Mathematics

Logarithm, Exponentiation, or Power

- A nice feature of the logarithmic function is that it can be used to count the number
  of digits of a given decimal a. This formula `(int)floor(1 + log10((double)a))`

Number Theory

Sieve of Eratosthenes: Generating List of Prime Numbers

```
typedef long long ll;

ll _sieve_size;
bitset<10000010> bs; // 10^7 is the rough limit
vll p; // compact list of primes

void sieve(ll upperbound) { // range = [0..upperbound]
  _sieve_size = upperbound+1; // to include upperbound
  bs.set(); // all 1s
  bs[0] = bs[1] = 0; // except index 0+1
  for (ll i = 2; i < _sieve_size; ++i) if (bs[i]) {
    // cross out multiples of i starting from i*i
    for (ll j = i*i; j < _sieve_size; j += i) bs[j] = 0;
    p.push_back(i); // add prime i to the list
    }
}

bool isPrime(ll N) { // good enough prime test
  if (N < _sieve_size) return bs[N]; // O(1) for small primes
  for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N; ++i)
    if (N%p[i] == 0)
      return false;
  return true; // slow if N = large prime
} // note: only guaranteed to work for N <= (last prime in vll p)^2
// inside int main()
sieve(10000000); // up to 10^7 (<1s)

printf("%d\n", isPrime((1LL<<31)-1)); // 8th Mersenne prime
printf("%d\n", isPrime(136117223861LL)); // 104729*1299709
```

Functions Involving Prime Factors

numPF(N): Count the number of prime factors of integer N.

```
int numPF(ll N) {
  int ans = 0;
  for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i)
    while (N%p[i] == 0) { N /= p[i]; ++ans; }
  return ans + (N != 1);
}
```

numDiv(N): Count the number of divisors of integer N.

```
int numDiv(ll N) {
  int ans = 1; // start from ans = 1
  for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i) {
    int power = 0; // count the power
    while (N%p[i] == 0) { N /= p[i]; ++power; }
    ans *= power+1; // follow the formula
  }
  return (N != 1) ? 2*ans : ans; // last factor = N^1
}
```

EulerPhi(N): Count the number of positive integers < N that are relatively prime to N

- Recall: Two integers a and b are said to be relatively prime (or coprime) if gcd(a, b) = 1

A better algorithm is the Euler's Phi (Totient) function

```
ll EulerPhi(ll N) {
  ll ans = N; // start from ans = N
  for (int i = 0; (i < (int)p.size()) && (p[i]*p[i] <= N); ++i) {
    if (N%p[i] == 0) ans -= ans/p[i]; // count unique
    while (N%p[i] == 0) N /= p[i]; // prime factor
  }
  if (N != 1) ans -= ans/N; // last factor
  return ans;
}
```

Modified Sieve

If the number of different prime factors has to be determined for many (or a range of) integers

```
int numDiffPFarr[MAX_N+10] = {0}; // e.g., MAX_N = 10^7
  for (int i = 2; i <= MAX_N; ++i)
    if (numDiffPFarr[i] == 0) // i is a prime number
```

```
    for (int j = i; j <= MAX_N; j += i)
      ++numDiffPFarr[j]; // j is a multiple of i
```

Similarly, this is the modified sieve code to compute the Euler Totient function:

```
int EulerPhi[MAX_N+10];
for (int i = 1; i <= MAX_N; ++i) EulerPhi[i] = i;
for (int i = 2; i <= MAX_N; ++i)
  if (EulerPhi[i] == i) // i is a prime number
    for (int j = i; j <= MAX_N; j += i)
      EulerPhi[j] = (EulerPhi[j]/i) * (i-1);
```

Greatest Common Divisor & Least Common Multiple

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

```
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a%b); }
int lcm(int a, int b) { return a / gcd(a, b) * b; }
```

Modular Arithmetic

The following are true involving modular arithmetic:

- (a + b) % m = ((a % m) + (b % m)) % m
- (a − b) % m = ((a % m) − (b % m)) % m
- (a × b) % m = ((a % m) × (b % m)) % m

Extended Euclidean Algorithm

Euclidean algorithm can also computes the coefficients of B´ezout identity $ax + by = gcd(a,b)$.

```
int extEuclid(int a, int b, int &x, int &y) { // pass x and y by ref
  int xx = y = 0;
  int yy = x = 1;
  while (b) { // repeats until b == 0
    int q = a/b;
    int t = b; b = a%b; a = t;
    t = xx; xx = x-q*xx; x = t;
    t = yy; yy = y-q*yy; y = t;
  }
  return a; // returns gcd(a, b)
}
```

Modular Multiplicative Inverse with Extended Euclidean Algorithm

compute x such that b × x = 1 (mod m).

```
int mod(int a, int m) { // returns a (mod m)
  return ((a%m) + m) % m; // ensure positive answer
}

int modInverse(int b, int m) { // returns b^(-1) (mod m)
  int x, y;
  int d = extEuclid(b, m, x, y); // to get b*x + m*y == d
  if (d != 1) return -1; // to indicate failure
  // b*x + m*y == 1, now apply (mod m) to get b*x == 1 (mod m)
  return mod(x, m);
}
```

Combinatorics

Fibonacci Numbers

```
int fib(int n){
        int f[n + 2];
        int i;
        f[0] = 0;
        f[1] = 1;
        for (i = 2; i <= n; i++) {
            f[i] = f[i - 1] + f[i - 2];
        }
        return f[n];
    }
};
```

Binomial Coefficients

We can compute a single (exact) value of C(n, k) with this formula: C(n, k) = n!/(n−k)! ×k!

C(n, 0) = C(n, n) = 1 // base cases.
C(n, k) = C(n-1, k-1) + C(n-1, k) // take or ignore an item, n > k > 0.

Catalan Numbers

```
ll Cat[MAX_N];
// inside int main()
Cat[0] = 1;
for (int n = 0; n < MAX_N-1; ++n) // O(MAX_N log p)
```

```
  Cat[n+1] = ((4*n+2)%p * Cat[n]%p * inv(n+2)) % p;
cout << Cat[100000] << "\n";
```

1. Cat(n) counts the number of distinct binary trees with n vertices,

Probability Theory

problems involving probability are either solvable with:

- Closed-form formula has to derive the require formula.
- Exploration of the search