05/10/2023
Thursday

Basic Mathematics for DSA

* PRIME NUMBER

- Naive Approach → Count primes (leetcode → 204)

Code →

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

bool isPrime(int &num){
    if(num<=1) return false;
    for(int i=2; i<num; i++){
        if(num%i==0){
            return false;
        }}
    return true;
}
int countPrimes(int &num){
    int count=0;
    for(int i=0; i<num; i++){
        if(isPrime(i)){
            count++;
        }
    }
    return count;
}
int main(){
    int num=20;
    int count=countPrimes(num);
    cout<<count<<endl;
}
```

output → 8

Time complexity → $O(num^2)$ i.e. $O(n^2)$

__/__/____

- **Square root approach → Count Primes**

→ Consider a number n.

→ It has two factors a and b such that when multiplied gives n i.e. $n = a \times b$.

→ Then, it must be the condition if a is greater than $\sqrt{n}$, then b must be smaller than $\sqrt{n}$.

i.e. $a > \sqrt{n}$ or $a < \sqrt{n}$

$b < \sqrt{n}$ $b > \sqrt{n}$

Both the factors cannot be greater than $\sqrt{n}$ at a time.

So, if n is a non-prime number, then its one factor must exists in the range $[0, \sqrt{n}]$

Code →

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

bool isPrime(int &num){
    if(num<=1){
        return false;
    }
    int n=sqrt(num);
    for(int i=2;i<=n;i++){
        if(num%i==0){
            return false;
        }
    }
    return true;
}
```

```
int countPrimes (int &num){
    int count = 0;
    for (int i= 0; i<num; i++){
        if (isPrime(i)){
            count ++;
        }
    }
return count;
}
int main() {
    int num = 20;
    int count = countPrimes (num);
    cout << count <<endl;
}
```

Output → 8

Time complexity = $O(num^{3/2})$ i.e. $O(n\sqrt{n})$

__/__/____

- **Sieve of Eratosthenes → Count Primes**

Approach →

- Traverse a loop from 2 ⟶ n-1.
- Assume all the numbers prime and mark them all true.
- Initialising from 2, mark all the multiples of 2 as non-prime. This is done by marking them false.
- Repeat the step above till n-1. Only for primes.
- Rest elements which are marked as prime i.e. true will be counted.

Code →

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

int countPrimes (vector<bool>&prime, int n){
    if (n==0){
      return 0;
    }
// handling 0 and 1 separately as they are non-prime
    prime[0]=prime[1]= false;

  int count =0;

  for (int i=2; i<n ; i++){
    if(prime[i]){
        count++;
      }
```

```
        int J = 2 * i;
        while (J < n){
            prime[J] = false;
            J += i;
        }
    }

    return count;
}

int main() {
    int n = 21;
    vector<bool> prime(n, true);
    int countIs = countPrimes(prime, n);
    cout << countIs << endl;
}
```

Output → 8

Time complexity → $n\left[\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \frac{n}{11} + \cdots\right]$

Inner loop

Outer loop

→ $O(n * (\log(\log n)))$

* GCD/HCF

(1) Euclid Algorithm →

$$gcd(a,b) = gcd(a-b,b) \text{ OR } gcd(a\%b,b)$$

Approach →

= Using Euclid Algorithm to find gcd.
= Apply this algorithm until one of the parameter becomes 0.

For instance,

= gcd (75, 125)
= gcd (125-75, 75) → gcd(50, 75)
= gcd (75-50, 50) → gcd(25, 50)
= gcd (50-25, 25) → gcd(25, 25)
= gcd (25-25, 25) → gcd (0, 25)

So, 25 is the answer.

(2) $LCM(a,b) * gcd(a,b) = a * b$

$$LCM(a,b) = \frac{a*b}{gcd(a,b)}$$

Code for finding GCD →

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

int gcd(int a, int b){

    while(a>0 && b>0){

        if(a>b){
            a-=b;
        }
        else{
            b-=a;
        }
    }
    return a==0 ? b : a;
}
int main(){
    int a= 75;
    int b= 125;
    int ans= gcd(a,b);
    cout<< ans;

}
```

Output → 25

* Modulo Arithmetic →

(1) $(a \% n) \rightarrow [0, ----, n-1]$

(2) • Modular Addition →

$$(a+b) \% m = (a \% m + b \% m) \% m$$

• Modular subtraction →

$$(a-b) \% m = (a \% m - b \% m + m) \% m$$

• Modular multiplication →

$$(a * b) \% m = (a \% m * b \% m) \% m$$

• Modular division →

$$(a/b) \% m \neq (a \% m / b \% m) \% m$$

Modular division directly is not possible.

• $((a \% m) \% m \% m) = a \% m$

* Slow Exponentiation →

Code →

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

int slowExponentiation(int a, int b){
    int ans = 1;
    for(int i=0; i<b; i++){
        ans = ans * a;
    }
    return ans;
}

int main(){

    cout << slowExponentiation(7,3)<<endl;

}
```

Output → 343

Time complexity → O(b)

* Fast exponentiation (a^b i.e a^b) →

I/P = a = 10
b = 5

O/P = 100000

Approach →

```
int ans = 1;
while (b > 0) {
    if (b & 1) {
        ans *= a;
    }
    a *= a;
    b >>= 1;
}
return ans;
```

Dry Run → a = 10, b = 5

1st iteration (5 > 0)
(5 & 1) → True
ans = 1 X 10 → ans = 10
a = 10 X 10 → a = 100
5 >>= 1 → b = 2

IInd iteration (2 > 0)
(2 & 1) → False
a = 100 X 100 → a = 10000
2 >>= 1 → b = 1

IIIrd iteration - (1>0)
(1&1) → True
ans = 10 x 10000 → ans = 100000
a = $10^4 \times 10^4 = 10^8$
1>>=1 → b = 0

IVth iteration - (0>0) → False
loop breaks
return ans → ans = 100000

Code →
```
#include<bits/stdc++.h>
using namespace std;
int fastExponentiation(int a, int b){
    int ans = 1;
    while(b>0){
        if(b&1){
            ans*=a;
        }
        a*=a;
        b>>=1; // right shift b by 1
    }
    return ans;
}
int main(){
    cout<<fastExponentiation(10,5)<<endl;
}
```

Time complexity → O(log b) → As after every iteration, b is getting reduced to half

* Modular Exponentiation for larger numbers →

I/P = a = 5
b = 4 , modulo = 7

O/P = 2

Explanation → $5^4 = 625$
Now, a^b % modulo ⇒ 625 % 7 = 2

Code →

```
#include<bits/stdc++.h>
using namespace std;
int modularExponentiation(int a, int b, int modulo){
    int ans = 1;
    while (b > 0){
        if (b & 1){
            ans = (ans * a) % modulo;
        }
        a = (a * a) % modulo;
        b >>= 1;
    }
    return ans % moulo;
}

int main(){
    cout << modularExponentiation(5,4,7) << endl;
}
```

Time complexity, → O(log b)

* Optimised Sieve → Count Primes

Approach →

- Optimisation 1 for inner loop →

Rather than starting from J = 2*i, start it from J = i*i. First unmarked number would be i*i, as others would have been marked by 2 to (i-1).

i x i → unmarked
i x (i-1)
i x (i-2)
i x (i-3)
.
.
.
i x 2
} Already marked

So, range of J would be $J \in [i*i \rightarrow n]$.

- Optimisation 2 for outer loop →

For $i \times i > n$

$i^2 > n$

$i > \sqrt{n}$, we could not check and loop breaks.

let n = 25

i = 6, i x i = 6 x 6 = 36

36 > 25 → Inner loop breaks

So, we will optimise the outer loop accordingly by making the range of loop $i \in [2, \sqrt{n}]$.

Code →

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

vector<bool> prime(int n){
    vector<bool> sieve(n, true);
    sieve[0]= sieve[1] = false;

    for(int i=2; i*i<=n; i++){          //optimisation 2
        if(sieve[i]){
            int J=i*i; //optimisation 1
            while(J<=n){
                sieve[J]= false;
                J+= i;
            }
        }
    }
    return sieve;
}
int main(){
    vector<bool> ans = prime(30);
    for(int i=0; i<ans.size(); i++){
        if(ans[i]){
            cout<<i<<" ";
        }
    }
}
```

Output → 2 3 5 7 11 13 17 19 23 29

Time complexity → $O(n * \log(\log n))$

* Segmented Sieve →

I/P = $l = 131$
$r = 140$

O/P = Primes in given range are : 131 137 139

Approach →

= Find base Primes. Base Primes will help to mark their multiples false which are present in range.
= Store these base Primes in a vector.
= Traverse a loop on the vector in which base Primes are stored.
= Calculate first multiple in range for the base primes.

```
if ⟶ (firstMultiple < l){
        firstMultiple += prime;
     }
// firstMultiple = (l / prime) * prime
```

l is the left and r is the right.

$l \leq Range \leq r$

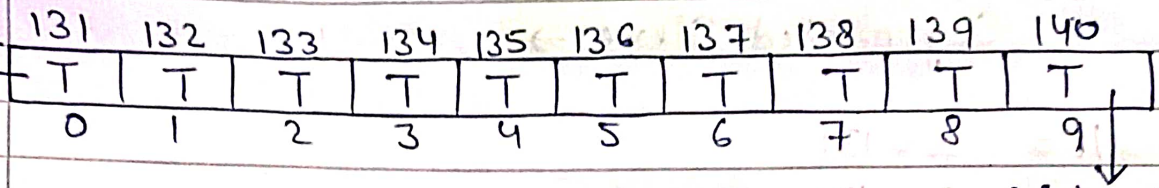
= Now, if first multiple >= l, then choose the max b/w // first multiple and prime * prime. prime * prime // because before that i.e. [prime * prime] all the // multiples of that particular prime are marked // false already if exists.

```
int J = max(firstMultiple, prime*prime);
while (J<=r){
    segSieve[J-l] = false;
    J += prime; }
```

Dry Run →

I/P

segSieve



Find base primes until $\sqrt{r} = \sqrt{140}$

$\boxed{\sqrt{r} = 11}$

Base primes are = 2, 3, 5, 7, 11

Iteration 1 → for 2,

$$\text{firstMultiple} = \left(\frac{131}{2}\right) * 2 = 130$$

⇒ (firstMultiple < l) ⟶ (130 < 131) → True

⇒ (firstMultiple += prime) → (130 + 2) → 132

⇒ J = max (firstMultiple, prime * prime)

J = max (132, 2*2) ⟶ J = 132

⇒ segSeive [J - l] = false;

segSeive [132 - 131] = segSeive [1] = false;

⇒ J += prime (until J <= r)

J += 2

132, 134, 136, 138, 140 are marked false by base Prime 2.

Iteration 2 → for 3 →

First Multiple = $\left(\frac{131}{3}\right) * 3 = 129$

(129 < 131) → True

(129 + 3) = 132

J = max (132, 3*3) → J = 132

segSeive [132 - 131] = false;

J += 3

132, 135, 138 are marked false by base Prime 3.

Iteration 3 → for 5 →

First Multiple = $\left(\frac{131}{5}\right) * 5 = 130$

(130 < 131) → True

(130 + 5) = 135

J = max (135, 5*5) → J = 135

segSeive [135 - 131] = false;

J += 5

135, 140 are marked false by base Prime 5.

Iteration 4 → for 7 →

FirstMultiple = (131/7) * 7 = 126

(126 < 131) → True

(126 + 7) = 133

J = max (133, 7*7) → J = 133

segSeive [133 - 131] = false;

J += 7

133, 140 are marked false by basePrime 7.

__/__/____

Iteration 5 → for 11 →

FirstMultiple = $\left(\frac{131}{11}\right) * 11 = 121$

(121 < 131) → True

(121 + 11) = 132

J = max (132, 11*11) → J = 132

segSeive [132 - 131] = false;

J += 11

132 is marked false by base Prime 11.

O/P

segSieve →

131	132	133	134	135	136	137	138	139	140
T	F	F	F	F	F	T	F	T	F
0	1	2	3	4	5	6	7	8	9

⇒ Return segSeive

Code →

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

// this function will find base Primes
vector<bool> findBasePrimes(int n) {
    vector<bool> sieve(n+1, true);
    sieve[0] = sieve[1] = false;

for (int i=2; i<= sqrt(n); i++) {
    if (sieve[i]) {
        int j = i*i;
        while (j<=n){
            sieve[j] = false;
            j += i;
    } } }
return sieve; }
```

```
// This function will return segmented sieve
vector<bool> segmentedSieve (int l, int r){
   // find basePrimes till sqrt r
    vector<bool>baseSieve= findBasePrime(sqrt(r));
   // store basePrimes      vector<int>basePrimesAre;
 for (int i=0; i<baseSieve.size(); i++){
       if(baseSieve[i]){
            basePrimesAre.push_back (i);
        }
    }

vector<bool> segSieve (r-l+1, true);
     if (l==0 || l==1){
          segSieve[l] = false;
       }

for (auto prime : basePrimesAre){
    int firstMultiple = (l/Prime) * prime;
     if (firstMultiple < l){
        firstMultiple += prime;
      }
     int j = max (firstMultiple, prime *prime);
     while (j<=r){
          segSieve [j-l]= false;
            j+=prime;
        }
     }
  return SegSieve;
}
```

```
int main(){
    int l = 131;
    int r = 140;
    vector<bool> sSieve = segmentedSieve(l, r);
    cout<<"prime in given range are: "<<endl;

    for (int i=0; i<sSieve.size(); i++){
        if (sSieve[i]){
            cout<< i + l <<" ";
        }
    }
}
```