Advanced Number Theory

- ✔ Factorization
- ✔ Sieve of Eratosthenes
- ✔ Prime Factorization
  - Smallest Prime Factor
  - Number of Factors
  - Sum of Factors
- ✔ Revision
  - Modular Arithmetic
  - Exponentiation
  - Euclidean Algorithm

Factorization: Find all factors of a number N

- Naive way
  - Check every number for factor from 1 to N
  - O(N)

- Factors occur in pairs
  - N = P * Q
  - Without loss of generality if P < Q => P <= sqrt(N)

- Efficient way
  - Check for every P from 1 to sqrt(N). Q = N / P
  - O(sqrtN)

Factorization: Find all factors of a number N

Implementation (Efficient way)

```
vector<long long> findFactors(long long n) {
    vector<long long> factors;
    for (long long d = 1; d * d <= n; d++) {
        if (n % d == 0) {
            factors.push_back(d);
            if (n / d != d) // d should be different from n / d
                factors.push_back(n / d);
        }
    }
    return factors;
}
```

Prime Factorization: Represent N as $p_1^{k1} p_2^{k2} \ldots\ldots p_m^{km}$

- Naive way
  - Find all factors of N.
  - Check each number for prime
  - Find how many times each prime divides N

- Efficient way
  - Only one of the prime factors of N can be > sqrt(N)
  - Iterate from 2 to sqrt(N)
  - Check if current number divides N. If yes, keep dividing N by that number as many times as possible
  - Invariant: any new number that will divide N now will be a prime number
  - O(sqrtN)

Prime Factorization: Represent N as $p_1^{k1} p_2^{k2} \ldots\ldots p_m^{km}$

Implementation (Efficient way)

```cpp
vector<long long> primeFactorization(long long n) {
    vector<long long> factorization;
    for (long long d = 2; d * d <= n; d++) {
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
    }
    if (n > 1) // checking for the only prime factor that is > sqrt(N)
        factorization.push_back(n);
    return factorization;
}
```

Sieve of Eratosthenes: Find all prime numbers from 1 to N

- Naive way
  - Iterate from 1 to N
  - Check if current number is prime: O(sqrtN) or O(logN) using Miller Rabin
  - Time: O(NrootN), Space: O(1)

- Efficient way
  - Iterate from 2 to N
  - Keep marking numbers as non primes by iterating on multiples of primes
  - If current number (X) is unmarked -> X is Prime
  - Mark all multiples of X as non-prime
  - Time: O(Nlog(logN)), Space: O(N). Proof

Sieve of Eratosthenes: Find all prime numbers from 1 to N

Implementation (Efficient way)

```
vector<bool> isPrime(n + 1, true);
isPrime[0] = isPrime[1] = false;

for (long long i = 2; i <= n; i++) {
    if (isPrime[i]) {
        for (long long j = i * i; j <= n; j += i) {
            // why iterate from i * i and not 2 * i
            isPrime[j] = false;
        }
    }
}
```

Smallest Prime Factor: Precomputing SPF for every N using Sieve

- Idea
  - Iterate through all the primes and for every multiple of that prime P, see if P is its smallest prime factor or not.
  - We can use the same idea as that in sieve to find all primes and iterate over only relevant multiples of P.
  - Time for precomputation: O(Nlog(logN))
- Use case
  - Finding the prime factorization of a number in O(logN) time

Efficient Prime Factorization using SPF

```
vector<pair<int, int>> primeFactorization(int x, vector<int>& spf){
    vector<pair<int, int>> ans;
    while(x != 1){
        int prime = spf[x];
        int cnt = 0;
        while(x % prime == 0){
            cnt++;
            x = x / prime;
        }
        ans.push_back({prime, cnt});
    }
    return ans;
}
void solve(){
    int maxN = 1e6;
    vector<bool> isPrime(maxN, true);
    vector<int> spf(1e6, 1e9);
    for(long long i = 2; i < maxN; i++){
        if(isPrime[i]){
            spf[i] = i;
            for(long long j = i * i; j < maxN; j += i){
                isPrime[j] = false;
                spf[j] = min(spf[j], (int)i);
            }
        }
    }
    vector<pair<int, int>> primeF = primeFactorization(36, spf);
}
```

Number and Sum of Divisors from Prime Factorization: Problem

- $N = p_1^{e1} p_2^{e2} \ldots\ldots p_k^{ek}$
  - Number of Divisors:

$$d(n) = (e_1 + 1) \cdot (e_2 + 1) \cdots (e_k + 1)$$

  - Sum of Divisors:

$$\sigma(n) = \frac{p_1^{e_1+1} - 1}{p_1 - 1} \cdot \frac{p_2^{e_2+1} - 1}{p_2 - 1} \cdots \frac{p_k^{e_k+1} - 1}{p_k - 1}$$

Modular Arithmetic: Link

$$(A + B) \% M = [(A \% M) + (B \% M)] \% M$$
$$(A - B) \% M = [(A \% M) - (B \% M) + M] \% M$$
$$(A * B) \% M = [(A \% M) * (B \% M)] \% M$$
$$(A / B) \% M = [(A \% M) * (B^{-1} \% M)] \% M$$

What is $B^{-1} \% M$ = Modular Inverse (Will be covered later)

Binary Modular Exponentiation: Link

Idea:

$$a^n = \begin{cases} 1 & \text{if } n == 0 \\ \left(a^{\frac{n}{2}}\right)^2 & \text{if } n > 0 \text{ and } n \text{ even} \\ \left(a^{\frac{n-1}{2}}\right)^2 \cdot a & \text{if } n > 0 \text{ and } n \text{ odd} \end{cases}$$

Time Complexity: O(log(n))

Binary Modular Exponentiation: Link

Implementation: (Don’t forget to take MOD when mentioned in problem)

Recursive

```
long long binpow(long long a, long long b) {
    if (b == 0)
        return 1;
    long long res = binpow(a, b / 2);
    if (b % 2)
        return res * res * a;
    else
        return res * res;
}
```

Iterative

```
long long binpow(long long a, long long b) {
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a;
        a = a * a;
        b >>= 1;
    }
    return res;
}
```

Euclidean Algorithm: Link

Theorem:

$$\gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise.} \end{cases}$$

Implementation:

Recursive

```
int gcd (int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd (b, a % b);
}
```

Iterative

```
int gcd (int a, int b) {
    while (b) {
        a %= b;
        swap(a, b);
    }
    return a;
}
```

Time Complexity: O(log(min(a, b))) Proof

Important GCD results

- GCD(a, b) = GCD(b, a)
- GCD(a, 0) = a
- GCD(a, b, c) = GCD(GCD(a, b), c) = GCD(a, GCD(b, c)) = GCD(b, GCD(a, c))
- GCD(a, b) >= GCD(a, b, c) >= GCD(a, b, c, d)
- GCD contains the minimum powers of primes
- LCM contains the maximum powers of primes
- GCD(a, b) * LCM(a, b) = a * b