Maths for DSA :

- Extraction of Digits : (%10, /10, *10)

Given n, print all divisors of n :

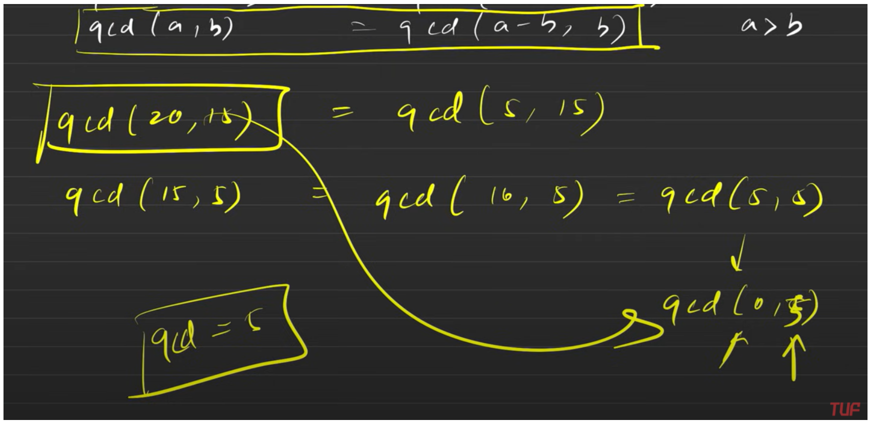
```
void printDivisors(int n) {
    vector<int> divisors;

    // Iterate from 1 to sqrt(n)
    for (int i = 1; i <= sqrt(n); i++) {
        if (n % i == 0) {
            // If divisors are equal, print only once
            if (n / i == i)
                cout << i << " ";
            else {
                // Otherwise print both
                cout << i << " ";
                divisors.push_back(n / i);
            }
        }
    }

    // Print divisors stored in the vector (greater than sqrt(n))
    for (int i = divisors.size() - 1; i >= 0; i--) {
    // why reverse ? Beacuse they would be stored in descending order
        cout << divisors[i] << " ";
    }
}
```

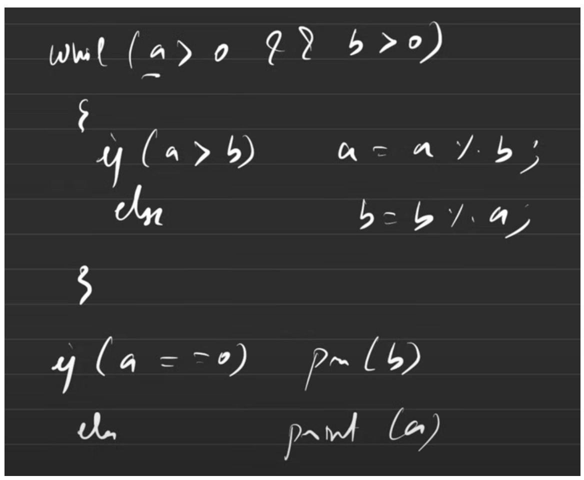
- Optimal way of checking if a number is prime or not. Use above algo and if the count == 2. we are done.

Given two numbers n1 and n2. Return their GCD

```
int gcd(int n1, int n2) {
    for(int i = min(n1, n2);i >= 1;i--) {
        if(n1 % i == 0 && n2 % i == 0) return i;
    }
}
```

Euclidean Algorithm :





Given n print all of its prime factors :

```
vector<int> findDivisors(int n) {
    vector<int> divisors;

    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) {
            divisors.push_back(i);
            while (n % i == 0) {
                n /= i;
            }
        }
    }
```

```
    // If n is still greater than 1, it must be prime and a divisor itself
    if (n > 1) {
        divisors.push_back(n);
    }

    return divisors;
}
```

Given a number n, return all the integers ≤ n

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

void SieveOfEratosthenes(int n)
{
    // Create a boolean array "prime[0..n]" and initialize
    // all entries it as true. A value in prime[i] will
    // finally be false if i is Not a prime, else true.
    bool prime[n + 1];
    memset(prime, true, sizeof(prime));

    for (int p = 2; p * p <= n; p++) {
        // If prime[p] is not changed, then it is a prime
        if (prime[p] == true) {
            // Update all multiples of p greater than or
            // equal to the square of it numbers which are
            // multiple of p and are less than p^2 are
            // already been marked.
            for (int i = p * p; i <= n; i += p)
                prime[i] = false;
        }
    }

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

Why till only root of n ?

- Any composite number m can be written as a product of two numbers, say a and b: m = a × b.
- If both a and b were greater than √n, their product a × b would be greater than n, which contradicts m ≤ n.
- Therefore, at least one of a or b must be less than or equal to √n.

Exponentiation by Squaring

```
long long power(long long base, long long exp) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) { // If exp is odd
            result *= base;
        }
        base *= base;
        exp /= 2;
    }
    return result;
}
```

Count Prime in a range L-R :

```
vector<int> sieveWithPrefixSum(int max) {
    vector<bool> is_prime(max + 1, true);
    vector<int> prefix_sum(max + 1, 0);
    is_prime[0] = is_prime[1] = false;

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

    // Build prefix sum array
    for (int i = 1; i <= max; ++i) {
        prefix_sum[i] = prefix_sum[i - 1] + (is_prime[i] ? 1 : 0);
    }

    return prefix_sum;
}

// Function to count primes in the range [L, R] using prefix sum array
int countPrimesInRange(int L, int R, vector<int>& prefix_sum) {
    if (L > R) return 0;
    if (L == 0) return prefix_sum[R];
```

```
    return prefix_sum[R] - prefix_sum[L - 1];
}
```

Smallest Prime Factor (SPF) | Prime Factorization |

As we give integers, keep returning their Prime Factorization

```
const int MAX = 1000000; // Adjust based on the problem's constraints

// Function to compute smallest prime factor (SPF) for every number up to max
std::vector<int> computeSPF(int max) {
    std::vector<int> spf(max + 1);
    for (int i = 2; i <= max; ++i) {
        spf[i] = i;
    }

    for (int i = 2; i * i <= max; ++i) {
        if (spf[i] == i) { // i is a prime number
            for (int j = i * i; j <= max; j += i) {
                if (spf[j] == j) {
                    spf[j] = i;
                }
            }
        }
    }

    return spf;
}

// Function to factorize a number using the SPF array
std::map<int, int> factorize(int num, const std::vector<int>& spf) {
    std::map<int, int> factors;
    while (num != 1) {
        int prime = spf[num];
        factors[prime]++;
        num /= prime;
    }
    return factors;
}

int main() {
    int L, R;
    cout << "Enter the maximum value for precomputation: ";
    cin >> MAX;

    // Compute SPF array
    vector<int> spf = computeSPF(MAX);
```

```
    cout << "Enter a number to factorize: ";
    int num;
    cin >> num;

    if (num > MAX) {
        cerr << "Number exceeds precomputed limit." << std::endl;
        return 1;
    }

    // Factorize the number
    map<int, int> factors = factorize(num, spf);

    cout << "Prime factors of " << num << " are: ";
    for (const auto& factor : factors) {
        cout << factor.first << "^" << factor.second << " ";
    }
    cout << endl;

    return 0;
}
```