Date......................

PRIMALITY TEST

- It is to determine if a given number is a prime number or not.

NAIVE APPROACH

```
bool isPrime(int n) {
    if(n == 1)
        return false;

    for(int i=2; i<n/2; i++)
        if(n%i == 0)
            return false;

    return true;
}
```

TIME COMPLEXITY

O(N)

BETTER APPROACH

- All divisors of a number N occur in pairs of (a,b) where

$\boxed{a*b = N}$

For example, factors of 12 = 1, 2, 3, 4, 6, 12

Pairs ⇒ (1,12), (2,6), (3,4)

Date.....................

STATEMENT: For a divisor pair(a,b) one of them lies below SQRT(N) and one lies above SQRT(N)

CASES:

- Both A and B are below SQRT(N)

$a < sqrt(N)$ $b < sqrt(N)$

But then $a*b < N$

Hence, this is NOT TRUE.

- Both A and B are above SQRT(N)

$a > sqrt(N)$ $b > sqrt(N)$

But then, $a*b > N$

Hence, this NOT TRUE

- One is above SQRT(N), one is below SQRT(N)

CASE A)

$b < sqrt(N) \rightarrow 1 < sqrt(N)/b$

$a = sqrt(N) * (1+x)$

Hence $a > sqrt(N)$

and vice-versa

THIS IS TRUE

Spiral

Date.......................

~~for~~ ~~int~~ bool isPrime {

Time Complexity : O(SQRT(N))

if(N==1)
    return false;

for(int i=2; i*i ~~<~~ <= N; i++)
    if(N%i==0)
        ~~return true~~; return false;

return ~~false~~ true;
}

Date..................

SIEVE OF ERATOSTHENES

Preprocessing Time : O(log(log N))
Answer Query: O(1)
Auxiliary space : O(N)

```
bool num[101] {0};

num[0] = num[1] = 1;

for (int i=2; i*i<N; i++) {
        if( ~~bool~~ num[i] ==0)
                for (int j = i*i ; j < 101; j+=i)
                        num[j] = 1;
}
```

MARKS ALL PRIME NUMBERS As '0' and COMPOSITE As '1'

PRIME FACTORIZATION

TIME COMPLEXITY : O(N)

10^9+7 → Prime

```
void findPrimefact(int N) {

        for(int i=2; i<=N; i++){
            if (N%i ==0){
                int count =0;
                while( N%i ==0) {
                    count ++;
                    N /=i;
                }
            cout << i << "^" << count << endl;
        }
}
```

Date.....................

OPTIMIZED APPROACH:

If N is a composite number, then there is at least 1 prime divisor of N below sqrt(N).

TIME COMPLEXITY : O(SQRT(N))

```
for(int i=2; i*i<=N; i++)
    if(N%i==0) {
        int cnt = 0;
        while (N%i==0)
            cnt++, N /= i;
        cout << i << " ^" << cnt << endl;
    }

if(N>1)
    cout<<N<<" ^"<<1;
```

Date..................

BINARY EXPONENTIATION

- Used to calculate a^n in $\log(n)$ time.

NAIVE APPROACH:

```
int power (int n, ~~base~~ int base) {

    int res = 1;                              Time Complexity : O(N)

    for (int i=0; i<n; i++)
        res = ~~res~~ res * base;

    return res;
}
```

OPTIMAL APPROACH:

```
int power(int a, ~~int n~~ int base) {        Time Complexity: O(log(N))

    int res = 1;
    while( ~~base~~ a ) {
        if ( ~~base~~ a % 2) {
            res *= a, n--;

        else
            a = a*a, n/=2;
    }
    return res;
}
```

Spiral

Date.....................

PRIME FACTORIZATION USING SIEVE

```
int num [51];

for (int i = 0; i < 51; i++)
    num[i] = -1;

for(int i=2; i<51; i++)  ⟶  for(int i=2; i*i <= 50; i++)
  if(num[i]==-1)
    for (int j= i; j< 51; j++) ⟶ for(int j = i*i ; j<51; j++)
      if (num[j] == -1;
        num[j] = i;
}
```

MATRIX EXPONENTIATION

- Given a matrix A, and an integer N, calculate A^N

Time Complexity = $O(M^3 * N)$ ⟶ Dimension

Optimized Complexity = O(M^3 * logN)