CP

Math

Modular Arithmetic

Modular arithmetic is the branch of arithmetic mathematics related with the “mod” functionality. Basically, modular arithmetic is related with computation of “mod” of expressions. Expressions may have digits and computational symbols of addition, subtraction, multiplication, division or any other.

Quotient Remainder Theorem:

`a = b x q + r where 0 <= r < b`

Modular Addition:

`(a + b) % m = ((a % m) + (b % m)) % m`

Modular Substraction:

```
((-b) % m) = (((- b) % m) + m) % m
(a - b) % m = ((a % m) + ((-b) % m)) % m
```

Modular Multiplication:

`(a x b) % m = ((a % m) x (b % m)) % m`

Modular Division:

`(a / b) % m = (a x (inverse of b if exists)) % m`

Moduler Inverse:

The modular inverse of `a mod m` exists only if a and m are relatively prime i.e. `gcd(a, m) = 1`. Hence, for finding the inverse of an under modulo m, if `(a x b) % m = 1` then b is the modular inverse of a.

Example: a = 5, m = 7, (a x b) % m = (5 x 3) % 7 = 1 hence, b=3 is modulo inverse of 5 under 7.

Programetically we can **Bruteforce** b from 1 to m-1 and check for which b, (a x b) % m = 1.

Fermat's Little Theorem:

Let p be a prime which does not divide the integer a, then $a^{p-1} \equiv 1 \pmod{p}$.

Proof:

Start by listing the first p-1 positive multiples of a: $a, 2a, 3a, \dots (p-1)a$

Suppose that ra and sa are the same modulo p, then we have $r \equiv s \pmod{p}$, so the p-1 multiples of a above are distinct and nonzero; that is, they must be congruent to $1, 2, 3, \dots, p-1$ in some order. That means for prime p, modulo p of 1 to $p-1$ multiplies of a [$a, 2a, 3a, \dots (p-1)a$] will be distinct [$1, 2, 3, \dots, p-1$] for each multiplies in random order.

Now multiply all these congruences together and we find,

$$a\,(2a)\,(3a)\,\dots\,((p-1)a) \equiv 1.2.3.\dots.(p-1) \pmod{p}$$
$$\Rightarrow a^{p-1}\,(p-1)! \equiv (p-1)! \pmod{p}$$
$$\Rightarrow a^{p-1} \equiv 1 \pmod{p}$$ [Divide both side by (p-1)!]

[Proved]

Modulo Inverse Under Prime Modulo:

For prime p, $a \times (1/a) = 1 \pmod{p}$ [$a \times (1/a) = 1$]. So $(1/a)$ is the modulo inverse of a under modulo p.

From Fermat's Little Theorem,

$$a^{p-1} \equiv 1 \pmod{p}$$
$$\Rightarrow a^{p-2} \equiv (1/a) \pmod{p}$$

Hence, modulo inverse of a under modulo p is $(a^{p-2} \bmod p)$.

So, moduler division for prime modulo p, $(a/b) \bmod p = (a * (b^{p-2} \bmod p)) \bmod p$

Modular Exponentiation (Big Mod):

Finding $a^n \bmod m$ is the modular exponentiation. a^n where a and n are large that a^n won't fit in long long integer, that's why doing mod won't help.

To solve this problem we can use ***Bruteforce*** technique to iterate over n and multiply a and doing mod [`ans = (ans * a) mod m`]. But cpmpexity of this is O(n). But we can reduce the complexity to $\log_2(n)$.

$$a^{100} = a^{50} * a^{50}$$
$$a^{50} = a^{25} * a^{25}$$
$$a^{25} = a^{12} * a^{12} * a$$
$$a^{12} = a^{6} * a^{6}$$
$$a^{6} = a^{3} * a^{3}$$
$$a^{3} = a^{1} * a^{1} * a$$

At every step we reduce ***n*** to ***n/2***.

Code Snippet:

```
long long int power(long long int a, long long int n, long long int
mod)
```

```
{
        long long int x;
        if (n == 1)
                return a % mod;
        x = recursive_power(a, n / 2);
        if (n % 2)
                return (((x * x) % mod) * a) % mod;
        else
                return (x * x) % mod;
}
```

Related Problem:

1. Print the last k digit of a^n (in decimal).

Solution: Procedure is same as modular exponentiation but we have to mod it with 10^k.

Code Snippet:

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

long long int power(long long int a, long long int n, long long
mod)
{
        long long int x;
        if (n == 1)
                return a % mod;

        x = power(a, n / 2, mod);
        if (n % 2)
                return (((x * x) % mod) * a) % mod;
        else
                return (x * x) % mod;
}

int numberOfDigits(long long int n)
{
        int cnt = 0;
        while (n)
        {
                n /= 10;
                cnt++;
        }
        return cnt;
}

void last_k_digit(long long int a, long long int n, long long in
{
        long long int mod = 1, digit;
        for (int i = 0; i < k; i++)
                mod *= 10; // mod=10^k
        digit = power(a, n, mod);
        for (int i = 0; i < k - numberOfDigits(digit); i++)
                cout << 0;
        if (digit)
                cout << digit << endl;
}

int main()
{
        long long int a, n, k;
        a = 3, n = 100000, k = 9; // Last k digit of a^n

        // Decimal form
        last_k_digit(a, n, k);

        return 0;
}
```

2. Print the last k digit of a^n (in binary).

Solution: Procedure is same as above but we have to mod it with 2^k.

Code Snippet:

```
long long int last_k_digit_binary(long long int a, long long int
long long int k)
{
        long long int x;
        if (n == 1)
                return a % mod;

        x = last_k_digit_binary(a, n / 2, k);
        // If we use % sign we have to use 2^k = (1<<k), or if w
& then we have to use (2^k)-1 = (1<<k)-1
        if (n % 2)
                return (((x * x) % (1 << k)) * a) % (1 << k);
        else
                return (x * x) % (1 << k);
}
```

3. Print something modulo 2^{32} or modulo 2^{64}.

Solution: Taking all the veriable as ***unsigned int*** related to the calculation will give modulo 2^{32}. It is safe to take all the veriable as unsigned int. Similarly, tking all the veriable as ***unsigned long long int*** related to the calculation will give modulo 2^{64}. We don't have to mod separately.

modulo 2^{32} → unsigned int

modulo 2^{64} → unsigned long long int

4. Multiply large integers under large modulo or avoid overflow in

modular multiplication.

Solution: Given an integer a, b, m. Find (a * b) mod m, where a, b may be large and their direct multiplication may cause overflow. Therefore we use the basic approach of multiplication i.e., a * b = a + a + … + a (b times). Now easily compute the value of addition (under modulo m) without any overflow in the calculation. If we use **Bruteforce** apprach for adding a b times time complexity will be O(b). But we can reduce the complexity to O(log b) using the same technique we used in modular exponentiation. But insted of multiplying we will adding.

Code Snippet:

```
long long int multiplication_mod(long long int a, long long int
long long int mod)
{
        long long int x;
        if (b == 1)
                return a % mod;

        x = multiplication_mod(a, b / 2, mod);
        if (b % 2)
                return (((x + x) % mod) + a) % mod;
        else
                return (x + x) % mod;
}
```

Important Propertice:

1. (a % b) = a - floor(a / b) * b
2. (x % 2^k) = x & (2^k - 1) = x & ((1 << k) - 1)

N.B. Here % means mod.

Prime

Prime Number: A prime number (or a prime) is a natural number greater than 1 that is not a product of two smaller natural numbers.

For example, 5 is prime because the only ways of writing it as a product, 1 × 5 or 5 × 1, involve 5 itself.

Composite number: A natural number greater than 1 that is not prime is called a composite number.

For example, 4 is composite because it is a product (2 × 2) in which both numbers are smaller than 4.

Fundamental Theorem of Arithmetic: Every natural number greater than 1 is either a prime itself or can be factorized as a product of primes that is unique up to their order.
For example, `12 = 2`^{2}`.3`^{2}` = (2.2).(3.3)`

Miserable Fate of 1: 1 is not a prime number. Because it doesn't have exactly 2 divisors. If 1 were a prime number there would be Chaos! Because then every number could have represented using product of primes in different ways. For example: `6 = 2·3 = 2·3·1 = 2·3·1·1….` You got the idea right? 1 is not a composite number(according to Wiki) either. Because it has less than 2 divisors.

Primality Test:
To check if a number is prime or not, we just have to check if a number between `2` to `√n` (inclding) divides the n.
Complexity: `O(√n)`
Code Snippet:

```
bool isPrime(int n)
{
        for (int i = 2; i * i <= n; i++)
                if (n % i == 0)
                        return false;

        return true;
}
```

Sieve of Eratosthenes:
It is an easy and efficient way to find all the prime number under a specific number.
Procedure:
1. The number which are devided by the current (For the initial its the first prime) prime, flag them.
2. The next unflaged unmber would be current prime. Repeat the step 1 till square root of n

Complexity: `O(n log(log(n)))`
Code Snippet:

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

```

```
typedef long long ll;

const int N = 1e8 + 9;
bool arr[N]; // Has lowest execution time compare to bitset and
vector.
// bitset<N> arr;
// vector<bool> arr(N);

int main()
{
        ll n;
        cin >> n;
        vector<ll> prime;

        arr[0]=arr[1]=1;
        for(ll i=2; i*i<=n; i++)
        {
                if(!arr[i])
                {
                        // we can start from both j=i or j=2, using i
is more optimized but it wouldn't reduce much time complexity.
                        for(ll j=i; i*j<=n; j++)
                        arr[i*j]=1;
                }
        }

        for(ll i=0; i<=n; i++) if(!arr[i]) prime.push_back(i);
        for(auto it : prime) cout << it << " ";
        cout << endl;

        return 0;
}
```

Smallest Prime Factor(SPF):

The *smallest* number greater than 1 that divides n is also the smallest prime factor of n.

Complexity: O(√n)

Code Snippet:

```
ll SPF(int n)
{
        for (int i = 2; i * i <= n; i++)
                if (n % i == 0)
                        return i;

        return n;
}
```

Prime Factorization:

Prime factorization is the decomposition of a number into a product of smaller prime numbers.

Enumerate:

The implementation is same as the smallest prime factor, just remove the factor from n and repeat the process.

Complexity: `O(√n)`

Code Snippet:

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

int main()
{
        int n, m, tmp_n;

        cin >> n;
        tmp_n=n;
        vector<int> v;

        for (int i = 2; i * i <= n; i++)
        {
                if (n % i == 0)
                {
                        while (n % i == 0)
                                v.push_back(i), n /= i;
                }
        }

        if (n > 1)
                v.push_back(n);

        cout << "Prime fectorization of " << tmp_n << ": \n";
        for (auto it : v)
                cout << it << " ";

        return 0;
}
```

Prime Factorization Using Sieve of Eratosthenes:

The implementation is same as the “Sieve of Eratosthenes”, just find the smallest prime fector for 1 to n and then find the prime fectors for specific number using the help from smallest prime factor.

Complexity: `O(n log(log(n)))`

Code Snippet:

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

```

```
typedef long long ll;
#define endl "\n"

const int N = 1e6 + 9;

int main()
{
        ll n, q;
        bool ok;

        vector<ll> spf(N);
        for (ll i = 2; i < N; i++)
                spf[i] = i;

//      Sieve
        vector<bool> flg(N);
        for (ll i = 2; i * i < N; i++)
        {
                if (!flg[i])
                {
                        for (ll j = i; j < N; j += i)
                                flg[j] = true, spf[j] = min(spf[j],
i);
                }
        }

        cin >> q;
        while (q--)
        {
                cin >> n;
                vector<ll> factors;

                while (n > 1)
                        factors.push_back(spf[n]), n /= spf[n];

                for (auto it : factors)
                        cout << it << " ";
                cout << endl;
        }

        return 0;
}
```

Prime Gap:

A prime gap is the difference between two successive prime numbers. The n-th prime gap, denoted g_n or $g(p_n)$ is the difference between the $(n + 1)$-th and the n-th prime numbers, i.e. $g_n = p_{n+1} - p_n$. We have $g_1 = 1$, $g_2 = g_3 = 2$, and $g_4 = 4$. The sequence (g_n) of prime gaps has been extensively studied; however, many questions and conjectures remain unanswered. There is no exact formula for g_n. Cramer's conjecture states that $g_n = O(\log(p_n)^2)$ which is indeed very low.

Goldbach’s Conjecture (Two Primes with given Sum):

Every even integer greater than 2 can be expressed as the sum of two primes.

Find the prime numbers using Sieve algorithm. Then one by one subtract a prime from N and then check if the difference is also a prime. If yes, then express it as a sum and keep a count.

Complexity: `O(n log(log(n)))`

Code Snippet:

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

typedef long long ll;
typedef unsigned long long ull;
#define endl "\n"

const ll N = 100 + 10;
vector<bool> isPrime(N, true);
vector<ll> primes;

void seive()
{
        isPrime[0] = isPrime[1] = false;
        for (ll i = 2; i * i <= N; i++)
        {
                for (ll j = i + i; j <= N; j += i)
                {
                        isPrime[j] = false;
                }
        }

        for (ll i = 0; i < N; i++)
        {
                if (isPrime[i])
                        primes.push_back(i);
        }
}

int main()
{
        ll n;
        ll c = 0;
        cin >> n;
        seive();

        for (ll i = 0; primes[i] <= n / 2; i++)
        {
                if (isPrime[n - primes[i]])
                {
                        cout << primes[i] << " + " << n - primes[i]
<< " = " << n << endl;;
                        c++;
                }
        }
        cout << c << endl;

        return 0;
}
```

Related Problem:

1. Find the next prime number greater than n where n can be up to 10^9.

Solution: The first solution that pops up in your head is to brute force over all numbers $>n$ until we find a prime. We can check if it is a prime or not in $O(\sqrt{n})$. According to Cramer's conjecture after roughly $O\log(10^9)^2 \approx 30^2 \approx 900$ operation, we will hit a prime! So the complexity will be $O(900 \cdot \sqrt{n})$ which is okay!

Euler's Totient Function

Euler's Totient Function: Euler's totient function, also known as *phi-function* $\varphi(n)$, counts the number of integers between 1 and n *inclusive*, which are coprime to n. Two numbers are coprime if their greatest common divisor equals 1 (1is considered to be coprime to any number).

Here are values of $\varphi(n)$ for the first few positive integers:

n	phi(n)
1	1
2	1
3	2
4	2

n	phi(n)
5	4
6	2
7	6
8	4
9	6
10	4
11	10
12	4
13	12
14	6
15	8
16	8
17	16
18	6
19	18
20	8

Properties:

→ *n=1*

If n is 1, then 1 itself is the only number which is coprime to it.

Therefore $\varphi(1)=1$

→ *n is prime*

If n is a prime number, then gcd(n, m) = 1 for all $1 \le m < n$.

Therefore, $\varphi(n)=n-1$

→ *n is power of prime* – *$\varphi(n)= \varphi(p^a)$*

Since, $n=p^a$, we can be sure that $\gcd(p,n)\neq 1$. Since both n and p are divisible by p. Therefore, the following numbers which are divisible by p are not coprime to n, $p,2p,3p\ldots p^2,(p+1)p,(p+2)p\ldots(p^2)p,(p^2+1)p\ldots(p^{a-1})p$. There are exactly $p^a/p=p^a-1$ numbers which are divisible by p. So, there are $n-p^{a-1}$ numbers which are coprime to n.

Hence, $\varphi(n) = \varphi(p^a) = n-(n/p) = p^a-(p^a/p) = p^a(1-(1/p) = p^a\times((p-1)/p)$

→ *$\varphi()$ is Multiplicative* – *$\varphi(m \times n)$*

It means, if m and n are coprime, then $\varphi(m\times n)=\varphi(m)\times\varphi(n)$.

Let the prime factorization of n be $p_1^{a1}p_2^{a2}\ldots p_k^{ak}$. Now, obviously p_i nad p_j are coprime to each other.

Since φ function is multiplicative, we can simply rewrite the function as:

$$\varphi(n)=\varphi(p_1^{a1}p_2^{a2}\ldots p_k^{ak})$$

$\varphi(n)=\varphi(p_1^{a1})\times\varphi(p_2^{a2})\ldots\times\varphi(p_k^{ak})$

We can already calculate $\varphi(p^a)=p^a\times((p-1)/p)$.

So our equationg becomes:

$\varphi(n)=\varphi(p_1^{a1})\times\varphi(p_2^{a2})\ldots\times\varphi(p_k^{ak})$

$\varphi(n) = (p_1^{a1}\times((p_1-1)/p_1)) \times (p_2^{a2}\times((p_2-1)/p_2))\ldots\ldots\ldots\ldots \times (p_k^{ak}\times((p_k-1)/p_k))$

$\varphi(n) = (p_1^{a1} \times p_2^{a2} \times p_k^{ak}) \times (((p_1-1)/p_1) \times ((p_2-1)/p_2)\ldots\ldots\ldots\ldots \times ((p_k-1)/p_k))$

$\varphi(n) = n \times (((p_1-1)/p_1) \times ((p_2-1)/p_2)\ldots\ldots\ldots\ldots \times ((p_k-1)/p_k$

Euler's Totient:

Using Prime Factorization:

The idea is basically based on prime factorization.

Complexity: $O(\sqrt{n})$

Code Snippet:

```
int phi(int n) {
	int result = n;
	for (int i = 2; i * i <= n; i++)
	{
		if (n % i == 0)
		{
			while (n % i == 0)
				n /= i;
			result -= result / i;
		}
	}
	if (n > 1)
		result -= result / n;
	return result;
}
```

Using Sieve of Eratosthenes (For a range):

It is based on sieve algorithm. It is generally used for a range (1 to n) of numbers.

Complexity: $O(n \log(\log(n)))$

Code Snippet:

```
void sieve_phi(int n) {
	vector<int> phi(n + 1);
	for (int i = 0; i <= n; i++)
		phi[i] = i;
	for (int i = 2; i <= n; i++)
	{
		if (phi[i] == i)
		{
```

```
                                for (int j = i; j <= n; j += i)
                                        phi[j] -= phi[j] / i;
                        }
                }
        }
```

Divisor

Divisor: In mathematics, a divisor of an integer `n`, also called a factor of `n`, is an integer `m` that may be multiplied by some integer to produce `n`. In this case, one also says that `n` is a multiple of `m`. An integer `n` is divisible or evenly divisible by another integer `m` if `m` is a divisor of `n`; this implies dividing `n` by `m` leaves no remainder.

Enumerating Divisors

If we have to find the divisors of **n**, we only have to enamurate till `√n`.

Suppose take `n=36`,

`1 X 36`
`2 X 18`
`3 X 12`
`4 X 9`
`6 X 6`

So the divisors are `1, 2, 3, 4, 6, 9, 12, 18, 36` and `√36 = 6`.
Terefore if we enamurate till `√n` we will be able to find all the divisors of `n`.

Complexity: `O(√n)`

Code Snippet:

```
void (long long int n)
{
        for (ll i = 1; i * i <= n; i++) // i * i <= n is same as
sqrt(n) but not safe because of double precision
        {
                if (n % i == 0)
                {
                        cout << i << " ";
                        if (n / i != i)
                                cout << (n / i) << " ";
                }
        }
}
```

Odd Number of Divisors

The ***SQUARE*** numbers has only odd number of divisors.

For example,

--Number--	--Divisors--	--Count--
1	1	1
2	1, 2	2
3	1, 3	2
4	1, 2, 4	3
5	1, 5	2
6	1, 2, 3, 6	4
7	1, 7	2
8	1, 2, 4, 8	4
9	1, 3, 9	3
10	1, 2, 5, 10	4

From the table we can see that only square numbers 1, 4, 9 has odd numbers of divisors.

Number of Divisors:

Enumerate**:**

Enumerate the divisors of n and return the count.

Complexity: `O(√n)`

Code Snippet:

```
int NOD(int n)
{
        int nod=0;
        for (ll i = 1; i * i <= n; i++)
        {
                if (n % i == 0)
                {
                        nod++;
                        if (n / i != i)
                                nod++;
                }
        }
        return nod;
}
```

Using Prime Factorization:

It should be obvious that the prime factorization of a divisor `d` has to be a subset of the prime factorization of n, e.g. $6=2.3$ is a divisor of $60=2^2.3.5$. So we only need to find all different subsets of the prime factorization of n. So if a prime factor p appears e times in the prime factorization of n, then we can use the factor p up to e times in the subset. Which means we have $e+1$ choices.

Therefore if the prime factorization of n is $p_1^{e1}.p_2^{e2}....p_k^{ek}$, where p_i are distinct prime numbers, then the number of divisors is:

$$NOD(n)=(e_1+1).(e_2+1)...(e_k+1)$$

Complexity: $O(\sqrt{n})$

Code Snippet:

```
int NODPF(int n)
{
        int nod=1, p;
        for (ll i = 2; i * i <= n; i++)
        {
                if (n % i == 0)
                {
                        p = 0;
                        while (n % i == 0)
                                p++, n /= i;
                        p++;
                        nod *= p;
                }
        }

        if (n > 1)
                nod *= 2;

        return nod;
}
```

Using Sieve of Eratosthenes (For a range):

It is based on sieve agorithm. It is generally used for a range (1 to n) of numbers.

Complexity: $O(n \log(\log(n)))$

Code Snippet:

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

typedef long long ll;
typedef unsigned long long ull;

int main()
{
        ll n;
        cin >> n;
        vector<ll> nod(n + 10);

        // .for (ll i = 1; i <= n; i++)
        //      for (ll j = 1; i * j <= n; j++)
        //              nod[i * j]++;

        // More optimized but little bit complex to understand
        for (ll i = 1; i * i <= n; i++)
        {
                for (ll j = i; i * j <= n; j++)
                {
                        if (i == j)
                                nod[i * j]++;
                        else
                                nod[i * j] += 2;
                }
        }

        for (ll i = 1; i <= n; i++)
                cout << i << " --> " << nod[i] << endl;

        return 0;
}
```

Upper Bound of Number of Divisors:

$NOD(n) \leq n$

Further improvement, $NOD(n) \leq (n/2)+1$

Further improvement, $NOD(n) \leq 2.\sqrt{n}$

Further improvement, $NOD(n) \approx 2.\sqrt[3]{n}$

$2.\sqrt[3]{n}$ can be used safely as the upper bound of NOD. Apparently, this approximation has been tested for $N \leq 10^{18}$, which is large enough to be used in programming contests.

Sum of Divisors:

Using Prime Factorization:

Using prime factorization concept sum of the divisors can be found very easily.

If only prime factor on n is p_1^{e1} then the divisors are $1,\ p_1,\ p_1^2\ ,\ p_1^3\ ,\ ....\ p_1^{e1}$. Then sum of the divisors is $1+p_1+p_1^2+p_1^3+....+p_1^{e1} = (p_1^{e1+1}-1)/(p_1-1)$.

If the prime factorization of n is $p_1^{e1}.p_2^{e2}$ then the sum of the divisors is $(1+p_1+p_1^2+p_1^3+....+p_1^{e1}).(1+p_2+p_2^2+p_2^3+....+p_2^{e2}) = ((p_1^{e1+1}-1)/(p_1-1)).((p_2^{e2+1}-1)/(p_2-1))$.

A similar argument can be made if there are more then two distinct prime factors.

Using Sieve of Eratosthenes (for a range):

It is based on seive agorithm. It is generally used for a range (1 to n) of numbers.

Complexity: `O(n log(log(n)))`

Code Snippet:

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

typedef long long ll;
typedef unsigned long long ull;

int main()
{
        ll n;
        cin >> n;
        vector<ll> sod(n + 10);

        //..    for (ll i = 1; i <= n; i++)
        //..            for (ll j = 1; i * j <= n; j++)
        //..                    sod[i * j] += i;

        // More optimized but little bit complex to understand
        for (ll i = 1; i * i <= n; i++)
        {
                for (ll j = i; i * j <= n; j++)
                {
                        if (i == j)
                                sod[i * j]+=i;
                        else
                                sod[i * j] += (i+j);
                }
        }

        for (ll i = 1; i <= n; i++)
                cout << i << " --> " << sod[i] << endl;

        return 0;
}
```

Legendre's formula (Given p and n, find the largest x such that p^x divides n!).

- n! is multiplication of {1, 2, 3, 4, ...n}.
- How many numbers in {1, 2, 3, 4, ... n} are divisible by p?

→ Every p'th number is divisible by p in {1, 2, 3, 4, ... n}.

Therefore in n!, there are ⌊n/p⌋ numbers divisible by p. So we know that the value of x (largest power of p that divides n!) is at-least ⌊n/p⌋.

- Can x be larger than ⌊n/p⌋ ?

  → Yes, there may be numbers which are divisible by p^2, p^3, …
- How many numbers in {1, 2, 3, 4, ….. n} are divisible by p^2, p^3, …?

  → There are ⌊n/(p^2)⌋ numbers divisible by p^2 (Every p^2'th number would be divisible). Similarly, there are ⌊n/(p^3)⌋ numbers divisible by p^3 and so on.
- What is the largest possible value of x?

  → So the largest possible power is ⌊n/p⌋ + ⌊n/(p^2)⌋ + ⌊n/(p^3)⌋ + …… Note that we add only ⌊n/(p^2)⌋ only once (not twice) as one p is already considered by expression ⌊n/p⌋. Similarly, we consider ⌊n/(p^3)⌋ (not thrice).

Complexity: `O(log`$_p$`n)`

Code Snippet:

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

typedef long long ll;
typedef unsigned long long ull;
#define endl "\n"

// Using recursion
// ll largest_power(ll n, ll p)
// {
//     if (n == 0)
//         return 0;
//     return n / p + largest_power(n / p, p);
// }

ll largest_power(ll n, ll p)
{
        ll pw = 0;
        while (n)
        {
                n /= p;
                pw += n;
        }

        return pw;
}

int main()
{
        ll n, p;
        cin >> n >> p;
        cout << largest_power(n, p) << endl;

        return 0;
}
```

Related Problem:

1. Find the numbers under n which has odd number of divisors.

Solution: The number would be 1 to square root of n.

Code Snippet:

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

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

        cout << "Count of numbers till " << n << " which has odd
number of divisors: " << sqrt(n) << endl;
        cout << "The numbers are: ";
        for (int i = 1; i * i <= n; i++) // i * i <= n is same a
<= sqrt(n) but not safe because of double precision
                cout << i << " ";

        return 0;
}
```

2. Which number has exactly 3 divisors?

Solution: The square of a prime number has only exactly 3 divisors. According to prime fectorization if and only if $n=p^2$ only then `NOD(n)=(2+1)=3`, where p is the prime fector of n.

3. Which number has exactly 4 divisors?

Solution: The qube of a prime number or multiplication of 2 prime number has only 4 divisors. According to prime fectorization if $n=p^3$ or `n=p.q` only then `NOD(n)=(3+1)=4` or `NOD(n)=(1+1).(1+1)=(2.2)=4`.

4. Count trailing zeroes in factorial of a number.

Solution: A trailing zero is always produced by prime factors 2 and 5. So we have to count the number of 5s and 2s in the prime factors of n! and take the minimum as answer. If we observe, then we will see that the number of 5s is always less than or equal to the number os 2s. Hence, if we can count the number of 5s then our job is done. We can use ***Legendre's Formula*** to find the number of 5s in the prime fectors of n!.

Complexity: $O(\log_5 n)$

Code Snippet:

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;

typedef long long ll;
typedef unsigned long long ull;
#define endl "\n"

ll trailing_zeros(ll n)
{
        if (n == 0)
                return 0;
        return n / 5 + trailing_zeros(n / 5);
}

int main()
{
        ll n;
        cin >> n;
        cout << trailing_zeros(n) << endl;

        return 0;
}
```

5. Count Divisors of factorial.

Solutions:

Using Seive algorithm find the primes less then or equal to the number of factorial.

Apply the Legendre’s formula for all the primes found in previous step.

Complexity: `O(n log(log(n)))`

Code Snippet:

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

typedef long long ll;
typedef unsigned long long ull;
#define endl "\n"

const ll N = 100 + 10;
vector<bool> isPrime(N, true);
vector<ll> primes;

void seive()
{
        isPrime[0] = isPrime[1] = false;
        for (ll i = 2; i * i <= N; i++)
        {
                for (ll j = i + i; j <= N; j += i)
                {
                        isPrime[j] = false;
                }
        }

        for (ll i = 0; i < N; i++)
        {
                if (isPrime[i])
                        primes.push_back(i);
        }
}

int main()
{
        ll n;
        cin >> n;
        seive();

        ll nod = 1;
        for (ll i = 0; i < primes.size() && primes[i] <= n; i++)
        {
                ll p = 0, x = n;
                while (x)
                {
                        x = x / primes[i];
                        p += x;
                }
                p++;
                nod *= p;
        }

        cout << nod << endl;

        return 0;
}
```

Divisibility

Divisibility by 2: If the last digit contains the digit `0, 2, 4, 6, 8` then the number is divisible by 2.

Divisibility by 5: If the last digit contains the digit `0, 5` then the number is divisible by 5.

Divisibility by 3: If the digit-sum is divisible by `3` then the number is divisible by 3.

Divisibility by 9: If the digit-sum is divisible by `3` then the number is divisible by 9.

Digit-Sum Divisibility Property: As 3 or 9 divide base (10) then the remainder is 1. Because of this property that if we devide the base with any number and the remainder is 1 then for those number and base the digit-sum divisibility comes in.

For exampel if the base is 31 then,

31%2=1
31%3=1
31%5=1
31%6=1
31%10=1
31%15=1
31%30=1

So for 31 based numbers if the digit-sum is divisible by 2 then the number is also divisible by 2 also. Same goes for 3, 5, 6, 10, 15, 30.

Divisibility by 4: If the number formed by the last two digit of a number is divisible by 4 then the number is divisible by 4.

Beacuse every exponential number of 10 () except 1→(10^0) and 10→(10^1) is divisible by 4. So 100 is divisible by 4. Hence, every other number divisible by 100 is also divisible by 4 also. So if the remainder of a number divided by 100 is divided by 4 then the number is also divided by 4.

So for example 4124 = 41*100 + 24. Here, 41*100 is divisible by 4 as 100 is divisible by 4. Hence we have to check 24 is divisible by 4 or not.

Another exampel 51234 = 512*100 + 34. Here 512*100 is divisible by 4 as 100 is divisible by 4. Hence we have to check 34 is divisible by 4 or not.

Divisibility by 6: The number which is divisible by all the prime factor of 6 → (2, 3) is also divisible by 6.

Prime Factor Divisibility Property: If n is divided by all the prime factor of x, then n is divisible by x also.

For example all the number divisible by 30 is also divisible the prime factor of 30 → (2, 3, 5).

Divisibility by 11: Sum the odd positioned *(from right)* digit and substract the even positioned *(from right)* digit. If the result is divisible by the 11 then the number is divisible by 11 also.

10%11=10 or we can also say that 10%11=-1 (anti clockwise).

So, 1234%11 = $(1*10^3 + 2*10^2 + 3*10^1 + 4)$%11

$= 1*(-1)^3 + 2*(-1)^2 + 3*(-1)^1 + 4$

= -1 + 2 - 3 + 4

Hence, odd positioned *(from right)* digit should be added and even positioned *(from right)* digit should be substract. Then the absulute value of the result should be checked that if it is divisible by 11 or not.

For example 12347253, |3-5+2-7+4-3+2-1|=|-5|. If the result is divisible 11 then number is divisible 11.

Divisibility and Large Numbers

How to check if a number $a \le 10^{100000}$ is divisible by $b \le 10^9$?.

In this case we will enumerate from first to last *(from left)* digit and check for the modulo.

If a large number *abcde* and check whether it is divisible by *x* then,

I1 → (0*10)%x + a%x
I2 → (I1*10)%x + b%x
I3 → (I2*10)%x + c%x
I4 → (I3*10)%x + d%x
I5 → (I4*10)%x + e%x

Complexity: `O(n)`

Code Snippet:

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

typedef long long ll;
typedef unsigned long long ull;
#define endl "\n"

int main()
{
        string s;
        ll x;
        cin >> s >> x;

        ll rem = 0;
        for (int i = 0; i < s.length(); i++)
        {
                rem = (rem * 10) % x + (s[i] - '0') % x;
        }

        if (rem == 0)
                cout << "Divisible" << endl;
        else
                cout << "Not Divisible" << endl;

        return 0;
}
```

K Consecutive Divisibility
Problem: You are given an integer `n (1≤n≤100)`. Find an integer array $a_1, a_2, \ldots, a_n$ of length n such that for each subarray, the product of the elements of that subarray is divisible by the length of the subarray. All ai should be $\leq 10^9$.

Theorem: The product of every `k` consecutive number is divisible by `k`.

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Product of every 2 consecutive number is divisible by 2, because in every 2 consecutive number there is at least a number which is divisible by 2.

Product of every 3 consecutive number is divisible by 3, because in every 3 consecutive number there is at least a number which is divisible by 3.

Product of every 4 consecutive number is divisible by 4, because in every 4 consecutive number there is at least a number which is divisible by 4.

Product of every 5 consecutive number is divisible by 5, because in every 5 consecutive number there is at least a number which is divisible by 5.

Divisible by All:

Problem 1: Count the numbers between [L,R] which are divisible by m.

Solution: `⌊R/m⌋ - ⌊L/m⌋`

Problem 2: How to check if a number x is divisible by both a and b?

Solution: If x is divisible by lcm(x,y), then x is divisible by both x and y.

Problem 3: How to check if a number x is divisible by all a, b and c?

Solution: If x is divisible by lcm(x,y,z), then x is divisible by all x, y and z.

Problem 4: Counts numbers in between [L,R] which are divisible by all array elements of the given array of size n.

Solution: `⌊R/lcm(array elements)⌋ - ⌊L/lcm(array elements)⌋`

Problem 5: Find numbers in between [L,R] which are divisible by all array elements of the given array of size n.

Solution:

1. Calculate the LCM of all the elements of given arr[]
2. Now, check the **LCM** for these conditions:
    1. If **(LCM < L and LCM*2 > R)**, then print -1.

2. If **(LCM > R)**, then print -1.
3. Now, take the nearest value of **L** (between **L** to **R**) which is divisible by the **LCM**, say **i**.
4. Now, start printing **i** and increment it by **LCM** every time after printing, until it becomes greater than **R**.

Pair Sums Divisibility:

Given an array and positive integer k, count the total number of pairs in the array whose sum is divisible by k.

Solution: Modulo every numbers in the array by k and keep the frequency of those numbers. Now the elements {a[i], k-a[i]} and {a[i]==0} will make pairs (a[i] will make pair with k-a[i] and the elements a[i]==0 s will make pair among themselves). Keep a flag whether the elements were used in pairs or not.

Complexity: `O(nlog(n))` as map is used.

Code Snippet:

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

typedef long long ll;
typedef unsigned long long ull;
#define endl "\n"

int main()
{
        ll k, n;
        cin >> n >> k;
        vector<ll> arr(n);
        map<ll, ll> frq;
        map<ll, bool> flg;

        for (ll i = 0; i < n; i++)
                cin >> arr[i], arr[i] %= k, frq[arr[i]]++,
flg[arr[i]] = true;

        ll cnt = 0;
        for (ll i = 0; i < n; i++)
        {
                if (flg[arr[i]])
                {
                        // arr[i] == k - arr[i] means they can m
pair among themselvs as their sum is divisible by k
                        // arr[i] == 0 means the numbers are
individually divisible by k. So they can make pair among themsel
                        if (arr[i] == k - arr[i] || arr[i] == 0)
                        {
                                ll x = frq[arr[i]] - 1;
```

```
                                        cnt += (x * (x + 1)) / 2;
                                }
                                else
                                        cnt += frq[arr[i]] * frq[k - arr[i]];

                                flg[arr[i]] = flg[k - arr[i]] = false;
                        }
        }

        cout << cnt << endl;

        return 0;
}
```

Subarray Sums Divisibility:

Given an array of positive and/or negative integers and a value k, count of all sub-arrays whose sum is divisible by K.

Solution:

Let there be a subarray (i, j) whose sum is divisible by k

`sum(i, j) = sum(0, j) - sum(0, i-1)`

Sum for any subarray can be written as `q*k + rem` where `q` is a quotient and `rem` is remainder

Thus,

`sum(i, j) = (q1 * k + rem1) - (q2 * k + rem2)`
`sum(i, j) = (q1 - q2)k + rem1-rem2`

We see, for `sum(i, j)` i.e. for sum of any subarray to be divisible by `k`, the RHS should also be divisible by `k`.

`(q1 - q2)k` is obviously divisible by `k`, for `(rem1-rem2)` to follow the same,

`rem1 = rem2` where,

`rem1 = sum of subarray (0, j) % k`
`rem2 = sum of subarray (0, i-1) % k`

Complexity: `O(n)`

Code Snippet:

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

typedef long long ll;
typedef unsigned long long ull;

int main()
{
        ll k, n;
        bool ok, flg;
```

```
	cin >> n >> k;
	vector<ll> arr(n);
	map<ll, ll> mod;
	for (ll i = 0; i < n; i++)
		cin >> arr[i];

	ll sum = 0;
	for (ll i = 0; i < n; i++)
		sum += arr[i], mod[((sum % k) + k) % k]++;

	ll cnt = 0;
	cnt += mod[0];
	for (auto it : mod)
	{
		if (it.second > 1)
			cnt += ((it.second - 1) * (it.second)) / 2;
	}
	cout << cnt << endl;

	return 0;
}
```

GCD and LCM

GCD: The greatest common divisor (GCD) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers. For two integers *x, y,* the greatest common divisor of *x* and *y* is denoted *gcd(x,y)*. For example, the GCD of 8 and 12 is 4, that is, gcd(8,12)=4.

Properties:

- Every common divisor of `a` and `b` is a divisor of `gcd(a,b)`
- The gcd is a commutative function: `gcd(a,b) = gcd(b,a)`
- The gcd is an associative function: `gcd(a,gcd(b,c)) = gcd(gcd(a,b),c)`
- The gcd of three numbers can be computed as `gcd(a,b,c) = gcd(gcd(a,b),c)`, or in some different way by applying commutativity and associativity. This can be extended to any number of numbers.
- `gcd(a,b) = gcd(a−b,b)`

In C++ builtin function `__gcd(a, b)` used to find the gcd of a and b.
Complexity: `O(log(min(a,b)))`
Code Snippet:

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

typedef long long ll;
typedef unsigned long long ull;

ll gcd(ll a, ll b)
{
        if (a == 0)
                return b;
        return gcd(b % a, a);
}

int main()
{
        ll a, b;

        cin >> a >> b;
        cout << __gcd(a, b) << endl;
        cout << gcd(a, b) << endl;

        return 0;
}
```

LCM: The least common multiple, lowest common multiple, or smallest common multiple of two integers *a* and *b*, usually denoted by *lcm(a,b)*, is the smallest positive integer that is divisible by both *a* and *b*.

Suppose

$a=2^3.3^4.5^1$

$b=2^2.3^5.7^2$

So, $gcd(a,b) = 2^{min(2,3)}.3^{min(4,5)}.5^{min(1,0)}.7^{min(0,2)} = 2^2.3^4.5^0.7^0$

$lcm(a,b) = 2^{max(2,3)}.3^{max(4,5)}.5^{max(1,0)}.7^{max(0,2)} = 2^3.3^5.5^1.7^2$

Therefore, $gcd(a,b).lcm(a,b) = (2^{min(2,3)}.3^{min(4,5)}.5^{min(1,0)}.7^{min(0,2)}) . (2^{max(2,3)}.3^{max(4,5)}.5^{max(1,0)}.7^{max(0,2)})$

$\Rightarrow gcd(a,b).lcm(a,b) = 2^{2+3}.3^{4+5}.5^{0+1}.7^{0+2}$

$\Rightarrow gcd(a,b).lcm(a,b) = (2^3.3^4.5^1).(2^2.3^5.7^2)$

$\Rightarrow gcd(a,b).lcm(a,b) = a.b$

$\Rightarrow lcm(a,b) = (a.b)/gcd(a,b)$

Complexity: $O(\log(min(a,b)))$

Code Snippet:

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

typedef long long ll;
typedef unsigned long long ull;

```

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

int main()
{
        ll a, b;

        cin >> a >> b;
        cout << (a * b) / __gcd(a, b) << endl;
        cout << (a * b) / gcd(a, b) << endl;
        // ((a / gcd(a, b)) * b); //avoids integer overflow

        return 0;
}
```

Euclid's Algorithm:

`GCD(a,b)=GCD(b,a%b)`

Proof:

The Euclidean Algorithm works on the principle `GCD(a,b)=GCD(b,a%b)`. If we can prove this, then there will be no doubt about the algorithm.

Let `g=GCD(a,b)` and `a=k×b+r`, where `k` is a non-negative integer and `r` is the remainder. Since `g` divides `a`, `g` also divides `k×b+r`. Since `g` divides `b`, `g` also divides `k×b`. Therefore, `g` must divide `r` otherwise `k×b+r` won't be divisible. So we proved that `g` divides `b` and `r`.

Now lets say we have `g′=gcd(b,r)`. Since `g′` divides both `b` and `r`, it will divide `k×b+r`. Therefore, `g′` will divide `a`.

Now, can g and g′ be two different numbers? No. We will prove this using contradiction.

Let's say that `g>g′`. We know that `g` divides both `b` and `r`. So how can `gcd(b,r)` be `g′` when we have a number greater than `g′` that divides both `b` and `r`? So `g` cannot be greater than `g′`. Using the same logic, we find there is a contradiction when `g<g`′.

Therefore, the only possibility left is `g=g′`.

∴`GCD(a,b) = GCD(b,r) = GCD(b,a%b)`.

Co-primes:

Two integers a and b are coprime, relatively prime or mutually prime if the only positive integer that is a divisor of both of them is 1.

If `GCD(a,b)=1` then `a` and `b` are co-prime.

GCD and LCM of an Array using Prime Factor:

1. Find the primes using sieve.
2. Track (min/max) the number of each prime factor for the entire array.
   1) For ***GCD*** track ***minimum*** number of each prime factor.
   2) For ***LCM*** track ***maximum*** number of each prime factor.
3. Multiply the each prime factor tracked (min/max) number of times.
   1) For ***GCD*** multiply ***minimum*** number of times.
   2) For ***LCM*** multiply ***maximum*** number of times.

Related Problem:

1. Find the GCD of large number.

Solution: Suppose two numbers a and b. a is large enough to store in long long but b is so large that we have use string to store it. Hence, we will try to make it less than or equal to ‘a’ by taking it’s modulo with ‘a’. We will traverse the b and calculate the modulo.

Complexity: `O(min(a,b))`

Code Snippet:

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

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

        cout << "Count of numbers till " << n << " which has odd
number of divisors: " << sqrt(n) << endl;
        cout << "The numbers are: ";
        for (int i = 1; i * i <= n; i++) // i * i <= n is same a
<= sqrt(n) but not safe because of double precision
                cout << i << " ";

        return 0;
}
```

2. Find the LCM of an array.

Solution: Perform the lcm operation for every number of the array.

Complexity: `O(n)`

Code Snippet:

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

typedef long long ll;
typedef unsigned long long ull;
#define endl "\n"

```

```
ll gcd(ll a, ll b)
{
        if (a == 0)
                return b;

        return gcd(b % a, a);
}

int main()
{
        ll n;
        cin >> n;
        vector<ll> arr(n);

        for (ll i = 0; i < n; i++)
                cin >> arr[i];

        ll lcm = arr[0];
        for (ll i = 1; i < n; i++)
        {
                lcm = (arr[i] / gcd(arr[i], lcm)) * lcm; // formula
lcm(a,b) = (a*b)/gcd(a,b)
        }
        cout << lcm << endl;

        return 0;
}
```

3. Find the modular LCM of an array.

Solution: Find the primes using sieve. Then track the maximum number of each prime factor for the entire array. Last multiply the each prime factor maximum number of times.

Complexity: `O(n`)

Code Snippet:

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

typedef long long ll;
typedef unsigned long long ull;
#define endl "\n"

const ll N = 1e6 + 10;
const ll mod = 1e9 + 7;

vector<bool> isPrime(N, true);
vector<ll> primes;
map<ll, ll> maxFactor;
void sieve()
{
        isPrime[0] = isPrime[1] = false;
        for (ll i = 1; i * i <= N; i++)
        {
                if (isPrime[i])
                {
                        for (ll j = i + i; j <= N; j += i)
                                isPrime[j] = false;
                }
        }

        for (ll i = 0; i < N; i++)
                if (isPrime[i])
                        maxFactor[it] = max(maxFactor[it], c);
}

int main()
{
        sieve();

        ll n;
        cin >> n;
        vector<ll> arr(n);

        for (ll i = 0; i < n; i++)
                cin >> arr[i];

        for (ll i = 0; i < n; i++)
        {
                ll x = arr[i];
                for (auto it : primes)
                {
                        ll c = 0;
                        while (x % it == 0 && x >= it)
                                c++, x /= it;

                        maxFactor[it] = max(maxFactor[it], c);
                }
        }

        ll lcm = 1;
        for (auto it : maxFactor)
        {
                ll c = it.second;
                if (c)
                {
                        for (ll i = 0; i < c; i++)
                        {
                                lcm = (lcm * it.first) % mod;
                        }
                }
        }
        cout << lcm << endl;
        return 0;
}
```

Digital Root

The digital root (also repeated digital sum) of a natural number in a given radix is the (single digit) value obtained by an iterative process of summing digits, on each iteration using the result from the previous iteration to compute a digit sum. The process continues until a single-digit number is reached. For example, in base 10, the digital root of the number 12345 is 6 because the sum of the digits in the number is 1 + 2 + 3 + 4 + 5 = 15, then the addition process is repeated again for the resulting number 15, so that the sum of 1 + 5 equals 6, which is the digital root of that number. In base 10, this is equivalent to taking the remainder upon division by 9 (except when the digital root is 9, where the remainder upon division by 9 will be 0), which allows it to be used as a divisibility rule.

Propertise:

- If we multiply any number by 9, the digital root will always be 9.
- Adding 9 to a number does not change the digital root of that number.
- If we divide any number by 9, the digital root of that number will be the remainder.

Formula:

DR`(x)=(x−1) mod 9+1`

Combinatorics

Permutation

Combination

Data Structures

Algorithms

Bit Manupulation

Hashing

Hashing is a technique used in data structures to store and retrieve data efficiently. It involves mapping data to a specific index in a hash table using a hash function that enables fast retrieval of information based on its key. The important thing about hashing is that the operation like ***store, fetch*** can be done in ***O(1)*** time on average.

The size of the hash table (***arrary or vector***) can be declared as,

	int	**bool**
Local (inside main function)	10^6	10^7
Global	10^7	10^8

For greater value, ***map*** and ***unordered map*** can be used as hash table.
In map the operations like ***store*** and ***fetch*** can be done in ***O(log(n))***, but in

unordered map they can be done in ***O(1)*** most of the time but in worst case it can be ***O(n)***. unordered map use bin or bracket method. If the values are stored in the same bracket or bin because of internal hashing function then it would take little bit more time than others because of collisions.

Example:
values: 2, 5, 15, 16, 28, 18, 48, 78, 68, 59
    and let the bracket size be 10, tough in real life it is much bigger than this and little bit complecated.

```
bracket
0    →
1    →
2    →    2,
3    →
4    →
5    →    5, 15
6    →    16,
7    →
8    → 18, 28, 48, 68, 78,
9  →      59
```

2-3 Pointer

Dutch National Flag
    it's used to sort an array consisting of three distinct elements (usually represented as 0, 1, and 2) in linear time.

Statement:
    Given an array containing only 0s, 1s, and 2s, sort the array in ascending order.

Approach:
1. Use three pointers:
    - \``low`\` to keep track of the position of the next 0.
    - \``mid`\` to traverse the array.
    - \``high`\` to keep track of the position of the next 2.
2. The idea is to move 0s to the beginning, 2s to the end, and 1s in the middle.
3. Iterate over the array with the `mid` pointer:
    - If `nums[mid]` is 0, swap it with `nums[low]` and increment both `low` and `mid`.

- If `nums[mid]` is 1, just move `mid` ahead.
- If `nums[mid]` is 2, swap it with `nums[high]` and decrement `high` (keep mid unchanged since the new element at `mid` might still need sorting).

Complexity: `O(n)`

Code Snippet:

```
void sortColors(vector<int> &nums)
{
        int n = nums.size();
        int low = 0, mid = 0, high = n - 1;

        while (mid <= high)
        {
                if (nums[mid] == 0)
                {
                        swap(nums[mid], nums[low]);
                        low++, mid++;
                }
                else if (nums[mid] == 2)
                {
                        swap(nums[mid], nums[high]);
                        high--;
                }
                else if (nums[mid] == 1)
                        mid++;
        }
}
```

Array Analysis

Moore Voting Algorithm

Moore's Voting Algorithm is a simple and efficient algorithm used to find the majority element in an array.

Statement:

Given an array of size n, find the majority element. The majority element is the element that appears more than n/2 times. Assume that the array is non-empty and the majority element always exists in the array.

Approach:

1. Candidate Selection
   - Initialize a candidate and a count variable.

- Traverse through the array:
    - If count is 0, update candidate to the current element and set count = 1.
    - If the current element is the same as candidate, increment count.
    - If the current element is different, decrement count.

2. Verification:
    - After determining the candidate, traverse the array again to confirm that the candidate occurs more than n/2 times.

Complexity: `O(n)`

Code Snippet:

```
int findMajorityElement(vector<int>& nums) {
        int candidate = 0;
        int count = 0;

        // Phase 1: Find a candidate
        for (int num : nums) {
                if (count == 0) {
                        candidate = num;
                }
                count += (num == candidate) ? 1 : -1;
        }

        // Phase 2: Verify the candidate
        count = 0;
        for (int num : nums) {
                if (num == candidate) {
                        count++;
                }
        }

        // If the candidate is the majority element, return it;
otherwise, return -1
        return (count > nums.size() / 2) ? candidate : -1;
}
```

Kadane's Algorithm

Kadane's algorithm is used to find the maximum subarray sum in a given array. It's a simple and efficient algorithm that can be implemented in linear time complexity.

Statement:

Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Approach:

1. Initialization:

Initialize two variables:

- `sum` to store the maximum sum of the subarray ending at the current index.
- `maxSum` to store the maximum sum found so far across all subarrays.

2. Iteration:

- Traverse the array from the first to the last element.
- For each element `nums[i]`, sumup the element to the `sum`.
- If sum is greater than `maxSum`, update `maxSum`.
- If `sum<0` then update the `sum` as 0.

3. Result:

After the loop ends, `maxSum` will hold the maximum sum of the contiguous subarray.

```
int maxSubArray(vector<int> &nums)
{
        int maxSum = INT_MIN, sum = 0, n = nums.size();
        for (int i = 0; i < n; i++)
        {
                sum += nums[i];
                maxSum = max(maxSum, sum);
                if (sum < 0)
                        sum = 0;
        }

        // Alternative
        // for (int num : nums) {
        //     sum = max(sum + num, num);
        //     maxSum = max(maxSum, sum);
        // }

        return maxSum;
}
```