CodeForces Columbia SHP Algorithms Group

- **Message from Christian:** please join the following group: https://codeforces.com/group/lfDmo9iEr5

Introduction to Algorithms
Science Honors Program (SHP)
Session 7

Innokentiy, Eric, and Christian

Saturday, April 13, 2024

Innokentiy Kaurov (Co-instructor)

- 2023 International Olympiad in Informatics (IOI) Silver
- ICPC North America Championship in May 2024

Eric Yuang Shao (Co-instructor)

- 2023 International Physics Olympiad (IPhO) Bronze
- 2023 Putnam Competition Honorable Mention
- ICPC North America Championship in May 2024

Slide deck in github

- You may get to the link by:
    - https://github.com/yongwhan/
    - => yongwhan.github.io
    - => columbia
    - => shp
    - => session 7 slide

Overview

- **Combinatorics**
- Break (5-minute)
- **Modular Arithmetic and Number Theory**

Now, let's cover:

- **Combinatorics AKA Counting**
  - Binomial Coefficients and bijection techniques
  - Stars and Bars
  - Counting paths
  - Inclusion-Exclusion principle

Binomial Coefficients: Formulas

$$(a+b)^n = \binom{n}{0}a^n + \binom{n}{1}a^{n-1}b + \binom{n}{2}a^{n-2}b^2 + \cdots + \binom{n}{k}a^{n-k}b^k + \cdots + \binom{n}{n}b^n$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

For $n < k$, $\binom{n}{k} = 0$.

Proving combinatorial identities

1. Algebra

2. Bijection

Binomial Coefficients: Properties

$$\binom{n}{k} = \binom{n}{n-k}$$

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$$

Pascal's identity

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Binomial Coefficients: Properties

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

$$\sum_{m=0}^{n} \binom{m}{k} = \binom{n+1}{k+1}$$

Binomial Coefficients: Properties

For $n \geq 1$,

$$\binom{n}{0} + \binom{n}{2} + \cdots = \binom{n}{1} + \binom{n}{3} + \cdots = 2^{n-1}$$

Binomial Coefficients: Properties

$$1\binom{n}{1} + 2\binom{n}{2} + \cdots + n\binom{n}{n} = n2^{n-1}$$

Binomial Coefficients: Properties

$$\binom{n}{0}^2 + \binom{n}{1}^2 + \cdots + \binom{n}{n}^2 = \binom{2n}{n}$$

Pascal's identity revisited

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Pascal's Triangle: Implementation

```
const int maxn = ...;
int C[maxn + 1][maxn + 1];
C[0][0] = 1;
for (int n = 1; n <= maxn; ++n) {
    C[n][0] = C[n][n] = 1;
    for (int k = 1; k < n; ++k)
        C[n][k] = C[n - 1][k - 1] + C[n - 1][k];
}
```

Applications of binomial coefficients

Count the number of integer solutions $(x_1, x_2, \dots, x_k)$ to the equation

$$x_1 + x_2 + \cdots + x_k = n,$$

with $x_i \geq 0$ for all i.

Solution

Choose places for *k-1* separator bars.

$$\binom{n+k-1}{n}$$

Can we generalize this?

Count the number of integer solutions $(x_1, x_2, \ldots, x_k)$ to the equation

$$x_1 + x_2 + \cdots + x_k = n,$$

with $x_i \geq a_i$ for all i.

$$(x_1' + a_1) + (x_2' + a_2) + \cdots + (x_k' + a_k) = n$$

New problem:

$$x_1' + x_2' + \cdots + x_k' = n - \sum_{i=1}^{k} a_i,$$

subject to $x_i' \geq 0$.

Apply Stars and Bars:

$$\binom{\left(n - \sum_{i=1}^{k} a_i\right) + k - 1}{n - \sum_{i=1}^{k} a_i}$$

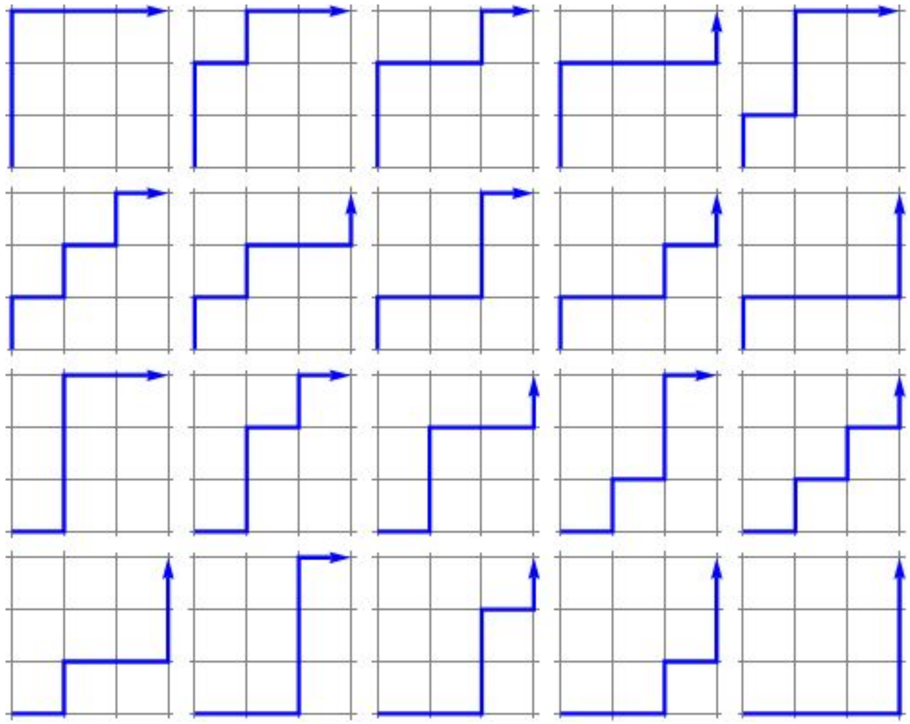
Counting paths

Paths in a grid

For a grid with *n* edges in each column and *m* edges in each row, how many paths are there from the bottom right cell to the upper left cell?

You can only go right or up

Example: $n=3$, $m=3$.

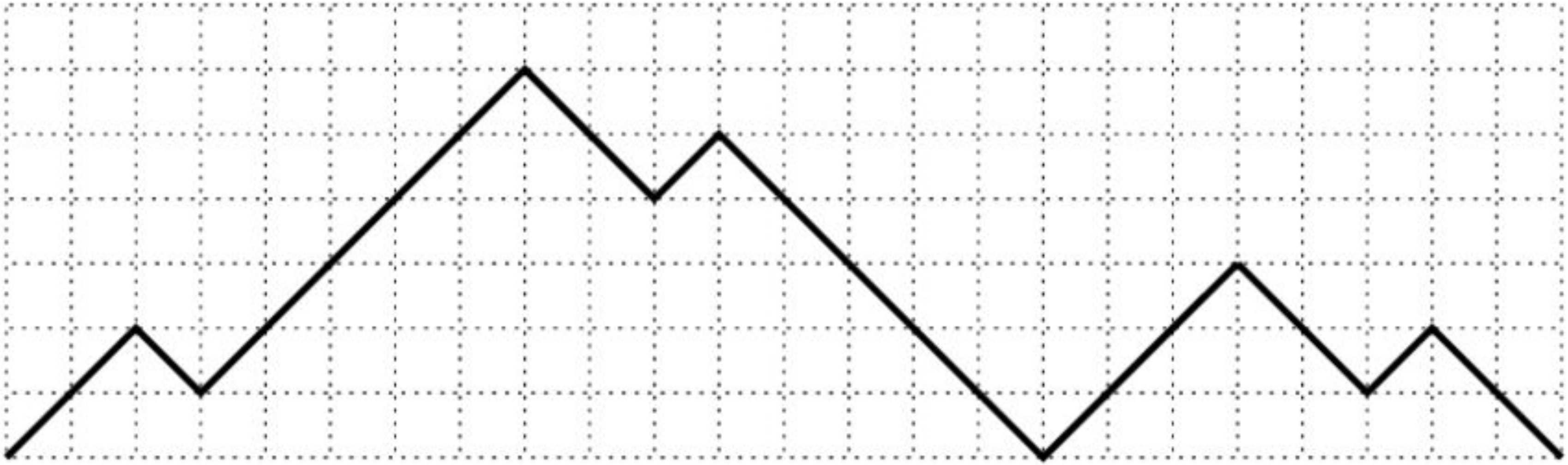


$$\binom{n+m}{n}$$

Dyck paths: paths on a plane

Start at **(0,0)** and **move to the right**. On each step, move diagonally upwards or downwards. Arrive at **(2*n*,0)** <u>without going below the *y*-axis.</u>

How many Dyck paths are there for a fixed *n*? Call this number C_n



Recursive formula?

$$C_0 = C_1 = 1$$

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}, n \geq 2$$

Bijection?

Let's count non-Dyck paths

- Count the paths that **do** go below the *y*-axis.
- Then, subtract this number from the total number of paths.

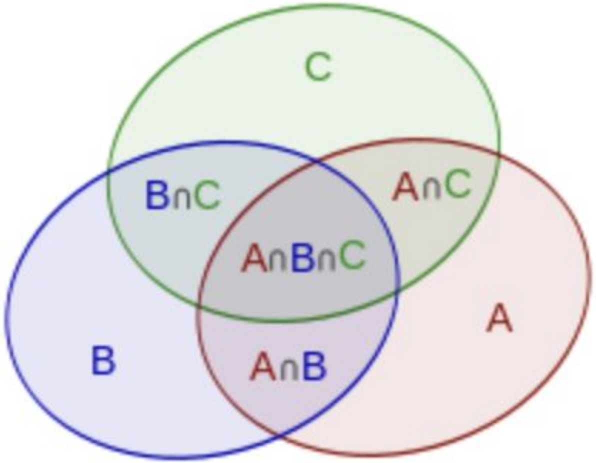
$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}, n \geq 2$$

$$C_n = \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1}\binom{2n}{n}, n \geq 0$$

Catalan Numbers: More applications

- Number of regular bracket sequence consisting of n opening and n closing brackets.
- The number of rooted full binary trees with $n + 1$ leaves (vertices are not numbered). A rooted binary tree is full if every vertex has either two children or no children.
- The number of ways to completely parenthesize $n + 1$ factors.
- The number of ways to connect the $2n$ points on a circle to form n disjoint chords.
- The number of non-isomorphic full binary trees with n internal nodes (i.e. nodes having at least one son).
- ...

The Inclusion-Exclusion Principle: Venn Diagrams



$$S(A \cup B \cup C) = S(A) + S(B) + S(C) - S(A \cap B) - S(A \cap C) - S(B \cap C) + S(A \cap B \cap C)$$

The Inclusion-Exclusion Principle

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{\emptyset \neq J \subseteq \{1,2,\ldots,n\}} (-1)^{|J|-1} \left|\bigcap_{j \in J} A_j\right|$$

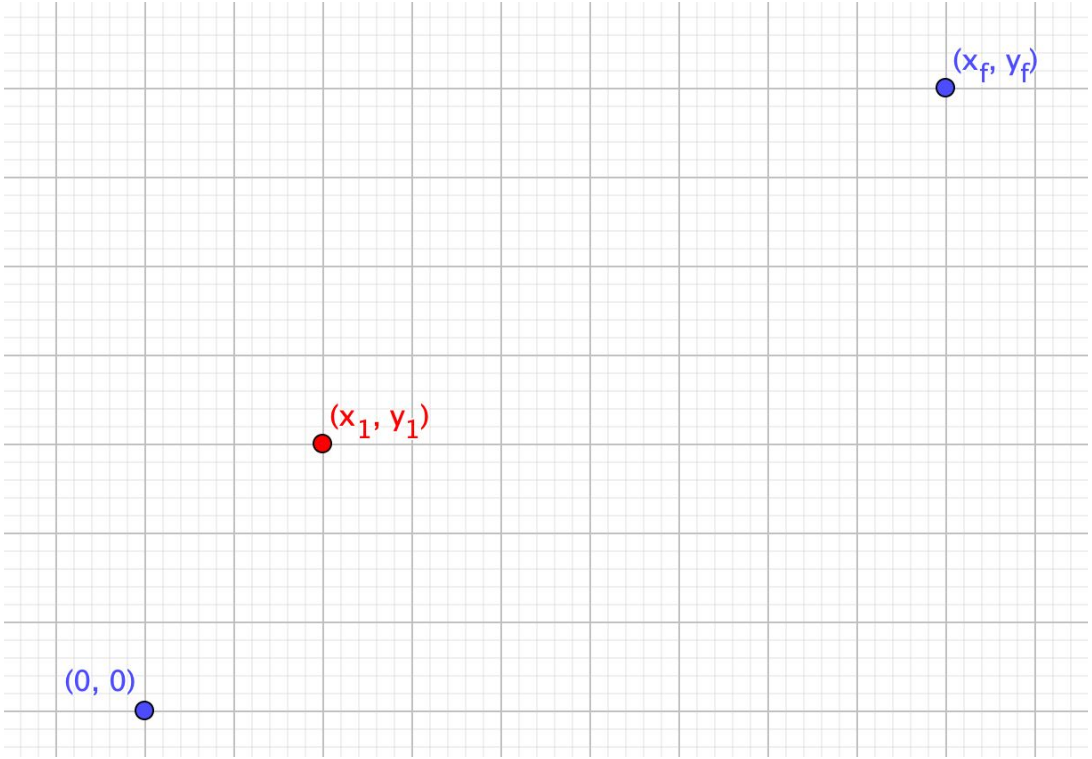
Example: Derangements

- Suppose a teacher wants *n* students to grade each other's tests, so they receive all tests and give one to each student at random. What is the probability that no one receives their own test?

Paths revisited: only up and right moves

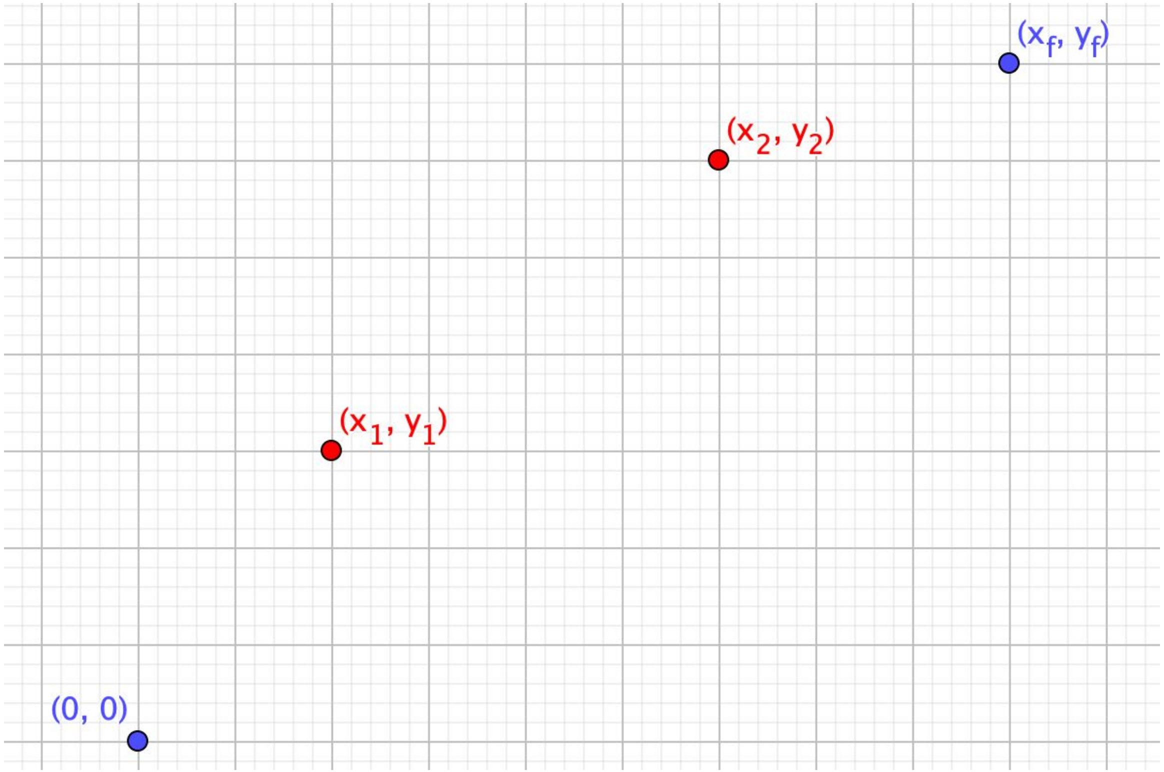
$$\begin{pmatrix} x_f + y_f \\ x_f \end{pmatrix}$$

Can't visit the red lattice point



$$\begin{pmatrix} x_f + y_f \\ x_f \end{pmatrix} - \begin{pmatrix} x_1 + y_1 \\ x_1 \end{pmatrix} \begin{pmatrix} (x_f - x_1) + (y_f - y_1) \\ x_f - x_1 \end{pmatrix}$$

Can’t visit any of the red lattice points



$$\binom{x_f + y_f}{x_f} - \left[\sum_{i=1}^{2} \binom{x_i + y_i}{x_i} \binom{(x_f - x_i) + (y_f - y_i)}{x_f - x_i}\right]$$

$$+ \binom{x_1 + y_1}{x_1} \binom{(x_2 - x_1) + (y_2 - y_1)}{x_2 - x_1} \binom{(x_f - x_2) + (y_f - y_2)}{x_f - x_2}$$

Computing binomial coefficients modulo large prime

$$\binom{n}{k} \equiv n! \cdot (k!)^{-1} \cdot ((n-k)!)^{-1} \mod m.$$

Factorial precomputation

$$n! = n \cdot (n-1)!$$

1. Precalculate factorials, take inverses naïvely.

$$\text{Precomputation: } \mathcal{O}(N).$$

$$\text{Query: } \mathcal{O}(\log \text{MOD}).$$

Can we do better?

Factorial precomputation improved

$$(n!)^{-1} = \left((n+1)!\right)^{-1}(n+1)$$

2. Precalculate factorials up to *N*, find inv_fact(*N*), propagate the answer down. Now you have an array of factorials and their inverses, so you can do lookups for each query.

$$\text{Precomputation: } \mathcal{O}(N + \log \text{MOD}) \approx \mathcal{O}(N).$$

$$\text{Query: } \mathcal{O}(1).$$

Attendance

BREAK #1

Now, let's cover:

- **Modular Arithmetic**
  - Modular Inverse
  - Extended Euclidean Algorithm
  - Linear Congruence Equation
  - Chinese Remainder Theorem
  - Primitive Root
- And if time permits,
  - Discrete Logarithm
  - Discrete Root

Inclusion-Exclusion in number theory

The number of relative primes in a given interval

- Given two numbers `n` and `r`, count the number of integers in the interval `[1, r]` that are relatively prime to `n` (their greatest common divisor is `1`).

The Main Idea

- We will denote the prime factors of n as p_i ($i = 1, ..., k$).

- How many numbers in the interval `[1, r]` are divisible by p_i?

Solution Sketch

- The answer to this question is: $\mathbf{r/p_i}$.

- However, if we simply sum these numbers, some numbers will be counted several times (those that share multiple p_i as their factors).

- Therefore, it is necessary to use the inclusion-exclusion principle.
- We will iterate over all 2^k subsets of p_i's, calculate their product and add or subtract the number of multiples of their product.

Implementation

```
int solve (int n, int r) {
    vector<int> p;
    for (int i=2; i*i<=n; ++i)
        if (n % i == 0) {
            p.push_back (i);
            while (n % i == 0)
                n /= i;
        }
    if (n > 1)
        p.push_back (n);
```

Implementation

```
    int sum = 0;
    for (int msk=1; msk<(1<<p.size()); ++msk) {
        int mult = 1, bits = 0;
        for (int i=0; i<(int)p.size(); ++i)
            if (msk & (1<<i)) ++bits, mult *= p[i];
        int cur = r / mult;
        if (bits % 2 == 1) sum += cur;
        else sum -= cur;
    }
    return r - sum;
}
```

Totient Function

- Euler's totient function, also known as **ϕ-function** $\phi(n)$, 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.

n	1	2	3	4	5	6	7	8	9	10	11	12
$\phi(n)$	1	1	2	2	4	2	6	4	6	4	10	4

Totient Function

$$\phi(p) = p - 1.$$

$$\phi(p^k) = p^k - p^{k-1}.$$

$$\phi(ab) = \phi(a) \cdot \phi(b).$$

$$\phi(ab) = \phi(a) \cdot \phi(b) \cdot \frac{d}{\phi(d)}$$

Totient Function

$$\phi(n) = \phi({p_1}^{a_1}) \cdot \phi({p_2}^{a_2}) \cdots \phi({p_k}^{a_k})$$

$$= \left({p_1}^{a_1} - {p_1}^{a_1-1}\right) \cdot \left({p_2}^{a_2} - {p_2}^{a_2-1}\right) \cdots \left({p_k}^{a_k} - {p_k}^{a_k-1}\right)$$

$$= p_1^{a_1} \cdot \left(1 - \frac{1}{p_1}\right) \cdot p_2^{a_2} \cdot \left(1 - \frac{1}{p_2}\right) \cdots p_k^{a_k} \cdot \left(1 - \frac{1}{p_k}\right)$$

$$= n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_k}\right)$$

Totient Function: Implementation ($n^{1/2}$)

```
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;
}
```

Totient Function: Implementation (n log log n)

```
void phi_1_to_n(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;
        }
    }
}
```

Modular Inverse

- A modular multiplicative inverse of an integer a is an integer x such that ax is congruent to 1 modular some modulus m.

- We want to find an integer x so that:

$$a \cdot x \equiv 1 \mod m$$

- We will also denote x simply with a^{-1}.

Modular Inverse: Extended Euclidean algorithm

- Consider the following equation with unknown x and y:

$$a \cdot x + m \cdot y = 1$$

- This is a Linear Diophantine equation in two variables. When gcd(a, m)=1, the equation has a solution which can be found using the extended Euclidean algorithm. Note that gcd(a, m)=1 is also the condition for the modular inverse to exist.

Modular Inverse: Extended Euclidean algorithm (con't)

- Now, if we take modulo m of both sides, we can get rid of $m \cdot y$ and the equation becomes:

$$a \cdot x \equiv 1 \mod m$$

- Thus, the modular inverse of a is x.

Modular Inverse: Extended Euclidean algorithm (con't)

```
int x, y;
int g = extended_euclidean(a, m, x, y);
if (g != 1) {
  cout << "No solution!";
}
else {
  x = (x % m + m) % m;
  cout << x << endl;
}
```

Extended Euclidean Algorithm

- The extended Euclidean algorithm finds a way to represent GCD in terms of a and b. So, it finds coefficients x and y for which:

$$a \cdot x + b \cdot y = \gcd(a, b)$$

Extended Euclidean Algorithm (con't)

- The changes to the Euclidean algorithm are very simple:
  - We can see that the algorithm ends with $b=0$ and $a=g$.
  - For these parameters we can easily find coefficients, namely $g \cdot 1+0 \cdot 0=g$.
  - Starting from these coefficients $(x,y)=(1,0)$, we can go backwards up the recursive calls.
  - All we need to do is to figure out how the coefficients x and y change during the transition from (a,b) to $(b,a\%b)$.

Extended Euclidean Algorithm (iterative)

- Alternatively, we can keep track of the two triples (a, 1, 0) and (b, 0, 1)
- Apply logic of Euclidean algorithm on these two triples, subtracting and multiplying as you would with vectors until you end up with the triple (g, x, y)
- Each triple acts in a sense as a set of “instructions” for how to produce the first number by multiplying a and b by constants and adding them up

Extended Euclidean Algorithm: Implementation #1

```
int gcd(int a, int b, int& x, int& y) {
  if (b == 0) {
    x = 1;
    y = 0;
    return a;
  }
  int x1, y1;
  int d = gcd(b, a % b, x1, y1);
  x = y1;
  y = x1 - y1 * (a / b);
  return d;
}
```

Extended Euclidean Algorithm: Implementation #2

```
int gcd(int a, int b, int& x, int& y) {
  x = 1, y = 0;
  int x1 = 0, y1 = 1, a1 = a, b1 = b;
  while (b1) {
    int q = a1 / b1;
    tie(x, x1) = make_tuple(x1, x - q * x1);
    tie(y, y1) = make_tuple(y1, y - q * y1);
    tie(a1, b1) = make_tuple(b1, a1 - q * b1);
  }
  return a1;
}
```

Modular Inverse: Binary Exponentiation

- Another method for finding modular inverse is to use **Euler's Totient Theorem**, which states that the following congruence is true if a and m are relatively prime:

$$a^{\phi(m)} \equiv 1 \mod m$$

- where ϕ is totient function.

Modular Inverse: Binary Exponentiation (con't)

- Note that a and m being relative prime is the condition for the modular inverse to exist.

- If m is a prime number, this simplifies to Fermat's little theorem:

$$a^{m-1} \equiv 1 \mod m$$

Modular Inverse: Binary Exponentiation (con't)

- For an arbitrary (but coprime) modulus m:

$$a^{\phi(m)-1} \equiv a^{-1} \mod m$$

- For a prime modulus m:

$$a^{m-2} \equiv a^{-1} \mod m$$

- From here, we can use the binary exponentiation.

Modular Inverse: Binary Exponentiation (con't)

```
ll inv(ll a, ll m) {
    return exp(a, phi(m)-1, m);
}

ll invp(ll a, ll m) {
    return exp(a, m-2, m);
}
```

Linear Congruence Equation

- We just need to solve:

$$a \cdot x \equiv b \pmod{n}$$

- When gcd(a,n) = 1 (coprime), we just need to find the multiplicative inverse:

$$x \equiv b \cdot a^{-1} \pmod{n}$$

Linear Congruence Equation (con't)

- When gcd(a,n) = 1 (coprime), we just need to find the multiplicative inverse:

$$x \equiv b \cdot a^{-1} \pmod{n}$$

- If not, let g := gcd(a,n). If b is not divisible by g, then there is **no solution**! If g divides b, then by dividing both sides of the equation by g (i.e. dividing a, b and n by g), we get:

$$a' \cdot x \equiv b' \pmod{n'}$$

Linear Congruence Equation (con't)

- We get x' as solution for x.
- It is clear that x' will also be a solution of the original equation. However, it will not be the only solution. It can be shown that the original equation has exactly g solutions, given by:

$$x_i \equiv (x' + i \cdot n') \pmod{n} \quad \text{for } i = 0 \ldots g - 1$$

- Summarizing, we can say that the number of solutions of the linear congruence equation is equal to either gcd(a,n) or 0.

Chinese Remainder Theorem

- Where m_i are pairwise coprime, let:

$$m = m_1 \cdot m_2 \cdots m_k$$

- Where a_i are some given constants, suppose we have:

$$\begin{cases} a & \equiv & a_1 \pmod{m_1} \\ a & \equiv & a_2 \pmod{m_2} \\ & \vdots & \\ a & \equiv & a_k \pmod{m_k} \end{cases}$$

Chinese Remainder Theorem (con't)

- The original form of CRT then states that the given system of congruences always has ***one and exactly one solution*** modulo m.

- For example,

$$\begin{cases} a & \equiv & 2 \pmod{3} \\ a & \equiv & 3 \pmod{5} \\ a & \equiv & 2 \pmod{7} \end{cases}$$

- has the solution 23 modulo 105.

Chinese Remainder Theorem (con't)

$$x \equiv a \pmod{m} \quad \overset{\text{equivalent}}{\Longleftrightarrow} \quad \begin{cases} x \equiv a_1 \pmod{m_1} \\ \quad \vdots \\ x \equiv a_k \pmod{m_k} \end{cases}$$

Chinese Remainder Theorem: Solution for Two Moduli

$$\begin{cases} a \equiv a_1 \pmod{m_1} \\ a \equiv a_2 \pmod{m_2} \end{cases}$$

- We want:

$$a \pmod{m_1 m_2}$$

- Using the **Extended Euclidean Algorithm** we can find **Bézout** coefficients n_1, n_2 such that

$$n_1 m_1 + n_2 m_2 = 1$$

Chinese Remainder Theorem: Solution for Two Moduli

$$n_1 \equiv m_1^{-1} \pmod{m_2}$$
$$n_2 \equiv m_2^{-1} \pmod{m_1}$$

Chinese Remainder Theorem: Solution for Two Moduli

$$n_1 \equiv m_1^{-1} \pmod{m_2}$$
$$n_2 \equiv m_2^{-1} \pmod{m_1}$$

$$a = a_1 n_2 m_2 + a_2 n_1 m_1 \bmod m_1 m_2$$

Chinese Remainder Theorem: Solution for Two Moduli

- We can check:

$$\begin{aligned}
a &\equiv a_1 n_2 m_2 + a_2 n_1 m_1 && \pmod{m_1} \\
&\equiv a_1(1 - n_1 m_1) + a_2 n_1 m_1 && \pmod{m_1} \\
&\equiv a_1 - a_1 n_1 m_1 + a_2 n_1 m_1 && \pmod{m_1} \\
&\equiv a_1 && \pmod{m_1}
\end{aligned}$$

- The same holds for m_2 by symmetry!

Chinese Remainder Theorem: Solution for Two Moduli

- **Uniqueness**

$$x \equiv a_i \pmod{m_i} \quad y \equiv a_i \pmod{m_i}$$

Chinese Remainder Theorem: Solution for Two Moduli

- **Uniqueness**

$$x \equiv a_i \pmod{m_i} \qquad y \equiv a_i \pmod{m_i}$$

$$x - y \equiv 0 \pmod{m_i}$$

Chinese Remainder Theorem: Solution for Two Moduli

- **Uniqueness**

$$x \equiv a_i \pmod{m_i} \qquad y \equiv a_i \pmod{m_i}$$

$$x - y \equiv 0 \pmod{m_i}$$

$$x - y \equiv 0 \pmod{m_1 m_2}$$

Chinese Remainder Theorem: Solution for Two Moduli

- **Uniqueness**

$$x \equiv a_i \pmod{m_i} \qquad y \equiv a_i \pmod{m_i}$$

$$x - y \equiv 0 \pmod{m_i}$$

$$x - y \equiv 0 \pmod{m_1 m_2}$$

$$x \equiv y \pmod{m_1 m_2}$$

Chinese Remainder Theorem: Solution for General Case

- **Inductive Solution**
  - As m_1m_2 is coprime to m_3, we can inductively repeatedly apply the solution for two moduli for any number of moduli.

Chinese Remainder Theorem: Solution for General Case

- **Inductive Solution**
  - As m_1m_2 is coprime to m_3, we can inductively repeatedly apply the solution for two moduli for any number of moduli.
- **Direct Construction**

$$M_i := \prod_{i \neq j} m_j \qquad N_i := M_i^{-1} \bmod m_i$$

Chinese Remainder Theorem: Solution for General Case

- **Inductive Solution**
  - As $m_1 m_2$ is coprime to m_3, we can inductively repeatedly apply the solution for two moduli for any number of moduli.
- **Direct Construction**

$$M_i := \prod_{i \neq j} m_j \qquad N_i := M_i^{-1} \bmod m_i$$

$$a \equiv \sum_{i=1}^{k} a_i M_i N_i \pmod{m_1 m_2 \cdots m_k}$$

Chinese Remainder Theorem: Solution for General Case

- We can check this is indeed a solution:

$$\begin{aligned} a &\equiv \textstyle\sum_{j=1}^{k} a_j M_j N_j && \pmod{m_i} \\ &\equiv a_i M_i N_i && \pmod{m_i} \\ &\equiv a_i M_i M_i^{-1} && \pmod{m_i} \\ &\equiv a_i && \pmod{m_i} \end{aligned}$$

Chinese Remainder Theorem: Implementation

```
struct Congruence { ll a, m; };
ll crt(vector<Congruence> const& cs) {
  ll M = 1, ret = 0;
  for (auto const& c : cs)
    M *= c.m;
  for (auto const& c : cs) {
    ll a_i=c.a, M_i=M/c.m, N_i = mod_inv(M_i,c.m);
    ret=(ret+a_i*M_i%M*N_i) % M;
  }
  return ret;
}
```

CRT: Solution for not coprime moduli

- In the not coprime case, a system of congruences has **exactly one** solution modulo **lcm**(m_1, m_2, ..., m_k) or has **no** solution at all.
- Where

$$p_1^{n_1} p_2^{n_2} \cdots p_k^{n_k}$$

- is a prime factorization of m_i, the following are equivalent!

$$a \equiv a_i \pmod{m_i} \qquad\qquad a \equiv a_i \pmod{p_j^{n_j}}$$

CRT: Solution for not coprime moduli (con't)

- Because originally some moduli had common factors, we will get some congruences moduli based on the same prime, however possibly with **different** prime powers.
- The congruence with the **highest** prime power modulus will be the strongest congruence of all congruences based on the same prime number.
- If there are no contradictions, then the system of equation has a solution.
    - We can ignore all congruences except the ones with the highest prime power moduli. These moduli are now coprime. So, we are back to coprime moduli!

Primitive Root

- A number g is called a primitive root modulo n if every number coprime to n is congruent to a power of g modulo n.
  - g is a primitive root modulo n if and only if for any integer a such that gcd(a,n)=1, there exists an integer k such that:

$$g^k \equiv a \pmod{n}$$

- k is then called the **index** or **discrete logarithm** of a to the base g modulo n. g is also called the **generator** of the multiplicative group of integers modulo n.

Primitive Root (con't)

- Primitive root modulo n exists if and only if:
  - **n** is **1**, **2**, **4**, or
  - **n** is power of an odd prime number ($n = p^k$), or
  - **n** is twice power of an odd prime number ($n = 2p^k$).

- This theorem was proved by Gauss in 1801.

Primitive Root: Naive Idea

- From **Lagrange's theorem**, we know that the index of any number modulo n must be a divisor of $\phi(n)$. Thus, it is sufficient to verify for all proper divisor $d \mid \phi(n)$ that g^d is not 1 modulo n. **We can do better!**

Primitive Root: Better Idea

- First, find ϕ(n) and factorize it: $p_1^{a_1} \cdots p_s^{a_s}$
- Then iterate through all numbers g in [1, n], and for each number, to check if it is primitive root, we do the following:
  - Calculate

$$g^{\frac{\phi(n)}{p_i}} \pmod{n}$$

  - If all the calculated values are different from 1, then g is a primitive root.

Practice Problems: Modular Arithmetic

- https://codeforces.com/problemset/problem/300/C
- https://codeforces.com/problemset/problem/622/F
- https://codeforces.com/problemset/problem/717/A
- https://codeforces.com/problemset/problem/896/D
- https://codeforces.com/problemset/problem/687/B
- https://codeforces.com/gym/101853/problem/G
- https://codeforces.com/contest/1106/problem/F

References

- https://cp-algorithms.com/algebra/module-inverse.html
- https://cp-algorithms.com/algebra/linear_congruence_equation.html
- https://cp-algorithms.com/algebra/chinese-remainder-theorem.html
- https://cp-algorithms.com/algebra/discrete-log.html
- https://cp-algorithms.com/algebra/discrete-root.html
- https://cp-algorithms.com/algebra/primitive-root.html

Again, CodeForces Columbia SHP Algorithms Group

- Please join the following group:
  https://codeforces.com/group/lfDmo9iEr5

Strings on April 20!

- On **April 20**, we will cover:
  - **Strings: Fundamentals**
  - **Strings: Matchings**

Slide Deck

- You may **always** find the slide decks from:
  - **https://github.com/yongwhan/yongwhan.github.io/blob/master/columbia/shp**

THANK YOU

Discrete Logarithm

- For given integers a, b, and m, the discrete logarithm is an integer x satisfying:

$$a^x \equiv b \pmod{m}$$

Discrete Logarithm

- For given integers a, b, and m, the discrete logarithm is an integer x satisfying:

$$a^x \equiv b \pmod{m}$$

- **baby-step giant-step algorithm**, an algorithm to compute the discrete logarithm proposed by Shanks in 1971, which has the time complexity $O(m^{1/2})$.
    - This is a meet-in-the-middle algorithm because it uses the technique of separating tasks in half.

Discrete Logarithm (con't)

- Write:

$$x = np - q$$

- Then,

$$a^{np-q} \equiv b \pmod{m}$$

Discrete Logarithm (con't)

- Write:

$$x = np - q$$

- Then,

$$a^{np-q} \equiv b \pmod{m}$$
$$a^{np} \equiv ba^q \pmod{m}$$

Discrete Logarithm

- So, let's write it as:

$$f_1(p) = f_2(q)$$

- Compute f_1 for all possible values of p and sort them and call it L.
- Compute f_2 for all possible values of q and find it in L using binary search/set.
- The time complexity is $O((m/n + n) \log m)$, which is minimized when n is $m^{1/2}$. Then, the time complexity can become $O(m^{1/2} \log m)$.
- We can remove log m by avoiding binary exponentiation!

Discrete Logarithm: When a and m are not coprime

- Let g = gcd(a,m) > 1. Clearly a^x mod m is divisible by g.
- If b is not divisible by g, there is no solution for x.
- If b is divisible by g, let:

$$a = g\alpha, b = g\beta, m = g\nu$$

- Then,

$$\begin{aligned} a^x &\equiv b \mod m \\ (g\alpha)a^{x-1} &\equiv g\beta \mod g\nu \\ \alpha a^{x-1} &\equiv \beta \mod \nu \end{aligned}$$

- We can apply baby-step giant-step algorithm here!

Discrete Root

- Given a prime n and two integers a and k, find all x for which:

$$x^k \equiv a \pmod{n}$$

Discrete Root

- Given a prime n and two integers a and k, find all x for which:

$$x^k \equiv a \pmod{n}$$

- Use discrete logarithm!

Discrete Root: one solution

- Let g be a primitive root modulo n.
- We can easily discard the case where a=0. In this case, obviously there is only one answer: x = 0.

- Otherwise,

$$(g^y)^k \equiv a \pmod n$$

- where

$$x \equiv g^y \pmod n$$

Discrete Root: one solution (con't)

- So, using discrete logarithm, we can find y satisfying:

$$(g^k)^y \equiv a \pmod{n}$$

- Having found y_0, one of the solutions will be:

$$x_0 = g^{y_0} \pmod{n}$$

Discrete Root: all solutions

- We know:

$$x^k \equiv g^{y_0 \cdot k + l \cdot \phi(n)} \equiv a \pmod{n} \forall l \in Z$$

- So, we recover,

$$x = g^{y_0 + \frac{l \cdot \phi(n)}{k}} \pmod{n} \forall l \in Z$$

- where the fraction is an integer. Equivalently:

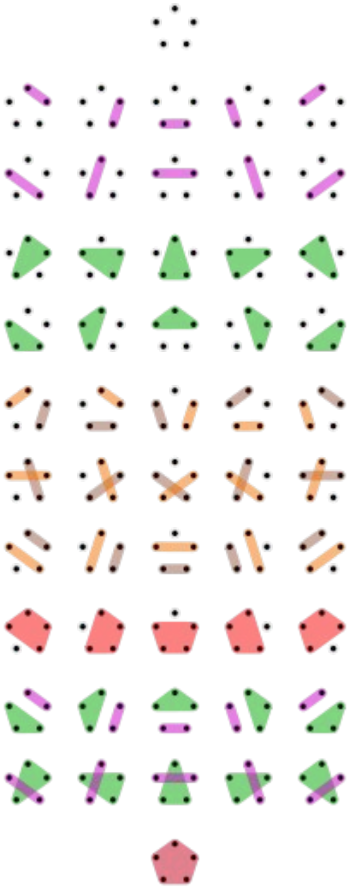
$$x = g^{y_0 + i \frac{\phi(n)}{gcd(k, \phi(n))}} \pmod{n} \forall i \in Z$$

Bell Numbers

- Bell numbers count **the possible partitions of a set**.
- For example, when n=3 (e.g., {a,b,c}), we have:
  - {{a},{b},{c}};
  - {{a},{b,c}};
  - {{b},{a,c}};
  - {{c},{a,b}};
  - {{a,b,c}};

Bell Numbers (A000110)

- 1
- 1
- 2
- 5
- 15
- 52
- 203
- 877
- 4140
- ...



Bell Numbers: Recurrence & Explicit

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

Binomial coefficient

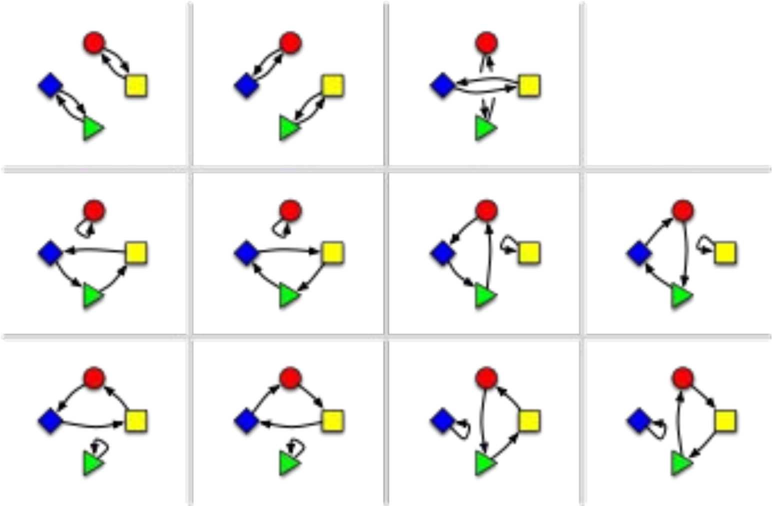
$$B_n = \sum_{k=0}^{n} \left\{ {n \atop k} \right\}$$

Stirling number of second kind

number of ways to partition a set of cardinality n into exactly k nonempty subsets

Stirling numbers of the first kind

- Count permutations according to their number of cycles (counting fixed points as cycles of length one)



$$\begin{bmatrix} 4 \\ 2 \end{bmatrix} = 11$$

Stirling numbers of the first kind: Recurrence

$$\begin{bmatrix} n+1 \\ k \end{bmatrix} = n \begin{bmatrix} n \\ k \end{bmatrix} + \begin{bmatrix} n \\ k-1 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1 \quad \begin{bmatrix} n \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ n \end{bmatrix} = 0$$

Stirling numbers of the first kind: Explicit

$$s(n, n-p) = \frac{1}{(n-p-1)!} \sum_{0 \le k_1, \ldots, k_p : \sum_1^p m k_m = p} (-1)^K \frac{(n+K-1)!}{k_1! k_2! \cdots k_p! \, 2!^{k_1} \, 3!^{k_2} \, \cdots (p+1)!^{k_p}}$$

Stirling numbers of the second kind

- the number of ways to partition a set of n objects into k non-empty subsets

Stirling numbers of the second kind: Recurrence

$$\left\{ {n+1 \atop k} \right\} = k \left\{ {n \atop k} \right\} + \left\{ {n \atop k-1} \right\} \quad \text{for } 0 < k < n$$

$$\left\{ {n \atop n} \right\} = 1 \quad \text{for } n \geq 0 \quad \text{and} \quad \left\{ {n \atop 0} \right\} = \left\{ {0 \atop n} \right\} = 0 \quad \text{for } n > 0.$$

Stirling numbers of the second kind: Explicit

$$\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n$$