but using two nested for loops and count those pairs whose sum is divisible by 'K'. The time complexity of this approach is $O(N^2)$.

```
int countKdivPairs(int A[], int n, int K)
{

    int count = 0;

    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {

            if ((A[i] + A[j]) % K == 0)

                count++;
        }
    }

    return count;
}

int main()
{
    int n;
    cin >> n;
    int a[n];
    for (int i = 0; i < n; i++)
        cin >> a[i];
    int k;
    cin >> k;

    cout << countKdivPairs(a, n, k);

    return 0;
}
```

technique. We will separate elements into buckets depending on their (value mod K). When a number is divided by K then the remainder maybe 0, 1, 2, up to (k-1). So take an array to say **freq[]** of size K (initialised with Zero) and increase the value of freq[A[i]%K] so that we can calculate the number of values giving remainder j on division with K.

Implementation

```
#include <bits/stdc++.h>
using namespace std;

int countKdivPairs(int A[], int n, int K)
{
    int freq[K] = {0};

    for (int i = 0; i < n; i++)
        ++freq[A[i] % K];

    int sum = freq[0] * (freq[0] - 1) / 2;

    for (int i = 1; i <= K / 2 && i != (K -
        sum += freq[i] * freq[K - i];

    if (K % 2 == 0)
        sum += (freq[K / 2] * (freq[K / 2]
    return sum;
}

int main()
{
    int n;
    cin >> n;
```

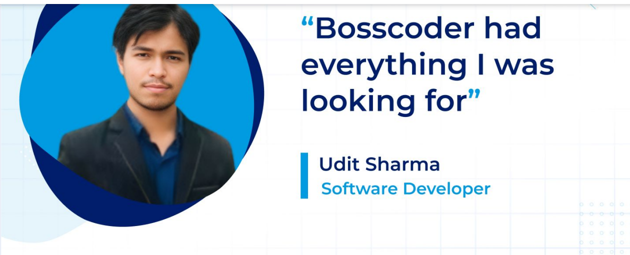
BOSSCODER ACADEMY UPSKILL WITH US

```
        cin >> a[i];
    int k;
    cin >> k;

    cout << countKdivPairs(a, n, k);

    return 0;
}
```

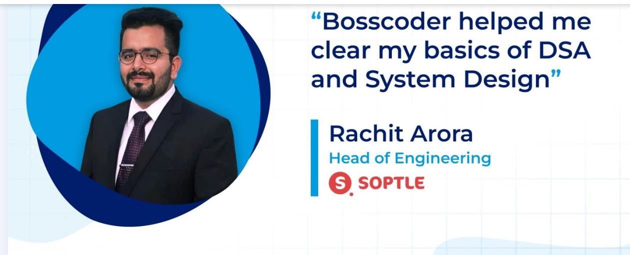
Time complexity: O(N)
Space Complexity: O(K), since K extra space has been taken.



Bosscoder had everything I was looking for

Read

Date: 12th May, 2023



Bosscoder helped me clear my basics of DSA and System Design

Read

Date: 5th May, 2023

View All blogs

Helping ambitious learners upskill themselves & shift to top product based companies.

Lets hear all about it.

Who are we

About us
Blog
Attend a FREE Event
Privacy Policy
Terms & Condition
Pricing and Refund Policy

Contact Us

Email: ask@bosscoderacademy.com

Follow us on

LinkedIn
Youtube
Instagram
Telegram
Reviews on Quora