## Week 38: Advanced Number Theory – Primality Testing & Integer Factorization

**Topics:** - Miller-Rabin Primality Test (Deterministic for 64-bit integers) - Pollard's Rho Algorithm for Integer Factorization - Fermat's Factorization Method - Modular Exponentiation Review - Applications: Cryptography (RSA), Large Number Factoring, Competitive Problems

**Weekly Tips:** - Miller-Rabin is a probabilistic test but can be made deterministic for 64-bit range. - Pollard's Rho is efficient for factoring numbers up to ~1e18. - Always use modular multiplication to avoid overflow when working with large numbers. - Factorization + primality testing is common in hard number theory problems. - Combine trial division for small primes with Pollard's Rho for efficiency.

**Problem 1: Miller-Rabin Primality Test Link:** Primality Test Reference **Difficulty:** Advanced

**C++ Solution with Explanation Comments:**

```
#include <bits/stdc++.h>
using namespace std;
using u128 = unsigned __int128;
using u64 = unsigned long long;

u64 modmul(u64 a, u64 b, u64 m){
    return (u128)a*b % m;
}

u64 modpow(u64 a, u64 d, u64 m){
    u64 r=1;
    while(d){
        if(d&1) r=modmul(r,a,m);
        a=modmul(a,a,m);
        d>>=1;
    }
    return r;
}

bool isPrime(u64 n){
    if(n<2) return false;
    for(u64 p:{2,3,5,7,11,13,17,19,23,29,31,37}){
        if(n%p==0) return n==p;
    }
    u64 d=n-1,s=0;
    while((d&1)==0){ d>>=1; s++; }
    for(u64 a:{2ULL,325ULL,9375ULL,28178ULL,450775ULL,9780504ULL,1795265022ULL})
{
        if(a%n==0) continue;
        u64 x=modpow(a,d,n);
        if(x==1 || x==n-1) continue;
```

```
        bool comp=true;
        for(u64 r=1;r<s;r++){
            x=modmul(x,x,n);
            if(x==n-1){ comp=false; break; }
        }
        if(comp) return false;
    }
    return true;
}

int main(){
    u64 n; cin>>n;
    cout<<(isPrime(n)?"Prime":"Composite")<<endl;
}
```

**Explanation Comments:** - Uses modular exponentiation for fast power checks. - Deterministic bases cover all 64-bit integers. - O(log n) per test, very efficient.

**Problem 2: Pollard's Rho Factorization Link:** Pollard's Rho Reference **Difficulty:** Advanced

**C++ Solution with Explanation Comments:**

```
#include <bits/stdc++.h>
using namespace std;
using u64 = unsigned long long;
using u128 = __uint128_t;

u64 modmul(u64 a,u64 b,u64 m){ return (u128)a*b % m; }
u64 modpow(u64 a,u64 d,u64 m){
    u64 r=1;
    while(d){ if(d&1) r=modmul(r,a,m); a=modmul(a,a,m); d>>=1; }
    return r;
}

u64 f(u64 x,u64 c,u64 n){ return (modmul(x,x,n)+c)%n; }

u64 rho(u64 n){
    if(n%2==0) return 2;
    mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
    while(true){
        u64 x=rng()%(n-2)+2, y=x, c=rng()%(n-1)+1, d=1;
        while(d==1){
            x=f(x,c,n);
            y=f(f(y,c,n),c,n);
            d=gcd<u64>(x>y?x-y:y-x,n);
            if(d==n) break;
```

```
        }
        if(d>1 && d<n) return d;
    }
}

int main(){
    u64 n; cin>>n;
    if(n==1){ cout<<1; return 0; }
    if(n%2==0){ cout<<2; return 0; }
    u64 factor=rho(n);
    cout<<factor<<endl;
}
```

**Explanation Comments:** - Uses random function `f(x) = x^2 + c mod n`. - Finds nontrivial gcd with `n` to get a factor. - Works efficiently for large composites (~1e18).

**Applications:** - Fast primality checks for cryptographic problems. - Factoring numbers in competitive problems with large constraints. - Building blocks for RSA-like cryptographic challenges.

---

**End of Week 38** - Learn Miller-Rabin and Pollard's Rho for ACM-ICPC problems involving primes and factors. - Practice both primality testing and factorization on large numbers.